Array Bounds Check Elimination in the Context of Deoptimization ⋆

Thomas Würthinger, Christian Wimmer, Hanspeter Mössenböck
Institute for System Software, Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University Linz, Austria

Abstract

Whenever an array element is accessed, Java virtual machines execute a compare instruction to ensure that the index value is within the valid bounds. This reduces the execution speed of Java programs. Array bounds check elimination identifies situations in which such checks are redundant and can be removed. We present an array bounds check elimination algorithm for the Java HotSpot™ VM based on static analysis in the just-in-time compiler.

The algorithm works on an intermediate representation in static single assignment form and maintains conditions for index expressions. It fully removes bounds checks if it can be proven that they never fail. Whenever possible, it moves bounds checks out of loops. The static number of checks remains the same, but a check inside a loop is likely to be executed more often. If such a check fails, the executing program falls back to interpreted mode, avoiding the problem that an exception is thrown at the wrong place.

The evaluation shows a speedup near to the theoretical maximum for the scientific SciMark benchmark suite and also significant improvements for some Java Grande benchmarks. The algorithm slightly increases the execution speed for the SPECjvm98 benchmark suite. The evaluation of the DaCapo benchmarks shows that array bounds checks do not have a significant impact on the performance of object-oriented applications.

Key words: Java, array bounds check elimination, just-in-time compilation, optimization, performance

1. Introduction

To ensure safe execution of programs within a virtual machine, every illegal memory access must be intercepted. For field accesses, this is done by type checking and verification of the field offset at compile time. Array accesses, however, require a run-time check to verify that the specified index is within the bounds of the array. In case of Java, the lower bound of an array is always zero, while the upper bound is the length of the array minus one. If the index is not within this range, an `ArrayIndexOutOfBoundsException` must be thrown. The overhead introduced by such checks can be significant, especially for mathematical applications, but checks fail only in rare cases. When it can be proven at compile time that a check never fails, it can be omitted. Such a check is said to be *fully redundant*.

⋆ This work was supported by Sun Microsystems, Inc.
Email addresses: `wuerthinger@ssw.jku.at` (Thomas Würthinger), `wimmer@ssw.jku.at` (Christian Wimmer), `moessenboeck@ssw.jku.at` (Hanspeter Mössenböck).

There are situations where checks are not fully redundant, but the total number of executed checks can be reduced by moving checks or combining several checks into a single one. For example, a check performed within a loop is likely to be executed more often than a check before the loop. The total number of dynamically performed checks can be reduced by replacing such a check with another one. In this case, the check is said to be *partially redundant*.

One important property that must be kept in mind when eliminating or moving checks in Java programs is that the semantics must stay the same. When a check fails, the exception must be thrown at the correct code position of the failing array access. It is not allowed to just stop the program when an array is accessed out of its valid bounds.

This paper describes our array bounds check elimination algorithm that tries to minimize the total number of dynamically executed checks in Java programs. It works as an additional optimization step on the just-in-time compiler's intermediate language, which is in static single assignment form. It eliminates checks that can be proven to be fully redundant and inserts additional instructions to be able to remove partially redundant checks by grouping multiple checks or moving checks out of loops.

To avoid loop versioning, i.e. the duplication of code, and nevertheless retain the exception semantics of Java, we use the facilities of the Java HotSpot^{TM} VM to switch back from compiled to interpreted code. Our algorithm focuses on program structures that are common to Java programs and eliminates the checks with a low impact on compile time. This is important because it is integrated into a fast just-in-time compiler where the additional time needed for compilation decreases the total execution speed.

We implemented our analysis for the client compiler of the Java HotSpot^{TM} VM. This paper contributes the following:

- We present a fast algorithm for array bounds check elimination that is suitable for a just-in-time compiler.
- We preserve the exception semantics of Java by using deoptimization.
- We show how to handle integer overflow when checking bound conditions.
- The evaluation shows the impacts of our algorithm on several benchmarks. We compare our results to the speedup theoretically achievable by array bounds check elimination.

In addition to our previous conference paper [27], this extended journal version contributes the following new elements:

- We present an in-depth analysis of our algorithm with more examples and details.
- We discuss problems of bounds check elimination in the context of on-stack-replacement of methods.
- We compare our approach with the bounds check elimination of the Java HotSpot^{TM} server compiler.
- We evaluate our algorithm with additional benchmarks, which show that bounds check elimination has a significant impact on scientific applications, but hardly any impact on object-oriented applications.

2. Motivational Example

Listing 1 shows a fragment of the method `String.hashCode()` that calculates the hash value of a `String` object. The character array `value` is accessed within the loop.

```
int hashCode = 0;
for (int i = offset; i < limit; i++) {
  hashCode = hashCode * 31 + value[i];
}
```

Listing 1. Calculating the hash code of a `String`.

According to the semantics of an array access in Java, the method could be rewritten as shown in Listing 2. The array is accessed only when the index `i` is within the valid bounds. Otherwise, an exception is thrown. In this example, the condition will always be true and the code with the exception will never be executed because `limit` is computed correctly. A run-time exception is viewed as a program bug, so in a correct program no array bounds check should ever fail. This makes bounds checks different from explicitly coded if-statements, where both the if-branch and the else-branch are expected to be executed. Although the

compiler cannot assume that the bounds check will never fail, it can assume that the bounds check will most likely not fail. Therefore, better optimizations are reachable when array bounds checks are treated differently from other conditions.

```
int hashCode = 0;
for (int i = offset; i < limit; i++) {
  if (i >= 0 && i < value.length) {
    // safely access array
    hashCode = hashCode * 31 + value[i];
  } else {
    // exceptional case, should never occur
    throw new ArrayIndexOutOfBoundsException();
  }
}
```

Listing 2. Hand-coded array bounds check.

3. System Overview

The main components of the Java HotSpot^{TM} VM include the run-time system, the garbage collector, and the interpreter. Furthermore, two different just-in-time compilers are available, called the *client compiler* and the *server compiler*. The server compiler [20] performs aggressive optimizations and produces fast machine code, however the time needed to compile a method is high. This is acceptable for long-running server applications, but not for interactive desktop applications where response time is more important than peak performance. The client compiler [11,18] achieves a high compilation speed by omitting time-consuming optimizations.

Both compilers can apply optimizations on optimistic assumptions. If an optimization is invalidated later, e.g. because of dynamic class loading, the VM can *deoptimize* the machine code [14] at discrete points, called *safepoints*. Execution is stopped and reverted back to the interpreter. The local variables and the current operand stack of the interpreter are reconstructed from the values of the registers and the memory.

Figure 1 shows the main components of the client compiler. The compiler is invoked only for frequently executed methods. It transforms the Java bytecodes of the input method to machine code with the help of two intermediate data structures, called the high-level intermediate representation (HIR) and the low-level intermediate representation (LIR). The instructions are organized in basic blocks, which are groups of sequentially executed instructions. Only exceptions allow control flow to exit the predefined execution order and jump to an exception block.

At the end of a basic block, control flow instructions that jump to other basic blocks like `goto` or `if` are allowed. A block is linked with all its predecessors and successors. The HIR is in static single assignment (SSA) form [8], i.e. there is always only a single point of assignment for each instruction. When control flow merges, phi instructions are inserted to merge different values of a variable. Data dependencies replace the local variables and the operand stack used by the Java bytecodes.

After the HIR is constructed, various global optimizations are performed, including global value numbering [4] and elimination of null checks. Our new bounds check elimination algorithm operates on the HIR just before the generation of the LIR. It is applied after all other optimizations on the HIR were performed because it can profit from them.

The LIR is close to a three operand machine code, but still platform independent. It is used for linear scan register allocation [26]. From the LIR, the final platform-dependent machine code is generated.

Two LIR instructions are necessary to perform the bounds check for every HIR instruction that accesses an array: a compare instruction of the array index and the array length, and a conditional branch to an out-of-line code stub that throws the exception if the check fails. As the lower bound of an array is always 0, the check whether the index is within the valid range can be reduced to a single unsigned compare. The

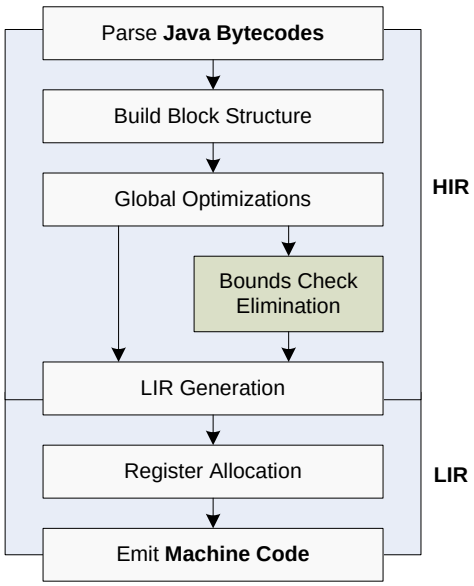


Fig. 1. Structure of the client compiler.

comparison if `a` is at least 0 and smaller than `b` can be checked by regarding `a` as an unsigned value and testing only whether it is smaller than `b`.

The current production version of the client compiler does not perform any kind of sophisticated array bounds check elimination. A bounds check is eliminated only when both the index and the length of the array are compile-time constants. This case is rare.

Our algorithm is implemented as a separate optimization phase just before LIR generation. It marks those array access instructions with a flag whose bounds checks are redundant and adds additional HIR instructions in case of partially redundant checks. When generating the LIR for an array access instruction, the flag is used to decide whether the bounds check must be emitted.

Our algorithm does not perform interprocedural analysis. This would require checking the bytecodes of all methods, also of those that never get compiled. Additionally, dynamic class loading could invalidate interprocedural information and therefore lead to additional deoptimizations.

4. Conditions

Our array bounds check elimination algorithm maintains conditions for index variables to decide whether a given index is within the correct bounds. We keep the kind of conditions as simple as possible without significantly reducing the number of eliminated checks in average Java programs. In comparison to other approaches [3,22], we do not use an inequality graph. Instead, the algorithm keeps a condition of the following form for every instruction `x` that computes an integer value:

$$i_{lower} + c_{lower} <= x <= i_{upper} + c_{upper}$$

Variables i_{lower} and i_{upper} refer to HIR instructions (i.e. the values produced by them), while c_{lower} and c_{upper} are integer constants. If the instruction parts of a condition are missing, the bounds are compile-time constants. Initially, when nothing about the bounds is known yet, every instruction has the condition

$$\texttt{MIN} <= x <= \texttt{MAX}$$

where `MIN` and `MAX` denote the minimum and maximum possible values of a 32-bit signed integer. Even when the bounds of a variable are a conjunction or disjunction of several conditions, the algorithm stores only a single condition. This is a loss of information, but a practical simplification in most cases. It makes the algorithm simpler and faster because it reduces the amount of data that must be processed.

Maintaining only a single condition requires that the algorithm must combine several conditions into a single one. Two conditions can be conjuncted or disjuncted. The upper and the lower bound are treated separately by these operations. The algorithm needs to be conservative, as a wrong condition could mean that a necessary bounds check is removed. In the following example two conditions are disjuncted, i.e. the algorithm knows only that at least one of them is true:

$$a <= x \text{ or } b <= x$$

If we do not know whether a is greater than b or vice versa, we have no new lower bound for `x`. A disjunction of conditions can occur for example after an `if` statement when the values of `x` coming from different possible control flow paths are merged.

In the case of a conjunction, we can safely take either of the two conditions as the new condition for `x`. The algorithm takes the later obtained condition as the new one. In the following example, the resulting condition for `x` will be that `x` is at least `b`, when we do not know whether `a` is greater than `b` or not:

$$a <= x \text{ and } b <= x$$

4.1. *Dominator Tree*

Our algorithm benefits from the SSA form of the HIR where each variable is assigned only at a single place in the program, i.e. its value does not change after its definition. But even if a method is in SSA form, the conditions for a variable are not the same at different points of the method. Control flow instructions like `if` do not modify the value of the operands, but do modify their bounds in the succeeding basic blocks.

The algorithm processes the blocks in a dominator-based order and maintains a stack of conditions for each variable, where the topmost stack element is the current valid condition. A dominator of a block is a block that is always executed before the block itself is executed. A range condition that holds in one of the dominators also holds in the block itself. It can only get stronger. Therefore, the algorithm uses a pre-order traversal of the dominator tree. When a block is processed, the condition for a variable is the conjunction of all conditions on the variable in the parent blocks in the dominator tree. Using this approach the algorithm avoids building an extended SSA form like in [3]. To be able to merge the conditions of variables from different control flows, the blocks are processed such that the predecessors of a block are processed before the block itself is processed. Only loop headers are an exception.

The calculation of the dominator of a block is done in one of the earlier phases during compilation. Our algorithm takes the results and constructs the dominator tree from this information. Figure 2 shows the control flow graph of an example method and the corresponding dominator tree. Block B1 is the first block of the method and will always be executed before any other block is executed. Therefore, this block is the root of the dominator tree and is processed first by our algorithm. Block B2 is a loop header, i.e. when entering the loop, this block is always executed first.

In the dominator tree, all blocks of a loop are children of the loop header block. The two alternatives B3 and B6 are merged at block B4. All three blocks are immediate children of the loop header in the dominator tree. Our algorithm processes blocks B3 and B6 before block B4. Because of this, the conditions for values merged by phi instructions of block B4 are calculated before the condition for the phi instruction. The final processing order is shown on the right side of each block.

4.2. *Initial Conditions for Instructions*

Generally, the initial condition of an integer instruction is that it is at least the minimum integer value and at most the maximum integer value. For some instructions, however, more specific initial conditions can be derived. The lower and upper bounds of a constant instruction can be set to the constant itself. Another example is the *bitwise and* operator `&`. If one of its operands is a positive constant, then the result is always at least zero and at most the constant operand. Similarly, bounds can be computed for the *modulo* operator `%`. The bounds for addition or subtraction instructions with constants involved are derived from the

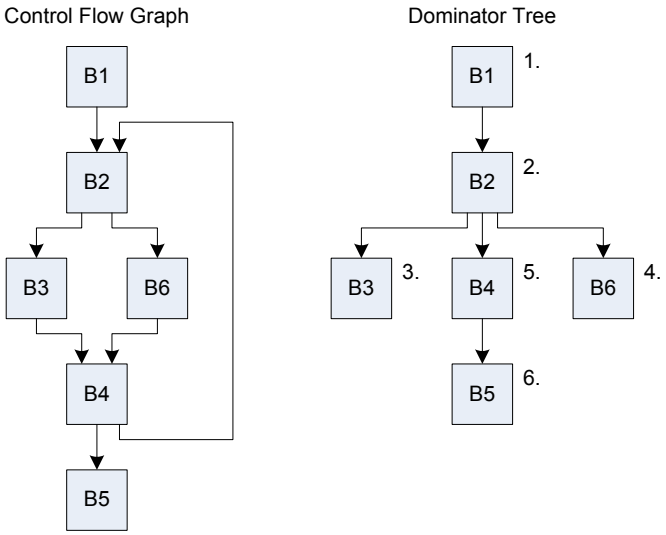


Fig. 2. Dominator tree and processing order.

bounds of the non-constant operand. The Java if-operator with two constant values, e.g. `(a > b) ? 0 : 1`, gets the range between the two constants as its initial bounds.

Listing 3 is a fragment of the method `Hashtable.get()` of the Java collection library. Our algorithm derives from the arithmetic operations that the array access does not require a bounds check. Figure 3 shows how the initial conditions are calculated. First, we derive that `hash & C` is always positive, because one of the operands of the bitwise and operation is positive. An array length must also always be positive, so we have a modulo operation with two positive operands. We can derive that the result must be between 0 and the second operand minus one. The resulting condition is then sufficient for removing the bounds check.

```
final int C = 0x7FFFFFFF;
int hash = key.hashCode();
int index = (hash & C) % tab.length;
Entry e = tab[index];
```

Listing 3. Fragment of the method `Hashtable.get()`.

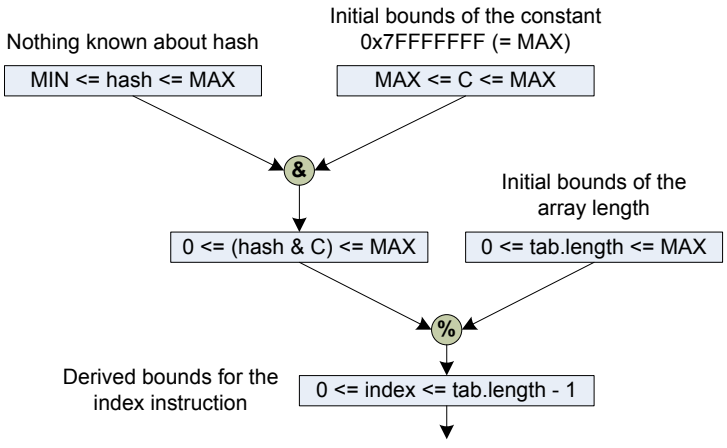


Fig. 3. Example for calculating initial bounds of instructions.

5. Elimination of Bounds Checks

We distinguish three cases where bounds checks can be eliminated: If the index variable is guaranteed to be below the array length, the check can be simply omitted. If the check is in a loop, the array length is loop invariant, and the maximum value of the index variable is known, then the check can be moved out of the loop. Finally, several checks involving the same array in the same basic block can be grouped together to one check.

5.1. *Fully Redundant Checks*

Checks are marked as fully redundant when it can be statically proven that the index expression is always nonnegative and smaller than the array length. Figure 4 presents the HIR of the Java method `get()` shown in Listing 4. The method is split into four blocks. At the end of block 1, an `if` instruction checks whether `p <= a.length`, and an `if` instruction in block 2 checks whether `p > 0`. If both conditions hold, the array load is performed in block 3 and the value is returned. The last block, which returns 0 if one of the conditions does not hold, is omitted for simplicity.

```
int get(int[] a, int p) {
  if (p <= a.length && p > 0) {
    return a[p - 1];
  }
  return 0;
}
```

Listing 4. Example method `get()`.

In this example, the root of the dominator tree is block 1 as this block is the start of the method. Block 2 is an immediate child of block 1 and also the parent of block 3. The bounds check elimination algorithm therefore processes the blocks in the same order as they are numbered.

Figure 5 shows how the conditions for the expressions `p` and `p-1` are derived. At every `if` instruction, new conditions for the affected expressions are combined with previously obtained conditions using a conjunction. In case of two-operand operations like additions or subtractions where one operand is constant, the conditions are modified to reflect this operation. In Figure 5, for example, the condition for `p` is converted into a condition for `p-1` by subtracting 1 from the constant part of both the lower and the upper bound.

Using the condition of the index expression `p-1`, it can be proven that the index is always within the valid range. The lower bound is nonnegative and the upper bound is smaller than the array length, so the array bounds check is fully redundant and can be omitted.

Blocks that do not dominate a block with an array access instruction in it are not processed because conditions derived in these blocks cannot be used to eliminate a bounds check. When building the basic blocks from the Java bytecodes, a flag for the corresponding block is set when such an instruction is added. Blocks that need not be processed are removed from the dominator tree before the analysis.

When executing the SPECjvm98 benchmark suite, 72% of all compiled methods do not contain an array access at all, so they are not processed by the algorithm. Additionally, 64% of the blocks can be eliminated because they do not dominate blocks containing bounds checks. In contrast to other approaches such as [12], we do not reduce the whole instruction graph to contain only instructions used as array indices because this would require disproportionally many changes of the HIR and the run-time cost for maintaining the conditions for all integer instructions is low.

5.2. *Loop-Invariant Checks*

The number of fully redundant checks in an average Java program is not high because it is required that the array length is compared with the index expression before the array access. In contrast, the method

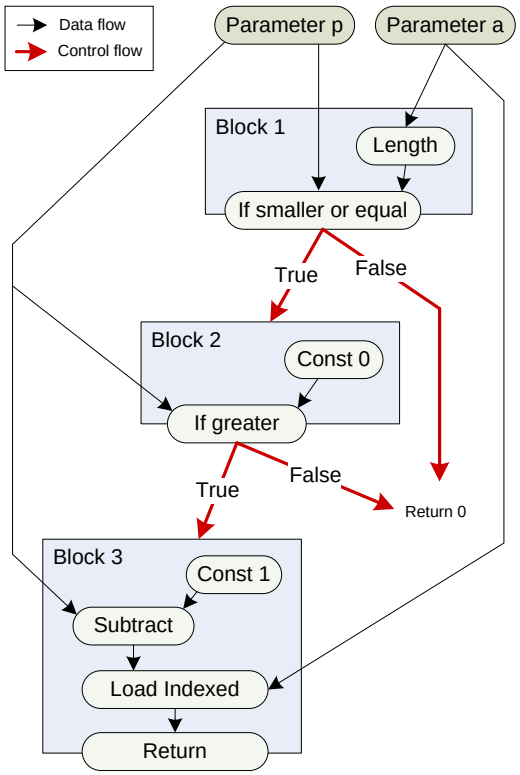


Fig. 4. HIR of the method `get()`.

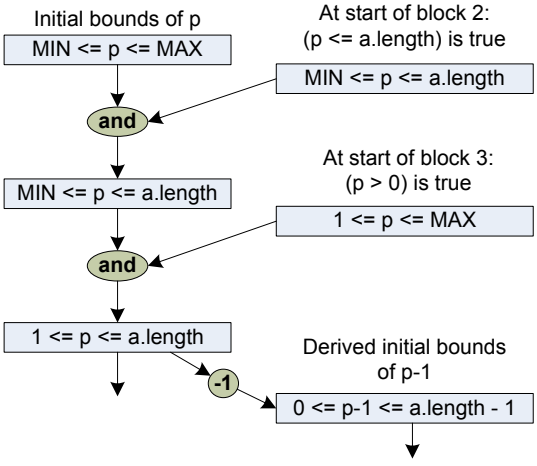


Fig. 5. Conditions for the values of the method `get()`.

`clear()` shown in Listing 5 uses a frequent code pattern where an array is accessed in a loop, but the array length is not checked explicitly.

The loop variable is used as the array index. The array and the variable `x` are not changed within the loop, i.e. they are *loop-invariant*. At the point of the array access, the upper bound of the variable `i` is known when looking at the loop condition `i <= x-1`. The lower bound can be inferred from the fact that the start value of `i` is 0 and `i` is never decreased.

Figure 6 shows the HIR of the method. The important part for proving that `i` only increases is the `phi` function. Phi functions are necessary in the SSA form to merge different values of the same variable when

```
void clear(int[] a, int x) {
  for (int i = 0; i < x; i++) {
    a[i] = 0;
  }
}
```

Listing 5. Example method `clear()`.

control flow joins. In the example, the phi function merges the values of `i` that come from the two predecessor blocks 1 and 3.

When processing block 2, our algorithm detects that the `phi` and the `add` instruction form a cycle with no other instructions involved. It derives from this construct that the value of `i` is always increased within the loop. Therefore, only the upper bound of the phi instruction must be set to `MAX`, while the lower bound is the start value of `i` before the loop, i.e. the constant 0.

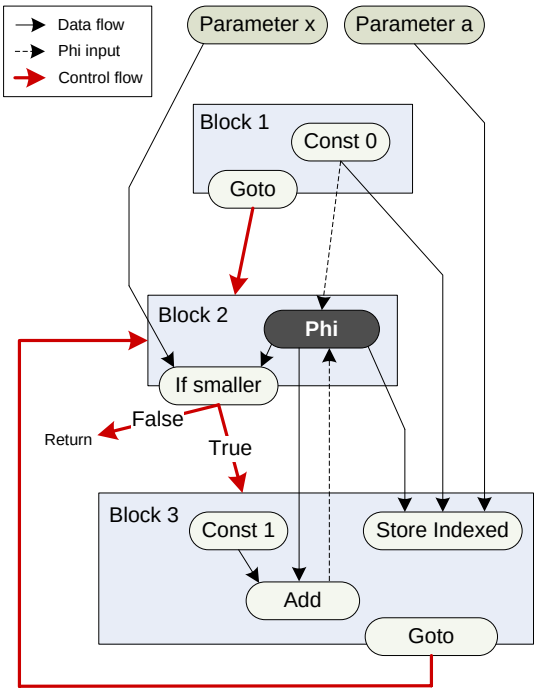


Fig. 6. HIR of the method `clear()`.

At the beginning of block 3, the preceding `if` instruction has been evaluated, so an upper bound for the `phi` instruction is known too. The two conditions are combined using a conjunction, so the resulting condition on the index variable at the array access instruction is

$$0 <= i <= x - 1$$

This condition is not sufficient to fully eliminate the check, because it does not incorporate the actual array length. The upper bound of the index as well as the array are parameters, so the method could be called with the parameter `x` being greater than the length of the array `a` and the method could throw an exception. The check is not fully redundant, so it cannot be eliminated by an intraprocedural analysis.

Because the length of the array and the bounds of the index do not change within the loop, the check is partially redundant. It can be replaced with a check before the loop. While an instruction in a loop is likely to be executed multiple times, the instruction before the loop is executed exactly once. Only if the loop is never executed because the parameter `x` is 0, the new bounds check is unnecessary, but this overhead can be neglected.

In the example, the variable `x` as well as the constant 0 are both *loop-invariant*. The bounds check can be replaced by a check before the loop whether `x` exceeds the array length. Finding out whether an instruction is loop-invariant can be done in constant time using the dominator tree. Any currently referenced instruction must be defined in a block that lies on the path between the current block and the root of the dominator tree. When the block of an instruction lies between the loop header block and the root block, then the instruction is loop-invariant.

A problem however arises because of the exception semantics of Java. Even in optimized machine code, an exception must be thrown at the same point during program execution as the interpreter would throw it. So we must not throw the exception before the loop even if we know that a bounds check fails at some point during the loop execution. If the exception would be thrown for example after 10 iterations, then these 10 iterations must be executed. It is therefore not allowed to insert a normal bounds check before the loop.

A common solution to this problem uses loop versioning (see for example [19]). An optimized version of the loop without bounds checks is executed when it is known that the checks are unnecessary, and the unmodified loop with bounds checks is executed otherwise. However, this solution duplicates the code of the loop and therefore bloats the method with backup code that is rarely or even never executed.

To avoid this, our algorithm inserts an instruction that triggers deoptimization if the check in front of the loop fails. The optimized machine code without the bounds check is then discarded and the method is executed in the interpreter instead, which will throw the exception at the correct point. In the example, the algorithm adds an instruction that triggers deoptimization if the parameter `x` is greater than the array length.

Figure 7 shows the HIR after the insertion of the deoptimization instruction. If the condition is true, the compiled machine code is thrown away and the interpreter continues executing the method.

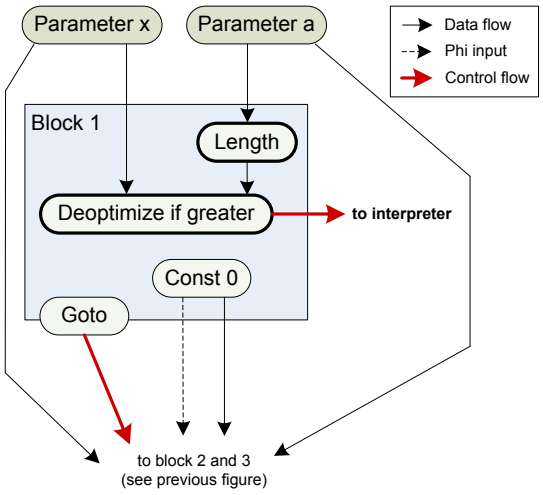


Fig. 7. After insertion of instructions.

We may even deoptimize when the program would have executed completely without an exception. This can be the case when the loop body is more complex and contains additional loop exits. The execution in the interpreter is slower, but after some time the method is recompiled again. To prevent cycles of compilation and deoptimization, we use a flag to ensure that the aggressive optimizations that caused a deoptimization are not applied again when a method is recompiled. So there is a benefit when the optimistic assumptions expressed by the deoptimize instructions at the beginning of the loop hold, but only a small penalty when they fail.

The deoptimize instruction before the loop accesses the length of the array and can therefore cause a `NullPointerException`. Again, the exception would be thrown at a wrong place. Therefore, we need to deoptimize instead of directly throwing the exception. In most cases, the null check is carried out implicitly

by the hardware [17]. We introduced a flag that marks an instruction such that the implicit null check will call deoptimization instead of throwing the exception.

5.3. *Overflow of Loop Variable*

The analysis that determines whether a variable of a Java program is always increasing within a loop must take integer overflow into account. The result of an addition that would be greater than the maximum integer value is a negative value. Therefore, we need to proof not only that the added value is at least zero, but also that no overflow can occur. Because our algorithm only processes the addition of constant values to loop variables, checking the first part is trivial.

Proving the second part, however, is impossible in most cases. Therefore, we need to make sure that deoptimization is called when the loop variable can overflow. This could be done by the following explicit check before the loop. As defined in Listing 5, `x` denotes the value never reached by the loop variable and `c` the constant that is added to the loop variable in each iteration:

$$\texttt{deoptimize if}\ \ x > MAX - c + 1$$

However, when the condition for the index variable is used to eliminate a bounds check, then the following deoptimize instruction is already inserted before the loop:

$$\texttt{deoptimize if}\ \ x > a.length$$

So the algorithm can safely assume that the loop variable cannot overflow if the condition

$$a.length <= MAX - c + 1$$

is always true. When the variable can overflow, deoptimization is called anyway, so we do not need to care about this case. As the length of an array must fit into a 32-bit signed integer value, this condition holds for sure if `c` is equal to 1. The maximum length of an array is also bound by the maximum heap size divided by the size of a single array element in bytes. So in most cases higher values for `c` are also acceptable.

The opposite case, when we want to show that a value is always decreasing, is simpler. The lower bound of the loop variable is checked to be at least zero by the deoptimize instruction. If this check succeeds, no subtraction of any positive value can cause an underflow.

5.4. *Grouping Checks*

Another way of reducing the number of executed bounds checks is to group multiple checks that affect the same array into a single one. We apply this optimization for bounds checks that are not removed by the previously discussed techniques. To simplify the analysis, it is limited to checks that occur within the same basic block and where the index expressions differ only by constants. Listing 6 shows the method `triple()` with three array stores. The bounds checks of all three stores can be folded into one check before the first store. Again, deoptimization is needed to ensure that the exception is thrown at the correct position.

```
void triple(int[] a, int i) {
  a[i] = 0;
  a[i+1] = 1;
  a[i+2] = 2;
}
```

Listing 6. Example method `triple()`.

Figure 8 shows the HIR representation of the method. The algorithm needs a single pass over the instructions of each basic block. For all array variables and index variables, it maintains the minimum and the maximum constants that are added to the index variable when a certain array is accessed. Array accesses whose bounds checks have been removed by previous optimizations are ignored. Instead of the bounds checks, two deoptimize instructions are inserted. They check whether the index variable plus the minimum

constant and the index variable plus the maximum constant are within the bounds of the array. When this condition holds, also all other array accesses involving the same index variable and the same array are safe.

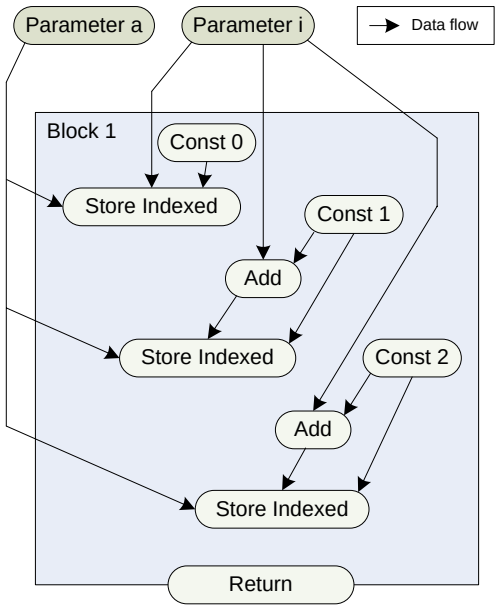


Fig. 8. HIR of the method `triple()`.

Grouping bounds checks is only profitable if more than two checks can be grouped together. This is because in Java the lower bound of an array is always 0 and therefore a normal bounds check can be done with a single unsigned compare instruction. For a grouped bounds check, however, separate compare instructions for the lower and the upper bound are necessary. In the example, the three bounds checks are replaced by the following two deoptimize instructions at the start of the basic block:

$$\texttt{deoptimize if} \quad i < 0$$

$$\texttt{deoptimize if} \quad i + 2 >= a.length$$

The calculation of $i + 2$ may cause an overflow. However, this case can be handled without any additional costs by using an unsigned $>=$ comparison, so that deoptimization is called when $i + 2$ overflows (the signed comparison $i >= a.length - 2$ would also be possible, but would require an additional arithmetic instruction). The comparison for < 0 of the first deoptimization instruction can also be performed by an unsigned comparison for $>= a.length$. This automatically handles the case when the addition of the index variable and the minimum constant leads to an underflow. The only case that is not handled automatically using these unsigned comparisons is when the difference between the lower and upper constant added to the index is greater than the maximum possible integer value. We do not optimize this uncommon case.

Array accesses with constant indices are treated separately. Here the lower bound can be checked at compile time, so we only maintain the maximum constant index for each array. Only one deoptimization instruction is needed, so grouping array accesses with constant indices is profitable for even only two checks.

When the constant added to the index variable is between the maximum and minimum constants added at previous array accesses, the bounds check is fully redundant and is immediately removed. So when the accesses at `i+1` and `i+2` in Listing 6 would be swapped, then the access to `i+1` would be safe. If `i+1` was out of the bounds of the array, then one of the previous array accesses would have thrown an exception.

6. Implementation Details

This section presents implementation details that are more specific to the Java HotSpot™ VM, like the handling of methods that are compiled for on-stack-replacement. Additionally, we compare our algorithm, which is tailored for the client compiler, with the algorithm for bounds check elimination in the current product version of the server compiler.

6.1. *Supporting Optimizations*

The array bounds check elimination algorithm is applied as one of the last optimization steps before the HIR is converted to the LIR. Other optimizations applied prior to our algorithm can lead to more bounds checks being eliminated. The built-in optimizations of the client compiler like constant folding and global value numbering help to identify stronger conditions on the index variables.

To increase the probability that a check can be moved out of a loop, we added a simple form of loop-invariant code motion. Constant expressions are moved out of loops. For arithmetic operations, both operands need to be loop-invariant.

Because of possible aliasing effects, we need to be conservative when moving field loads or array accesses. A field load is only moved out of the loop when there is no field store to this field within the loop. This conservative disambiguation by field name turned out to be sufficient. An array access is moved when the array and the index instructions are loop-invariant and there is no array store to an array of the same basic type within the loop. The basic type only disambiguates scalar types, but does not distinguish between different reference types for simplicity.

When moving instructions, we need to care about exceptions to occur at the correct code position. A field load or an array load can cause a `NullPointerException`. We call deoptimization instead of throwing the exception immediately to preserve the semantics.

6.2. *On-Stack-Replacement*

The Java HotSpot™ VM normally compiles a method only when it is frequently executed, i.e. when its invocation counter reaches a certain threshold. If a method contains a long-running loop, however, it is also possible to switch from interpreted to compiled code while the method is running. This is called *on-stack-replacement* (OSR) [15]. Such methods are compiled with an additional entry point that jumps directly into a loop to the point where the compilation was triggered. The local variables and the operand stack of the interpreter are transferred to the machine code in the OSR entry block.

Figure 9 shows an example of the HIR when a method is compiled with an OSR entry. In this case, the `phi` instructions for loop values in the loop header have three inputs. The third one is coming from the OSR block and represents the result of the loop iterations that were executed in the interpreter. Block 1 is never executed when the OSR entry is used. When on-stack-replacement occurs in a nested loop, additional phi instructions are also required for all outer loops.

After a method is compiled using on-stack-replacement, the next invocation of the method causes a normal compilation. Therefore, only a small fraction of the total execution time is spent in the OSR compiled machine code when a method is executed frequently. On the other hand, when a program consists of a method that is called only once and contains a long-running loop, then the method will only get OSR compiled. Removing array bounds checks in such methods can improve the run-time performance of the program. For most applications, the handling of OSR methods does not lead to an additional speed-up. Nevertheless, it is desirable that the difference between the OSR compiled and normally compiled machine code of a method is as low as possible, otherwise users could be confused by unexplainable performance differences.

When our algorithm is applied to OSR methods without any changes, it is only capable of removing few array bounds checks. It cannot assume any conditions on the values coming from the OSR entry block. Therefore, all conditions for them need to be cleared. To eliminate bounds checks in such methods, the algorithm must consider the fact that the values of the OSR entry are coming from the execution of the

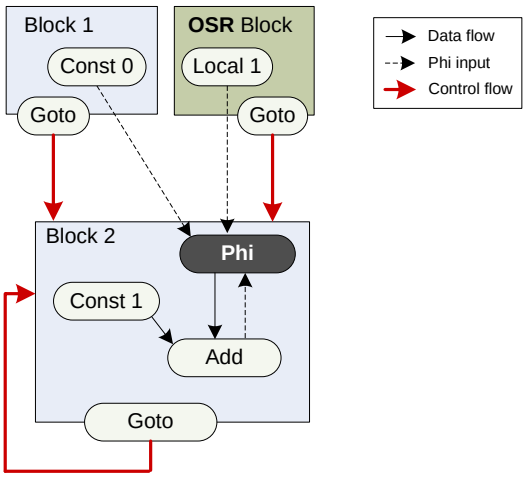


Fig. 9. HIR example for on-stack-replacement.

method in the interpreter. We group together the additional instructions that were inserted because of the OSR entry. For example, if a loop-invariant array is accessed within a loop that contains an OSR entry, the array is now represented by three instructions: the original instruction before the loop, the instruction that loads the array from the interpreter frame, and a phi instruction that merges the two previous instructions at the loop header.

It is safe to assume that the three instructions all refer to the same array. Therefore, we form a group from these three instructions. The algorithm starts by grouping the instruction in the OSR block and the phi instruction. Now there is a phi instruction with only one input coming from a different instruction. So those two instructions can be grouped too. This process is continued until no more instructions can be grouped. The number of instructions in a group depends on the loop nesting level, i.e. when the OSR entry is inside an inner loop, more than three instructions are grouped together.

Special care must be taken when a bounds check is moved out of a loop that has an OSR entry. The deoptimize instructions must be inserted twice: once as usual before the loop and a second time at the end of the OSR block, because the loop can also be entered from this block. When the checked bounds or the array is part of a group of instructions, then the instruction that represents the group at the insertion point must be found. The algorithm iterates through the instructions in decreasing order of their basic block's dominator depth, i.e. the distance in the dominator tree from the root block. The first instruction whose dominator depth is smaller than the loop header represents the group in the block before the loop.

In most cases, our algorithm removes the same number of bounds checks in the OSR compiled method as in the normal compilation of the method. Differences occur only when preceding optimization steps are restricted. For example, global value numbering and loop-invariant code motion do not optimize loops with OSR entries. If a bounds check is eliminated only because an instruction was loop-invariant, then it cannot be eliminated in the OSR compiled code.

Our optimization of OSR methods is based on the assumption that values coming from the interpreter have the same conditions as if the method was executed normally. This assumption holds only if the interpreter executed exactly the same bytecodes that are compiled, which holds for all normal application runs. However, advanced features of the Java HotSpot^{TM} VM to support interactive debugging sessions can change this behavior. It is possible to modify local variables arbitrarily using a debugger [25], and it is possible to replace the bytecodes of a method using *hot swapping* [9]. Analyzing the impact of these changes is too complex, so the only feasible solution is to disable our OSR algorithm during debugging.

6.3. *Comparison with Server Compiler*

The server compiler applies time-consuming global optimizations to produce faster machine code. When building its program dependence graph [7], array bounds checks are modeled as if-statements. Therefore, traditional compiler optimizations help removing bounds checks. The disadvantage of this approach is that the intermediate representation of the compiler gets more complicated, because additional instructions and control flow branches are necessary for each bounds check. This is in contrast to the approach of the client compiler, where the bounds checks are a part of the array access instruction. This section compares the bounds check elimination of the current product version of the server compiler with our research implementation for the client compiler.

Fully redundant bounds checks are automatically eliminated by the conditional constant propagation phase of the server compiler. The grouping of checks is included in the global value numbering phase. Similar to our approach, the server compiler groups checks where the same index instruction plus a constant offset and the same array are involved. In contrast to our algorithm, the server compiler also searches for possible groups in preceding blocks. Therefore, it is more optimistic when grouping checks. Our algorithm only groups checks within one basic block and it is therefore guaranteed that when we deoptimize, the `ArrayIndexOutOfBoundsException` or at least another exception must be thrown. The code generated by the server compiler possibly deoptimizes when no exception must be thrown.

Loop-invariant check motion is done as part of the bounds check elimination within counted loops. The server compiler does not distinguish between bounds checks and other conditional operations here. The loop is split in a pre-loop, a post-loop, and the main loop body. When one operand of a compare instruction involves a loop-invariant instruction and the other one depends on the counter variable, then the limits are adjusted such that the compare instruction in the main loop can be deleted. While the approach of the server compiler is more general and also optimizes general bounds checks, it heavily increases the code size and the compilation time by duplicating the loop body.

When the loop is not identified as a counted loop, the server compiler does not remove array bounds checks. Listing 7 shows an example method that the server compiler fails to optimize. The loop is not identified as a counted loop because it has two loop exits and therefore loop splitting is not applied. The array bounds check remains unchanged and is executed for each loop iteration. In contrast, our algorithm identifies the condition `0 <= i<= c-1` for the index variable. The condition that `c` is smaller or equal to the length of the array is loop-invariant and can therefore be checked by a deoptimization instruction before the loop. Our approach is optimistic, as it is not guaranteed that the exception is thrown when we revert execution back to the interpreter. Execution could exit the loop before the index reaches a value greater than the array length. The possible benefit of aggressive removal of bounds checks within a loop is high, while the additional cost when the condition does not hold is only a recompilation of the method without optimistic bounds check elimination.

```
for (int i = 0; i < c; i++) {
  if (check()) return;
  array[i] = 0;
}
```

Listing 7. Loop with additional exits.

7. Evaluation

This section evaluates our implementation of the algorithm in the Java HotSpot™ VM. It is currently integrated into the early access build b21 of the JDK 7 [24]. We measure the percentage of removed dynamic bounds checks, the impact on compilation speed, and the impact on execution speed.

All measurements were performed on an Intel Core™2 Quad processor Q6600 with four cores at 2.4 GHz, running Microsoft Windows XP. Each core has a separate L1 data cache of 32 KByte. Two cores together

share 4 MByte L2 cache, so there are 8 MByte L2 cache in total. The main memory of 2 GByte is uniformly accessed by all cores. The results were obtained using a 32-bit operating system and a 32-bit VM.

We use several benchmark suites for the evaluation. The SciMark 2.0 [21] and Java Grande [5] benchmarks perform scientific computations that usually operate on large arrays. Some mathematical kernels are contained in both benchmarks, but either the implementation or the size of the input data is different so that they have a different behavior. The SPECjvm98 benchmark suite [23] contains both numerical and object-oriented applications. Finally, the DaCapo benchmark suite [2] consists of large object-oriented applications.

7.1. *Eliminated Bounds Checks*

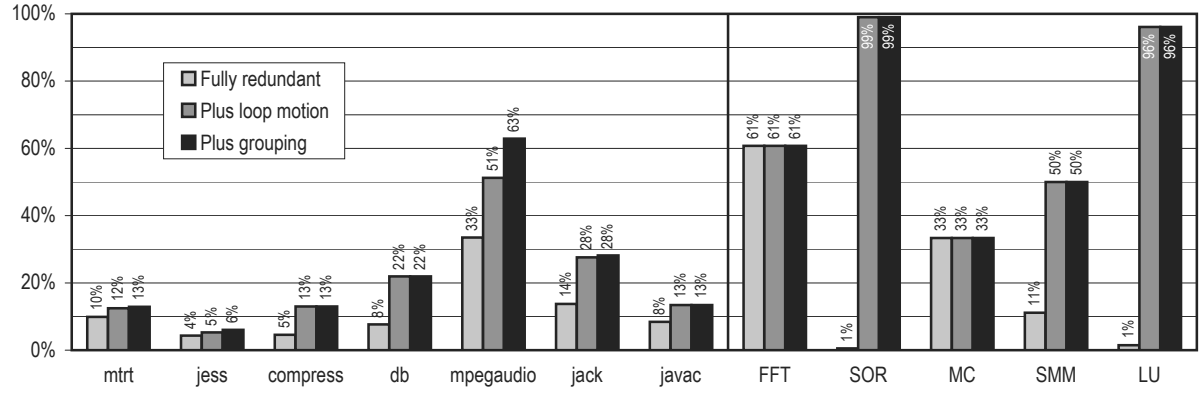
When comparing bounds check elimination algorithms, the percentage of removed checks is the most important criteria. We instrumented the generated machine code to increment counters before each array access and also before each newly inserted deoptimization instruction. Figure 10 shows the results of our algorithm when only fully redundant checks are removed, when the loop-invariant motion of checks is enabled, and finally also with the grouping of checks enabled. One deoptimization instruction is counted as if one bounds check was performed. This is reasonable because a bounds check and a check for deoptimization are expressed both by a compare instruction, a conditional branch, and in some cases also by an instruction that loads the length of an array.

Some fully redundant bounds checks are detected in most benchmarks, e.g. all bounds checks of the RayTracer benchmark can be eliminated. However, eliminating only fully redundant bounds checks would not be a generally applicable optimization. The removal of partially redundant checks by inserting a deoptimization instruction before the loop increases the number of removed bounds checks significantly for most mathematical benchmarks. Additional instructions must be inserted before the loop, so the efficiency is directly related to the number of loop iterations, i.e. the ratio of inserted checks before the loop and eliminated checks inside the loop. For example, one check before the loop eliminates 98 checks inside the loop for the SOR benchmark of SciMark and even 1498 checks for the SOR benchmark of the Java Grande suite. In contrast, this ratio is between 1.2 and 10 for all benchmarks of the DaCapo suite.

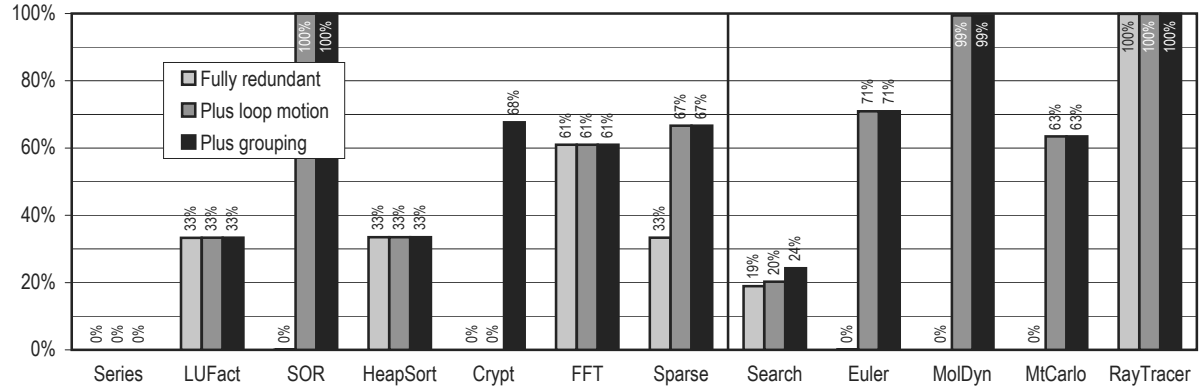
The grouping of checks is effective only in special cases, mainly because there is only an improvement if more than two checks are grouped. The number of additionally executed inserted checks for deoptimization is quite high because these checks are at the same loop nesting level as the original array bounds checks. The Crypt benchmark of Java Grande performs six array accesses on the same array in the most important method. Grouping eliminates these bounds checks, but inserts two checks for deoptimization. Therefore, about two thirds of the checks are eliminated. The benchmark mpegaudio of SPECjvm98 is another example where this optimization yields significantly better results.

In mathematical benchmarks that perform array operations within loops, our algorithm eliminates the majority of the checks. For the other benchmarks, the bounds checks that can not be eliminated are mostly checks of array accesses that are outside of loops, not fully redundant, and cannot be grouped. As the overhead of a method invocation is quite high, such bounds checks are not worth being eliminated. Section 7.2 shows that the theoretically possible speedup achievable by bounds check elimination for object-oriented applications is low.

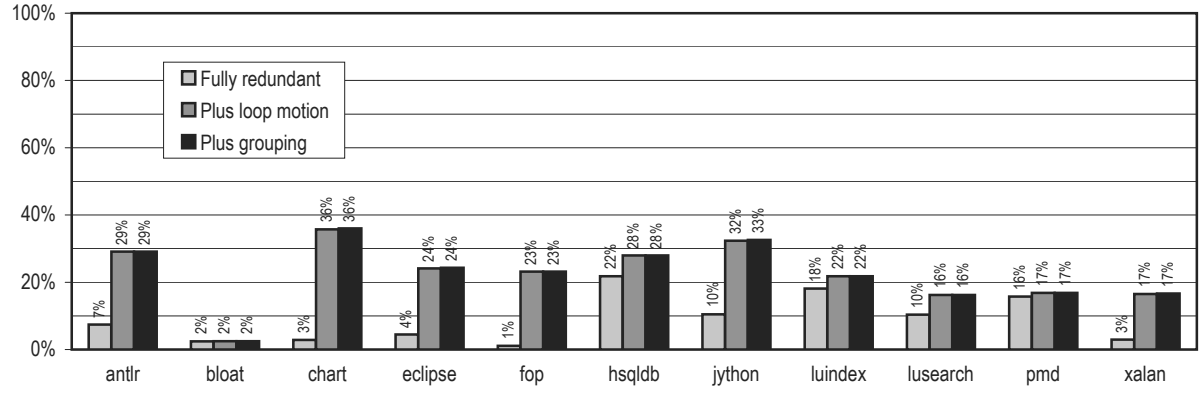
No array bounds check can be eliminated in the Series benchmark of Java Grande. The array is loaded from a field inside a loop just before the array access. Loop-invariant code motion of the field access would be necessary to move the bounds check out of the loop. This is however not possible because the loop contains calls to other methods, which could possibly change the field. An interprocedural analysis would be necessary to prove that the field is not changed. However, the benchmark results of the next section show that in this case such an optimization would not lead to any significant speedup. We believe that an intraprocedural analysis is sufficient in most cases because only bounds checks in the innermost loops are of interest. Such loops usually do not contain non-inlinable method calls.



(a) SPECjvm98 and SciMark 2.0 benchmarks



(b) Java Grande benchmarks (Section II and Section III)



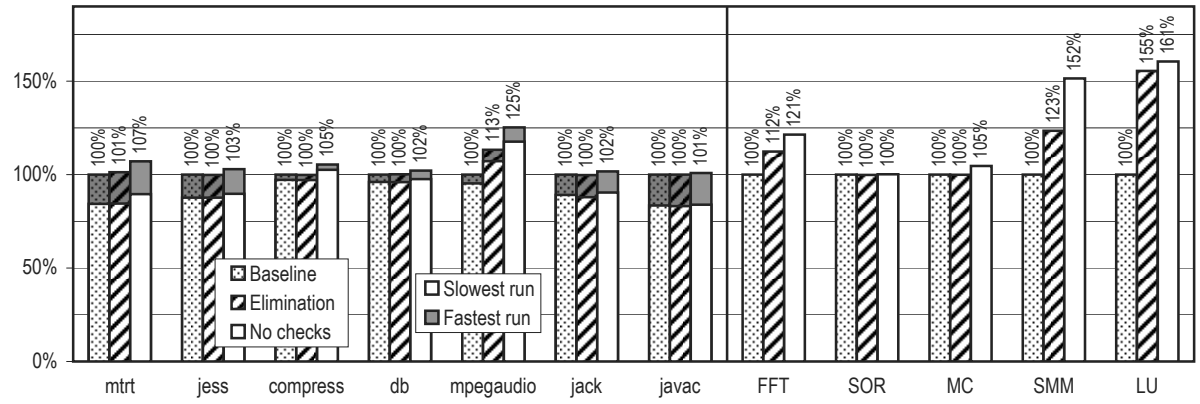
(c) DaCapo benchmarks

Fig. 10. Percentage of removed bounds checks (taller bars are better).

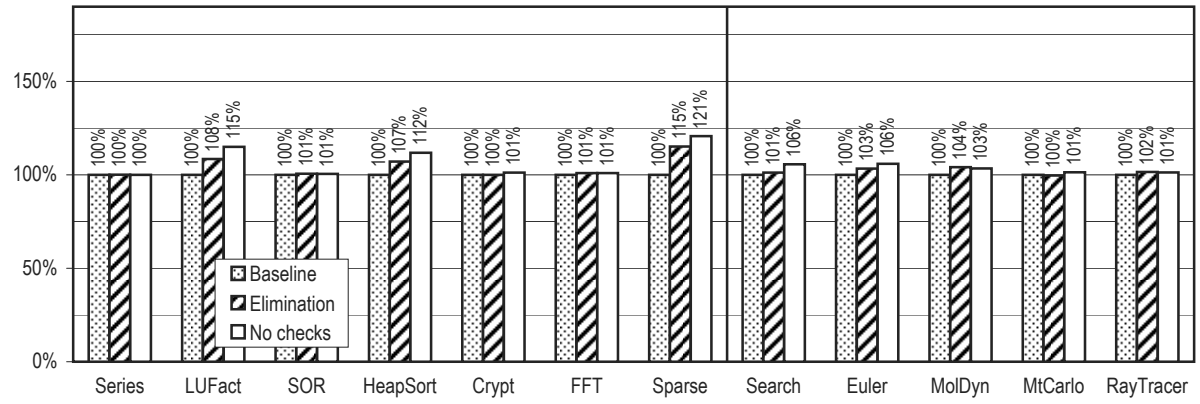
7.2. *Impact on Run Time*

The speedup obtained by array bounds check elimination depends on the type of application. If the most frequently executed methods contain array access instructions, elimination of bounds checks is effective. On the contrary, no speedup can be expected if only a low percentage of the execution time is spent accessing arrays. Therefore, we did not only measure the speedup achieved by our algorithm, but also the theoretically possible speedup reachable with bounds check elimination by generating no bounds checks at all. With this change, the Java VM does not conform to the specification, as an `ArrayIndexOutOfBoundsException` will

never be thrown and a program could freely access the memory and disable all security mechanisms by accessing an array with an incorrect index. However, the benchmarks are not affected by this change as the indices of all array accesses are within the correct bounds. Therefore, Figure 11 shows the benchmark results for three configurations: the baseline configuration with bounds checks, the impact of our bounds check elimination, and the artificial configuration without any bounds checks.



(a) SPECjvm98 and SciMark 2.0 benchmarks



(b) Java Grande benchmarks (Section II and Section III)

Fig. 11. Speedup when using the bounds check elimination algorithm (taller bars are better).

When executing the SPECjvm98 benchmarks for the first time, the compilation time is relevant in comparison to the total execution time. Therefore, we executed each benchmark several times and measured the slowest and the fastest execution time. The slowest and the fastest runs are shown on top of each other relative to the same baseline. For SciMark and the Java Grande benchmarks, the compilation time is insignificantly low and multiple runs show the same results.

Only for some of the benchmarks, the achievable speedup is above five percent. Even if an algorithm eliminates all bounds checks, it cannot cross this limit. For example, although our algorithm eliminates nearly all of the bounds checks in the SOR benchmark of SciMark, there is no measurable speedup because the array accesses are only responsible for a tiny fraction of the total execution time.

We do not report results for the DaCapo benchmark suite because bounds check elimination is not significant for any of these benchmarks. In the artificial configuration without any bounds checks, no benchmark has a speedup above 2%. Even though up to 36% of the bounds checks are eliminated for some benchmarks, the speedup is only about 1% for these benchmarks and therefore insignificant in practice.

The slowest runs of SPECjvm98 show that the slightly increased compilation time does not affect the start-up performance of an application. The benefit outweighs the analysis overhead. Bounds check elimination has no significant negative impact on the run time of any benchmark. The mpegaudio benchmark is one

of the two SPECjvm98 benchmarks that have a theoretically possible speedup of more than 5%. This is because the benchmark carries out calculations based on arrays, e.g. the discrete cosine transformation. Our algorithm achieves a bit more than one half of the possible speedup, which is consistent with the 63% eliminated bounds checks for this benchmark. In contrast, our algorithm fails to optimize the mtrt benchmark, because the most relevant bounds checks of this benchmark are neither fully redundant nor in a loop.

The highest speedups are achieved for the LU benchmark of SciMark. The theoretically possible speedup is 161%. Because our algorithm eliminates 96% of all bounds checks, our achieved speedup is 155%. Significant performance improvements are also achieved for other benchmarks of the SciMark and Java Grande suites.

The results reported in our previous paper [27] show a higher speedup for some of the SciMark benchmarks. For example, the LU benchmark was 197% faster with our bounds check elimination, with a theoretically possible speedup of 201%. These results were measured on an Intel Pentium 4 processor with 3.2 GHz. The improved processor architecture and the increased cache size of the current Intel Core processors reduce the impact of bounds check elimination for these benchmarks.

7.3. *Compile-Time Impact*

One important design goal of our algorithm is to have a small impact on the overall compilation time. It is designed for the use in a fast just-in-time compiler, so the time needed for bounds check elimination adds to the run time of an application. Figure 12 summarizes the compile-time results of the four benchmark suites. It contains the minimum and the maximum value observed in the according benchmarks.

	Compile time [msec]			Number of compiled methods			Number of bounds checks		
	Total	Bounds check elim.		Total	With bounds checks		Total	Eliminated	
SPECjvm98	23 - 304	0.5 - 1.5	0.5% - 2.2%	55 - 651	36 - 125	19% - 65%	98 - 1042	32 - 901	26% - 86%
SciMark 2.0	2 - 4	0.1 - 0.2	2.7% - 4.5%	6 - 11	5 - 9	82% - 100%	15 - 83	7 - 50	47% - 71%
Java Grande	2 - 47	0.1 - 1.4	1.2% - 3.9%	8 - 135	5 - 41	30% - 100%	18 - 1151	6 - 541	15% - 78%
DaCapo	233 - 3075	1.9 - 19.1	0.4% - 2.1%	429 - 2848	122 - 1201	22% - 42%	453 - 10754	115 - 2011	9% - 55%

Fig. 12. Compile-time characteristics of the benchmark suites.

The figure shows the different characteristics of scientific and object-oriented benchmarks. The mathematical computations of the SciMark and Java Grande benchmarks are packed in a low number of long-running methods. Some benchmarks require only 6 methods to be compiled, with an overall compilation time of only 2 milliseconds. Bounds check elimination requires 1.2% to 4.5% of the compilation time. This overhead is influenced by the number of compiled methods that contain bounds checks because the whole optimization phase is skipped when no bounds check is present.

The object-oriented structure of the DaCapo benchmarks lead to a high number of compiled methods. The compilation of several thousand methods leads to an overall compilation time of up to 3 seconds. The first run of such benchmarks is significantly slower than subsequent runs because of this overhead. Just one quarter to one half of the methods contain array bounds checks, therefore the overhead of bounds check elimination is only 0.4% to 2.1%. The SPECjvm98 benchmark suite contains both mathematical and object-oriented benchmarks, so the numbers are between the other kinds of benchmark suites.

The last group of columns in Figure 12 shows the static number of bounds checks that are generated or eliminated. The numbers vary greatly between the benchmarks and are mostly unrelated with the number of executed or eliminated bounds checks at run time, which were presented in Figure 10. This shows that it is unimportant how many array access instructions are optimized. Contrariwise, it is important to eliminate the few but important array access instructions that are inside frequently executed loops or methods.

Deoptimization because of loop-invariant or grouped array bounds checks does not have a significant impact on compile time because such cases occur rarely. At most two deoptimizations are performed in the DaCapo benchmarks, and one deoptimization occurs in the LU benchmark of SciMark. All other benchmarks

of SciMark, all Java Grande benchmarks, and all SPECjvm98 benchmarks do not require deoptimization because of our optimistic bounds check elimination.

8. Related Work

Gupta presents algorithms for bounds check elimination based on bound conditions [12,13]. One important difference to our algorithm is the way the case of an index being out of bounds is handled. While we must adhere to the Java exception semantics, Gupta does not care about throwing the exception at the correct place and therefore does not require a concept similar to our deoptimization. The basic ideas of grouping multiple checks into a single one as well as moving checks out of loops are similar to our algorithm. However, his algorithm does not take advantage of conditions introduced by conditional branches.

To reduce the execution time, Gupta's algorithm works on a reduced control flow graph, which contains only instructions that are involved in array accesses. While Gupta differentiates between the lower and the upper bound check, this does not make sense for our algorithm, as the costs for applying both or only one of them are the same in the Java HotSpot™ VM.

Bodík et al. present an algorithm that allows bounds check elimination for Java to be performed on demand [3], integrated into the just-in-time compiler of the Jikes RVM [6]. This way it is possible to use the algorithm only for array accesses that are frequently executed. The input to their algorithm is a representation similar to the HIR which is also in SSA form. They convert it to an extended SSA form in such a way that the conditions do not change within the lifetime of a value. Therefore, they need to insert additional `pi` instructions (in contrast to `phi` instructions of the SSA form) to represent new conditions for certain values, e.g. after `if` statements.

Our algorithm does not need to insert such instructions and solves the problem by the special pre-order traversal of the dominator tree. They use a full inequality graph instead of maintaining simplified conditions and perform an adapted shortest path algorithm to check whether an index is within the correct bounds or not. In case of conditional branches, our algorithm updates the bounds of the instructions, while their algorithm adds additional edges to the inequality graph.

Qian et al. perform an intraprocedural analysis and build inequality graphs for important points in a method [22]. They update the inequality graph of a block by merging the graphs of its predecessors. When processing loops, they do a fix-point calculation. They further improve their algorithm by doing an interprocedural field analysis and a special analysis for finding rectangular arrays. However, their algorithm is not capable of handling partially redundant checks. They integrated their algorithm into the Kaffee Java VM and into IBM's HPCJ and provide evaluation results for the mpegaudio benchmark and the SciMark benchmark suite.

All previously described approaches, including ours, use only information from the array itself and not from the context around the array. In contrast, the *related field analysis* of Aggarwal et al. builds a relationship between an array field and an integer field of the same object [1]. When it is known that the value of the integer field is always below or equal the length of the array referenced by the array field, and when the integer field is compared with the index variable that is used for an array access, then the bounds check of the array access is eliminated. This optimizes a frequently occurring code pattern used e.g. by the Java collection classes: Because the *capacity* of a collection is usually larger than the *size* of the collection, the explicit comparison that ensures that the index is below the size is sufficient to remove the bounds check that ensures that the index is below the capacity. Therefore, they can eliminate a significantly higher percentage of bounds checks from the SPECjvm98 benchmark suite. However, they require an interprocedural whole-program analysis of field accesses [10], which would be expensive and complicated to implement in the context of a just-in-time compiler.

Some approaches eliminate bounds checks by annotating Java bytecodes [16,28]. The advantage of this method is that the just-in-time compiler does not need to perform the elimination. A more complex analysis can be applied as the analysis time does not add to the execution time of the Java program. The disadvantages are larger class files and the need for verification of the annotations in the virtual machine.

9. Conclusions

We presented an array bounds check elimination algorithm for Java and evaluated our implementation in the Java HotSpot^{TM} VM. Our algorithm is based on an intermediate representation in SSA form and performs an intraprocedural constraint analysis on the index instructions. We remove partially redundant bounds checks and retain the correct exception semantics by using deoptimization. The algorithm is designed to eliminate bounds checks for typical Java programs. While having a low impact on the compilation time, it succeeds to eliminate the majority of all bounds checks and leads to a speedup close to the theoretical maximum for the scientific SciMark benchmark. There are plans to integrate the algorithm in one of the upcoming releases of the Java HotSpot^{TM} virtual machine.

Acknowledgements

We would like to thank the Java HotSpot^{TM} compiler team at Sun Microsystems, especially Kenneth Russell, Thomas Rodriguez and David Cox, for their persistent support, for contributing many ideas and for helpful comments on all parts of the Java HotSpot^{TM} VM.

References

[1] A. Aggarwal, K. H. Randall, Related field analysis, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2001, pp. 214–220.

[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, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2006, pp. 169–190.

[3] R. Bodík, R. Gupta, V. Sarkar, ABCD: Eliminating array bounds checks on demand, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2000, pp. 321–333.

[4] P. Briggs, K. D. Cooper, L. T. Simpson, Value numbering, Software: Practice and Experience 27 (6) (1997) 701–724.

[5] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey, A benchmark suite for high performance Java, Concurrency: Practice and Experience 12 (6) (2000) 375–388.

[6] M. G. Burke, J.-D. Choi, S. J. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, J. Whaley, The Jalapeño dynamic optimizing compiler for Java, in: Proceedings of the ACM Conference on Java Grande, ACM Press, 1999, pp. 129–141.

[7] C. Click, M. Paleczny, A simple graph-based intermediate representation, in: Papers from the ACM SIGPLAN Workshop on Intermediate Representations, ACM Press, 1995, pp. 35–49.

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

[9] M. Dmitriev, Safe class and data evolution in large and long-lived Java^{TM} applications, Tech. Rep. TR-2001-98, Sun Microsystems, Inc. (2001).

[10] S. Ghemawat, K. H. Randall, D. J. Scales, Field analysis: getting useful and low-cost interprocedural information, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 2000, pp. 334–344.

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

[12] R. Gupta, A fresh look at optimizing array bound checking, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1990, pp. 272–282.

[13] R. Gupta, Optimizing array bound checks using flow analysis, ACM Letters on Programming Languages and Systems 2 (1-4) (1993) 135–150.

[14] U. Hölzle, C. Chambers, D. Ungar, Debugging optimized code with dynamic deoptimization, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1992, pp. 32–43.

[15] U. Hölzle, D. Ungar, Optimizing dynamically-dispatched calls with run-time type feedback, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 1994, pp. 326–336.

[16] J. Hummel, A. Azevedo, D. Kolson, A. Nicolau, Annotating the Java bytecodes in support of optimization, Concurrency: Practice and Experience 9 (11) (1997) 1003–1016.

[17] M. Kawahito, H. Komatsu, T. Nakatani, Effective null pointer check elimination utilizing hardware trap, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, 2000, pp. 139–149.

[18] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, D. Cox, Design of the Java HotSpot™ client compiler for Java 6, ACM Transactions on Architecture and Code Optimization 5 (1) (2008) article 7.
[19] V. V. Mikheev, S. A. Fedoseev, V. V. Sukharev, N. V. Lipsky, Effective enhancement of loop versioning in Java, in: Proceedings of the International Conference on Compiler Construction, LNCS 2304, Springer-Verlag, 2002, pp. 101–115.
[20] M. Paleczny, C. Vick, C. Click, The Java HotSpot™ server compiler, in: Proceedings of the Java Virtual Machine Research and Technology Symposium, 2001, pp. 1–12.
[21] R. Pozo, B. Miller, SciMark 2.0, http://math.nist.gov/scimark2/ (1999).
[22] F. Qian, L. J. Hendren, C. Verbrugge, A comprehensive approach to array bounds check elimination for Java, in: Proceedings of the International Conference on Compiler Construction, LNCS 2304, Springer-Verlag, 2002, pp. 325–342.
[23] Standard Performance Evaluation Corporation, The SPECjvm98 Benchmarks, http://www.spec.org/jvm98/ (1998).
[24] Sun Microsystems, Inc., Java Platform, Standard Edition 7 Source Snapshot Releases, http://download.java.net/jdk7/ (2007).
[25] Sun Microsystems, Inc., Java™Platform Debugger Architecture, http://java.sun.com/javase/6/docs/technotes/guides/jpda/ (2007).
[26] C. Wimmer, H. Mössenböck, Optimized interval splitting in a linear scan register allocator, in: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments, ACM Press, 2005, pp. 132–141.
[27] T. Würthinger, C. Wimmer, H. Mössenböck, Array bounds check elimination for the Java HotSpot™ client compiler, in: Proceedings of the International Conference on Principles and Practice of Programming in Java, ACM Press, 2007, pp. 125–133.
[28] H. Xi, S. Xia, Towards array bound check elimination in Java™ virtual machine language, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 1999, pp. 110–125.