# Array Bounds Check Elimination for the Java HotSpot™ Client Compiler*

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

## 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 (40% on average). The algorithm also improves the execution speed for the SPECjvm98 benchmark suite (2% on average, 12% maximum).

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Languages, Performance

## Keywords

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

*This work was supported by Sun Microsystems, Inc.


*PPPJ 2007,* September 5–7, 2007, Lisboa, Portugal.
http://doi.acm.org/10.1145/1294325.1294343

## 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*.

There can also be 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 average 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™ 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 the 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 integrated our analysis in the client compiler of the Java HotSpot™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.

## 2. SYSTEM OVERVIEW

The main components of the Java HotSpot™ 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 [12] 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 [4, 10] 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 [7] 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 at the end of a basic block, control flow instructions 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 [3], 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 [2] 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 [17]. From the LIR, the final platform-dependent machine code is generated.

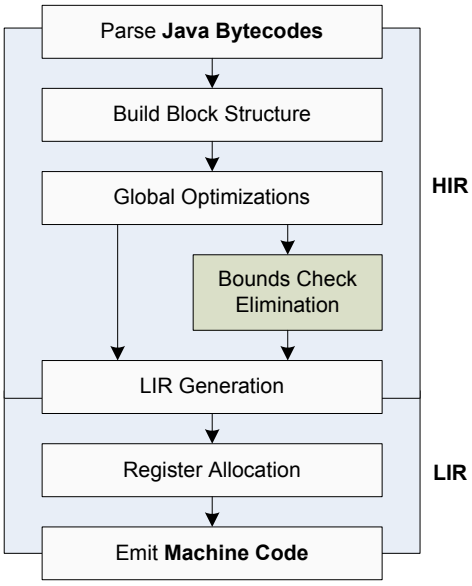


**Figure 1: Structure of the client compiler.**

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 comparison if `a` is greater or equal 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 only eliminated when both the index and the length of the array are compile-time constants. This is however a rare case.

The 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 if the bounds check must be emitted.

We do not perform an 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.

## 3. ALGORITHM

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 [1, 14], we do not build an inequality graph. Instead, the algorithm keeps a condition of the following form for every instruction `x` that computes an integer value:

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

The 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 bounds

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

where `MIN` and `MAX` denote the minimum and maximum possible values of a 32-bit signed integer. When two or more bounds are known for a variable, the algorithm tries to calculate the conjunction of them. The algorithm needs to be conservative, e.g. if the bounds involve values that are not compile-time constants, only one part of the conjunction is saved. When the algorithm knows that `x` is smaller or equal to `a` and it is smaller or equal to `b`, it can only save one of the two conditions. This is a loss of information, but a practical simplification in most cases.

## 3.1 Fully Redundant Checks

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 possible 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 [1].

Figure 2 presents the HIR of the Java method `get` shown in Listing 1. 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 a condition 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 1: 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.

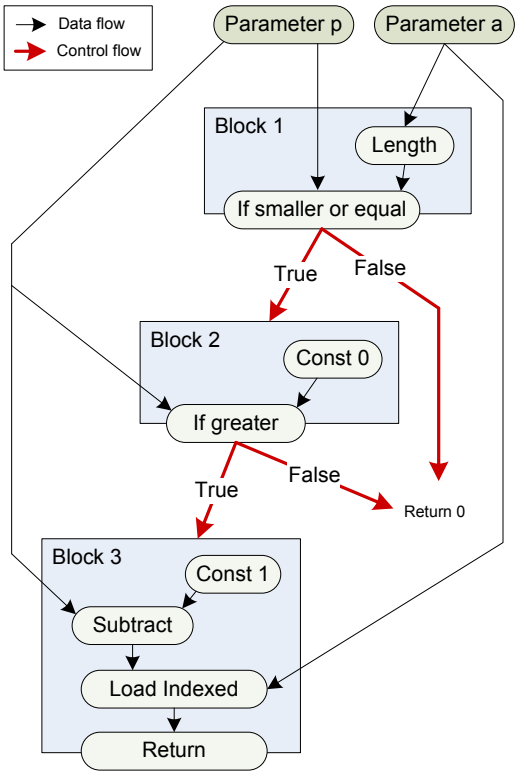


**Figure 2: HIR of method `get`.**

Figure 3 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 an and-operation. 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 3, for example, the condition for `p` is converted to 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 in the valid range. The lower bound is positive and the upper bound is smaller than the array length, therefore the array bounds check is fully redundant and can be omitted.

Blocks that do not dominate a block with an array access 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 bytecode, 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 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 [5], 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 runtime cost for maintaining the conditions for all integer instructions is low.

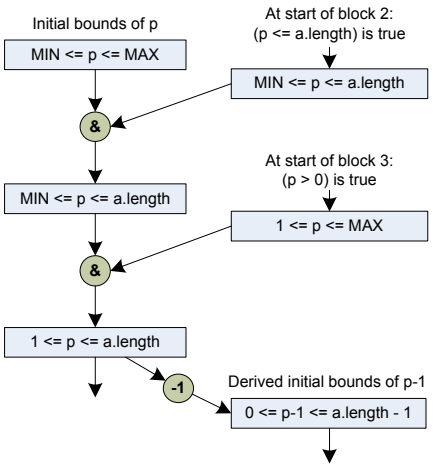


**Figure 3: Conditions for the values of method `get`.**

## 3.2 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 `clear` shown in Listing 2 uses a frequent code pattern where an array is accessed in a loop, but the array length is not checked explicitly.

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

**Listing 2: Example method `clear`.**

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 only increased.

Figure 4 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 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.

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

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

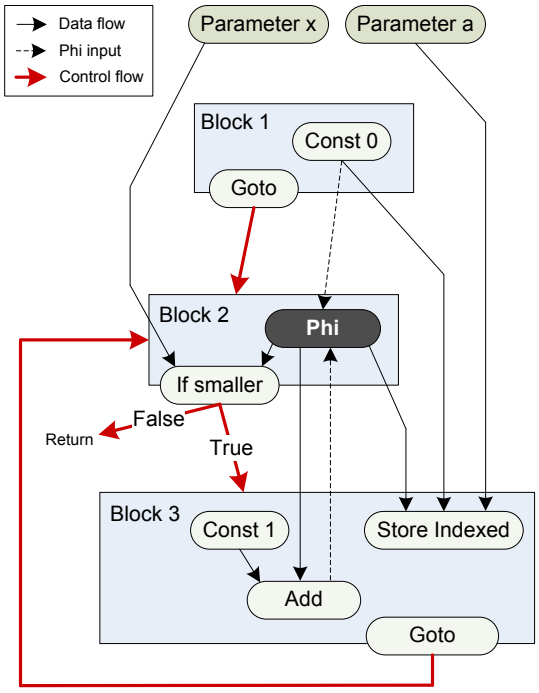


**Figure 4: HIR of method `clear`.**

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 bigger 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` does not exceed 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 the 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 [11]). 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 bigger than the array length.

Figure 5 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.

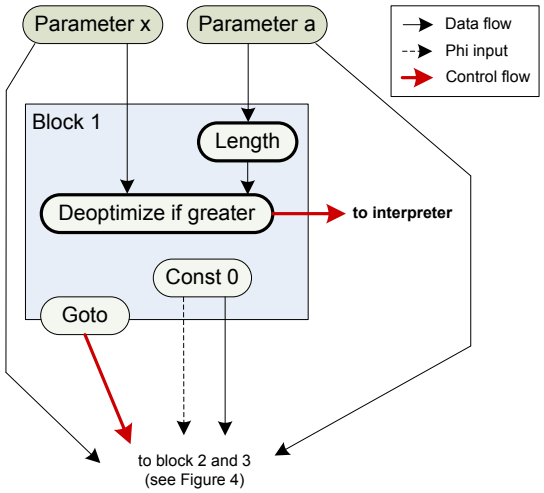


**Figure 5: After insertion of instructions.**

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 analysis that determines if 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 bigger than the maximum integer value is a negative value. Therefore, we need not only proof that the added value is bigger or equal to zero, but also that no overflow can occur. As 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 2, `x` denotes the value never reached by the loop variable and `c` the constant that is added to the loop variable in each iteration:

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

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:

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

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

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

is always true. If the variable can overflow, deoptimization is called anyway, so we do not need to bother 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 greater or equal to zero by the deoptimize instruction. If this check succeeds, no subtraction of any positive value can cause an underflow.

## 3.3 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 only differ by constants. Listing 3 shows the method `triple` with three array stores. The bounds checks of all three stores can be folded to 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 3: Example method `triple`.**

Figure 6 shows the HIR representation of the method. The algorithm needs a single pass over the instructions of each basic block. For all arrays and index variables, it maintains the minimum and the maximum constants that are added to the index variable when the array is accessed. 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.

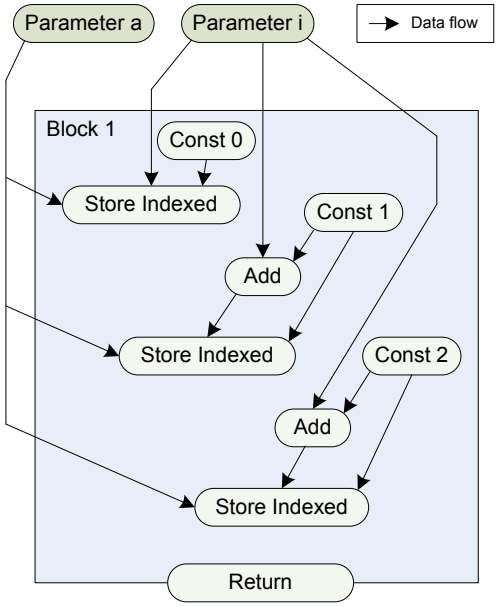


**Figure 6: HIR of method `triple`.**

When this condition holds, also all other array accesses involving the same index variable and the same array are safe. Checks of multiple array accesses with different constant indices are replaced by a single deoptimization instruction that checks whether the maximum constant is smaller than the array length.

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:

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

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

It is possible that the calculation of $i + 2$ results in 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 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.

## 3.4 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, additionally 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 a field of the same type within the loop. An array access is moved, when the array and the index expressions are loop-invariant and there is no array store to an array of the same type within the loop.

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 added an instruction flag to mark instructions that must not throw an exception immediately, but call deoptimization instead to fulfill the exception semantics.

## 3.5 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) [8]. 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 7 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 with an OSR entry, the next invocation of the method causes a normal compilation without the OSR entry.

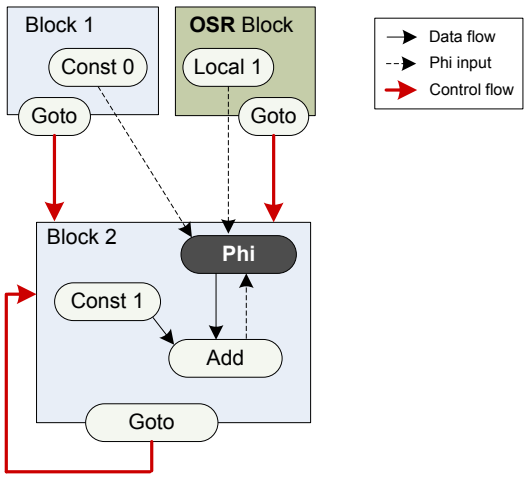


**Figure 7: HIR example for on-stack-replacement.**

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.

We enhanced the algorithm such that it takes benefit of the fact that the values of the OSR entry are coming from the execution of the method in the interpreter. This is done by ignoring a value from an OSR entry when updating the conditions of a phi instruction. The algorithm can assume that the conditions for the value coming from the interpreter are the same as for the phi instruction itself. 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.

# 4. EVALUATION

This section evaluates our implementation of the algorithm in the Java HotSpot™ VM. It is currently integrated into the early access build b04 of the JDK 7 [16]. We measure the percentage of removed dynamic bounds checks, the impact on the compilation speed, and the impact on the execution speed.

All measurements were performed on an Intel Pentium 4 processor 540 with 3.2 GHz and 1 GByte of main memory. The operating system was Windows XP Professional with Service Pack 2 installed. We did not modify the default configuration of the HotSpot™ VM, so the standard sizes for the heap are used.

For the evaluation we executed the two benchmark suites SPECjvm98 [15] and SciMark 2.0 [13]. The first one includes benchmarks with few array accesses like javac or jack, while the second one performs scientific computations that operate mostly on large arrays.

## 4.1 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 8 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. 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.

The removal of partially redundant checks by inserting a deoptimization instruction before the loop increases the removed bounds checks significantly for nearly all benchmarks. The grouping of checks is effective only in special cases, mainly because there is only an improvement if more than two checks are grouped and the number of additionally inserted deoptimize instructions is quite high. The benchmark mpegaudio is the only one where this optimization yields significantly better results.

In mathematical benchmarks that perform array operations within loops like mpegaudio, FFT, SOR, SMM and LU, 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 4.3 shows that the theoretically possible speedup achievable by bounds check elimination for such applications is low.

In the MC benchmark, none of the array bounds checks can be eliminated because the indices are fields and an interprocedural analysis would be necessary to prove that an exception cannot occur.

## 4.2 Impact on Compile Time

One important design goal of our algorithm is to have a small impact on the overall compilation time. The algorithm 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.

When enabling bounds check elimination, the time spent in the compiler is increased by 1.8% when executing the SPECjvm98 benchmark suite and increased by 5.6% when executing SciMark. The overhead for SciMark is higher because the relative number of array accesses is higher and more bounds checks are eliminated.

For all executed benchmarks, deoptimization because of a loop-invariant or grouped array bounds check is never needed. So in the common case there is no additional compilation overhead introduced by recompiling deoptimized methods.

## 4.3 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 very 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.

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, the compilation time is insignificantly low. Figure 9 shows the achieved speedup reached by our algorithm and the theoretically possible speedup of array bounds check elimination.

Only for four of the twelve benchmarks, the achievable speedup is above five percent. Even if an algorithm eliminates all bounds checks, it cannot cross this limit. For example, even though our algorithm eliminates 96% of the bounds checks in the SOR benchmark, there is no measurable speedup because the array accesses are only responsible for a small fraction of the total execution time.

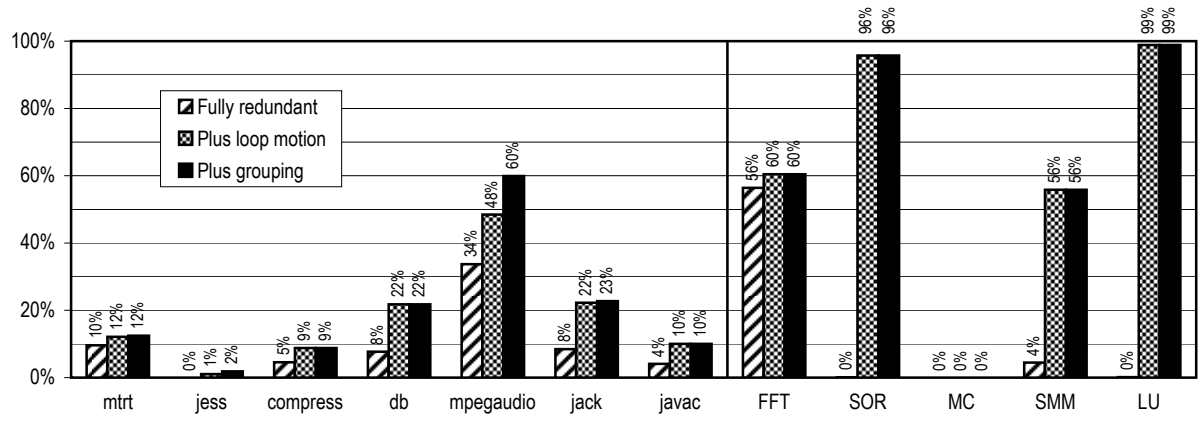


**Figure 8: Percentage of removed bounds checks (taller bars are better).**

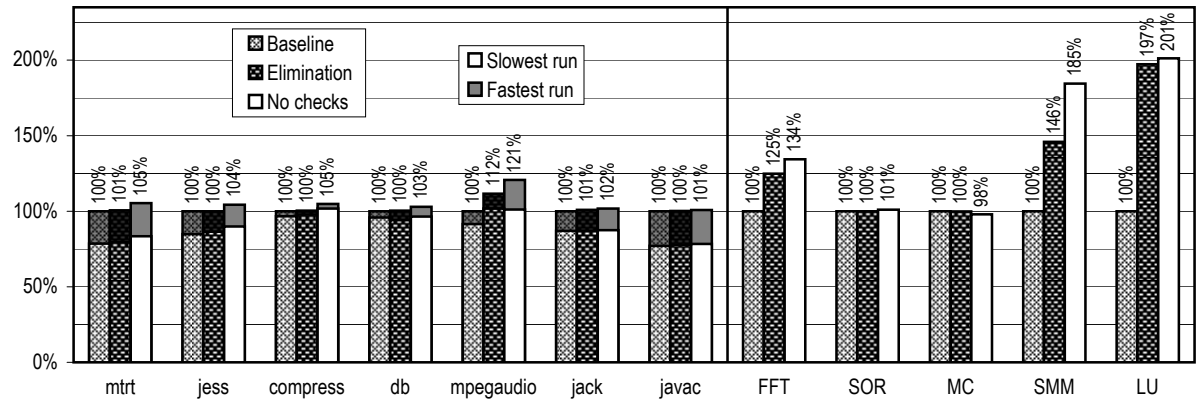


**Figure 9: Speedup when using the bounds check elimination algorithm (taller bars are better).**

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 is the only benchmark of the SPECjvm98 suite that has a theoretically possible speedup of more than 5%. This is because the benchmarks carries out calculations based on arrays, e.g. the discrete cosine transformation. The other SPECjvm98 benchmarks rarely access arrays, so their performance can hardly be improved by removing array bounds checks.

The best results are achieved for the LU benchmark. The theoretically possible speedup is 201%. Because our algorithm eliminates 99% of all bounds checks, the benchmark executes nearly twice as fast as before. Significant performance improvements are also achieved for the benchmarks SMM (146%), FFT (125%) and mpegaudio (112%).

## 5. RELATED WORK

Gupta presents algorithms for bounds check elimination based on bound conditions [5, 6]. 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$^{TM}$ VM.

Bodík et al. present an algorithm that allows bounds check elimination for Java to be performed on demand [1]. 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 to represent new conditions for certain values, e.g. after `if` conditions.

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 [14]. 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.

Some approaches eliminate bounds checks by annotating Java bytecodes [9, 18]. 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.

# 6. CONCLUSIONS

We presented an array bounds check elimination algorithm for Java and evaluated our implementation in the Java HotSpot™ 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™ virtual machine.

## Acknowledgements

We would like to thank the Java HotSpot™ 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™ VM.

# 7. REFERENCES

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

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

[3] 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.

[4] 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.

[5] R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282. ACM Press, 1990.

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

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

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

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

[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™client compiler for Java 6. Technical report, 2007.

[11] V. V. Mikheev, S. A. Fedoseev, V. V. Sukharev, and N. V. Lipsky. Effective enhancement of loop versioning in Java. In *Proceedings of the International Conference on Compiler Construction*, pages 101–115. LNCS 2304, Springer-Verlag, 2002.

[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, 2001.

[13] R. Pozo and B. Miller. *SciMark 2.0*, 1999. http://math.nist.gov/scimark2/.

[14] F. Qian, L. J. Hendren, and C. Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proceedings of the International Conference on Compiler Construction*, pages 325–342. LNCS 2304, Springer-Verlag, 2002.

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

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

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

[18] H. Xi and 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*, pages 110–125. IBM Press, 1999.