Methods Used For Redundant Array Bounds-check Elimination And The Gains of Variable Range Analysis

Author: Bashir Nayal
Supervisor: Atze van der Ploeg

July 17, 2022

Abstract

Memory-safe languages ensure that when memory is accessed, it is always within the bounds of said memory region. This makes these languages more reliable in certain settings. Safe languages are an excellent choice for security-sensitive applications. Unfortunately, there is a performance penalty associated with this. In order to mitigate some of these penalties, compilers for such languages utilize various techniques to reduce the costs caused by security checks. This paper summarizes the literature on the methods used to alleviate the overhead introduced by the array-bounds check that safe languages insert. Furthermore, a benchmark is performed using a prototype language to measure the extent of the cost reduction that can be achieved.

1 Introduction

Memory safety is essential in systems where failure might have catastrophic implications. However, security checks can sometimes hinder performance significantly. As an example, programs that have the Array Bounds-Check can be twice as slow as their counterparts[1]. This trade-off makes deciding on a programming language, for a certain system, a difficult task.

Languages like c are very popular in systems design because they ensure speed. However, these systems are sometimes susceptible to vulnerabilities that can cause loss of availability and/or confidentiality due to memory issues. Although there have been significant improvements by compilers to prevent hijacking of processes[2][3], there is still potential for human-error. When fast execution is critical, languages with memory safety are not always a choice due to the overhead associated with them. Many studies have been carried out in order to achieve memory safety property while sacrificing as little performance as possible.

This paper explains how the bounds-check insertion is implemented and when it is safe to remove bounds-checks that are redundant. The paper then introduces the existing methods used to achieve the elimination of these checks. Finally, a prototype language is used to measure the performance improvement when removing redundant checks using variable range analysis.

2 Array Bounds-Check

The Array bounds-check is a technique used to ensure that the variable used to index a memory location is within the bounds of that array. In order to achieve this, the compiler needs to keep track of all the declared arrays as well as their respective sizes. At compile time, the bounds-checks are inserted using this information. Figure 1 shows how the code is transformed after performing the bounds-check pass.

before bounds-check insertion

```
1    int index = 0;
2    int arr[] = new int[512];
3    while (index < 64) {
4        arr[index] = index;
5        index += 1;
6    }
```

after bounds-check insertion

```
1    int index = 0;
2    int arr[] = new int[512];
3    while (index < 64) {
4        if (index >= arr.length || index < 0) {
5            throw new ArrayIndexOutOfBoundsException();
6        }
7        else {
8            arr[index] = index;
9            index += 1;
10       }
11   }
```

Figure 1: Array memory access before and after bounds-check insertion

2.1 Bounds-Checks Overhead

The addition of the bounds-checks increases the number of instructions that the machine executes. Moreover, this branching can interfere with other optimizations. Take instruction scheduling as an example. This is an essential compiler optimization that minimizes wasting clock cycles for many CPU architectures, such as RISC or x86[4]. If the memory location of the array size happens to cause a cache-miss, the introduction of this branching can cause the CPU pipeline to stall.

Additionally, for languages like Java, the semantics of exception handling are very strict. This means that if the bounds-check is inside a loop, hoisting it outside the loop should not result in an exception being thrown from the wrong program point[5]. In other words, if a bounds-check cannot be safely removed, it may hinder other optimizations.

Based on the issues associated with bounds-checks, It is hence desirable to remove array-bounds checks as much as possible, while still ensuring that no out-of-bounds indexing is possible.

2.2 Redundant Array Bounds-Checks

An array bounds-check is said to be redundant when it can be proven that the index used to to address the memory is within the bounds of the array. Put differently, for all possible values that a variable $index$ can take at run-time, it must hold that:

1. $index < array.length$
2. $index \geq 0$

Consider the code generated by the bounds-check pass in figure 1. It is clear that during the execution of this program, the if statement will never evaluate to true. Using simple logic, it can be deduced that the value of the variable $index$ will always be between 0 and 64 inclusive. Moreover, the array's size is known at compile time-512 in this case. With this data at hand, there is enough information to prove that both conditions 1 and 2 are always satisfiable. This means that both code snippets shown in figure 1 have the exact same behaviour. This is a rather simple example but it serves to showcase how downgrading an inefficient bounds-check insertion can be to a program.

3 Eliminating Redundant Bounds Checks

There are a handful of methods utilized in order to achieve optimal elimination of redundant bounds-checks. This section will summarize the major techniques and introduce an implementation inspired from aspects of some of these methods.

3.1 Theorem Provers

Large programs can quickly have very complex control and data flow. This makes it challenging to reason about the logic involved and renders it prone to errors. This was the motivation behind decades-long research in the area of automatic software verification.

For this technique, one of the major problems it faces is the ability for the proof-checker to determine the loop-invariant property for a given loop. The need for that is to be able to reason about what is happening inside the loop, and use this information to prove the validity of the bounds-checks.

A loop has the loop-invariant property if there exists an expression $invariant$ in which it holds that:

1. $invariant$ is true before the execution of the loop
2. $invariant$ is true after each iteration of the loop
3. $invariant$ is true after exiting the loop

Once the loop-invariant property is found, it can be used to construct a rule that expresses the correctness requirement on the loop[6]. This loop-invariant expression does not contribute to the semantics of the loop.

Unfortunately, it is not enough to find a loop-invariant. $true$ is a loop-invariant that always satisfies the correctness condition but is not a strong invariant[7]. Finding a correct and strong invariant is an undecidable task. However, there are methods that heuristically approximate it[7] that give acceptable results.

In the context of bounds-checking, the loop invariant needs to encode the bounds-check expression. This would ensure that if the correctness of the loop holds, then there are no violations of the array bounds within that loop. In which case, it can be proven that there are no additional bounds-checks needed.

For a simple loop that sequentially traverses an array, the invariant is straight-forward, namely (index >= 0 and index < arr.length). It gets trickier with more complex loops. Figure 2 is an example where some control flow is involved. For this loop, the invariant that encodes the bounds-check is as follows: (index > arr.length && !some_condition()) || ((index > arr.length - 5) && some_condition()). This demonstrates how difficult it can get for a loop invariant expression to be found.

As established above, finding the loop invariant is very important and quite difficult. There has been many research conducted towards finding reliable ways to automatically find it[8][9]. Some languages, such as Eiffel, offer programmers the option to provide the loop-invariant. While this might solve the invariant problem, it pushes this responsibility to the programmer, which makes it inconvenient.

```
1     int index = 0;
2     int arr[] = new int[64];
3     while (index < arr.length) {
4        if (some_condition()) {
5            arr[index + 5];
6        }
7        else {
8            arr[index];
9        }
10       index += 1;
11    }
```

Figure 2: toy example of a loop with minimal control flow

The best current solution to this problem is a combination of automatic search and human annotation to aid the process[7].

3.2 Data Flow Analysis

This section describes two papers that utilize this technique. Data flow analysis is used for many global optimizations. It is done by means of first constructing a graph that represents the program. In this graph the nodes are instructions and the edges are possibilities of control flow between the nodes at execution-time[10].

3.2.1 Variable Range Analysis

This analysis was introduced by Harrison[11]. It is used for many optimizations, like elimination of redundant tests which includes redundant bounds-checks. It is also used for choosing the appropriate data representation at compile time, such as choosing the bitwidth of a variable.

What follows is a summary of how the analysis is done as described in Harrison's paper[11]:

The paper uses the common use-definition (UD) and definition-use (DU) chains. The data structures UD and DU represent a variable use followed by all previous definitions, or a variable definition followed by all preceding uses, respectively. In addition to those, Harrison introduces 2 other chains, namely, use-test (UT) and test-use (DT). These chains are generated from program points where variables are used in the conditional branch statements.

The initialization step of this algorithm is calculating the data flow graph of the program using the data flow chains mentioned previously. For each program point, the reachable variables' range are initialized to $[-\infty, \infty]$. It is important to mention that in the data flow graph, the algorithm introduces 2 nodes for each variable that is used inside a conditional branch statement. Each node corresponds to whether the control flow path is $true$ or $false$. These 2 nodes, along with the DT and DU chains are used to account for branching when calculating the ranges.

Referring back to the first code snippet in Figure 1. When the algorithm reaches the program point at line 3, the condition $index < 64$ is used to populate the 2 introduced nodes for the variable $index$. The node that corresponds to the $true$ branch is populated with the range $[-\infty, 64]$ and the $false$ branch is populated with the range $[64, \infty]$.

The algorithm iteratively inspects the program points and propagates range information until no changes

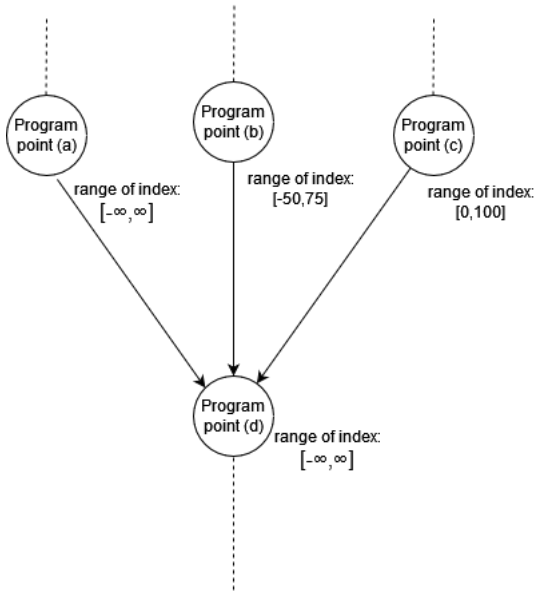

Figure 3: A segment of some data flow graph that shows the propagation of variable range information of a variable called index.

can be made, in which case, the algorithm halts. The propagation of the range information is similar to that of constant propagation except that the compiler needs to do operations on ranges. For a given program point, if the variable range originated from different data flow points, the resulting range at that program point is then the union of the ranges from the preceding points.

While this analysis seems promising, it suffers from some shortcomings. Firstly, the introduction of 2 nodes for each variable is not very desirable. This is due to the increase in the data the compiler needs to keep track of when doing this analysis. Because this analysis uses the UD and DU chains extensively, it means that the compiler needs to maintain global information about each variable that is reachable at a specific point in the data flow graph. This could increase the compilation time significantly the larger the code being compiled is.

Secondly, because the propagation of the ranges is done through the union of the ranges from the previous nodes, it causes loss of information if there is one preceding node with a large range. This is shown in figure 3. In a setting where this analysis is used for bounds-checks elimination, this loss of information means that there will be some bounds-checks that are redundant but are not able to be removed. Suppose that, at run-time, it is very rarely that the program reaches program point (d) from program point (a), the bounds-checks need to be present for whenever the program point (a) is reached.

The accuracy of this analysis is limited by the extent of granularity loss caused by variable range propagation as seen in figure 3. This can make this analysis unreliable for programs with complex data flow.

3.2.2 The ABCD Algorithm

This algorithm was developed by Gupta, et al[12] for the java Just-In-Time compiler. It operates directly on the LLVM IR and was designed with emphasis on the compilation time overhead given the dynamic nature of Java. It solves some of the issues that variable range analysis faces which are discussed further.

This algorithm constructs a global constraint system. This constraint system is encoded by means of the

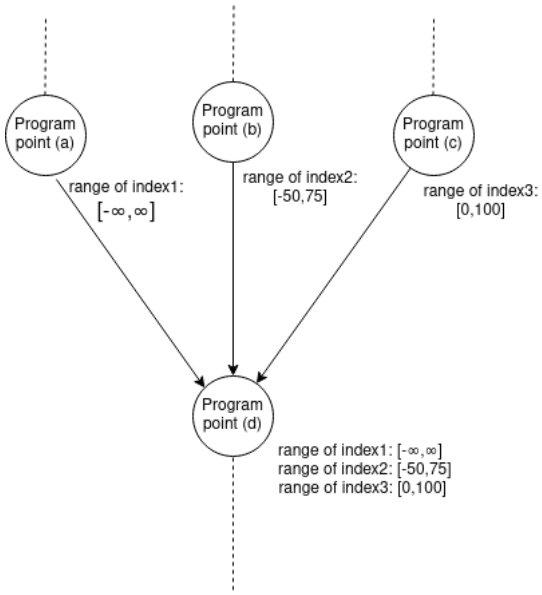

Figure 4: A segment of some data flow graph that shows the propagation of variable range information of a variable called index after the SSI conversion.

Static Single Information (SSI) form. The SSI form that Gupta, el al introduced is built from the Static Single Assignment (SSA) form by splitting the live range of each variable at certain program points. In this case, these program points are variable assignments and conditionals. This allows variables to carry information inferred from conditional statements[13] which the analyser can then use.

From a global view, what the SSI form ensures is that each variable in the program is unique to each program point. There is hence no need to keep track of the state of the variables at each program point.

After the SSI form is constructed, an inequality graph is built using the constraints that are in the LLVM IR. In essence, each constraint is a node and the edges between them are the relation between these constraints. The details of how this graph is generated are explained by Gupta, et al[12].

In order to determine whether a bounds-check is redundant, there are different strategies in which the inequality graph can be used to prove redundancy[12]. For example, finding the shortest path between the index and size to be less than 0 is enough to prove that a bounds-check is redundant.

Because this algorithm uses the SSI form, it has the advantage of the variables carrying more information as established earlier. This means that in cases where a variable has the possibility to originate from multiple different preceding program points, it can be distinguished which basic block it is coming from. In other words, this solves the problem shown in figure 3 that the variable range analysis suffers from. Information propagation with variable renaming using SSI can be seen in figure 4.

One drawback that this algorithm has is that the conversion from SSA to SSI has non-negligible overhead for compilation time[13]. Furthermore, because variables are introduced to ensure the SSI-property, it can cause the program to run out of registers to allocate and hence be forced to use the RAM, which has a significant impact on performance.

3.3 Avoiding bounds-checks

One way to avoid the bounds-checks overhead is to remove the need to check array bounds altogether. Most high level languages offer the iterator type to traverse over different data structures. Whenever the end of the memory is reached, null is returned. This is useful because it can then be used for the loop's condition, which eliminates the need for a loop induction variable.

4 Implementation

This section briefly describes a language developed to deliver some bench-marking results. Additionally, it details how the bounds-checks are added and eliminated. Finally, it presents an example to show the steps involved in calculating the redundancy of a bounds-check.

The prototype language that was developed for this paper is a subset of the c language. It supports most control flow constructs, int and array types, and constant strings. The frontend of this language emits Low Level Virtual Machine (LLVM) code. All the transformations and optimizations mentioned below are executed on this intermediate representation.

4.1 Bounds-check insertion

This pass iterates over the functions present in the module and inspects each instruction. Once a GetElementPtr (GEP) Instruction is found, it checks whether the memory being accessed is related to an array. If this is the case, a function call to *__check_bounds__* (seen in figure 5) is inserted before that GEP instruction. The arguments to this call are the index of this access and the size of the array. In order to provide the size of the array, the use-definition chain is traversed.

This pass naively adds bounds-checks before all array accesses; nevertheless, checks that may be readily proved to be unnecessary will be removed in any case.

```
1   void __check__bounds__(int index, int size) {
2       if (index < 0 || index >= size) {
3           printf("Error: memory violation\n");
4           exit(1);
5       }
6       return;
7    }
```

Figure 5: Function body of the bounds checker

4.2 Calculating Variable Range

Before this analysis is run, 2 passes are executed. The first pass is constant propagation. If the variable used to address memory originated from constants that are known at compile-time (as shown in figure 6), the bounds-check associated with such access can be easily validated for memory violation without the need for range calculations.

The second pass is memory-to-register (mem2reg), which mainly promotes memory references to register references. What makes it significant to the range analysis is the fact that mem2reg introduces PHI nodes. These nodes are particularly useful because they offer information about the possible paths that the

variable in question might originate from. PHI nodes will also be used to propagate the range of information between different basic blocks.

```
1   int arr[512];
2   int a = 10;
3   int b = 20;
4   int index = a * b;
5   arr[index];
6   /* with constant propagation
7       line 5 is replaced with arr[200] */
```

Figure 6: Example of an application of constant propagation

Once these passes run, the compiler can now be setup to do the range analysis. In order to manipulate the ranges of variables during the analysis, it needs to be able to do operations on those ranges as described by Harrison[11]. For instance, given a variable var with range $[-10, 10]$, if the analysis encounters an instruction that increments var by 5, the range should be updated with $[-5, 15]$.

Because the compiler for the prototype language uses LLVM, it means that the UD and DU chains are already calculated. These can be used to quickly find the program points where information can be extracted. Put differently, if the intent was to calculate the variable range for var, it is permissible to limit the attention to the last point where var was (re)defined. This is exactly where the UD chain comes in handy. Similarly, the DU chain is used to propagate the information about the range at the program points where var is used.

There are multiple cases that need to be considered while doing this analysis. The following sections describe how these are done.

4.2.1 Operations on Variables

When there is an operation between two variables, both ranges for these variables need to be known. This is because the resulting range is constructed by executing this operation on the lower and upper bounds of these ranges. Suppose there are two variables $var1$ and $var2$ with their respective ranges $[0, 10]$ and $[-50, -10]$. A new variable $var3$ that is defined as $var3 = var1 * var2$ has the following range:

$$[-50 * 0, -10 * 10] = [-100, 0]$$

The other operations' effects on the ranges follow the same logic.

4.2.2 Conditional Statements

A conditional branch splits the range of variable(s) used when that variable is part of the statement. In figure 8 on line 8, two pieces of information can be extracted from this compare instruction. Firstly, the upper bound of the variable $\%0$ is 64 at the start of $while.start$ basic block. Secondly, the lower bound of the variable $\%0$ is 64 at the start of $while.end$ basic block. This information is then combined with what is already known about $\%0$ to construct its range at the appropriate program point.

Figure 7 shows how range will differ depending on which program point in the control graph the variable is in.

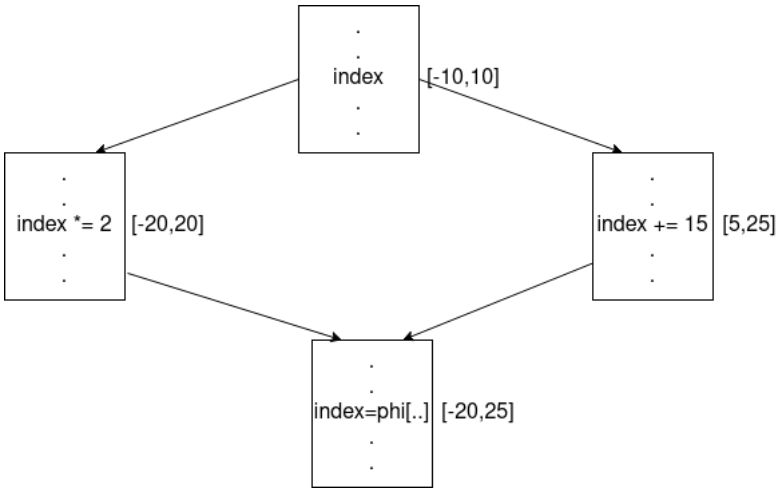

Figure 7: Graph that shows how the range of a variable will change depending on which program point it is present at.

4.2.3 Loops Influence on Variable Range

Loops are nothing more than conditional branching except they form a cycle in the control flow graph. This creates cyclic dependency on the data which makes it so that it usually needs multiple passes before the information about variables that reside inside the loop is propagated. Moreover, to be able to calculate ranges of variables inside a loop, it is very important to find the induction variable.

4.2.4 An Example of Variable Range Calculation

This section shows the exact steps that are taken to calculate the variable range for the array index in figure 8.

When the scanner that goes through the basic blocks reaches line 12, using the UD chain, it determines that the variable $\%index.0$ was last defined at line 7. The variable range analysis starts by checking what type of instruction line 7 is. The instruction in this case is a phi node that originate from the basic blocks $entry$ and $while.start$. From the $entry$ basic block, the value range associated with it is [0,0]. However, to get information about the variable $\%1$, the analysis tool needs to analyze the $while.start$ basic block.

In the $while.start$ basic block at line 13, it is seen that the variable $\%1$ is defined by an addition operator of the value 1 and the variable $\%index.0$. Because $\%index.0$ is defined by $\%1$ and $\%1$ is defined by $\%index.0$, it means that the initial value has come from one of the other nodes in the PHI instruction–from the $entry$ basic block in this case. Knowing that the initial value has the range $[0, 0]$ and that the variable only increases, it can be concluded that the lower bound of the range is thus 0.

The current information gathered thus far is that after line 7, the lower bound of $\%index.0$ is 0. Further analysis is needed to hopefully gather enough information to determine the upper bound. On line 8, the variable $\%index.0$ is used in a compare instruction. This splits the range of $\%index.0$ into two halves at the value 64. Because this compares if $\%index.0$ is less than 64, it means the range of this variable will be split at 64. On line 9, the result of this compare instruction is used to conditionally branch. When $\%0$ is true, it means that $\%index.0$ is less than 64 and it jumps to $while.start$ basic block. This means in this basic block, the upper bound must be 64.

When $\%0$ is false, the code branches to the $while.end$ basic block. There, it can be concluded that the lower bound of $\%index.0$ is 64.

With this information gathered, it can be concluded what the final range of $\%index.0$ will be. After line 7 it is confirmed that the lower bound is 0 and it was established that in the $while.start$ basic block the upper range is 64. This yields the range $[0, 64]$ for the variable $\%index.0$ in the $while.start$ basic block.

On the other hand, it is known that in the basic block $while.end$, the upper range for $\%index.0$ is 64. However, there is not enough information to infer the lower range of it. This results in the range $[64, \infty]$ inside that basic block.

```
1  define i32 @main(i32 %argc, i8** %argv) {
2   entry:
3     %arr = alloca i32, i32 512, align 4
4     br label %while.cond
5
6   while.cond:    ;preds = %while.start, %entry
7     %index.0 = phi i32 [0, %entry],[%1, %while.start]
8     %0 = icmp slt i32 %index.0, 64
9     br i1 %0, label %while.start, label %while.end
10
11  while.start:  ; preds = %while.cond
12   call void @__check_bounds__(i32 %index.0, i32 512)
13   %1 = add i32 %index.0, 1
14   br label %while.cond
15
16  while.end:    ; preds = %while.cond
17   ret i32 0
18 }
```

Figure 8: LLVM IR generated from a simple loop traversal of an array in the first code snipped in figure 1

4.3 Bounds-check Elimination

After having calculated the value range of the variables, there is enough information to start identifying redundant bounds-checks. This pass iterates over the functions of the module. In each function, the instructions are inspected. Whenever a call to *__check_bounds__* is found, it is checked whether the range of the index that was calculated in the previous analysis does not violate the memory given the size of the array.

If a bounds-check is found to be redundant, it is added to a list that accumulates all the function calls that can be safely removed. At the end of the pass, any instructions that are in the accumulator list are removed.

4.4 Summary of The Algorithm

After having discussed all the pieces needed to achieve the range analysis, this section explains how how everything is put together and in what steps actions are taken. This algorithms assumes the bounds-checks are already inserted.

1. run the mem2reg and cp pass.
2. scan the IR for calls to $\_\_check\_bounds\_\_$.
3. when a call is found, calculate the range for the index variable as described in the previous sections.
4. if the range of the index passed to $\_\_check\_bounds\_\_$ always has a lower bound that is larger than 0 and an upper bound that is less than the size of the array. Then the function call is removed, otherwise, it is kept unchanged.
5. the algorithm is done when all the basic blocks are inspected.

5 Measuring Performance Difference

In this section, bench-marking is done on different programs and sorting algorithms. The system these tests are run on has a 6-Core Intel Core i7-8750H 64bits CPU. Additionally, it runs Ubuntu 20.04.4.

To avoid noise in the measurements, each instance of the benchmark is repeated 1000 times and the average of these measurements is used instead. Moreover, when the code is run, it is ensured that different runs of the same program are not bench-marked in succession. This is to avoid inflating the results of performance due to caching or branch prediction buffering.

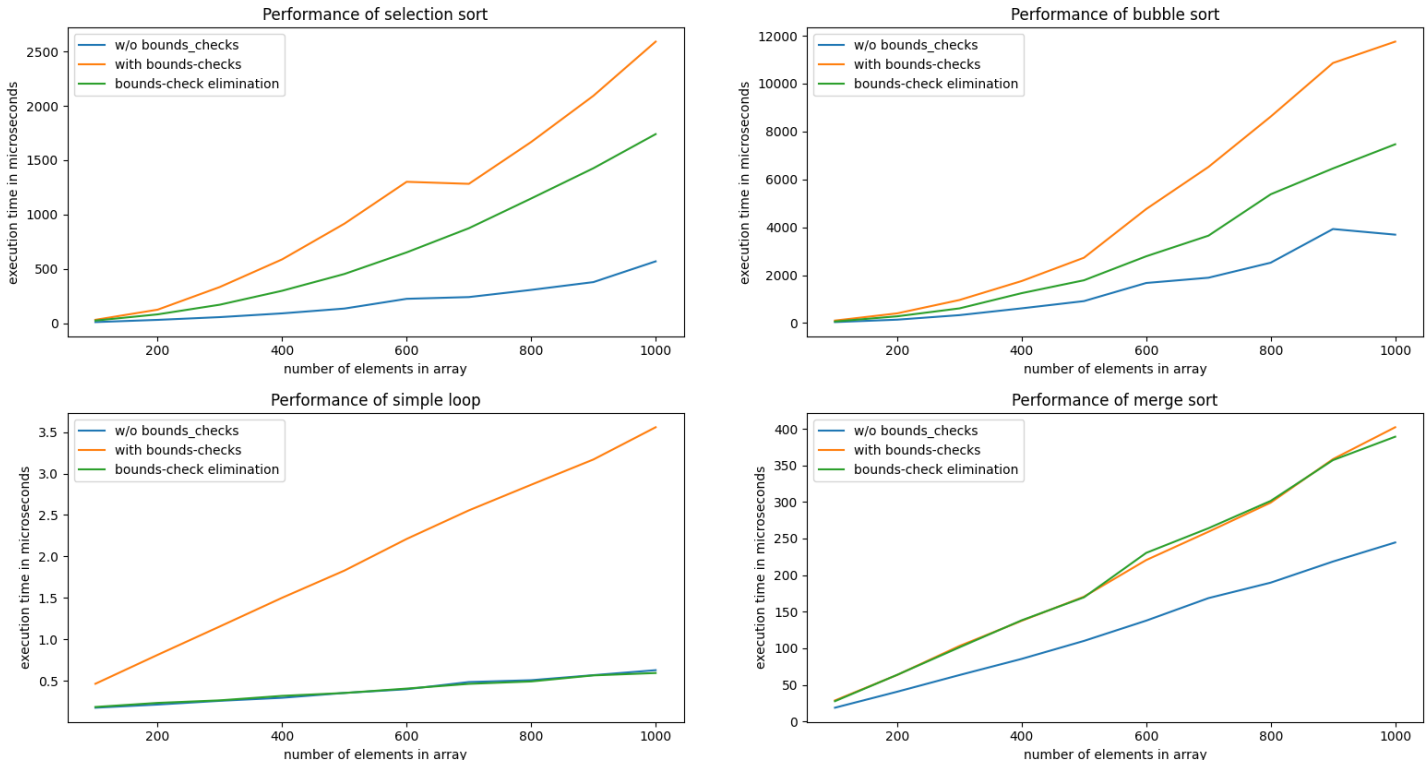

Figure 9: A few programs that show the difference in performance between the different compiler passes

It has been established that the nature of the code influences the usefulness of bounds-checks elimination. In cases with loops that can be easily analyzed for its induction variable, the range analysis can easily

extract enough information to completely eliminate bounds-checks. This can be seen in figure 9 program ***simple loop***.

When nested loops are present, the current implementation of bounds-checks elimination algorithm, as discussed in earlier sections, suffers. This is because the tool would need to keep track of all parent loops and reason about how many times each loop runs. This increases the complexity and resources needed significantly-memory and time. However, the tool can still do the analysis reasonably well, as seen in figure 9 in both selection and bubble sort algorithms. Although the resulting speed is lower than the original un-safe performance, for such algorithms, it is relatively cheap to fully remove the bounds-checks if the implementation is extended to handle nested loops.

```
    .
    .
1   entry.endif:
2     %i1.promoted = load i32, i32* %i1, align 4
3   entry.while.start:
4     %1035 = phi i32 [ %1041, %.while.start ], [ %i1.promoted, %entry.endif ]
5     call void @__check_bounds__(i32 %1035, i32 %1031)
6     %1037 = getelementptr i32, i32* %L, i32 %1035
    .
    .
```

Figure 10: IR snippet code generated from the bench-marked mergesort

Moreover, the tool is limited when pointer aliasing is involved. Hence, why the benchmark that was done on merge sort shows no improvements after running the bounds-checks elimination pass. Figure 10 shows a bounds-check that the analysis could not be removed due to this shortcoming. The variable $\%i1.promoted$ originates from a load instruction and is later used in the phi instruction on line 4. The result of this phi is then used to index the array at line 6. In order to get information about the range of this index ($\%1035$ in this case), the compiler needs to keep track of all the pointer aliases of $\%i1$. The issue of pointer aliasing is not limited to the range analysis, the ABCD algorithm with its SSI form would struggle in the same fashion.

Although it is possible to handle some instances where pointer aliasing is present, the implementation does not handle such cases as it is beyond the scope of this paper. It is, however, important to point out such issues.

The purpose of these measurements is to concretely show the potential overhead that bounds-checks can introduce as well as the performance that can be saved.

6 Conclusion

This paper has shown possible methods for bounds-checks to be optimized. It is very clear that the bounds-checks have a non-negligible overhead as seen in figure 9 even when optimized using variable range analysis.

Depending on the nature of the program, bounds-checks elimination may deliver significant speed-ups. However, this speedup in many cases still does not match the original speed of the program before bounds-checks were introduced. This makes it so that the bounds-checks pass is not always desirable for systems where every percentage of performance is of high value.

In systems where memory safely takes precedence over performance, and it is still important for compilers to produce code with high performance, this analysis is very applicable.

References

[1] F. Chow, "A portable machine-independent global optimizer - design and measurements," Ph.D. dissertation, 01 1983.

[2] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 985–999.

[3] W. H. Hawkins, J. D. Hiser, and J. W. Davidson, "Dynamic canary randomization for improved software security," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, ser. CISRC '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2897795.2897803

[4] M. A. Ertl and A. Krall, "Optimal instruction scheduling using constraint logic programming," in *Programming Language Implementation and Logic Programming*, J. Maluszyński and M. Wirsing, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 75–86.

[5] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[6] C. A. Hoare, "Proof of correctness of data representations," *Acta Inf.*, vol. 1, no. 4, p. 271–281, dec 1972. [Online]. Available: https://doi.org/10.1007/BF00289507

[7] G. C. Necula and P. Lee, "Compiling with proofs," Ph.D. dissertation, USA, 1998, aAI9918593.

[8] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Comput. Surv.*, vol. 46, no. 3, jan 2014. [Online]. Available: https://doi.org/10.1145/2506375

[9] A. Humenberger, M. Jaroschek, and L. Kovács, "Automated generation of non-linear loop invariants utilizing hypergeometric sequences," in *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 221–228. [Online]. Available: https://doi.org/10.1145/3087604.3087623

[10] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 194–206. [Online]. Available: https://doi.org/10.1145/512927.512945

[11] W. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Transactions on Software Engineering*, vol. 3, no. 03, pp. 243–250, may 1977.

[12] R. Bodík, R. Gupta, and V. Sarkar, "Abcd: Eliminating array bounds checks on demand," *SIGPLAN Not.*, vol. 35, no. 5, p. 321–333, may 2000. [Online]. Available: https://doi.org/10.1145/358438.349342

[13] R. S. B. M. A. B. Andre Luiz C. Tavares, Fernando M. Q. Pereira, "Efficient ssi conversion." Departamento de Ci^encia da Computac ao – UFMG, November 2009.