Optimizing Array Bound Checks Using Flow Analysis

Rajiv Gupta

Presented by Lauren Biernacki, Colton Holoday, Yirui Liu, Andrew McCrabb
Bounds Checking

Python

```python
array = [0, 1, 2]
for i in range(5):
    print (array[i]),
```

C++

```cpp
#include <iostream>
using namespace std;

int array[] = {0, 1, 2};
int main()
{
    for (int i=0; i<10; i++)
        cout << array[i] << " ";
    return 0;
}
```

Traceback (most recent call last):
File "main.py", line 3, in <module>
    print (array[i]),
IndexError: list index out of range

...Program finished with exit code 0
Stack Buffer Overflow Vulnerability

C++

```c
void target() {
    printf("You overflowed successfully, gg");
    exit(0);
}

void vulnerable(char* str1) {
    char buf[5];
    strcpy(buf, str1);
}

int main() {
    vulnerable("AAAAAAAAAAAA\xf0\x01\x01\x00");
    printf("This only prints in normal control flow");
}
```

Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer overflows

**Instruments code to:**

- Create poisoned *redzones* around stack objects
- Check *shadow memory* before each memory access
Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer overflows

Instruments code to:
- Create poisoned *redzones* around stack objects
- Check *shadow memory* before each memory access

"AddressSanitizer achieves efficiency without sacrificing comprehensiveness."

73% slowdown, 337% increased memory usage
Compile Time Optimizations for ASan

- Using dataflow techniques, such as the work done by Gupta, it should be possible to optimize ASan’s checks

- This could be applied to other memory safety protections, or simply bounds checking in general

```c
if (f) {
    a[i] = ...;
} else {
    a[i] = ...;
}
```

`Fully redundant checks`

```c
//Enough to check a[i] here
if (f) {
    a[i] = ...;
} else {
    a[i] = ...;
}
```

`Hoisting bounds checks`
Optimizing Array Bounds Checks

1. Local elimination
2. Global elimination
   a. Elimination algorithm
   b. Further optimization
3. How to deal with loops
4. Evaluation
Local Elimination

Before Optimization

\[
\begin{align*}
&\text{MIN}(a) \leq i + 1 \leq \text{MAX}(a) \\
&\text{temp} \leftarrow a[i+1] \\
&\text{MIN}(a) \leq i + 1 \leq \text{MAX}(a) \\
&a[i+1] \leftarrow a[i] \\
&\text{MIN}(a) \leq i \leq \text{MAX}(a) \\
&a[i] \leftarrow \text{temp}
\end{align*}
\]

After Optimization

\[
\begin{align*}
&\text{MIN}(a) \leq i, i + 1 \leq \text{MAX}(a) \\
&\text{temp} \leftarrow a[i+1] \\
&a[i+1] \leftarrow a[i] \\
&a[i] \leftarrow \text{temp}
\end{align*}
\]

Fig. 1. Local elimination of bound checks.
Global Elimination

- $10 \leq i \leq 50$
- $20 \leq i \leq 100$
- $5 \leq i \leq 200$
Global Elimination

Both checks *subsume* the last one.
By propagating bounds checks through the CFG we can determine which checks are redundant and eliminate them.

Both checks subsume the last one.
Available Checks ~ “A bound check $C$ is available at a program point $p$ if it is guaranteed that, along each path leading to point $p$, either $C$ is performed or a check that subsumes $C$ is performed.”

<table>
<thead>
<tr>
<th>Key Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Paths</td>
</tr>
</tbody>
</table>
Eliminating Redundant Checks

\[ 10 \leq i \leq 50 \]
\[ 20 \leq i \leq 100 \]
\[ 5 \leq i \leq 200 \]
Eliminating Redundant Checks

IN = 10 <= i <= 50

OUT = 10 <= i && i <= 50

IN = 20 <= i <= 100

OUT = 20 <= i && i <= 100

IN = 10 <= i && i <= 50 && 20 <= i && i <= 100

5 <= i <= 200

== 20 <= i && i <= 50
Let’s Formalize the Analysis

\[ C_{OUT}[B] = C_{GEN}[B] \lor \text{forward}(C_{IN}[B], B), \]
\[ C_{IN}[B] = \bigwedge_{P \in \text{Pred}(B)} C_{OUT}[P], \text{ where } B \text{ is not the initial block,} \]
\[ C_{IN}[B] = \emptyset, \text{ where } B \text{ is the initial block.} \]
Handling KILL Set

- Monotonic operations can retain checks through kill filter

20 \leq i \leq 100
i = i + 1

\text{OUT} = 20 \leq i \land i \leq 100

\text{forward}(C_{IN}[B], B) \{
S = \emptyset
\text{for each check } C \in C_{IN}[B] \text{ do}
\text{case } C \text{ of}
\begin{align*}
\text{lb} \leq v: & \\
\text{case AFFECT}(B, v) \text{ of}
\begin{align*}
\text{unchanged: } & S = S \cup \{ lb \leq v \} \\
\text{increment: } & S = S \cup \{ lb \leq v \} \\
\text{decrement: } & /* \text{the check is killed} */ \\
\text{multiply: } & S = S \cup \{ lb \leq v \} \\
\text{div}>1: & /* \text{the check is killed} */ \\
\text{div}<1: & S = S \cup \{ lb \leq v \} \\
\text{changed: } & /* \text{the check is killed} */
\end{align*}
\end{align*}
end case
\begin{align*}
v \leq ub: & \\
\text{case AFFECT}(B, v) \text{ of}
\begin{align*}
\text{unchanged: } & S = S \cup \{ v \leq ub \} \\
\text{increment: } & /* \text{the check is killed} */ \\
\text{decrement: } & S = S \cup \{ v \leq ub \} \\
\text{multiply: } & /* \text{the check is killed} */ \\
\text{div}>1: & S = S \cup \{ v \leq ub \} \\
\text{div}<1: & /* \text{the check is killed} */ \\
\text{changed: } & /* \text{the check is killed} */
\end{align*}
end case
\end{align*}
\}

Optimizing Array Bounds Checks

1. Local elimination
2. Global elimination
   a. Elimination algorithm
   b. Further optimization
3. How to deal with loops
4. Evaluation
Eliminating Redundant Checks

Before Optimization

\[
\begin{align*}
-5 \leq i &\leq 200 \\
\text{if ( ) then} &\quad -10 \leq i \leq 50 \\
\text{else} &\quad -20 \leq i \leq 100 \\
\text{fi} &\quad \text{Before Optimization}
\end{align*}
\]

After Modification

\[
\begin{align*}
-10 \leq i &\leq 100 \\
\text{if ( ) then} &\quad -10 \leq i \leq 50 \\
\text{else} &\quad -20 \leq i \leq 100 \\
\text{fi} &\quad \text{After Modification}
\end{align*}
\]

After Elimination

\[
\begin{align*}
-10 \leq i &\leq 100 \\
\text{if ( ) then} &\quad -i \leq 50 \\
\text{else} &\quad -20 \leq i \\
\text{fi} &\quad \text{After Elimination}
\end{align*}
\]
Formulating a Dataflow Analysis

Very-busy Checks ~ “A bound check $C$ is very busy at a program point $p$ if it is guaranteed that, along each path starting at point $p$, either $C$ is performed or a check that subsumes $C$ is performed.”

<table>
<thead>
<tr>
<th>Key Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Paths</td>
</tr>
<tr>
<td>Backward</td>
</tr>
<tr>
<td>$\land$ instead of $\lor$</td>
</tr>
</tbody>
</table>
Let’s Formalize the Analysis

Compute the set of very-busy checks at all points in the program

\[
C_{\text{IN}}[B] = C_{\text{GEN}}[B] \lor \text{backward}(C_{\text{OUT}}[B], B),
\]

\[
C_{\text{OUT}}[B] = \bigwedge_{S \in \text{Succ}(B)} C_{\text{IN}}[S], \quad \text{where } B \text{ is not the terminating block},
\]

\[
C_{\text{OUT}}[B] = \emptyset, \quad \text{where } B \text{ is the terminating block};
\]

\[
S_1 \land S_2 \land \cdots \land S_n
\]

\[
= \{C : \forall S_i, 1 \leq i \leq n, (C \in S_i \lor \exists C' \in S_i \land C' \text{ subsumes } C)\},
\]

\[
S_1 \lor S_2 \lor \cdots \lor S_n
\]

\[
= \{C : (\exists S_i, 1 \leq i \leq n, C \in S_i) \land (\nexists C' \in S_i, 1 \leq i \leq n, C' \text{ subsumes } C)\}.
\]
Let’s Formalize the Analysis

Compute the set of very-busy checks at all points in the program

\[ C_{IN}[B] = C_{GEN}[B] \lor \text{backward}(C_{OUT}[B], B), \]
\[ C_{OUT}[B] = \bigwedge_{S \in \text{Succ}(B)} C_{IN}[S], \text{ where } B \text{ is not the terminating block; } \]
\[ C_{OUT}[B] = \emptyset, \text{ where } B \text{ is the terminating block; } \]
\[ S_1 \land S_2 \land \cdots \land S_n \]
\[ = \{ C : \forall S_i, 1 \leq i \leq n, (C \in S_i \lor \exists C' \in S_i \land C' \text{ sub} ) \} \]
\[ S_1 \lor S_2 \lor \cdots \lor S_n \]
\[ = \{ C : (\exists S_i, 1 \leq i \leq n, C \in S_i) \land (\forall C' \in S_i, 1 \leq i \leq n, C \text{ based} ) \}. \]
Modifying Checks

If a check C’ is very busy at the point immediately following the check C, and C’ subsumes C, then C can be replaced by C’.

--- 5 ≤ i ≤ 200
  if () then
    -- 10 ≤ i ≤ 50
    ....
  else
    -- 20 ≤ i ≤ 100
    ....
  fi

Before Optimization

--- 10 ≤ i ≤ 100
  if () then
    -- 10 ≤ i ≤ 50
    ....
  else
    -- 20 ≤ i ≤ 100
    ....
  fi

After Modification

--- 10 ≤ i ≤ 100
  if () then
    -- i ≤ 50
    ....
  else
    -- 20 ≤ i
    ....
  fi

After Elimination
Optimizing Array Bounds Checks

1. Local elimination
2. Global elimination
   a. Elimination algorithm
   b. Further optimization
3. How to deal with loops
4. Evaluation
Propagating the Checks out of the Loops

Example:

Before Optimization

After Propagation

Fig. 7. Propagation of bound checks.
Propagating the Checks out of the Loops

Goal:
- Reduce the number of times the checks are executed

Algorithm:
- Identify the candidates (e.g. loop invariants) for propagation
  - Use-def Chain
  - Dominator Sets
- Check hoisting
- Propagate the checks out of the loop
Propagating the Checks out of the Loops

Another Example:

\[
\begin{align*}
&\text{for } i \leftarrow \text{min to max} \text{ do} \\
&\quad \text{if (inc) then } \quad \text{-- } \text{MIN}(a) \leq i \leq \text{MAX}(a) \\
&\quad \quad \text{sum } \leftarrow \text{sum } + a[i] \\
&\quad \quad \text{else } \quad \text{-- } \text{MIN}(a) \leq i \leq \text{MAX}(a) \\
&\quad \quad \text{sum } \leftarrow \text{sum } - a[i] \\
&\quad \text{fi} \\
&\text{od} \quad \text{Before Propagation}
\end{align*}
\]

\[
\begin{align*}
&\text{for } i \leftarrow \text{min to max} \text{ do} \\
&\quad \text{if (inc) then } \\
&\quad \quad \text{sum } \leftarrow \text{sum } + a[i] \\
&\quad \quad \text{else } \text{sum } \leftarrow \text{sum } - a[i] \\
&\quad \text{fi} \\
&\text{od} \quad \text{After Propagation}
\end{align*}
\]

Fig. 10. Propagation out of loops with known bounds for subscript variables.
Optimizing Array Bounds Checks

1. Local elimination
2. Global elimination
   a. Elimination algorithm
   b. Further optimization
3. How to deal with loops
4. Evaluation
## Experimental Evaluation

>80% of Bounds Checks Eliminated on Average

<table>
<thead>
<tr>
<th></th>
<th>UNOPT</th>
<th>L-elim</th>
<th>G-elim</th>
<th>Prop</th>
<th>Total Deleted</th>
<th>% Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>59,400</td>
<td>39,600</td>
<td>9,900</td>
<td>9,900</td>
<td>59,400</td>
<td>100%</td>
</tr>
<tr>
<td>Quick</td>
<td>271,184</td>
<td>72,784</td>
<td>10,014</td>
<td>54,347</td>
<td>137,145</td>
<td>51%</td>
</tr>
<tr>
<td>Queen</td>
<td>13,784</td>
<td>2,288</td>
<td>1,748</td>
<td>1,778</td>
<td>5,814</td>
<td>42%</td>
</tr>
<tr>
<td>Towers</td>
<td>556,262</td>
<td>261,944</td>
<td>97,844</td>
<td>0</td>
<td>359,788</td>
<td>65%</td>
</tr>
<tr>
<td>Lloop6</td>
<td>20,160</td>
<td>8,064</td>
<td>0</td>
<td>12,096</td>
<td>20,160</td>
<td>100%</td>
</tr>
<tr>
<td>FFT</td>
<td>37,414</td>
<td>24,568</td>
<td>0</td>
<td>5,930</td>
<td>30,498</td>
<td>82%</td>
</tr>
<tr>
<td>MatMul</td>
<td>1,043,200</td>
<td>640,000</td>
<td>256,000</td>
<td>147,200</td>
<td>1,043,200</td>
<td>100%</td>
</tr>
<tr>
<td>Perm</td>
<td>80,624</td>
<td>10,078</td>
<td>0</td>
<td>7,240</td>
<td>17,318</td>
<td>21%</td>
</tr>
</tbody>
</table>
Implications of This Work

1993
● Compilers came with “array bound check” flag
● Too much performance and memory overhead
● Gupta publishes this paper

Today
● Address Sanitizer used to provide comprehensive memory checks
● Still comes with high overheads
● We can apply these three optimizations from Gupta to reduce overheads
Conclusion

Comprehensive Bounds Checking
● Useful for Testing & Debugging
● 73% slowdown; 337% memory overhead

Pre-process bounds checks to eliminate many runtime checks
● Local & Global Elimination; Loop Propagation
● >80% Runtime bounds checks eliminated
Questions

“Optimizing Array Bound Checks Using Flow Analysis”
Backup Slides
Address Sanitizer (ASan)

- An open source tool created by Google, included in LLVM
- Used to identify memory errors, including buffer overflows
- Consists of two parts:
  - **Code Instrumentation** — Creates poisoned redzones around stack and global objects, instruments code to check shadow memory before each memory access
  - **Run-time Library** — Augments malloc() and free() to apply the above protections to the heap
Address Sanitizer (ASan)

Before:

```c
void foo() {
    char a[32];
    ...
    *address = ...;
    return;
}
```

After:

```c
void foo() {
    char redzone1[32];
    char a[32];
    char redzone3[32];
    int *shadow = MemToShadow(redzone1);
    // poison redzones
    shadow[0] = 0xffffffff;
    shadow[1] = 0x00000000;
    shadow[2] = 0xffffffff;
    ...
    if (IsPoisoned(address)) {
        ReportError(address);
    }
    *address = ...;
    ...
    // unpoison all
```
Address Sanitizer (ASan)

Before:

```
void foo() {
    char a[32];
    ...
    *address = ...;
    return;
}
```

After:

```
void foo() {
    char redzone1[32];
    char a[32];
    char redzone3[32];
    int  *shadow = MemToShadow(redzone1);
    // poison redzones
    shadow[0] = 0xffffffff;
    shadow[1] = 0x00000000;
    shadow[2] = 0xffffffff;
    ...
    *address = ...;
    // unpoison all
    return;
}
```

“AddressSanitizer achieves efficiency without sacrificing comprehensiveness.”

73% slowdown, 337% increased memory usage
Main Insights

Elimination:
- Eliminate redundant checks at compile time
- Analogous to constant folding and common subexpression elimination

Propagation:
- Propagate bound checks out of loops to reduce the number of run-time checks
- Analogous to loop invariant code motion optimization
Algorithm for Eliminating Redundant Checks

Fig. 3. Global elimination by modification of bound checks.