Dynamic and Feedback-Directed Optimizations

*meet the PL faculty*

**Rastislav Bodík**
Talk Outline

• Underlying theme
  • technology trends and personal philosophy

• Some ongoing projects:
  • **ABCD: Array-Bounds-Check Optimization for Java**
    • Denis Gopan
  • **Profile-Directed Program Specialization**
    • Subbu Sastry, Jim Smith
  • **VM-Sensitive Malloc**
    • Shai Rubin, Trishul Chilimbi
One important technology trend

µProc: 60%/yr.
DRAM: 7%/yr.

Processor-Memory Performance Gap: (grows 50% / year)

“Moore’s Law”

graph by D. Patterson
Processor-memory gap

• A pessimist says:
  • “memory is becoming slower”

  gap today: 100 cycles x 4-wide = 400 instrs
  gap in 6 years: 1,000 cycles x 8-wide = 8,000 instrs

• An optimist says:
  • “processor is becoming more intelligent”

  gap today: 400 instructions =
  • generate a 20-instruction loop and execute it 10 times

  gap in 6 years: 8,000 instructions =
  • generate a 200-instruction loop and execute it 300 times
ABCD: Eliminating Array-Bounds Checks on Demand

Denis Gopan  U of Wisconsin
Rajiv Gupta  U of Arizona
Vivek Sarkar  IBM TJ Watson
Why do we need array-bounds checks?

**Pro:** type-safe programs don’t “crash”

**Con:** some violations checked at run time

**Direct cost:** *executing* the checks
checks are frequent, expensive

**Indirect cost:** *preventing* optimization
checks block code motion of side-effect instructions

**Our goal:** safety without performance penalty

**How?** remove redundant checks
The Problem

- Checks must be removed at run-time
  - all checks must be present when the bytecode program enters the JVM

- Existing bounds-check optimizers:
  - their emphasis: precision
    - goal: all checks removed = statically type-safe
    - theorem prover, range propagation, types
  - their properties
    - too heavy-weight
    - notion of control flow is lost
      - how to add profile feedback?
Why optimize on demand?

- Optimize only the **few hot** checks

![Graph showing the relationship between number of static checks and percentage of dynamic checks analyzed (not necessarily removed).]
Why optimize on demand?

- optimize only the few hot checks

(checks analyzed) (not necessarily removed)

% of dynamic checks

number of static checks

checks analyzed

% of dynamic checks

mpegaudio

0 10 20 30 40 50 60 70 80 90 100
An ideal dynamic optimizer?

- A balance between power and economy
  - powerful just enough
  - minimize analysis work
  - reduce IR overhead
  - only common cases
  - efficient IR
  - reuse the IR

- Scalable
  - optimize only hot checks
  - no whole-program analysis
  - profile-directed
  + demand-driven
  - use “local” info
  + insert (cold) checks
High-level algorithm

for each hot array access $A[i]$ do

-- optimize upper-bound check
$ABCD( i < A.length )$

-- optimize lower-bound check
$ABCD( 0 \leq i )$

end for
High-level algorithm

for each hot array access $A[i]$ do

-- optimize upper-bound check
ABCD( $i < A.length$ )

-- optimize lower-bound check
ABCD( $0 \leq i$ )

drop for
Example

```java
i ← A.length
while ( ) {
    --i
    ..A[i]..
}
```
Example

\[
i \leftarrow A.\text{length} \\
\text{while ( ) } \{ \\
\quad --i \\
\quad \text{..}A[i]\text{..} \\
\} \\
\]

ABCD = SSA + shortest path
A simplified ABCD algorithm

1. build SSA

2. label edges with constraints

3. analyze $A[i_k]$: is $i_k < A.length$ always true?
\[ i_0 \leftarrow A\.length \]

\[ i_1 \leftarrow \varnothing(i_0, i_2) \]

\[ i_2 \leftarrow i_1 - 1 \]
\[ \ldots A[i_2] \ldots \]
\[ i_0 \leftarrow A.length \]

\[ i_1 \leftarrow \emptyset(i_0, i_2) \]

\[ i_2 \leftarrow i_1 - 1 \]

\[ \ldots A[i_2] \ldots \]

- \( A.length \)
- \( i_0 \)
- \( \emptyset \) \( i_1 \)
- \( i_2 \)
Simple ABCD

\[ i_0 \leftarrow \text{A.length} \]

\[ i_1 \leftarrow \emptyset(i_0, i_2) \]

\[ i_2 \leftarrow i_1 - 1 \]

\[ \ldots \text{A}[i_2] \ldots \]
\( i_0 \leftarrow A.\text{length} \)

\( i_1 \leftarrow \emptyset(i_0, i_2) \)

\( i_2 \leftarrow i_1 - 1 \)

\( \ldots A[i_2] \ldots \)
Simple ABCD

\[ i_0 \leftarrow A.length \]

\[ i_1 \leftarrow \emptyset(i_0, i_2) \]

\[ i_2 \leftarrow i_1 - 1 \]

.. A[i_2] ..
Simple ABCD

\[ i_0 \leftarrow A\text{.length} \]

\[ i_1 \leftarrow \emptyset(i_0,i_2) \]

\[ i_2 \leftarrow i_1 - 1 \]

\[ \ldots A[i_2] \ldots \]

\[ i_0 \leq A\text{.length} - 0 \]

\[ i_1 \leq i_0 - 0 \]

\[ i_2 \leq i_1 - 1 \]
Simple ABCD

\[
i_0 \leftarrow A.length
\]
\[
i_1 \leftarrow \emptyset(i_0, i_2)
\]
\[
i_2 \leftarrow i_1 - 1
\]
\[
.. A[i_2] ..
\]

\[
i_0 \leq A.length - 0
\]
\[
i_1 \leq i_0 - 0
\]
\[
i_2 \leq i_1 - 1
\]

weight(A.length \to i_2) = 1 \Rightarrow i_2 \leq A.length - 1
A simplified ABCD algorithm

1. build SSA

2. label edges with constraints

3. analyze $A[i_k]$:  
   
   *input:* a bounds check $i_k < A.length$  
   
   *algorithm:* find shortest path $p$ from $A.length$ to $i_k$  
   
   *output:* check is redundant if $weight(p) > 0$
ABCD with run-time checks

```c
f(int A[], int n)
{
    for (i=0; i < n; i++)
        // A[i]
}
```
f(int A[], int n)
{
    if (n <= A.length)
    {
        for (i=0; i < n; i++)
            A[i];
    }
}
ABC with run-time checks

\[ f(int A[], int n) \]
\[
\begin{array}{l}
\{ \\
\quad \textbf{if} \ (n \leq A.length) \\
\quad \text{for} \ (i=0; \ i < n; \ i++) \\
\quad \quad ..A[i].. \\
\}\n\end{array}
\]

\rightarrow \text{run-time checks eliminate need for whole-program analysis}
Classification of hot checks

checks removed [% of all dynamic checks]

examined
“removable”
ABCD
JIT-like
Classification of hot checks

checks removed [% of all dynamic checks]

examined
“removable”
ABCD
JIT-like

16
Summary

• Current speedup: modest, up to about 5%
  • direct cost: in Jalapeno, checks already very efficient
  • indirect cost: few global optimizations implemented (11/99)

• Analysis time
  • 4 ms / check = visit 10 SSA nodes / check
  • recall 20 checks yields 80% dynamic coverage
  ⇒ 80 ms to analyze a large benchmark!

• Precision:
  • can be improved with few extensions (ABCD → ABCDE)
  • remaining checks appear beyond compiler analysis

👉 Use ABCD for your bounds-check optimization
Profile-Directed Program Specialization

Subbu Sastry

Jim Smith
What is program specialization?

- A famous and effective optimization
  - it hard-codes part of the input into the program

- Example:

```c
while (...) {
    fprintf(fp, "Hello %d %s", i, str[i]);
}
```
Example (cont)

```c
fprintf (fp, format, ...) {
    for ( ; *format; format++) {
        switch (*format) {
            case '%':
                switch (*++format) {
                    case 'd': print_int(va_arg(int), fp); break;
                    case 's': fputs(va_arg(char*), fp); break;
                    ...
                }
            case '\': // handle \t, \n, ...
            ...
            default: fputc(*format, fp); break;
        }
    }
}
```
Example (cont)

```c
while (...){
    specialized-fprintf(fp, i, str[i]);
}

specialized-fprintf (fp, v1, v2) {
    fputs("Hello ", fp);
    print_int(v1, fp);
    fputc(' ', fp);
    fputs(v2, fp);
    fputs(v2, fp);
}```
Why did not specialization make it (yet) ?

Two open questions:

1. **Are there opportunities for specialization in programs that are not interpreters?**

2. **How to identify specialization regions automatically?**
Question 1: opportunities

• An experimental limit study revealed:
  
  • estimated speedup:

<table>
<thead>
<tr>
<th>perl</th>
<th>m88ksim</th>
<th>gcc</th>
<th>interp</th>
<th>printf</th>
<th>go</th>
<th>li</th>
<th>jpeg</th>
<th>raytrace</th>
<th>strata</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.77</td>
<td>1.40</td>
<td>1.38</td>
<td>2.31</td>
<td>1.42</td>
<td>1.02</td>
<td>1.10</td>
<td>1.08</td>
<td>1.05</td>
<td>1.03</td>
</tr>
</tbody>
</table>

➤ even non-interpreters have significant speedup potential!
So far, programmers had to manually annotate which variables should be hard-coded.

Our approach:
- hardware-assisted monitoring of values and
- subsequent specialization-region formation
Virtual-Memory-Sensitive Malloc

Shai Rubin

Trishul Chilimbi  Microsoft Research
Motivation

• Modern programs:
  • use malloc/new extensively
    • lots of objects on the heap
    • poor memory locality

• Optimization opportunity:
  • allocate objects to improve memory behavior
Prior work

- When an object is allocated, predict if it is going to be
  - hot / cold
  - long- / short-lived

- Send it to a heap dedicated to
  - hot objects
  - cold objects
Observation

- The hot heap is still too big
  - needs to be split

- Our approach:
  - when allocating objects, predict which of them will be accessed together
  - send them to the same heap
  - ideally, size of each heap < 1 page