Valence: Variable Length Calling Context Encoding

Tong Zhou, Michael R. Jantz, Prasad A. Kulkarni, Kshitij A. Doshi, Vivek Sarkar
void append(List list) {
    lock();
    list.append(…);
    unlock();
}
void append(List list) {
    lock();
    list.append(...);
    unlock();
}
void append(List list) {
    lock();
    list.append(...);
    unlock();
}

void bob_call() {
    lock();
    append(globalList);
    unlock();
}
void append(List list) {
    lock();
    list.append(...);
    unlock();
}

void bob_call() {
    lock();
    append(globalList);
    unlock();
}
```
void append(List list) {
    lock();
    list.append(...);
    unlock();
}

void caleb_call() {
    append(globalList);
}
```
Common Library Code

```java
void append(List list) {
    lock();
    list.append(...);
    unlock();
}

void caleb_call() {
    append(globalList);
}
```

Everything is thread-safe
void append(List list) {
    lock();
    list.append(...);
    unlock();
}

void caleb_call() {
    append(globalList);
}
• If we know the calling context where synchronization is unnecessary, how do we fix it automatically?
After Code Transformation

```c
void append(List list) {
    if (call_from_caleb()) {
        lock();
        list.append(...);
        unlock();
    }
    else {
        list.append(...);
    }
}
```
void append(List list) {
    if (call_from_caleb()) {
        lock();
        list.append(...);
        unlock();
    }
    else {
        list.append(...);
    }
}
Also Useful For ...

Better memory layout

Enhanced Debugging

Anomaly detection
However...

• For the current state-of-the-art approach, precise calling context checking could incur:
  • > 8x slowdown when querying at every call site.

Why this slow? What’s the real problem behind it?
Traditional Approach PCCE

- Use a single integer (cc) to encode all contexts
- Assign a unique number to each static context
  - Do integer addition and subtraction on call and call return
• Calling context: AJ
• cc: 0
• Calling context: AJG
• cc: 0 + 2 = 2
- Calling context: AJGI
- $cc: 2 + 4 = 6$
Problem 1

Massive amount of distinct static calling context for large code base

Linux kernel =>
Problem 1

Massive amount of distinct static calling context for large code base

Linux kernel =>

Contexts grow exponentially
Problem 1

Massive amount of distinct static calling context for large code base

Linux kernel =>

Contexts grow exponentially

Statically encoding this would overflow 64 bits!
PCCE Deals With Cycles

- Push tuple \(<\text{current } cc, \text{GJ}>\) onto a stack
- Reset \(cc\)
Context: AJGI

• Context: AJGI

Diagram:
A → B → D → E
A → J → G → F
A → I

Back Edge

cc: 6
• Context: AJGIJ
Problem 2

Need to save the entire acyclic context on each back edge.

• Too much redundant leads to inefficient querying
Problem 2

Need to save the entire acyclic context on each back edge.

• Too much redundant leads to inefficient querying

Redundancy accumulates
Two Identified Problems

• Current approach
  • > 8x slowdown when querying at every call site.

• Problem 1
  • Unscalable encoding for the massive amount of static calling contexts.

• Problem 2
  • Inefficient encoding for infinite amount of dynamic calling contexts.
Valence Solved Them All

• The most compact scalable precise calling context encoding

• Compared to the current state-of-the-art approach for SPEC CPU2017 benchmarks. On average, Valence achieves
  • > 60% space overhead reduction (from 4.3 to 1.6 64-bit words) for storing calling contexts.
  • > 70% time overhead reduction for querying calling contexts.
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
• Conclusion
Solution To Scalability Problem

• Encode the context in a logical statically-sized bit vector.
  • Naturally scalable

• Instrumentations
  • Before the call: append a value to the bit vector
  • After the call: pop out the value
Valence Acyclic Encoding

- Use a static bit vector to encode all contexts
- Assign a unique bit pattern to each static context
  - Do bits appending and popping on call and call return (logical)

← : append a binary number
Valence Acyclic Encoding

• Calling context: AJ
• cc: (nil)

← : append a binary number
Valence Acyclic Encoding

- Calling context: AJG
- \( cc: 01 \)

\( \leftarrow: \text{append a binary number} \)
Valence Acyclic Encoding

• Calling context: AJGI
• cc: 0110

← : append a binary number
• How do we statically determine what range in the bit vector to update at each call site?
• How to ensure each bit pattern is unique?
• Check out the algorithms in the paper.
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
• Conclusion
Solution To Cycle Problem

• Goal: to reduce encoding sizes and thus improve performance

• A different way to define cycle edges.

• First calculate the strongly-connected components (SCCs) of the call graph.
A Call Graph With Cycles

• Back edges: HE, GJ
A Call Graph With Cycles

- Five SCCs
• SCC EFH and DGJ have edges inside
  • Cycle edges
• Only need to store these edges for cycle encoding
  • Store in a dynamic bit vector
• Cyclic context: EF
• Cyclic context: EFH

cc: EF, FH
• Cyclic context: EFHE
The SCCs form an acyclic graph

Use the previous acyclic encoding scheme
Takeaways

• Encode call graphs without cycles
  • Use a statically-sized bit vector
  • Scale efficiently

• Encode cycles in the call graphs
  • Use a separate dynamically-sized bit vector
  • Space efficient
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
• Conclusion
We Want To Know...

• Acyclic call graph
  • how much more space Valence requires than PCCE?

• Cyclic call graph
  • how much more compact Valence is?
    • A more compact encoding makes querying more efficient (less memory traffic)

• Valence v.s. PCCE
  • Instrumentation overhead
  • Detection overhead
Evaluation

• Configurations
  • Each encoding strategy (PCCE, Valence) is implemented as a LLVM pass.

• Hardware
  • 3.30GHz Intel CPU and 16G DRAM.

• Benchmark
  • SPEC CPU2017 C/C++ Benchmark suite.
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
  • Problem 2 Evaluation (cyclic call graphs)
  • Problem 1 Evaluation (acyclic call graphs)
• Conclusion
# Benchmark Static Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Call Graph Statistics</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Edges</td>
<td>SCCs</td>
<td>Complex SCCs</td>
</tr>
<tr>
<td>gcc</td>
<td>19,011</td>
<td>131,388</td>
<td>17,182</td>
<td>459</td>
</tr>
<tr>
<td>mcf</td>
<td>32</td>
<td>126</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>1,048</td>
<td>22,820</td>
<td>1,040</td>
<td>12</td>
</tr>
<tr>
<td>namd</td>
<td>61</td>
<td>553</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>parest</td>
<td>1,315</td>
<td>15,080</td>
<td>1,234</td>
<td>46</td>
</tr>
<tr>
<td>povray</td>
<td>519</td>
<td>8,258</td>
<td>377</td>
<td>57</td>
</tr>
<tr>
<td>lbm</td>
<td>17</td>
<td>27</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>4,055</td>
<td>22,848</td>
<td>3,700</td>
<td>181</td>
</tr>
<tr>
<td>x264</td>
<td>367</td>
<td>2,318</td>
<td>366</td>
<td>1</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>87</td>
<td>573</td>
<td>87</td>
<td>3</td>
</tr>
<tr>
<td>imagick</td>
<td>915</td>
<td>18,864</td>
<td>807</td>
<td>40</td>
</tr>
<tr>
<td>leela</td>
<td>204</td>
<td>1,013</td>
<td>202</td>
<td>15</td>
</tr>
<tr>
<td>nab</td>
<td>79</td>
<td>730</td>
<td>79</td>
<td>10</td>
</tr>
<tr>
<td>xz</td>
<td>149</td>
<td>359</td>
<td>140</td>
<td>3</td>
</tr>
</tbody>
</table>
# Benchmark Static Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Call Graph Statistics</th>
<th>Cycle Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Edges</td>
</tr>
<tr>
<td>gcc</td>
<td>19,011</td>
<td>131,388</td>
</tr>
<tr>
<td>mcf</td>
<td>32</td>
<td>126</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>1,048</td>
<td>22,820</td>
</tr>
<tr>
<td>namd</td>
<td>61</td>
<td>553</td>
</tr>
<tr>
<td>parest</td>
<td>1,315</td>
<td>15,080</td>
</tr>
<tr>
<td>povray</td>
<td>519</td>
<td>8,258</td>
</tr>
<tr>
<td>lbm</td>
<td>17</td>
<td>27</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>4,055</td>
<td>22,848</td>
</tr>
<tr>
<td>x264</td>
<td>367</td>
<td>2,318</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>87</td>
<td>573</td>
</tr>
<tr>
<td>imagick</td>
<td>915</td>
<td>18,864</td>
</tr>
<tr>
<td>leela</td>
<td>204</td>
<td>1,013</td>
</tr>
<tr>
<td>nab</td>
<td>79</td>
<td>730</td>
</tr>
<tr>
<td>xz</td>
<td>149</td>
<td>359</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Call Graph Statistics</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nodes</td>
</tr>
<tr>
<td>gcc</td>
<td>19,011</td>
<td>131,388</td>
</tr>
<tr>
<td>mcf</td>
<td>32</td>
<td>126</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>1,048</td>
<td>22,820</td>
</tr>
<tr>
<td>namd</td>
<td>61</td>
<td>553</td>
</tr>
<tr>
<td>parest</td>
<td>1,315</td>
<td>15,080</td>
</tr>
<tr>
<td>povray</td>
<td>519</td>
<td>8,258</td>
</tr>
<tr>
<td>lbm</td>
<td>17</td>
<td>27</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>4,055</td>
<td>22,848</td>
</tr>
<tr>
<td>x264</td>
<td>367</td>
<td>2,318</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>87</td>
<td>573</td>
</tr>
<tr>
<td>imagick</td>
<td>915</td>
<td>18,864</td>
</tr>
<tr>
<td>leela</td>
<td>204</td>
<td>1,013</td>
</tr>
<tr>
<td>nab</td>
<td>79</td>
<td>730</td>
</tr>
<tr>
<td>xz</td>
<td>149</td>
<td>359</td>
</tr>
</tbody>
</table>

At most 15 bits to encode a cycle edge
## Cyclic Encoding Cost Estimation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PCCE (bits)</th>
<th>Valence (bits)</th>
<th>Valence/PCCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1E+06</td>
<td>424950</td>
<td>34%</td>
</tr>
<tr>
<td>mcf</td>
<td>6</td>
<td>1</td>
<td>17%</td>
</tr>
<tr>
<td>cactusBSSN</td>
<td>1802</td>
<td>469</td>
<td>26%</td>
</tr>
<tr>
<td>parest</td>
<td>2656</td>
<td>1600</td>
<td>60%</td>
</tr>
<tr>
<td>povray</td>
<td>62913</td>
<td>28560</td>
<td>45%</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>48932</td>
<td>17226</td>
<td>35%</td>
</tr>
<tr>
<td>x264</td>
<td>18</td>
<td>2</td>
<td>11%</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>228</td>
<td>48</td>
<td>21%</td>
</tr>
<tr>
<td>imagick</td>
<td>18144</td>
<td>5240</td>
<td>29%</td>
</tr>
<tr>
<td>leela</td>
<td>378</td>
<td>115</td>
<td>30%</td>
</tr>
<tr>
<td>nab</td>
<td>450</td>
<td>125</td>
<td>28%</td>
</tr>
<tr>
<td>xz</td>
<td>126</td>
<td>90</td>
<td>71%</td>
</tr>
</tbody>
</table>

**Geomean:** 49%
Average execution time for each CC query

Detection Overhead
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
  • Problem 2 Evaluation (cyclic call graphs)
  • Problem 1 Evaluation (acyclic call graphs)
• Conclusion
## Benchmark Static Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Acyclic Encoding Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCCE</td>
</tr>
<tr>
<td>gcc</td>
<td>214</td>
</tr>
<tr>
<td>mcf</td>
<td>6</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>28</td>
</tr>
<tr>
<td>namd</td>
<td>10</td>
</tr>
<tr>
<td>parest</td>
<td>25</td>
</tr>
<tr>
<td>povray</td>
<td>57</td>
</tr>
<tr>
<td>lbm</td>
<td>2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>42</td>
</tr>
<tr>
<td>x264</td>
<td>18</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>15</td>
</tr>
<tr>
<td>imagick</td>
<td>64</td>
</tr>
<tr>
<td>leela</td>
<td>13</td>
</tr>
<tr>
<td>nab</td>
<td>13</td>
</tr>
<tr>
<td>xz</td>
<td>10</td>
</tr>
</tbody>
</table>
# Benchmark Static Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Acyclic Encoding Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCCE</td>
</tr>
<tr>
<td>gcc</td>
<td>214</td>
</tr>
<tr>
<td>mcf</td>
<td>6</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>28</td>
</tr>
<tr>
<td>namd</td>
<td>10</td>
</tr>
<tr>
<td>parest</td>
<td>25</td>
</tr>
<tr>
<td>povray</td>
<td>57</td>
</tr>
<tr>
<td>lbm</td>
<td>2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>42</td>
</tr>
<tr>
<td>x264</td>
<td>18</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>15</td>
</tr>
<tr>
<td>imagick</td>
<td>64</td>
</tr>
<tr>
<td>leela</td>
<td>13</td>
</tr>
<tr>
<td>nab</td>
<td>13</td>
</tr>
<tr>
<td>xz</td>
<td>10</td>
</tr>
</tbody>
</table>

PCCE Cannot operate on 214-bit integer efficiently
### Benchmark Static Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Acyclic Encoding Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCCE</td>
</tr>
<tr>
<td>gcc</td>
<td>214</td>
</tr>
<tr>
<td>mcf</td>
<td>6</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>28</td>
</tr>
<tr>
<td>namd</td>
<td>10</td>
</tr>
<tr>
<td>parest</td>
<td>25</td>
</tr>
<tr>
<td>povray</td>
<td>57</td>
</tr>
<tr>
<td>lbm</td>
<td>2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>42</td>
</tr>
<tr>
<td>x264</td>
<td>18</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>15</td>
</tr>
<tr>
<td>imagick</td>
<td>64</td>
</tr>
<tr>
<td>leela</td>
<td>13</td>
</tr>
<tr>
<td>nab</td>
<td>13</td>
</tr>
<tr>
<td>xz</td>
<td>10</td>
</tr>
</tbody>
</table>

- **Average Acyclic Words:**
  - PCCE: 1.21
  - Valence: 1.29
### Benchmark Static Characteristics

#### Average Acyclic Words

- PCCE: 1.21
- Valence: 1.29

#### Low cost for scalability!

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Acyclic Encoding Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCCE</td>
</tr>
<tr>
<td>gcc</td>
<td>214</td>
</tr>
<tr>
<td>mcf</td>
<td>6</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>28</td>
</tr>
<tr>
<td>namd</td>
<td>10</td>
</tr>
<tr>
<td>parest</td>
<td>25</td>
</tr>
<tr>
<td>povray</td>
<td>57</td>
</tr>
<tr>
<td>lbm</td>
<td>2</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>42</td>
</tr>
<tr>
<td>x264</td>
<td>18</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>15</td>
</tr>
<tr>
<td>imagick</td>
<td>64</td>
</tr>
<tr>
<td>leela</td>
<td>13</td>
</tr>
<tr>
<td>nab</td>
<td>13</td>
</tr>
<tr>
<td>xz</td>
<td>10</td>
</tr>
</tbody>
</table>
Instrumentation Overhead

![Instrumentation overhead graph]

Execution time relative to default

Benchmark

- gcc
- mcf
- cactuBSSN
- namd
- parest
- povray
- ibm
- xalancbmk
- x264
- deepsjeng
- imagick
- leela
- nab
- xz
- geomean

PCCE
Valence
Table of Contents

• Overview
• Encode Acyclic Call Graphs
• Encode Call Graphs With Cycles
• Evaluation
• Conclusion
• We presented Valence: a precise context encoding scheme that is both scalable and low overhead to query.

• Overall, our approach reduces the length of calling context encoding from 4.3 words to 1.6 words on average (> 60% reduction), thereby improving the efficiency of applications that frequently store or query calling contexts.

• See how Valence can enhance some program analysis and software engineering fields.
Thank you! Questions?

Contact info: tz@gatech.edu
Back up slides
DeltaPath: A Scalable Version of PCCE

- Encode context using a list of \(<cc, \text{anchor node}>\) pairs
- Tuple list operation is difficult to implement efficiently
- Still inherits PCCE’s inefficient cycle encoding
DeltaPath: A Scalable Version of PCCE

• Divide the call graph into sub-graphs
  • Each subgraph is encodable with PCCE

• Introduce the notion of “anchor node” as entry point for each subgraph
DeltaPath: A Scalable Version of PCCE

Context integer space 1

Context integer space 2

Anchor: (root of a territory)