A New Idiom Recognition Framework for Exploiting Hardware-Assist Instructions

Dec. 10, 2007

Toshio Nakatani
Team Members

- Motohiro Kawahito
- Hideaki Komatsu
- Takao Moriyama
- Hiroshi Inoue
Hardware accelerators will play a more important role

- CPU frequency will continue to increase, but the pace is slowing down due to power and cooling limitations.
- Software will continue to demand faster processors.
- Future processors will include more H/W accelerators driven by the special hardware instructions, which we call **H/W-assist instructions**.
  - IBM System z (mainframe) has a coprocessor that supports character-translation.
  - Intel processors will have SSE4 to accelerate string and text processing instructions.
Example: searching for a single delimiter

```java
while(true) {
    if (bytes[index] == 13) break;
    index++;
}
```

```assembly
LA R2, 16(bytes, index) // Start address = bytes + index
LA R3, 16(bytes, bytes.length); // End address = bytes + bytes.length
LHI R0, 13 // Search character = 13
SRST R3, R2 // Scans for 13
LR index, R3 // index = R3
```

```
This is a pen.
```

Index = 10
SRST instruction performance on IBM System z 990

Larger numbers are better

Number of characters processed by SRST

million characters / sec

w/ SRST

w/o SRST

x7

Larger numbers are better

w/ SRST

w/o SRST
Example: searching for multiple delimiters

bytes:
This is a pen. \[13\]0

\[\text{index}\]

while(true) {
    b = bytes[index];
    if (b < 16)
        if (b != 10) break;
    index++;
}

// Intermediate language
index = TRT(bytes, index, BooleanTable) // TRT: TRANSLATE AND TEST

16x16 BooleanTable for TRT

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x50</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0x60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x70</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0x90</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0xA0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xB0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0xC0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xD0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0xE0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xF0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

LA R3, BooleanTable // R3 = BooleanTable
LA R1, 16(bytes, index) // R1 = bytes+index
TRT 0(256, R1), 0(R3) // Scans for -128 to 16 except 10
LR index, R1 // index = R1
Why do we need idiom recognition?

- Compilers need to find an idiom, called idiom recognition, to generate the corresponding H/W-assist instruction for using the H/W accelerator.

Example: Idiom of delimiter search

```java
while(true) {
    if (bytes[index] op C) break;
    index++;
}
```

**Single delimiter**

```java
index = SRST(bytes, index, C)
```

**Multiple delimiters**

```java
index = TRT(bytes, index, Table)
```

- `op` will match equality or inequality, such as `==`, `<=`, `!=`, ...
- `C` will match any constant.
Exact pattern matching cannot optimize Programs 1 to 3.

Program 1: (Separated code)

```java
    t = bytes[index];
    while(true) {
        if (t == 13) break;
        index++;
        t = bytes[index];
    }
```

Program 2: (Additional code)

```java
    while(true) {
        t = bytes[index];
        if (t == 13) break;
        index++;
    }
    u = t; // Used after the loop
```

Program 3: (Different order)

```java
    while(true) {
        if (bytes[index++] == 13) break;
    }
```

The case for exact matching:

```java
    while(true) {
        if (bytes[index] == 13) break;
        index++;
    }
```
We can use the SRST instruction for all of these examples

Program 1: (Separated code)
```
t = bytes[index];
while(true) {
    if (t == 13) break;
    index++;
    t = bytes[index];
}
```
```
index = SRST(bytes, index, 13)
```

Program 2: (Additional code)
```
while(true) {
    t = bytes[index];
    if (t == 13) break;
    index++;
}
```
```
t = bytes[index]
```
```
u = t // Used after the loop
```

Program 3: (Different order)
```
while(true) {
    if (bytes[index++] == 13) break;
}
```
```
index = SRST(bytes, index, 13)
```
```
index++
```
```
t = bytes[index]
```
```
u = t // Used after the loop
```
We can use the TRT instruction for all of these examples

Program 4: (Multiple IFs & Separated node)
```java
int t = bytes[index];
while(true) {
    if (t < 16)
        if (t != 10) break;
    index++;
    t = bytes[index];
}
```

Program 5: (Multiple IFs & Different order)
```java
index++;
while(true) {
    index++;
    t = bytes[index];
    if (t < 16)
        if (t != 10) break;
}
```

16x16 Boolean Table for TRT

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x70</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x80</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0xF0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Outline

- Background
- Our approach for idiom recognition
- Experiments on the IBM System z platform
- Summary
Our Approach for Idiom Recognition

- **Step 1:** Find more potential candidates by using a **topological embedding algorithm** [Fu '97]
  
  Computational order is $O(|V_P||E_T| + |E_P|)$
  
  \[ \begin{align*}
    V_P & : \text{Nodes of the idiom graph} \\
    E_P & : \text{Edges of the idiom graph} \\
    E_T & : \text{Edges of the target graph}
  \end{align*} \]

- **Step 2:** **Attempt to transform each candidate to exactly match the idiom** by applying three code transformations
  
  - partial peeling
  - forward code motion
  - replication of store nodes

  We give up if the transformed graph does not match the idiom.
Exact Matching vs. Topological Embedding (Part 1)

- Topological embedding *matches* if there is a path in the target graph corresponding to each edge in the idiom.

**Exact Matching**

<table>
<thead>
<tr>
<th>Idiom</th>
<th>Target Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>a [\downarrow] b [\downarrow] c</td>
<td>a [\downarrow] b [\downarrow] c</td>
</tr>
</tbody>
</table>

**Topological Embedding**

<table>
<thead>
<tr>
<th>Idiom</th>
<th>Target Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>a [\downarrow] b [\downarrow] c</td>
<td>a [\downarrow] b [\downarrow] Y [\downarrow] Z [\downarrow] b [\downarrow] c</td>
</tr>
</tbody>
</table>
Exact Matching vs. Topological Embedding (Part 2)

- Topological embedding *matches* if there is a path in the target graph corresponding to each edge in the idiom.
Our approach can convert these variations

- load from an array
- check it with constants
- increment the index

Separated Node
- Direct Conversions

Multiple IFs
- Additional Node
- Need Graph Transformations

Different Order
Our approach can convert these variations

Different Order

Partial peeling

Forward code motion

Idiom

Additional Node

Replication of store nodes

while (true) {
    if (bytes[index] == 13) break;
    index++;
}

u = t; // Used

while (true) {
    if (bytes[index] == 13) break;
    index++;
}

aS

b

c

S
We can use the SRST instruction for all of these examples

Program 1: (Separated node)
```java
index = SRST(bytes, index, 13)
t = bytes[index];
while(true) {
    if (t == 13) break;
    index++;
    t = bytes[index];
}
```

Program 2: (Additional node)
```java
index = SRST(bytes, index, 13)
t = bytes[index];
if (t == 13) break;
index++;
```

u = t; // Used after the loop

Program 3: (Different order)
```java
index = SRST(bytes, index, 13)
index++
```

Replication of store nodes
Forward code motion
We can use the TRT instruction for all of these examples

```
index = TRT(bytes, index, BooleanTable)
```

Program 4: (Multiple IFs & Separated node)
```
t = bytes[index];
while(true) {
    if (t < 16)
        if (t != 10) break;
    index++;
    t = bytes[index];
}
```

Program 5: (Multiple IFs & Different order)
```
index++
```
```
while(true) {
    index++;
    t = bytes[index];
    if (t < 16)
        if (t != 10) break;
}
```

16x16 BooleanTable for TRT

```
<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 A B C D E F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0x10</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0x70</td>
</tr>
<tr>
<td>0x80</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0xFB</td>
</tr>
<tr>
<td>0xF0</td>
</tr>
</tbody>
</table>
```

Partial peeling
Coverage of exact matching and our approach

- All Programs
- Topological Embedding (Step 1)
- Exact Matching by Transformations (Step 2)
- Exact Matching
Outline

- Background
- Our approach for idiom recognition
- Experiments on the IBM System z platform
- Summary
Experiments on the IBM System z platform

- Environment: System z 990 2084-316, 64-bit, 8 GB RAM, Linux

- Three algorithm variants:
  - **Baseline**: Perform exact pattern matching.
  - **Our approach**: Perform our approach in addition to the baseline
  - **No idiom**: Disable both exact matching and our approach.

- Measured success ratio for converting loops
  - JCK API tests (J2SE) including 3.7 million loops

- Measured performance improvements and compilation time overhead
  - Micro-benchmarks for J2SE class files
  - IBM XML Parser
  - SPECjvm98
  - SPECjbb2000
## Exploited Idioms

<table>
<thead>
<tr>
<th>Idiom Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findbytes</td>
<td>searching for delimiters</td>
</tr>
<tr>
<td>arraytranslate</td>
<td>converting character codes</td>
</tr>
<tr>
<td>countDigits</td>
<td>counting digits of integers</td>
</tr>
<tr>
<td>intToString</td>
<td>converting integers to strings</td>
</tr>
<tr>
<td>memcpy</td>
<td>copying memory</td>
</tr>
<tr>
<td>memset</td>
<td>filling memory</td>
</tr>
<tr>
<td>memcmp</td>
<td>comparing memory</td>
</tr>
</tbody>
</table>
High-level Flow Diagram

- Baseline
- Our Approach

Loop Canonicalization & Loop Versioning

Canonicalize each loop

Exact Matching

Find many candidates

Topological Embedding

Transform to match the idiom

Graph Transformations

Faster Code
Success ratio for converting the JCK API tests (J2SE)

Our Approach

Exact Matching (Baseline)

1.75x improvement

0.0% 5.0% 10.0% 15.0% 20.0% 25.0% 30.0%

16.1% 28.2% 1.0%

100% means 3.7 million loops that the JIT compiler tried to optimize
Performance improvements observed in Micro-Benchmarks and XML Parser

Larger numbers are better

- **Micro-Benchmarks (J2SE)**
  - Java/lang/String.compareTo()
  - Java/io/BufferedReader.readLine()

- **IBM XML Parser**
  - Throughput (Baseline=100%)
  - Average length of delimiter search loops

- **Our approach**
- **No idiom**

**Throughput (Baseline=100%)**
- **Our approach**
- **No idiom**

**Average length of delimiter search loops**
- **Our approach**
- **No idiom**

Larger numbers are better.
Performance improvements observed for SPEC benchmarks

Larger numbers are better

SPECjvm98

SPECjbb2000

Improvement (Baseline=100%)

Warehouses

Larger numbers are better
Summary

- Hardware accelerators will play a more important role.
- We developed a new approach for idiom recognition, which is much more powerful than exact matching:
  - Higher conversion ratio – 1.75x for J2SE libraries using JCK
- Our approach significantly improved performance for XML parser by 64% with small compilation time overhead of 0.28%.
- For future work, we would like to:
  - support more idioms, graph transformations, and architectures
  - investigate other graph representations, such as a program dependence graph (PDG)
Small compilation time overhead

- Our approach consumes only \textbf{0.28\% to 0.37\%} of the total compilation time.

<table>
<thead>
<tr>
<th></th>
<th>XML Parser</th>
<th>SPECjvm98</th>
<th>SPECjbb2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our Approach</td>
<td>0.28%</td>
<td>0.37%</td>
<td>0.28%</td>
</tr>
<tr>
<td>Remaining compilation</td>
<td>99.72%</td>
<td>99.63%</td>
<td>99.72%</td>
</tr>
<tr>
<td>All</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Thank you