Tokyo Research Laboratory

A New Idiom Recognition Framework for Exploiting Hardware-Assist Instructions

Dec. 10, 2007

Toshio Nakatani

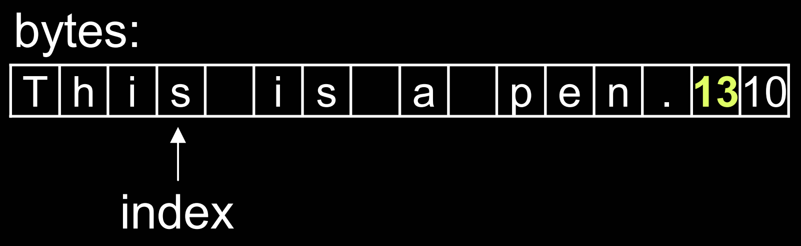
Team Members

- 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

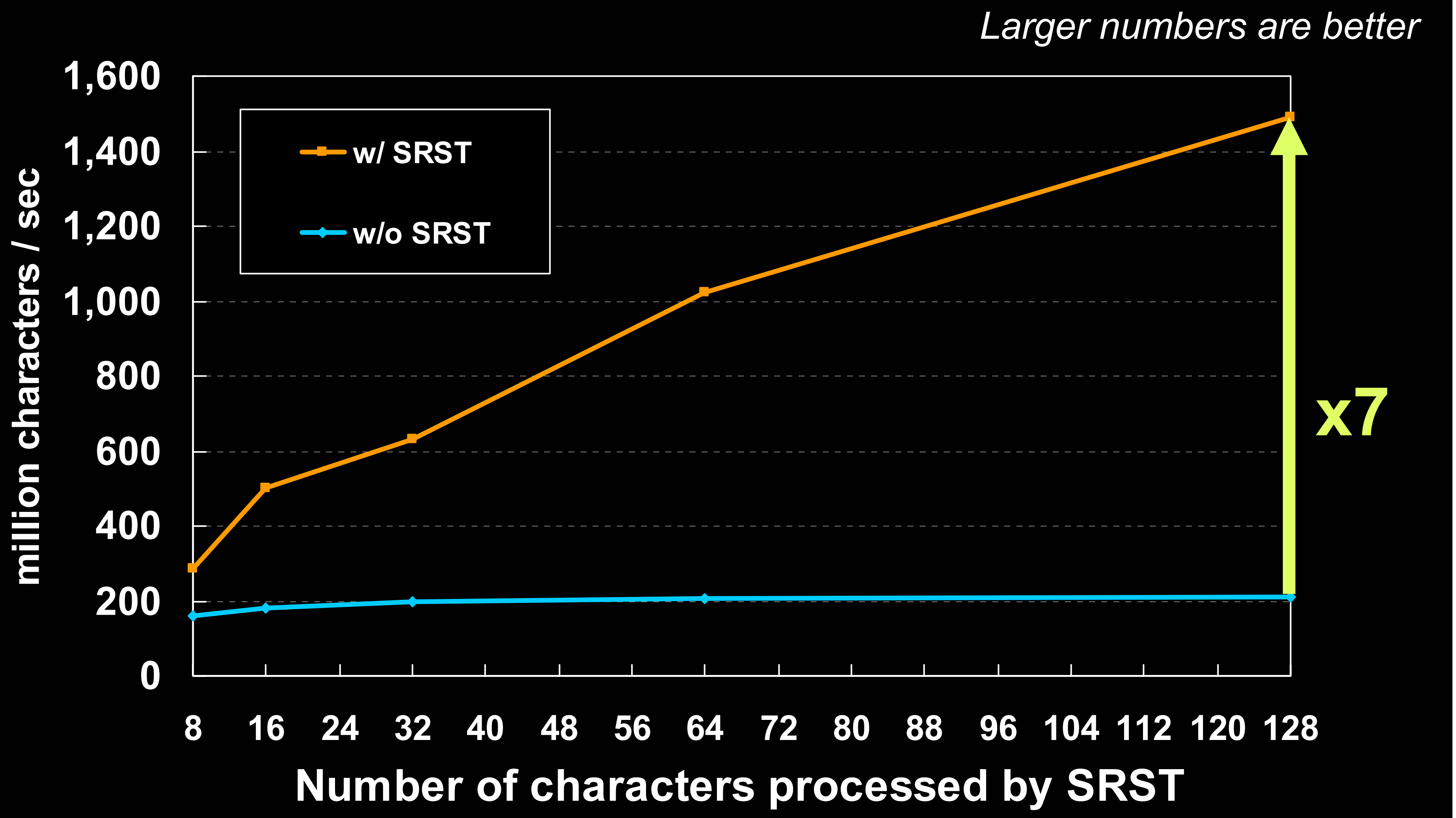


```
while(true) {
    if (bytes[index] == 13) break;
    index++;
}
```

```
// Intermediate language
index = SRST(bytes, index, 13)        // SRST: SEARCH STRING
```

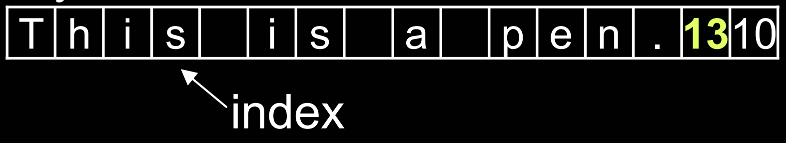
```
LA      R2, 16(bytes, index)          // Start address = bytes + index
LA      R3, 16(bytes, bytes.length);  // End address = bytes + bytes.length
LHI     R0, 13                        // Search character = 13
SRST    R3, R2                        // Scans for 13
LR      index, R3               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

SRST instruction performance on IBM System z 990



Example: searching for multiple delimiters

bytes:



T	h	i	s		i	s		a		p	e	n	.	13	10

index

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

16x16 *BooleanTable* for TRT

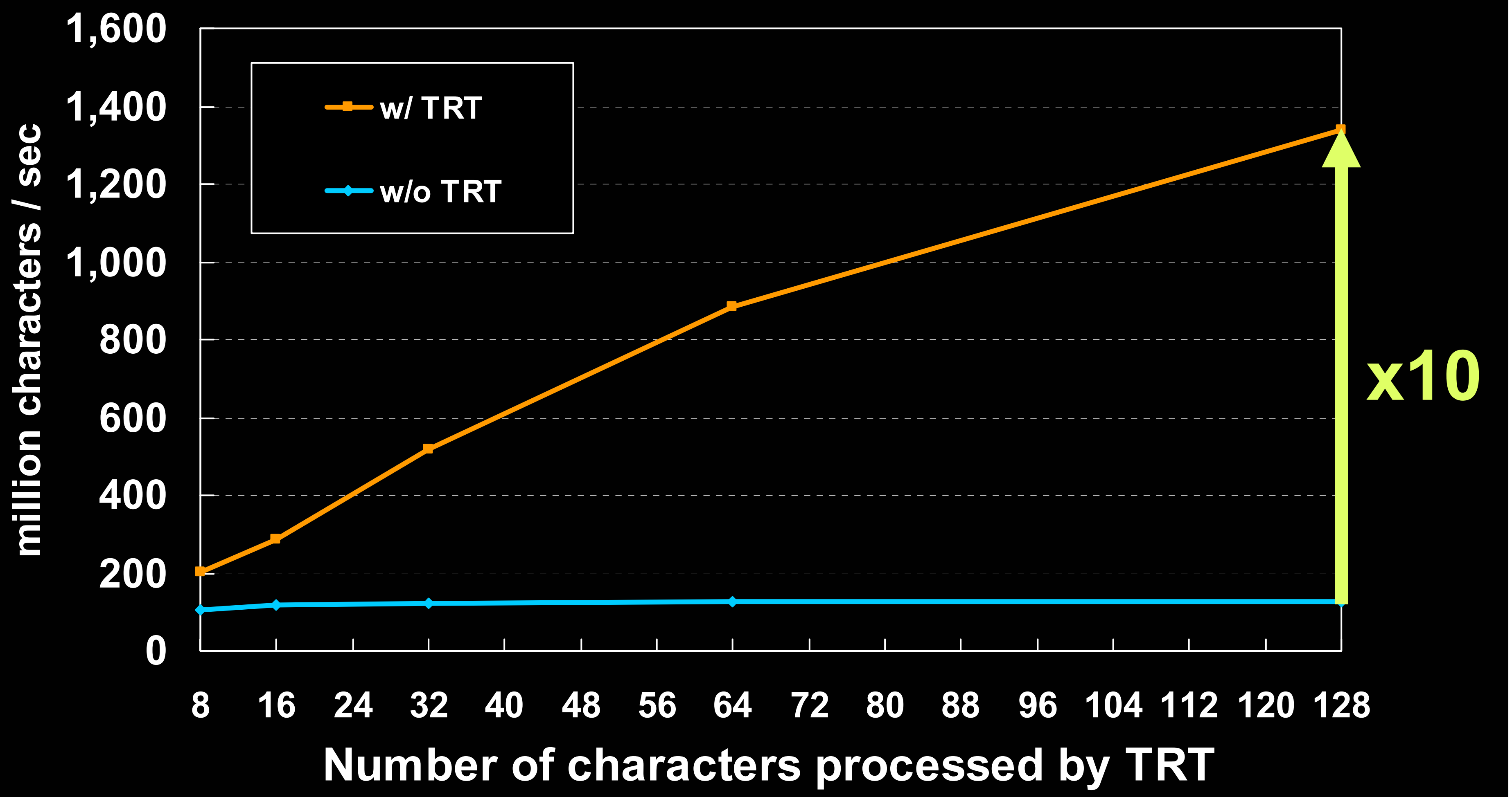
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
0x10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:									:							
0x70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x80	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
:									:							
0xE0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0xF0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

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

```
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
```

TRT instruction performance on IBM System z 990

Larger numbers are better



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

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

op will match equality or inequality, such as “==“, “<=“, “!=“, …

C will match any constant.

Single delimiter

index = SRST(bytes, index, ***C***)

Multiple delimiters

index = TRT(bytes, index, ***Table***)

Exact pattern matching cannot optimize Programs 1 to 3.

Program 1: (Separated code)

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

Program 2: (Additional code)

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

Program 3: (Different order)

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

The case for exact matching:

```
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++;
}
u = t; // Used after the loop
```

→

```
index = SRST(bytes, index, 13)
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++
```

We can use the TRT instruction for all of these examples

16x16 *BooleanTable* for TRT

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
0x10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:									:							
0x70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x80	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
:									:							
0xE0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0xF0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Program 4: (Multiple IFs & Separated node)

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

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

Program 5: (Multiple IFs & Different order)

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

index++
index = TRT(bytes, index, *BooleanTable*)

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 $O(|V_P||E_T| + |E_P|)$

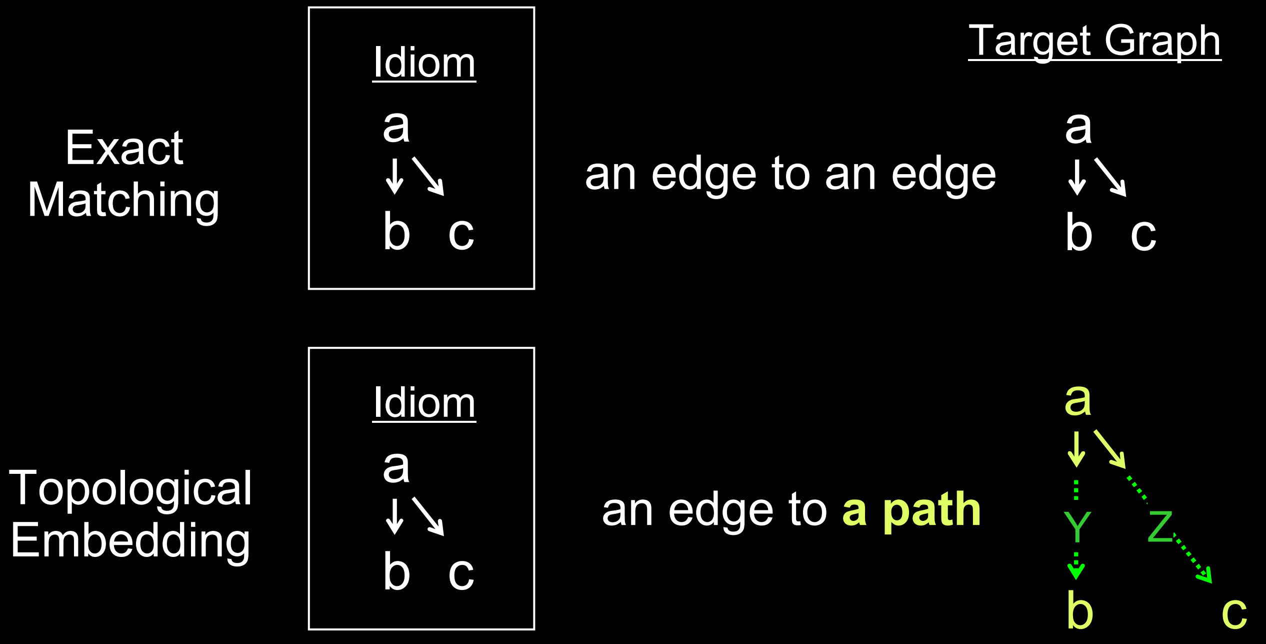
V_P: Nodes of the idiom graph
E_P: Edges of the idiom graph
E_T: Edges of the target graph

- **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



Target Graph

Exact Matching

Idiom

an edge to an edge

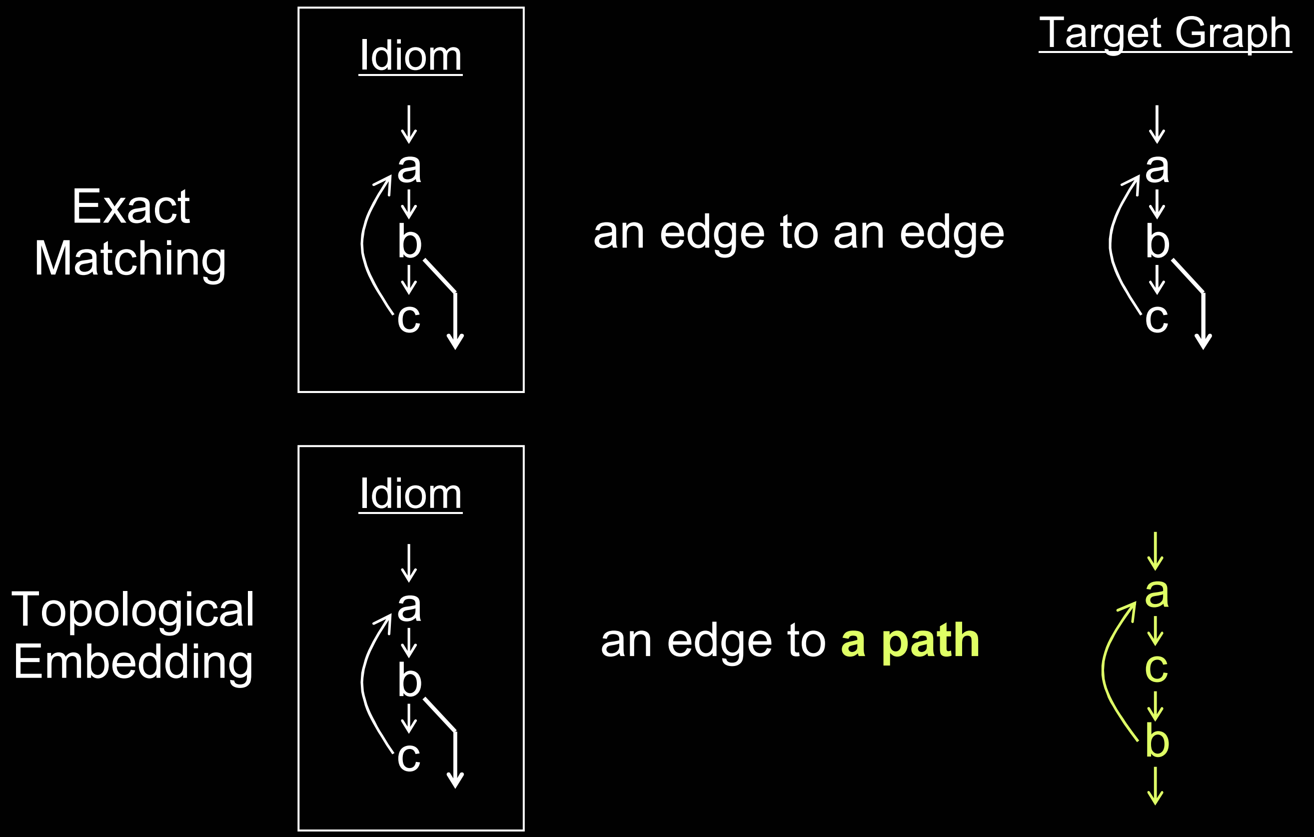
Topological Embedding

Idiom

an edge to **a path**

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



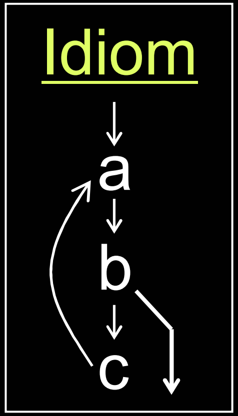
Exact Matching

an edge to an edge

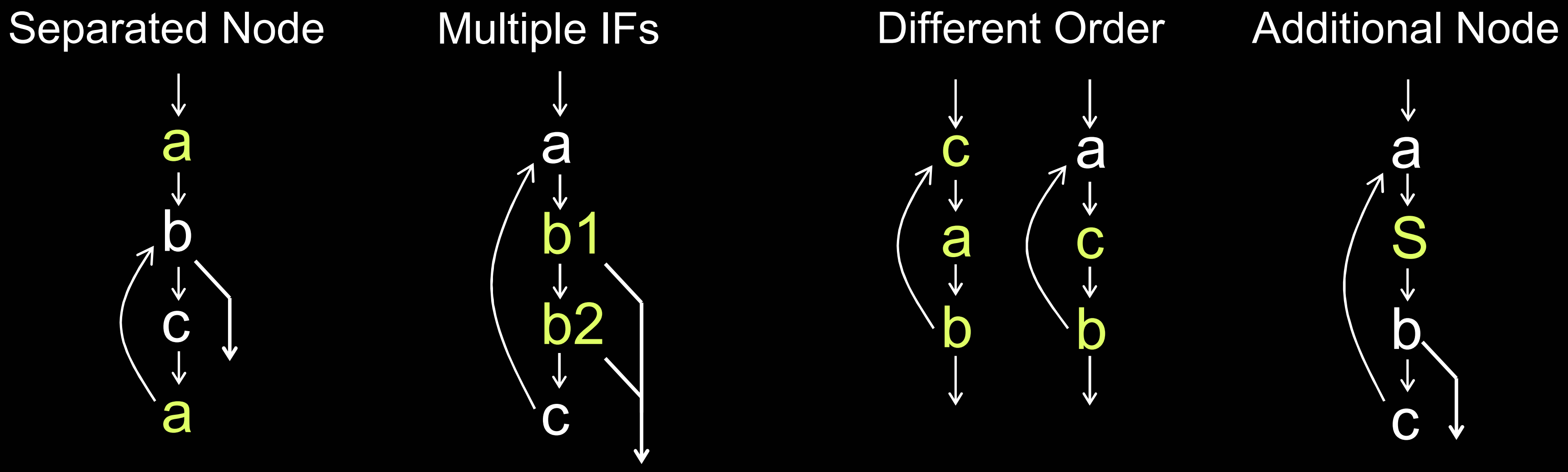
Topological Embedding

an edge to **a path**

Our approach can convert these variations



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

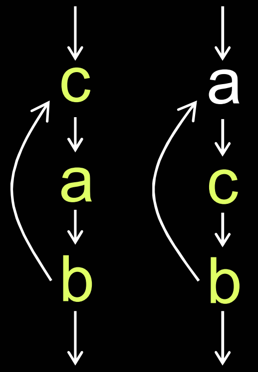


Direct Conversions

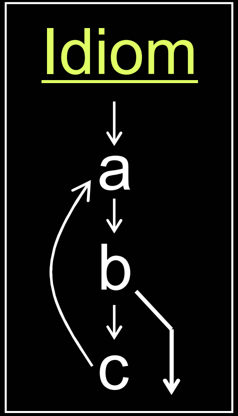
Need Graph Transformations

Our approach can convert these variations

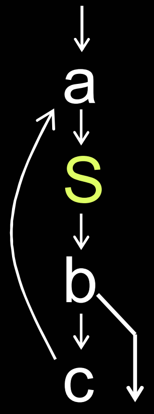
Different Order



Idiom

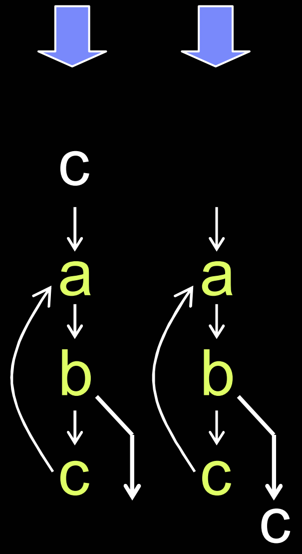


Additional Node



Partial peeling

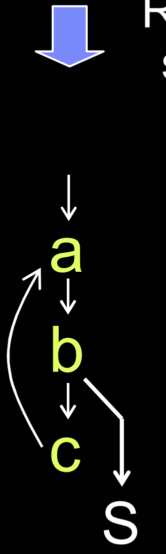
Forward code motion



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

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

Replication of store nodes



We can use the SRST instruction for all of these examples

Program 1: (Separated node)

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

→

```
index = SRST(bytes, index, 13)
```

Program 2: (Additional node)

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

Replication of store nodes

→

```
index = SRST(bytes, index, 13)
t = bytes[index]
u = t // Used after the loop
```

Program 3: (Different order)

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

Forward code motion

→

```
index = SRST(bytes, index, 13)
index++
```

We can use the TRT instruction for all of these examples

16x16 *BooleanTable* for TRT

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
0x10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:								:								
0x70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x80	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
:								:								
0xE0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0xF0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Program 4: (Multiple IFs & Separated node)

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

→

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

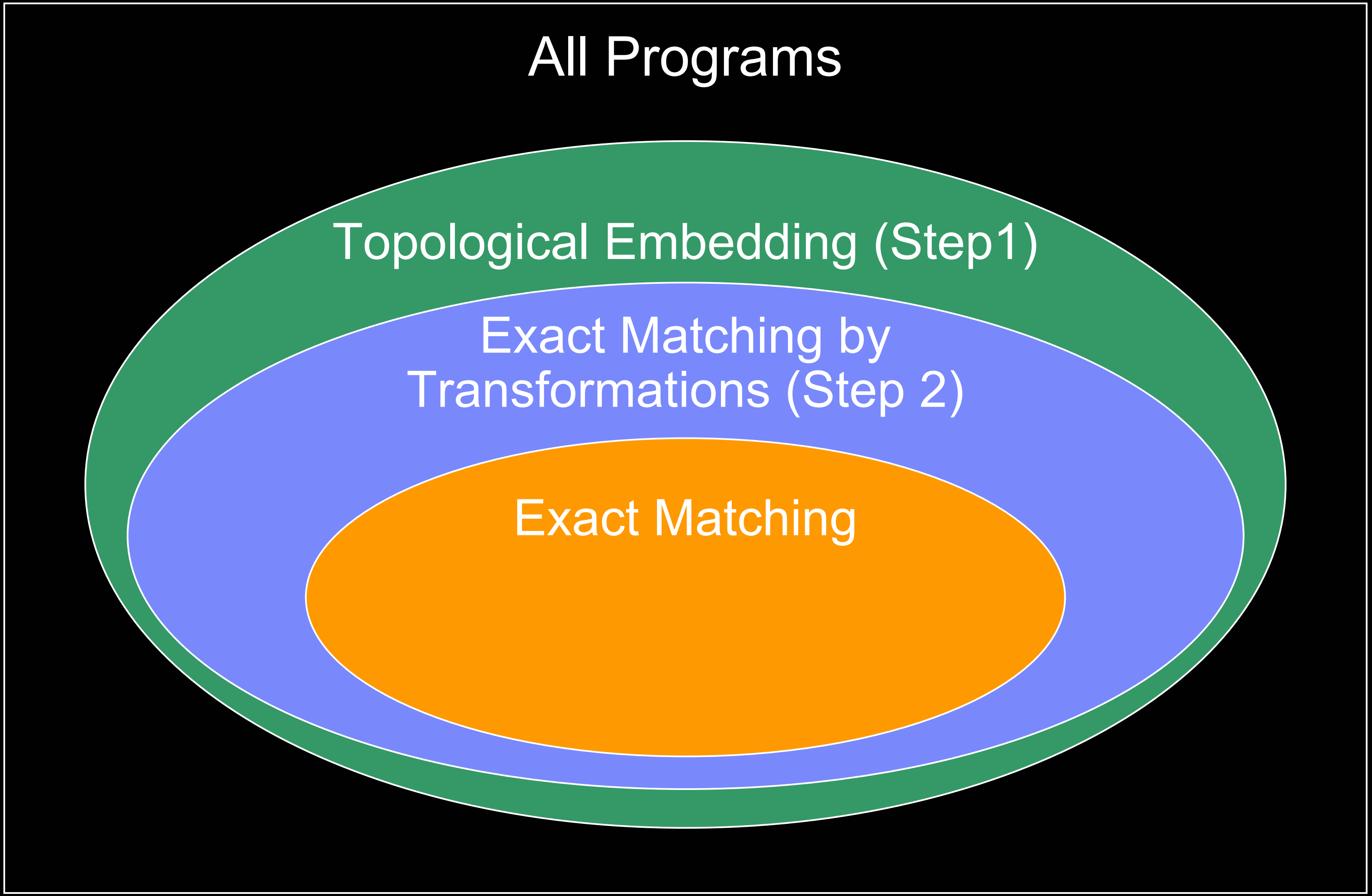
Program 5: (Multiple IFs & Different order)

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

Partial peeling →

index++
index = TRT(bytes, index, *BooleanTable*)

Coverage of exact matching and our approach



All Programs

Topological Embedding (Step1)

Exact Matching by Transformations (Step 2)

Exact Matching

Outline

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

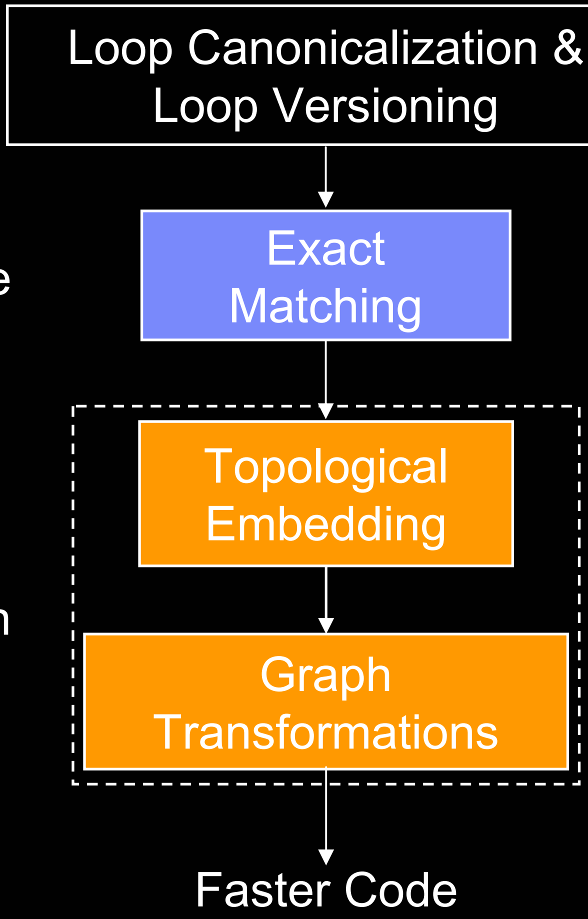
Experiments on the IBM System z platform

- 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

Idiom Name	Description
findbytes	searching for delimiters
arraytranslate	converting character codes
countDigits	counting digits of integers
intToString	converting integers to strings
memcpy	copying memory
memset	filling memory
memcmp	comparing memory

High-level Flow Diagram



Loop Canonicalization & Loop Versioning

Canonicalize each loop

🟦 Baseline

Exact Matching

🟧 Our Approach

Topological Embedding

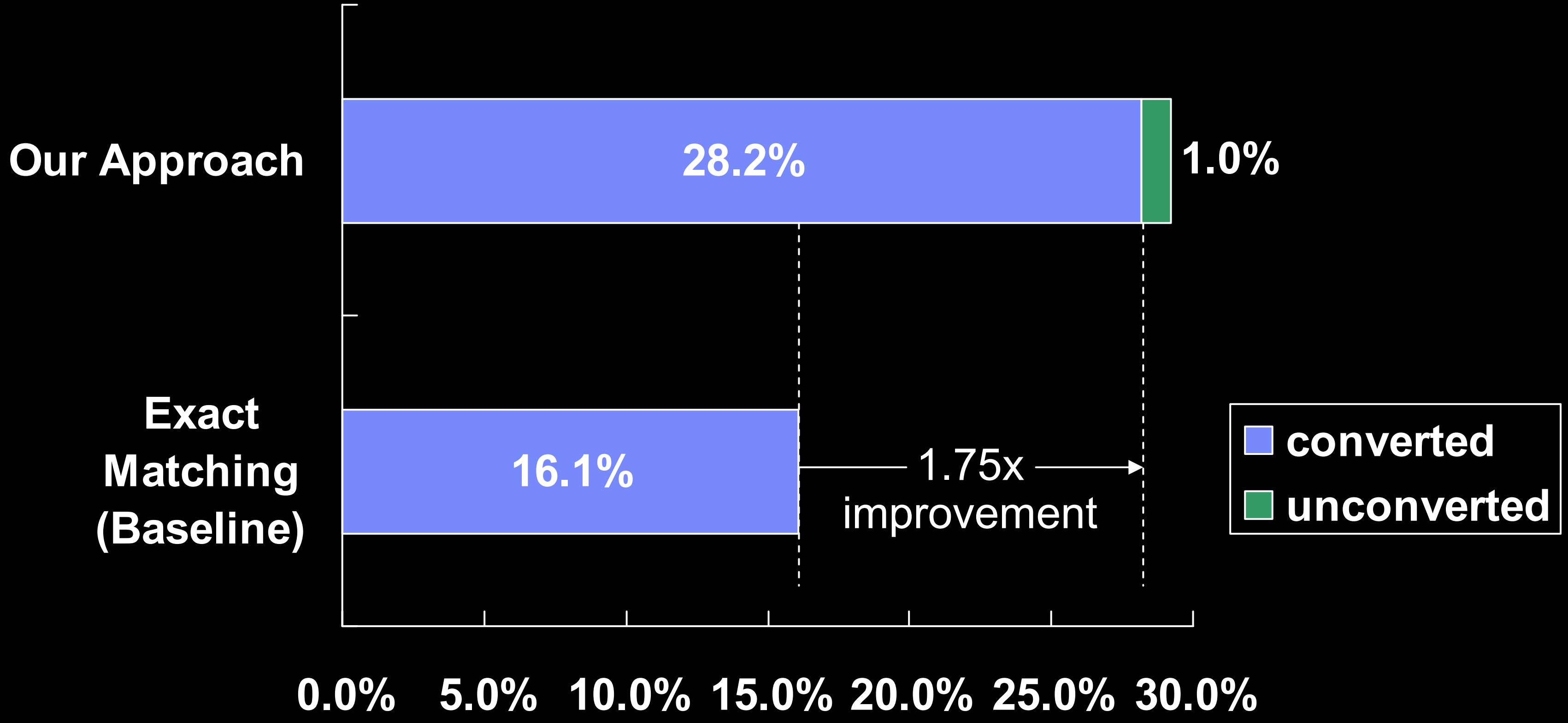
Find many candidates

Graph Transformations

Transform to match the idiom

Faster Code

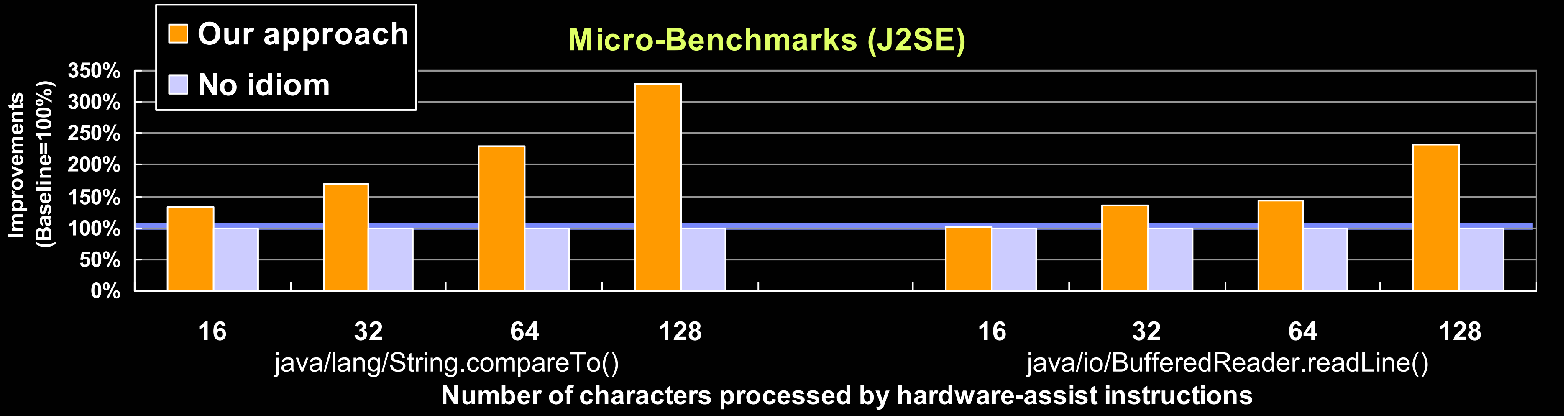
Success ratio for converting the JCK API tests (J2SE)

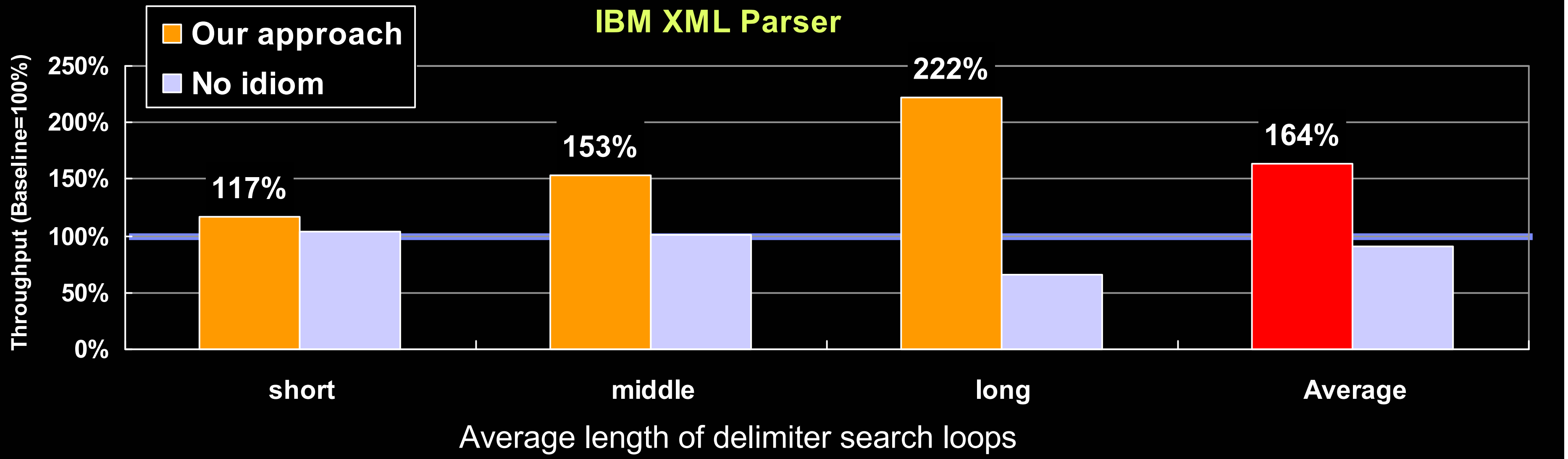


100% means 3.7 million loops that the JIT compiler tried to optimize

Performance improvements observed in Micro-Benchmarks and XML Parser

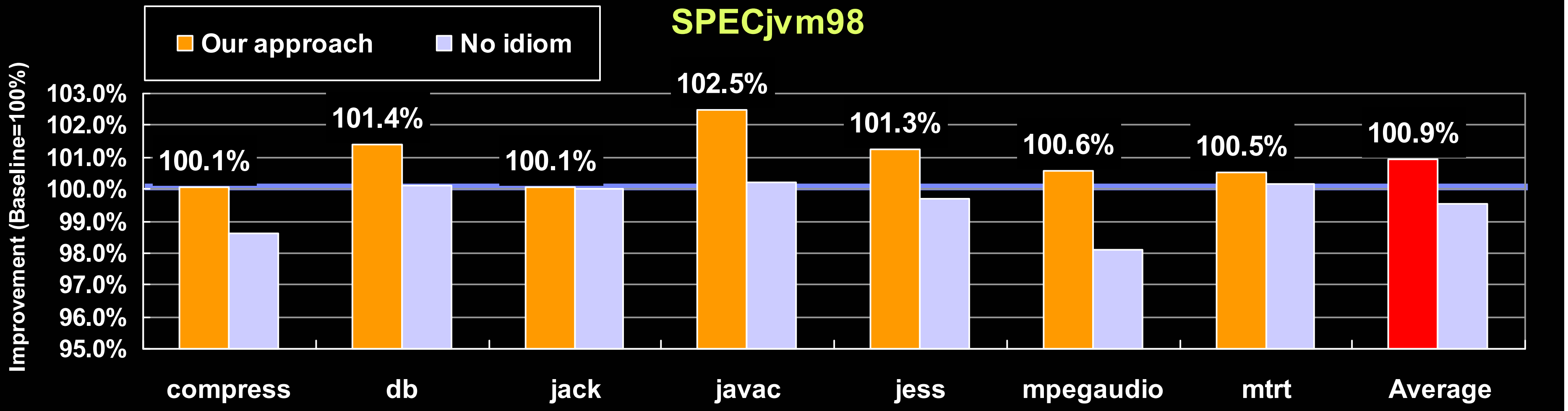
Larger numbers are better

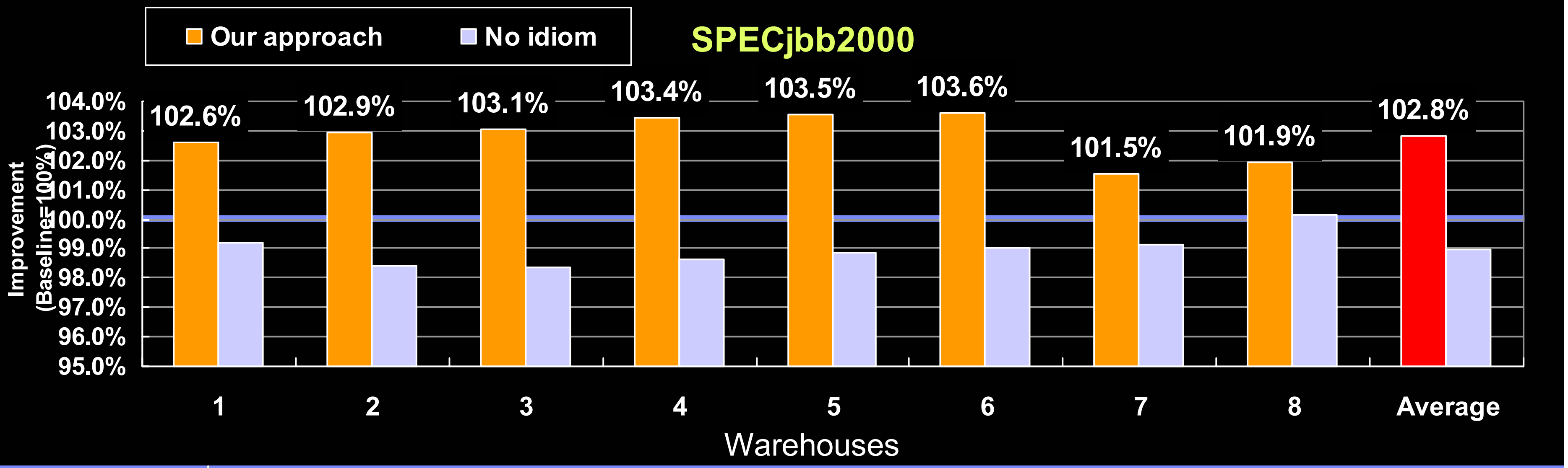




Performance improvements observed for SPEC benchmarks

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 **0.28% to 0.37%** of the total compilation time.

	XML Parser	SPECjvm98	SPECjbb2000
Our Approach	0.28%	0.37%	0.28%
Remaining compilation	99.72%	99.63%	99.72%
All	100%	100%	100%

Thank you