January 12, 2006

RT0639
Computer Science 12 pages

Research Report

A New Idiom Recognition Framework for Exploiting Accelerators

Motohiro Kawahito, Hideaki Komatsu, Takao Moriyama, Hiroshi Inoue, Toshio Nakatani

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A New Idiom Recognition Framework for Exploiting Hardware-Assist Instructions

Motohiro Kawahito Hideaki Komatsu Takao Moriyama Hiroshi Inoue Toshio Nakatani
IBM Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato, Kanagawa, 242-8502, Japan
{ jl25131, komatsu, moriyama, inouehrs, nakatani }@jp.ibm.com

Abstract

Modern processors support hardware-assist instructions (such as TRT and TROT instructions on IBM zSeries) to accelerate certain functions such as delimiter search and character conversion. Such special instructions have often been used in high performance libraries, but they have not been exploited well in optimizing compilers except for some limited cases. We propose a new idiom recognition technique derived from a topological embedding algorithm [4] to detect idiom patterns in the input program more aggressively than in previous approaches. Our approach can detect a pattern even if the code segment does not exactly match the idiom. For example, we can detect a code segment that includes additional code within the idiom pattern. We implemented our new idiom recognition approach based on the Java Just-In-Time (JIT) compiler that is part of the J9 Java Virtual Machine, and we supported several important idioms for special hardware-assist instructions on the IBM zSeries and on some models of the IBM pSeries. To demonstrate the effectiveness of our technique, we performed two experiments. The first one is to see how many more patterns we can detect compared to the previous approach. The second one is to see how much performance improvement we can achieve over the previous approach. For the first experiment, we used the Java Compatibility Kit (JCK) API tests. For the second one we used IBM XML parser, SPECjvm98, and SPCjbb2000. In summary, relative to a baseline implementation using exact pattern matching, our algorithm converted 75% more loops in JCK tests. We also observed significant performance improvement of the XML parser by 64%, of SPECjvm98 by 1%, and of SPECjbb2000 by 2% on average on a z990. Finally, we observed the JIT compilation time increases by only 0.32% to 0.44%.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – compilers, optimization.

General Terms Algorithms, Performance, Design, Experimentation

Keywords idiom recognition, hardware-assist instructions, VMX, topological embedding, Java, JIT

1. Introduction

Idiom recognition is an application of pattern matching to compilers, and it has been used to search for specific patterns in code sequences and to replace them with faster code [22][23][24]. It is also useful for exploiting hardware-assist instructions, which are becoming important as the rate of increase of processor frequencies is declining due to power and cooling limitations.

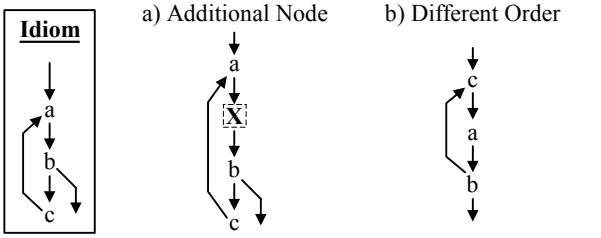


Figure 1. Drawbacks of previous approaches

Traditional approaches for idiom recognition compare the target pattern against the idiom pattern for an exact match [22][23][24]. They fail to recognize it when the idiom pattern does not appear exactly as expected in the target pattern. **Figure 1** shows some drawbacks of previous approaches in more detail. In Figure 1 (a), the idiom cannot be recognized because it has the additional node 'X'. In Figure 1 (b), the idiom cannot be recognized because the order of the nodes is different.

To overcome such limitations, we propose a new idiom recognition technique. Our new approach consists of two phases. For the first phase, using a variant of a topological embedding algorithm **[4]**, we find all of the code segments that contain one of the idiom graphs in a program. We can find candidates even if the code segment does not exactly match the idiom as shown in Figure 1. For the second phase, we attempt to transform the candidate graphs to the idiom graphs using various graph transformation techniques. If we succeed, we convert the modified graph into faster code by generating a hardware-assist instruction. If we fail, we tell the Java programmers or compiler developers about all of the potentially idiomatic candidates for suggesting further performance improvements.

Unlike previous approaches, we can detect all of the graphs in Figure 1 and potentially transform them to the idiom graphs automatically. For example, we can move the node 'X' out of the loop if 'X' has no dependence on the other nodes in the same loop in Figure 1 (a) and replicate the node 'c' outside of the loop to align the loop entry in Figure 1 (b). In addition, our approach can transform them even if 'X' has a dependence on the other nodes. We will explain these transformations in Section 3.4. As a result, our algorithm can convert many more candidates to faster code for the maximum use of hardware-assist instructions than previous approaches.

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

We implemented our new idiom recognition algorithm based on the Java Just-In-Time (JIT) compiler that is part of the J9 Java Virtual Machine. For IBM zSeries [6][26], we supported several important idiom patterns: searching for delimiters, converting character codes, copying memory, filling memory, comparing memory, and converting integers (32-bit and 64-bit) to strings. To demonstrate the effectiveness of our technique, we performed two experiments. The first one examines how many additional patterns we can detect beyond the previous approach. The second one studies the performance improvements we can achieve over the previous approach. For the first experiment, we used the JCK [12] API tests. For the second one, we used IBM XML parser, SPECjvm98, and SPCjbb2000. In summary, relative to a baseline implementation using exact pattern matching, our algorithm converted 75% more loops in the JCK tests. We also observed significant performance improvement of the XML parser by 64%, of SPECjvm98 by 1%, and of SPECjbb2000 by 2% on average on a z990 [6]. Finally, we observed the JIT compilation time increases by only 0.32% to 0.44%.

We also supported the same idioms on IBM pSeries. In the Appendix, we explain how we implemented delimiter searches using vector instructions. We also show some experiments on IBM pSeries.

1.1 Our Contributions

- **A New Idiom Recognition Approach:** Our two-phase idiom recognition algorithm can find variations of idiom patterns in the input program more aggressively than previously known algorithms and automatically transform them into the idiom. As a result, it can take full advantage of hardware-assist instructions.
- **Exploitation of special hardware-assist instructions:** We demonstrate how these instructions can be exploited in commercial applications.

The rest of the paper is organized as follows. Section 2 describes previous work. Section 3 describes our approach. Section 4 covers the performance results obtained in our experiments. Section 5 offers some concluding remarks.

2. PREVIOUS WORK

First, we discuss two common approaches for exploiting hardware accelerators by using special hardware-assist instructions. One is by library calls (or intrinsic inlining), and the other is by idiom recognition. Second, we discuss several known techniques to improve the effectiveness of idiom recognition.

Regarding library calls in Java, the IBM JIT compilers [5][29], for example, can generate optimized code for System.arraycopy(), which is one of the most frequently used intrinsics. They can also generate a special machine instruction corresponding to each method in the Math class library, such as Math.sin(). However, programmers have to explicitly call those libraries to use these instructions.

Regarding idiom recognition, there are two families of techniques. The first family recognizes a specific instruction sequence from an acyclic region to convert it to faster code [21]. This technique is widely used in optimizing compilers. For example, the IBM JIT compiler provides a table of frequently used bytecode sequences as idioms to mitigate the inefficiency in code generation caused by stack semantics [27]. Clark *et al.* proposed an approach [2] that extracts a specific instruction sequence from several basic blocks, but it is still limited to an acyclic region. Superword-Level Parallelism (SLP) is an approach to exploit SIMD instructions [16] for optimizing a loop body. It unrolls a loop beforehand, and then it recognizes vectorizable instructions from a basic block. Thus, it is designed for a loop whose body consists of a single basic block. Shin *et al.* extends SLP in the presence of control flow [25], but it is still limited to an acyclic region.

The second family recognizes a specific instruction sequence including a cycle to parallelize numerical programs [22][23][24]. They compare the instruction sequence of the loop body with each pre-defined idiom. We call this an "**exact match**". However, it often fails to catch idioms when programmers slightly change a program. For example, it cannot recognize the examples in Figure 1. Metzger proposed a combination of idiom recognition and algorithm recognition [19]. This approach first replaces idioms in a graph with a single node that represents the idiom. Next, it parses the resulting graph according to algorithm plans. If a complete match occurs, then the code can be replaced by alternate implementations. This approach also relies on an exact match, and thus it still misses many opportunities.

For improving the effectiveness of idiom recognition, there has been a lot of research [21] into parallelizing or vectorizing loops for numerical programs by applying various loop optimizations, such as loop canonicalization, loop versioning, loop distribution, and loop fusion. For example, our baseline compiler performs loop canonicalization and loop versioning. They are effective to expose specific patterns for idiom recognition. For the case of Java, however, it is rare to find those loops which can be candidates for loop distribution or loop fusion. One reason is that Java programmers tend to use many method calls, which make data dependence analysis difficult for loop transformations. Method inlining mitigates this problem, but we cannot necessarily inline all method calls because of the code expansion problem. Indeed, in our experiments we observed that graph transformations were not able to be performed because the target loops include one or more method calls. Another reason is that the multi-dimensional arrays of Java are allocated as arrays of arrays unlike the dense-array of FORTRAN. Thus, we cannot assume that the length of each array of the first dimension is same. As a result, we can only eliminate exception checks from the innermost loop by applying loop versioning technique. This limits the cases to which loop optimizations can be applied.

It is also known that abstract interpretation techniques [3][17] or symbolic analysis techniques [1] help improve the effectiveness of idiom recognition. Abstract interpretation techniques are fast enough for JIT compilers. For example, we applied an abstract interpretation technique [11] for software prefetching in our JIT compilers. On the other hand, the symbolic analysis techniques are powerful but more time consuming. Thus, our baseline compiler, for example, performs faster optimization techniques based on dataflow analysis, such as induction variable analysis, range analysis, alias analysis, and class/field/array privatization in earlier phases. In addition, it also performs traditional optimizations in advance, such as inlining, copy propagation, dead code elimination, code specialization, exception check elimination, and partial redundancy elimination. These optimizations help find as many candidates as possible.

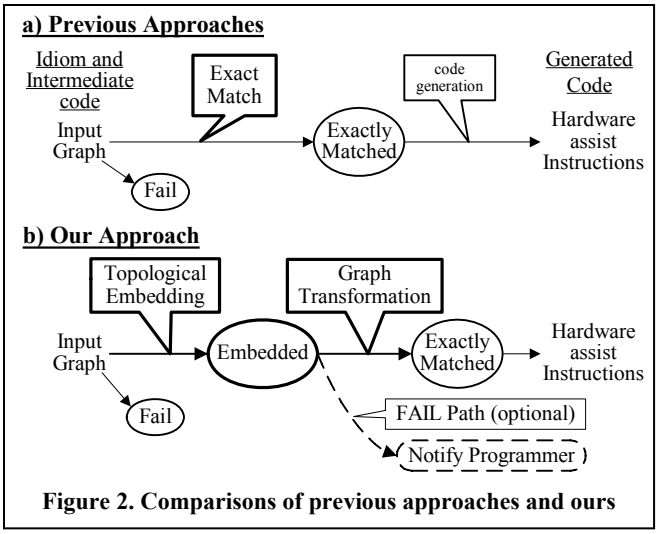


Figure 2. Comparisons of previous approaches and ours

3. OUR APPROACH

Our approach overcomes the problems of the previous approaches as described in Section 2 with a more flexible algorithm to search for code fragments of the patterns for partial graph matching. **Figure 2** shows a comparison of our approach and previous approaches. Our new approach consists of two phases. In the first phase, we find all of the code segments in a program that contain one of the idiom graphs, even if the sequence of the code appears to be different from the idiom. In the second phase, we attempt to transform each code segment into one corresponding to the idiom graph using various code transformation techniques available in the compiler. In addition, we can provide hints if the graph transformations fail. That is, the compiler can tell the Java programmers or compiler developers about all of the potentially idiomatic candidates in order to suggest further performance improvements. When successful, we first transform an input loop to a special node (e.g. memcpy, memset, memcmp) at the intermediate language (IL) level, and then our system generates a code sequence corresponding to the node for each platform.

Figure 3 shows a flow diagram of our algorithm. First, we transform each loop to our graph representation. Next, we apply two pre-filters described in Section 3.2: (1) exclude those idioms which are unlikely to be matched and (2) exclude some rarely iterated loops based on the runtime profile information and depending on the idiom. These reduce the number of candidate idiom graphs to search for with the topological embedding algorithm. Next, we search for each idiom by applying our algorithm described in Section 3.3. Next, we attempt to match the idiom by applying the graph transformations described in Section 3.4. If the transformed graph matches the idiom, we can replace it with a special node corresponding to the idiom to generate a faster code sequence. For our algorithm, we can easily support a new idiom by adding an idiom graph and the corresponding code generation pattern without modifying the algorithm.

Table 1 shows the supported idioms. These idioms are architecture independent. We use special hardware instructions both on IBM zSeries and IBM pSeries. We will describe more details in Section 3.6 and the Appendix.

Section 3.1 describes the advantages of a topological embedding algorithm and our modifications to the original algorithm. Section

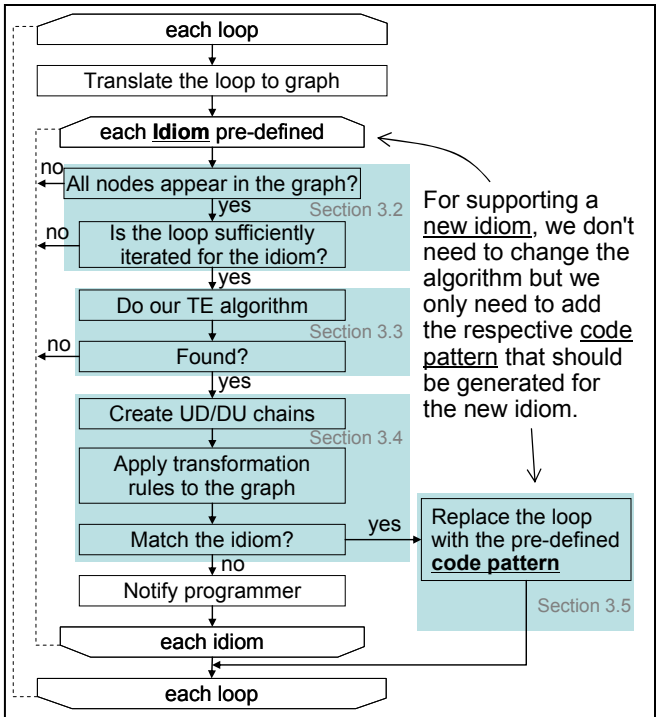


Figure 3. Flow diagram of our algorithm

3.2 describes the pre-filters, which reduce the number of candidate idiom graphs. Section 3.3 describes the first phase of our algorithm, which finds all of the code segments that contain one of the idiom graphs in a program. Section 3.4 describes the second phase of our algorithm, which attempts to transform the candidate graphs into the idiom graph. Section 3.5 describes an analysis required for generating the code pattern. Section 3.6 describes generated code for the IBM zSeries.

Table 1. Supported Idioms

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

3.1 Advantages of Topological Embedding

In this section, we briefly describe the advantages of the topological embedding (TE) algorithm [4]. We consider ordered labeled directed graph pattern matching and topological embedding problems, where an ordered labeled directed graph is a directed graph in which every node is associated with a label, and the left-to-right order of siblings is significant. For exact pattern matching, a directed graph P matches a directed graph T if there is a mapping f from the nodes in P to the nodes in T such that f preserves label, degree for internal nodes in P, and the parent relationship. TE relaxes the restriction on preserving the parent relationship by requiring f to preserve the ancestor relationship, i.e., for each node α in P, the ith child of α from the left can be mapped to either the

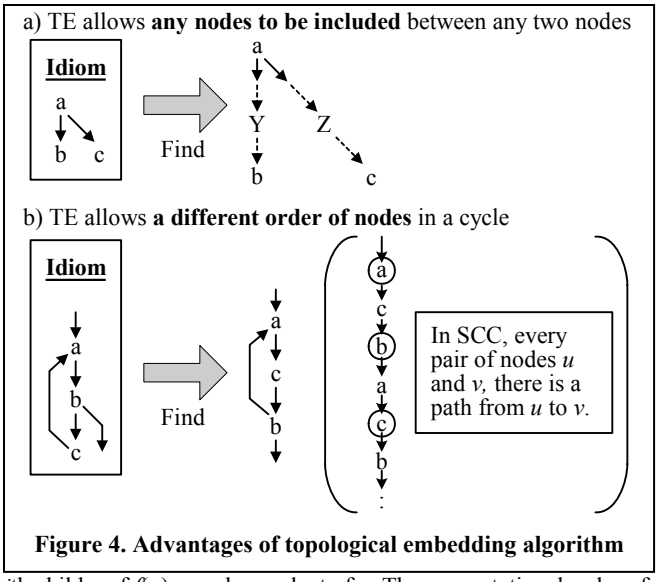


Figure 4. Advantages of topological embedding algorithm

*i*th child *c* of $f(\alpha)$ or a descendant of *c*. The computational order of the TE algorithm is $O(|V_P|\ |E_T| + |E_P|)$ [4]. Here, *V* and *E* are nodes and edges, respectively.

Figure 4 shows two advantages of the TE algorithm compared to exact matching. One is that it allows any nodes to be included between any two nodes of the idiom graph as shown in Figure 4(a). The other is that it allows a different order of nodes in a cycle. In a strongly connected component (SCC), for every pair of nodes *u* and *v*, there is a path from *u* to *v* and a path from *v* to *u*. In Figure 4(b), the idiom is a loop whose body consists of nodes "*abc*". When an input program is "*acb*", unfolded infinite tree is "*acbacb*…" As you can see, there is a path from *a* to *b*, *b* to *c*, and *c* to *a*. As a result, we can recognize this input program as a candidate.

3.1.1 Our modifications to the TE algorithm

We modified the original TE algorithm as follows:

- As we mentioned, the original TE assumes that the left-to-right order of siblings is significant. However, this limits the ability to detect commutative operations. We check all of the operand patterns for commutative operations, such as additions, multiplications, and so on.
- We use a wild-card node, which matches several opcodes (labels) in a target graph. We use it to find multiple if statements and variables.

3.2 Pre-filters

There are two reasons to perform pre-filtering. Controlling compilation time increases is a very critical problem for JIT compilers. In addition, using a special hardware-assist instruction has one disadvantage. While it can greatly improve performance for a sufficiently long input sequence, it could degrade performance for a very short input sequence because of the startup costs.

In Figure 3, there are two pre-filters before our TE algorithm. The first pre-filter checks that all nodes in the idiom appear in the target graph. For each idiom and the graph, we create a bit-vector whose bits represent the opcodes. For example, if the graph includes a byte array load (baload), then the corresponding bit of the

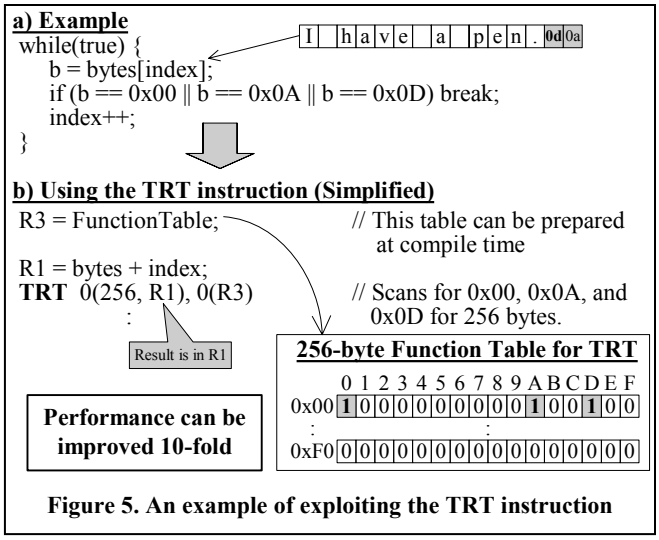


Figure 5. An example of exploiting the TRT instruction

bit-vector is on. We compare the bit-vector of each idiom graph with that of the target graph to exclude those idioms which cannot be matched.

The second pre-filter excludes rarely iterated loops if the hardware-assist instruction corresponding to the idiom has a large startup cost. For the TRT instruction, we cannot estimate the actual search length, because it depends on the content of each input array. For predicting the search length, we use runtime profile information. We compute the ratio of the frequency of the inside block over that of the outside block for each loop and exclude the rarely iterated loops from the candidates.

Note that the startup costs for each special hardware-assist instruction varies on the processor. The use of profiling in a JIT environment allows our algorithm to be tuned to the platforms and specific processor models on which the application is running.

3.3 Finding the Candidate Graphs

In this section, we describe how we find the candidate graphs by using our topological embedding algorithm. There are five steps in our algorithm:

1. Translate input intermediate language (IL) code into our graph representation, which consists of:
   - Nodes: IL nodes (or wildcard nodes for idiom graphs)
   - Type 1 Edges: Operand edges
   - Type 2 Edges: Control flow edges
2. Find candidates among the leaf nodes in the abstract syntax trees [21] (Type 1 Edges) created in Step 1
3. Use the TE algorithm to walk through Type 1 Edges from every leaf to the root and find the candidate nodes. In this step, we appropriately check commutative operands.
4. Use the TE algorithm to walk through Type 2 Edges from the exit to the entry while checking the relationships between all the ancestors and descendants for each node found in Step 3
5. Extract the smallest sub-graph that includes all of the nodes in the current idiom candidate

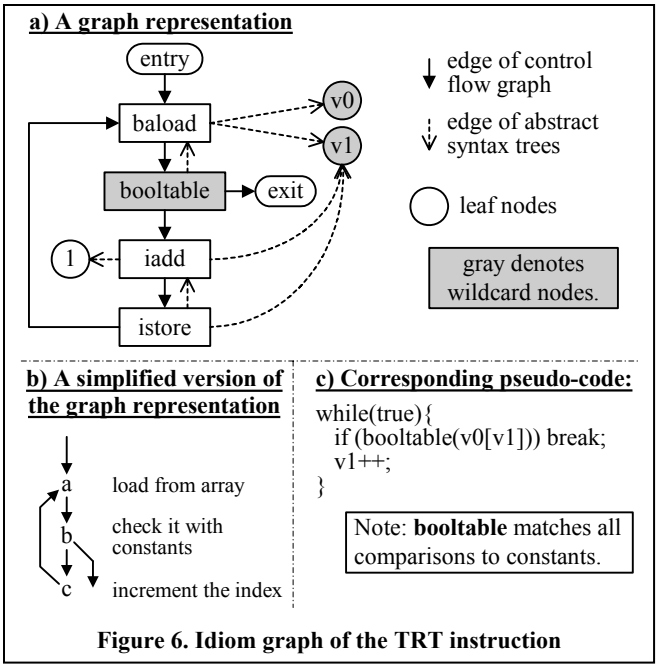


Figure 6. Idiom graph of the TRT instruction

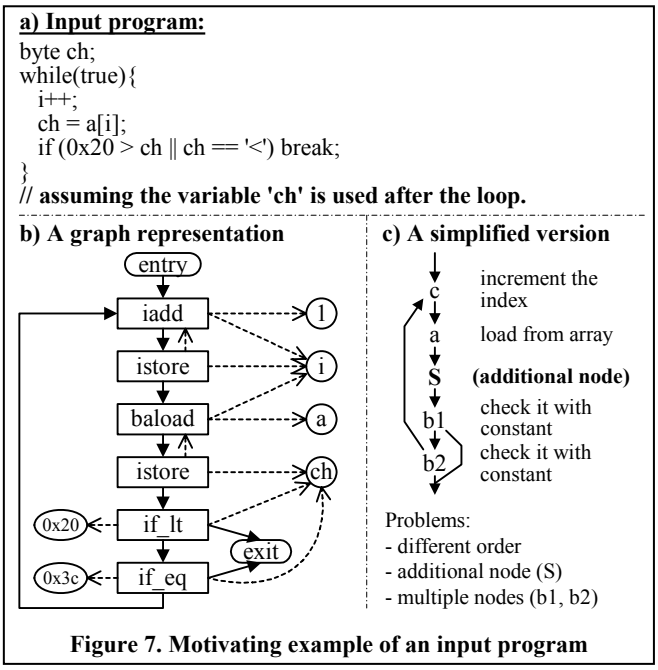


Figure 7. Motivating example of an input program

3.3.1 Example: TRT instruction

We clarify our algorithm for finding the candidate graphs using examples. The TRANSLATE AND TEST (TRT) instruction [10] on IBM zSeries can be used to search for characters with special meanings in a byte-array. To indicate which characters have a special meaning, we need to prepare a function table as a 256-byte array. In the table, non-zero values signify special characters. In this paper, we call them "**delimiters**".

Figure 5 shows an example of exploiting the TRT instruction. The example in Figure 5(a) searches for 0x00, 0x0A, and 0x0D in a byte array. In this example, the result address must point to 0x0D (carriage return). We can convert this example code to faster code using the TRT instruction as shown in Figure 5(b). We need to prepare the function table by setting non-zero values for the table entries of the delimiters. By using this conversion, performance can be improved 10-fold, depending on the search length. Such loops are often found in text processing programs such as XML parsers. In actual programs, these if-conditions can vary according to what characters are assumed to act as delimiters, and thus an exact match is difficult to find using such loops.

Figure 6(a) shows an idiom graph for the TRT instruction. It consists of nodes and two kinds of edges. Here, "baload" means "load byte from array" [18]. In the graph, there are two kinds of wildcard nodes, variables and the special node "**booltable**". Variable nodes in the idiom match all variables in the target graph. The node booltable matches all comparisons of the child and any constants. We used the node booltable not only for the TRT instruction but also for other idioms, such as character conversions. We can also use the simplified graph representation as shown in Figure 6(b) for later explanations. Figure 6(c) shows the pseudo-code corresponding to the idiom.

Figure 7 shows a motivating example of an input program. Step 1 of the algorithm described in Section 3.3 translates the input program (a) to the graph representation (b). In Step 2, we find candidate leaf nodes by analyzing their ancestors. For example, since the parents of the variable 'v1' in Figure 6(a) are 'baload', 'iadd', and 'istore', the variable 'i' in Figure 7(b) becomes a candidate for 'v1'.

In this example, there are three difficulties for previous approaches trying to detect a candidate: (1) The order of the nodes in the loop body is different from that of the idiom. (2) There is an additional node "store into the variable 'ch'". (3) There are multiple if statements. As we mentioned in Figure 4, we solve the first and the second problems by using the topological embedding algorithm. In addition, we solve the third problem by using a wildcard node, which can match two if statements. After performing Steps 3 to 5 in Section 3.3, the result is the successful detection of an optimization candidate.

3.4 Graph Transformations

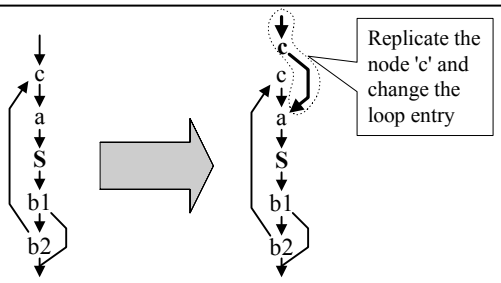
Because our first phase in Section 3.3 may find graphs whose program patterns are different from the idioms, we need to transform the candidate graphs to the idiom graphs. Before graph transformations, we create UD/DU chains to analyze data dependences of the variables. We have implemented three graph transformations: (1) partial peeling of a loop body, (2) replicating store nodes outside of loops, and (3) code motion. We prepared a list of transformations for each idiom. This phase calls each transformation in the list and checks whether the modified target graph matches the idiom graph.

In this phase, we do not directly modify the input intermediate language code yet because it is difficult to undo those transformations, and because unneeded transformations may degrade performance. For example, if store nodes are replicated by the technique of Section 3.4.2 but the loop cannot be transformed, then it will degrade performance. Instead, we modify our internal graph and store some compensation code for the entry and each exit point. If the idiom recognition finally decides that the loop can be transformed, we will generate compensation code and the special IL node.

P: Pattern graph, T: Target graph

pTop = the next node of the entry of *P*;
for (each *t* from the entry to the exit in *T*){
  if (*t* corresponds to *pTop*) return; // already aligned
  if (*t* is in a cycle) break;
}
lastNode = *firstNode* = *t*;
for (each *t* from *firstNode* to the exit in *T*){
  if (*t* corresponds to *pTop*) break;
  lastNode = *t*;
}
idealLoopEntry = *t*;
regionR = from *firstNode* to *lastNode* in *T*;
for (each *t* in *regionR*)
  if (there is a parent of *t* outside of *regionR*) return;
Add every node in *regionR* to the compensation block of the loop entry;
Modify the loop entry to *idealLoopEntry*;

Figure 8. Algorithm of partial peeling



Modified program:

```
byte ch;
i++;                          // This expression is replicated.
while(true){
  ch = a[i];                  // Change the loop entry
  if (0x20 > ch || ch == '<') break;
  i++;
}
```

Figure 9. After performing partial peeling

3.4.1 Partial peeling of a loop body

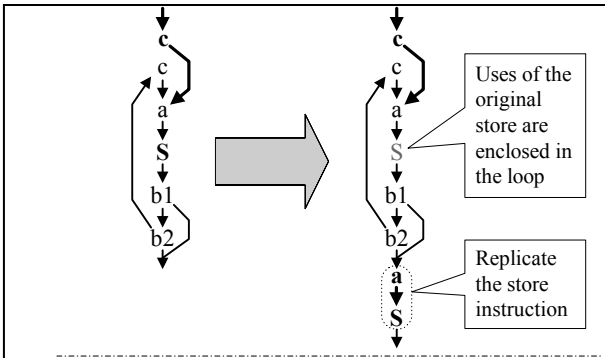
The purpose of this transformation is to align the loop entry. **Figure 8** shows our algorithm. This transformation replicates the region from the loop entry to the ideal loop entry outside of the loop, and then it modifies the entry point to match the ideal one.

Figure 9 shows the transformation result of Figure 7. In the example of Figure 7(c), the loop entry is the node 'c', but it should be the node 'a'. Thus, this transformation replicates the node 'c' outside of the loop and changes the loop entry to the node 'a'.

3.4.2 Replicating store nodes

The example in Figure 7 includes an additional node, a store to the variable 'ch'. As we mentioned in Figure 7, we are assuming that the variable 'ch' will be used after the loop. In this case, we cannot ignore the store node. Previous approaches give up on this case because the expression "ch = a[i]" has data dependences for the succeeding if-statement.

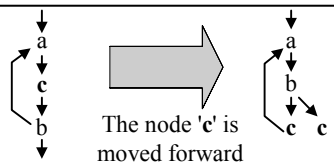
This transformation replicates the store node outside of the loop. By using this transformation, all uses of the original stores are enclosed within the loop. In other words, the variable 'ch' in the loop is now used only to pass the array value to some nodes in the loop. Therefore, we can ignore the original stores for idiom recognition.



Modified program:

```
byte ch;
i++;                          // This expression is replicated.
while(true){
  ch = a[i];                  // Change the loop entry
  if (0x20 > ch || ch == '<') break;
  i++;
}
ch = a[i];                    // This expression is replicated.
```

Figure 10. After performing store replication



a) Original program:

```
while(true){
  if (a[i++] == 0x20) break;
}
```

b) Modified program:

```
while(true){
  if (a[i] == 0x20) break;
  i++;
}
i++;
```

Figure 11. Example of code motion

This transformation is similar to partial dead code elimination (PDE) [14]. Unlike the PDE technique, it moves store nodes beyond their uses. We assure that neither the variable of the store node nor the right-hand-side (RHS) expression is changed between the original point and the loop exit point. For moving memory access expressions in Java, Kawahito *et al.* discussed possible barriers and alias analysis in [13].

Figure 10 shows the transformation result of Figure 9. In this example, because neither the variable 'ch' nor the RHS expression 'a[i]' is changed between these two positions, we can replicate it outside of the loop. Through this replication, we can transform the loop to faster code using the TRT instruction. As you can see in Figure 10, the store replication itself worsens the performance of the program. Thus, we need to cancel such a transformation if the loop cannot be transformed to the faster code.

3.4.3 Code motion

This transformation is similar to the previous transformation "replicating store nodes", but the purpose of this transformation is to reorder the nodes to match the idiom graph. To date, we implemented only forward code motion, because the partial peeling

a) Input program:

```
byte ch;
while(true){
  i++;
  ch = a[i];
  if (0x20 > ch || ch == '<') break;
}
// assuming the variable 'ch' is used after the loop.
```

b) Modified program:

```
byte ch;
i++;                          // This expression is replicated.
while(true){
  ch = a[i];                  // Change the loop entry
  if (0x20 > ch || ch == '<') break;
  i++;
}
ch = a[i];                    // This expression is replicated.
```

c) After Optimization

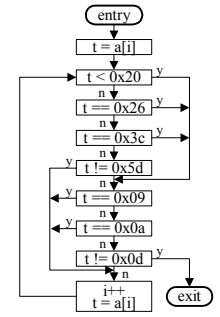
```
byte ch;
i++;                          // This expression is replicated.
i = TRT(a, i, FunctionTable);
ch = a[i];                    // This expression is replicated.
```

Figure 12. Transitions of the input program in Figure 7

a) Input program

```
t = a[i];
while (true) {
  if ((0x20 <= t && t != 0x26 && t != 0x3C && t != 0x5D)
      || t == 0x09 || t == 0x0A || t == 0x0D)
      i++;
  else
      break;
  t = a[i];
}
```

b) Graph representation



c) Function Table for TRT (256 bytes)

	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**	0	0	**1**	**1**	0	**1**	**1**
0x10	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0x20	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
0x30	0	0	0	0	0	0	0	0	0	0	0	0	**1**	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**
:									:							
0xF0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

('1' means exit conditions)

Figure 13. Complex if-statements

covers some of the transformations required for backward code motion.

In **Figure 11**, we see that the node 'c' is not placed at the ideal location but is between the nodes 'a' and 'b'. Note that there is control dependence between nodes 'c' and 'b'. We use a form of the busy code motion algorithm [15] in the opposite direction, which moves an instruction if its execution count is not increased. We add a barrier immediately before the ideal position in order to correctly stop the movement of the node. As a result, the node 'c' is moved after the node 'b' and outside of the loop, as shown in (b), and we can now convert the loop to the TRT instruction.

3.5 Analysis when generating the code pattern

So far we have discussed how we recognized and transformed target graphs into which the idiom graph can be topologically embedded. In this section, we describe an analysis required for generating the code pattern of a few idioms using booltable.

Assumption 1: We already know the target graph matches the idiom graph.
Assumption 2: We already know the local variable *T* that is used for both the array loads and the comparisons.

$$\mathrm{In}(n) = \Big(\bigcup_{m \in \mathrm{Pred}(n)} \mathrm{edge}(n, m, \mathrm{In}(m)) \Big) \cup \mathrm{GEN}(n)$$

GEN(*n*): set of possible values of the array, if there is assignment from the array to the variable *T* in *n*. Otherwise, the empty set Φ.

```
edge(n, m, In){
  if (m's last instruction is an if statement that compares
                                the variable T and a constant){
    taken = set of values of the variable T when the condition is met
    not_taken = ~taken;
    return ((the condition is met along the edge from m to n) ?
                taken : not_taken) ∩ In;
  } else   return In;
}
```

Figure 14. Algorithm for computing the exit conditions

Figure 12 shows the transitions of the input program in Figure 7. Finally, we obtained the optimized program in Figure 12(c) by converting the code sequence enclosed in the dashed-box in Figure 12(b) to TRT(), which is a faster code sequence including the TRT instruction. Because we introduced wildcard nodes into the topological embedding algorithm (which allows any node to be included), we can find multiple if statements corresponding to the special node booltable.

Multiple if statements are sometimes very complex as in **Figure 13**. In this example, we note here that the node "load from array" is separated into nodes inside and outside of the loop. Our approach can successfully recognize it as a candidate with the idiom in Figure 6(a).

Here, we need to create a function table for the TRT instruction. We perform a forward dataflow analysis to compute the exit conditions as shown in **Figure 14**, which is a kind of a value range analysis similar to the one by the approach of Uh *et al.* [30].

In the example of Figure 13(a), GEN will be "-128 to 127" at "t = a[i]", because it is a byte array. After performing the dataflow analysis, we will obtain the exit conditions at the block "*exit*" point in Figure 13(b). Finally, we need to convert the signed value range (-128 to 127) to the unsigned value range (0 to 255). As a result, we can compute the function table for the TRT instruction as shown in Figure 13(c).

3.6 Generated code on IBM zSeries

In this section, we describe the generated code on IBM zSeries for the idioms in Table 1. As shown in Figure 2, our approach first transforms an input loop into a special node (e.g. memcpy, memset, memcmp) at the IL level, and then generates a code sequence corresponding to the node on each platform. We have already explained the first idiom in Table 1 on IBM zSeries, searching for delimiters[1]. The following sections describe the generated code for the second to sixth idioms in Table 1.

[1] In the Appendix, we describe how we perform delimiter searches by using VMX instructions on pSeries.

a) Example: Byte to Char

```
while (srcOffset < limit) {
  int b0 = byteArray[srcOffset];
  if (b0 < 0 || b0 == 0x0D) break;
  charArray[dstOffset] = (char)b0;
  srcOffset++;
  dstOffset++;
}
```

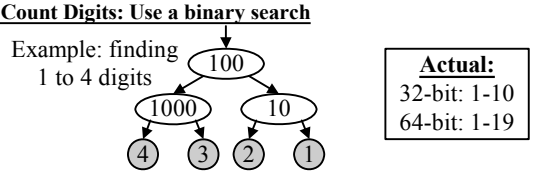
b) Function Table for TROT (512 bytes)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	0	1	2	3	4	5	6	7	8	9	A	B	C	**80**	E	F
0x10	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
0x20	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
:									:							
0x70	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7D	7F
0x80	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**
:									:							
0xF0	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**	**80**

(80: Exit condition)

Figure 15. Example of exploiting the TROT instruction

a) Count Digits: Use a binary search



b) Extract each Digit: Use CVD(G) and UNPKU instructions

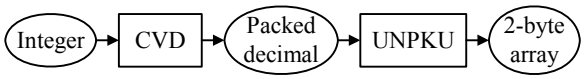


Figure 16. Optimizing converting integers to strings

3.6.1 Converting character codes

The IBM zSeries has the following four instructions for simple character conversions [10]:

- TROO: A **one**-byte array to a **one**-byte array (byte to byte)
- TROT: A **one**-byte array to a **two**-byte array (byte to char)
- TRTO: A **two**-byte array to a **one**-byte array (char to byte)
- TRTT: A **two**-byte array to a **two**-byte array (char to char)

We are finding many opportunities to use the TROT and TRTO instructions in XML parsers, such as conversions from UTF-8 to Unicode and vice versa. These instructions also have a function table, which provides a conversion table and an exit condition.

Figure 15 shows an example of exploiting the TROT instruction, which converts a byte array to a double-byte array. We can convert the input program in (a) to faster code using the TROT instruction. Figure 15(b) shows the function table. In this example, we assume that 0x80 signifies the exit condition. We choose an exit value in the range where the loop exits. We also use the wildcard node booltable to handle flexible if-statements, such as the example of the TRT instruction as shown in Figure 13.

3.6.2 Converting integers to strings

For converting integers to strings, we found the following hot loops:

- Count the digits of the integer using "divide by 10"
- Extract each digit by using "divide by 10" and store it into a double-byte array

Our JIT compiler already improved these loops by replacing the divisions with multiplications, but we can improve them further. For counting the number of digits of an integer value, we replace it with a binary search as shown in **Figure 16(a)**. We actually generate bigger trees for counting the digits of 32-bit and 64-bit integer values. This is an example of converting a slower algorithm to a faster algorithm. Therefore, it means that we can use idiom recognition not only for hardware-assist instructions but also for other improvements, such as algorithm conversions. Because we did not use special instructions for this transformation, we can use it for all architectures.

For extracting each digit of an integer value, we replace the original code with a code sequence using two special instructions on IBM zSeries. We use the CONVERT TO DECIMAL (CVD) and the UNPACK UNICODE (UNPKU) instructions [10] as shown in Figure 16(b). Note that the CVDG instruction can handle a 64-bit integer. The CVD instruction converts an integer to packed decimal data. The UNPKU instruction converts the packed decimal data to a double-byte array.

3.6.3 The other idioms

We can convert loops for copying memory, for filling memory, and for comparing memory into special instructions on IBM zSeries. A loop copying memory can be converted to the MOVE (MVC) instruction [10]. A loop filling memory can be converted to the EXCLUSIVE OR (XC) or the MVC instructions. For filling with zero, we can use the XC instruction. For filling with another value, we can use the MVC instruction with a 1-byte destructive overlap [10]. A loop for comparing memory can be converted to the COMPARE LOGICAL (CLC) or the COMPARE LOGICAL CHARACTER LONG (CLCL) instructions [10].

4. EXPERIMENTS

We measured two metrics in our experiments: (1) how many loops we converted and (2) performance improvements. We used the Java Compatibility Kit (JCK) [12] to see how effective our new algorithm is in finding idioms in comparison to the existing one. For JCK, we used the highest optimization level in compiling every method to find the maximum coverage of our algorithm in finding the idioms we supported. Other than that, we did not set any special JIT compiler options for running the JCK.

To evaluate the performance improvements, we used micro-benchmarks for J2SE class files, IBM XML parser, SPECjvm98, and SPECjbb2000. For the XML parser, we measured three different XML documents: small (567 bytes), medium (52,845 bytes), and large (787,487 bytes). We used the default JIT settings for these measurements. That is, the execution frequency of each method decides the execution mode (in the interpreter or in the JIT compiler) and the optimization level. We did not set any special JIT compiler options for measuring performance. For the SPECjvm98 and SPECjbb2000, we also used the default JIT settings.

We implemented our new idiom recognition approach by modifying the Java JIT compiler. We measured the following variants:

- **Baseline:** Perform an exact pattern matching loop recognition. This compares each IL node in a loop to a pre-defined template. If all of the IL nodes in the loop match the pre-defined template, it will convert the loop to faster code. In

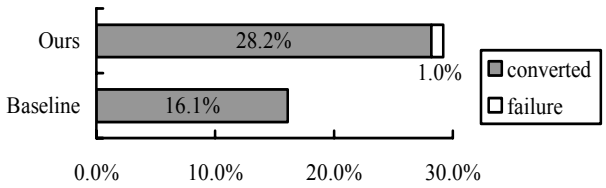


Figure 17. Coverage of our approach for 3,724,925 loops that the JIT compiler tried to optimize in JCK API tests

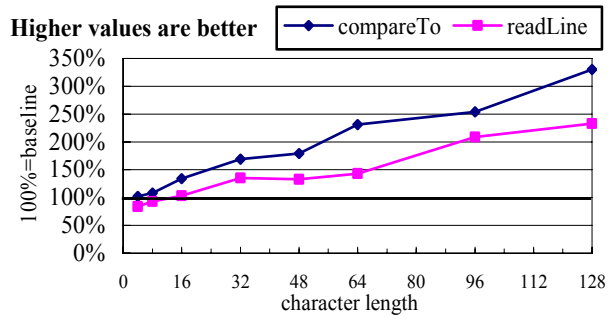


Figure 18. Performance improvements of micro-benchmarks for the J2SE class library on IBM zSeries

order to find as many candidates as possible, we performed traditional optimizations beforehand, such as loop canonicalization, loop versioning, copy propagation, dead code elimination, range analysis, induction variable analysis, alias analysis, exception check elimination, partial redundancy elimination, class/field/array variable privatization, inlining, code specialization, and so on. For delimiter searches, it handles a single if-statement.

- **Ours:** Used our algorithm described in this paper in addition to the baseline. We performed the pattern matching loop recognition in the baseline algorithm first and then applied our algorithm.
- **Disable all:** Disable both the pattern matching loop recognition and our algorithm.

All of the experiments were conducted on a zSeries 990 2084-316 (sixteen 64-bit 1.2 GHz processors with 8 GB of RAM), and running z/Linux.

4.1 Coverage in the JCK API tests

Figure 17 shows how many loops we converted on IBM zSeries for the JCK API tests, which invokes many variants of methods in the J2SE class library. Because the class library is frequently used in Java programs, that coverage is very important. The JIT compiler tried to optimize all of the innermost loops (3,724,925). The topological embedding found 29.2% of them, and our algorithm finally converted 28.2% of them. In contrast, the baseline algorithm using exact matching converted 16.1% of them successfully. Relative to a baseline implementation using exact pattern matching, our algorithm succeeded in finding 81% more candidates (=(29.2/16.1)-1) and ultimately converted 75% more candidates (=(28.2/16.1)-1).

We still have two areas for further improvements. We can create new idioms to convert some non-candidates (from the remaining 70.8%). We can also create new graph transformations to convert some of current failures (1.0%). We are investigating several transformation failures, but we have not yet found examples transformable by the compilers. Those loops include additional nodes that have data dependences upon values of an array. Thus, we cannot separate those nodes from the original loop by using loop distribution or code motion techniques.

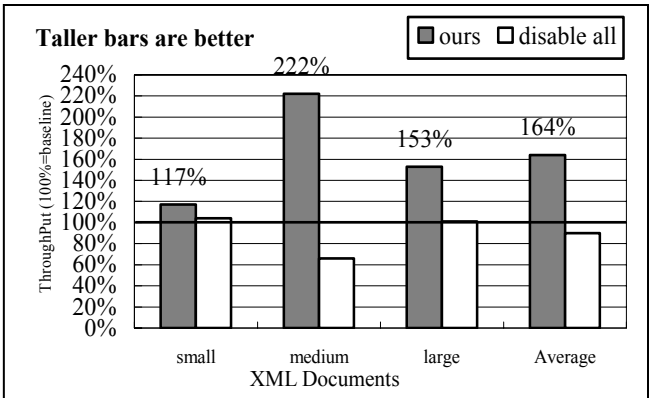


Figure 19. Performance improvements of XML parser using special hardware-assist instructions on IBM zSeries

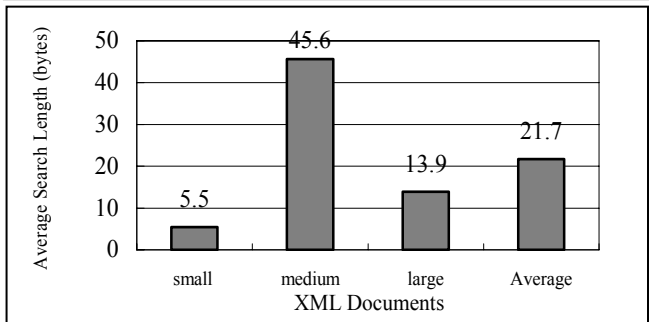


Figure 20. Average search length of the delimiter search loops replaced by our approach

4.2 Performance Improvements

Figure 18 shows the performance improvements of the micro benchmarks for the J2SE class library. We picked two frequently used methods, java/lang/String.compareTo and java/io/BufferedReader.readLine, where the code motion described in Section 3.4.3 and the replication of store nodes described in Section 3.4.2 are necessary, respectively. As can be seen, we obtained good performance improvements for those methods.

Figure 19 shows the performance improvements for the XML parser over our baseline on IBM zSeries[2]. In this figure, the X-axis shows the XML documents. The labels refer to the file sizes. Our approach improves performance for all of the XML documents. We found that exploiting the TRT instruction is particularly effective. Regarding graph transformations, the partial peeling described in Section 3.4.1 and replicating store nodes as described in Section 3.4.2 are particularly important. The baseline compiler using simple pattern matching also improves the performance, especially for the medium size XML document. In summary, our approach improves performance by 64% on average and by up to 122% (2.22x). Since parsing XML documents is done very often

[2] The Appendix shows performance improvements on IBM pSeries.

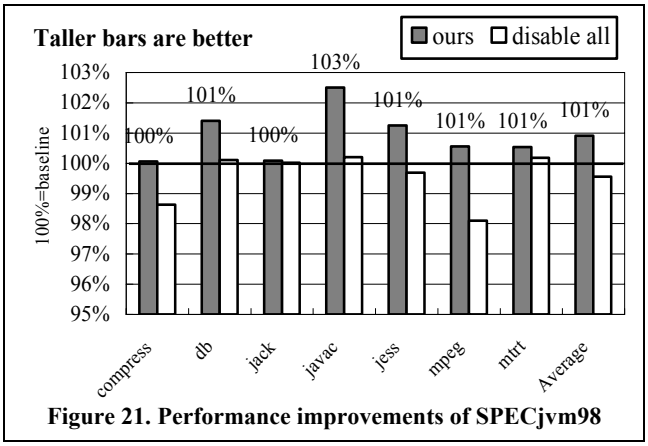


Figure 21. Performance improvements of SPECjvm98

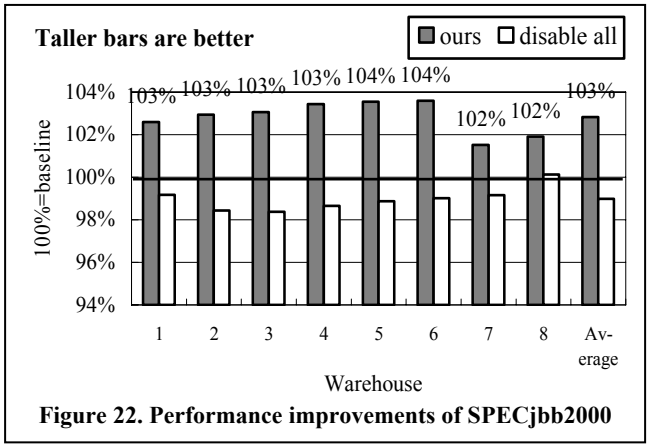


Figure 22. Performance improvements of SPECjbb2000

in Web applications, this result is very significant for the real world.

Figure 20 shows the average search length of the delimiter search loops replaced by our approach. As we mentioned in Section 3.2, while a special hardware-assist instruction greatly improves performance for long data blocks, it could degrade performance for very short data blocks because of its startup costs. Figure 20 can tell the reason for the performance differences in parsing the three XML files in Figure 19.

Figure 21 and **Figure 22** show performance improvements for SPECjvm98 and SPECjbb2000 on IBM zSeries, respectively. We did not find a significant improvement in comparison to the XML parser results. This is because many hardware-assist instructions in the IBM zSeries are targeted at text processing. There are fewer opportunities in these benchmarks than in the XML parser.

4.3 Compilation Time

We have two filters to reduce compilation time by excluding:

- rarely executed methods
- idioms unlikely to be matched

Recent JIT compilers use multiple optimization levels [28], which are driven by the hotness of each method. Our idiom recognition algorithm is performed only at higher optimization levels.

As we mentioned in Figure 3, we exclude those idioms which are unlikely to be matched against the target loop to limit the extra compilation time. We can consider the nodes of an idiom, and if a graph is missing any of those nodes, we already know no topological embedding exists. For each idiom and the graph, we create a bit-vector whose bits represent the opcodes. We compare the bit-vector of every idiom graph with that of the target graph to exclude those idioms which are unlikely to be matched. This minimizes the number of candidate graphs passed to the topological embedding algorithm. In our experiment, we excluded 90% of idioms by this filter. If an idiom is more complex, this filter will more effectively exclude unmatchable idioms. This is because a complex idiom has many characteristics that we can use in this filter. Therefore, we think that the compilation time increase would not change much even if more idioms and more complicated idioms were considered.

We measured the breakdown of the JIT compilation times for the XML parser, SPECjvm98, and SPECjbb2000 on IBM zSeries, as shown in **Table 2**. As we mentioned, our approach performed the pattern matching loop recognition in the baseline algorithm first and then applied our algorithm. In summary, our algorithm increases the total compilation time by only 0.32% to 0.44%, while it achieves significant performance improvements, as shown in Section 4.2.

Table 2. Breakdown of JIT compilation times of our approach

	XML parser	SPECjvm98	SPECjbb2000
Our algorithm	0.28%	0.37%	0.28%
Pattern matching loop recognition (Baseline code)	0.10%	0.07%	0.04%
The rest	99.62%	99.56%	99.68%

5. CONCLUSION

We presented a new idiom recognition technique for dynamic compilers to detect code segments that contain one of the given idiom patterns and to generate faster code by exploiting the hardware accelerators available on the target processors. We are exploiting several special hardware-assist instructions on IBM zSeries and VMX instructions on some models of the IBM pSeries. Our new approach uses a topological embedding algorithm to detect an idiom pattern from the target program in a more flexible manner. Unlike previous approaches, we can detect an idiom pattern even if the code segment does not exactly match the pattern.

Our approach has three features. First, it can find more candidates by utilizing the topological embedding algorithm. Second, it automatically transforms the candidates to idiom graphs to convert the modified graphs into faster code. Finally, even if the graph transformations fail, we can tell the Java programmers or compiler developers about the potential candidates in order to suggest further performance improvements. Our current implementation provides the location of the potential candidate in the Java source code and the pseudo-code corresponding to the idiom.

We implemented our new idiom recognition approach based on the Java Just-In-Time (JIT) compiler that is part of the J9 Java Virtual Machine, and we supported several important idioms. To demonstrate the effectiveness of our technique, we performed two experiments. The first one is to see how many more patterns we can detect over the previous approach. The second one is to see how much more performance improvement we can achieve over

the previous approach. For the first experiment, we used the JCK API tests. For the second experiment, we used IBM XML parser with various XML files, SPECjvm98, and SPCjbb2000. In summary, relative to a baseline implementation using exact pattern matching, our algorithm converted 75% more loops in the JCK tests. We also observed significant performance improvement of the XML parser by 64%, of SPECjvm98 by 1%, and of SPECjbb2000 by 2% on average on a z990. Finally, we observed that the JIT compilation time increases by only 0.32% to 0.44%.

For future work, we plan to support more idioms and graph transformations. Because we want to minimize the increases in compilation time, we did not create rich graph representations, such as a program dependence graph. We plan to investigate which graph representation is actually most effective. In addition, we plan to support some hardware-assist instructions on other architectures, such as IA-32 or the Cell Broadband Engine architecture.

6. ACKNOWLEDGEMENT

We would like to thank Mike Fulton of the IBM Toronto Laboratory for implementing the pattern matching loop recognition approach in the baseline code and for helpful discussions. We also thank Charles Webb of the IBM Poughkeepsie Laboratory, and Arie Tal and Erik Charlebois of the IBM Toronto Laboratory for helpful discussions. We also thank IBMJ-HRS for supplying support to check the wording of this paper.

REFERENCES

[1] W. Blume and R. Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proceedings of the 1994 International Conference on Parallel Processing*, pp. 233-238, 1994.

[2] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. *In Proc. of the 32nd Annual International Symposium on Computer Architecture*, pp. 272-283, June 2005.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In ACM Symposium on Principles of Programming Languages. (POPL '77)*, pp. 238-252, 1977.

[4] J.J. Fu, Directed Graph Pattern Matching and Topological Embedding, *Journal of Algorithms*, 22(2):372-391, February 1997.

[5] N. Grcevski, A. Kielstra, K. Stoodley, M.G. Stoodley, V. Sundaresan: Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. *Virtual Machine Research and Tech Symposium 2004*: pp. 151-162

[6] IBM Corp., *IBM Mainframe*, http://www-03.ibm.com/servers/eserver/zseries/

[7] IBM Corp., *IBM PowerPC Architecture*, http://www-03.ibm.com/chips/power/powerpc/

[8] IBM Corp., *IBM System p5 servers*, http://www-03.ibm.com/systems/p/

[9] IBM Corp., *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, http://www-3.ibm.com/chips/techlib/ techlib.nsf/techdocs/FBFA164F824370F987256D6A006F424D

[10] IBM Corp. *z/Architecture Principles of Operation*, SA22-7832-04, September 2005

[11] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani, "Stride Prefetching by Dynamically Inspecting Objects," *ACM Programming Language Design and Implementation (PLDI 2003)*, pp. 269-277, 2003.

[12] Java Compatibility Kit, https://jck.dev.java.net/

[13] M. Kawahito, H. Komatsu, and T. Nakatani. Partial redundancy elimination for access expressions by speculative code motion, *Software: Practice and Experience*, Vol. 34, No. 11, pp. 1065-1090, 2004.

[14] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. *In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 147-158, Orlando, Florida, June 1994.

[15] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, Vol. 17, No. 5, pp. 777-802, 1995.

[16] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *In Proceedings of the Conference on Programming Language Design and Implementation*, pp.145-156, 2000.

[17] M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*. Vol. 26, No. 3, pp. 413-463, 2004

[18] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1996).

[19] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. *In IJCAI-95 Workshop Program Working Notes*, 1995.

[20] Motorola Corp., *AltiVec Technology Programming Interface Manual*, 1999, http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

[21] S.S. Muchnick. *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, Inc., 1997.

[22] S. Pinter, R. Pinter, Program Optimization and Parallelization Using Idioms, *ACM Transactions on Programming Languages and Systems*, vol. 14, 1994, pp.305-327

[23] B. Pottenger, R. Eigenmann, Idiom recognition in the Polaris parallelizing compiler, Proceedings of *the 9th international conference on Supercomputing*, pp. 444-448, 1995

[24] H. Sato, Array Form Representation of Idiom Recognition System for Numerical Programs, *Proceedings of the 2001 conference on APL*, pp. 87-98, 2001

[25] J. Shin, M.W. Hall, J. Chame. Superword-level parallelism in the presence of control flow, *Symposium on Code Generation and Optimization*, pp. 165-175, 2005

[26] T.J. Slegel, E. Pfeffer, and J.A. Magee. The IBM eServer z990 microprocessor, *IBM Journal of Research and Development*, Vol. 48, No. 3/4, 2004

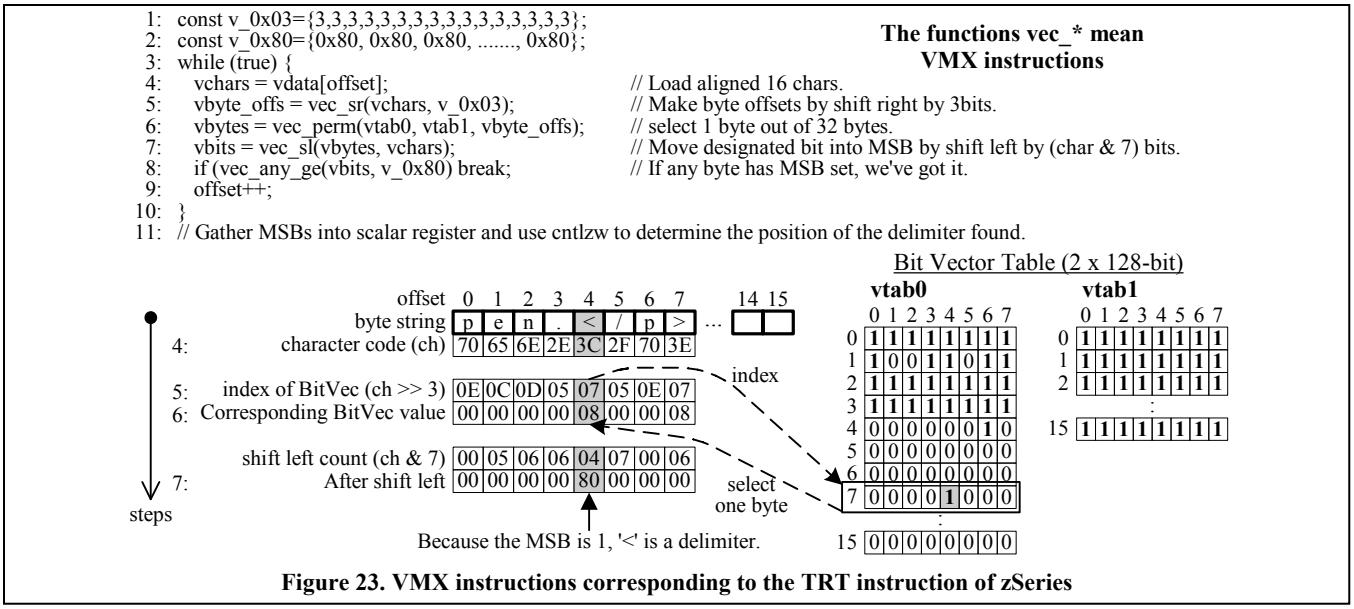


Figure 23. VMX instructions corresponding to the TRT instruction of zSeries

[27] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, T. Nakatani. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, 2000.

[28] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. "A Dynamic Optimization Framework for a Java Just-In-Time Compiler", In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2001

[29] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a Java just-in-time compiler for IA-32 platforms, *IBM Journal of Research and Development*, vol. 48, Num. 5/6, 2004

[30] G.R. Uh and D.B. Whalley. Effectively exploiting indirect jumps, *Software – Practice and Experience*, vol. 29, Num. 12, pp. 1061-1101, 1999

APPENDIX

Using the same idiom recognition framework, we also supported the same idioms described in Table 1 for IBM pSeries. As shown in Figure 2, our approach first transforms an input loop to a special node (e.g. memcpy, memset, memcmp) in the IL level, and then it generates a code sequence corresponding to the node on each platform. Only the code generation is different from that of IBM zSeries.

To begin with, we emulated those hardware-assist instructions of IBM zSeries, which are described earlier, by using the Vector Multimedia eXtension (VMX, also known as AltiVec or Velocity Engine) instructions [9][20] that are available on some models of the IBM PowerPC processors [7]. VMX provides 128-bit vector length that can be subdivided into sixteen 8-bit values, eight 16-bit values, or four 32-bit values. For the purpose of emulation, we use the instruction set for "sixteen 8-bit values".

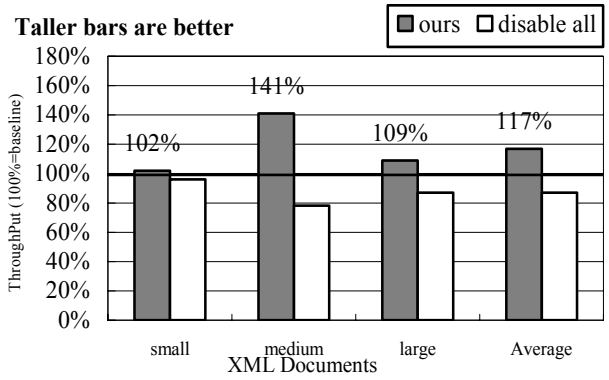


Figure 24. Performance improvements of XML parser using VMX instructions on IBM pSeries

As an example, we describe how we emulate delimiter searches by using VMX instructions in **Figure 23**. We convert a function table (which denotes delimiter characters) for the TRT instruction into a pair of 128-bit vector registers. Essentially, we look up the bit-vector in a 16-way parallel manner by using vector permute and vector shift operations. We assume that vtab0 and vtab1 are converted from the function table in Figure 13(c). This implementation effectively evaluates the following if statement for 16 characters as one step.

```
if ((BitVec[ch >> 3] << (ch & 7)) >= 0x80) break;
```

To see the effectiveness of our approach on IBM pSeries, we measured performance improvements of the XML parser. All the experiments were conducted on a BladeCenter JS20 (PowerPC 970FX 2.2GHz with 1 GB of RAM), and running Linux. **Figure 24** shows the performance improvements for the XML parser over our baseline on IBM pSeries. In summary, our approach improves performance by 17% on average and by up to 41%.