Idiom Recognition Framework Using Topological Embedding

MOTOHIRO KAWAHITO, HIDEAKI KOMATSU, TAKAO MORIYAMA,
HIROSHI INOUE, and TOSHIO NAKATANI, IBM Research - Tokyo

Modern processors support hardware-assist instructions (such as TRT and TROT instructions on the IBM System z) to accelerate certain functions such as delimiter search and character conversion. Such special instructions are often used in high-performance libraries, but their exploitation in optimizing compilers has been limited. We devised a new idiom recognition technique based on a topological embedding algorithm to detect idiom patterns in the input programs more aggressively than in previous approaches using exact pattern matching. 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 also propose an instruction simplification for the idiom recognition. This optimization analyzes all of the usages of the output of the optimized code for a specific idiom. If we find that we do not need an actual value for the output but only a value in a subrange, then we can assign a value in that subrange as the output. The code generation can generate faster code with this optimization. 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 the special hardware-assist instructions on the IBM System z and on some models of the IBM System p. To demonstrate the effectiveness of our technique, we performed two experiments. The first experiment was to see how many more patterns we can detect compared to the previous approach. The second experiment measured the performance improvements over the previous approaches. For the first experiment, we used the Java Compatibility Kit (JCK) API tests. For the second experiment we used the IBM XML parser, SPECjvm98, and SPCjbb2000. In summary, relative to a baseline implementation using exact pattern matching, our algorithm converted 76% more loops in JCK tests. On a z9, we also observed significant average performance improvement of the XML parser by 54%, of SPECjvm98 by 1.9%, and of SPECjbb2000 by 4.4%. Finally, we observed that the JIT compilation time increased 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

Additional Key Words and Phrases: Idiom recognition, hardware-assist instructions, VMX, topological embedding, Java, JIT

ACM Reference Format:

Kawahito, M., Komatsu, H., Moriyama, T., Inoue, H., and Nakatani, T. 2013. Idiom recognition framework using topological embedding. *ACM Trans. Architect. Code Optim.* 10, 3, Article 13 (September 2013), 34 pages. DOI: <http://dx.doi.org/10.1145/2512431>

An earlier version of this article was presented at the ACM SIGPLAN Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2006.

Authors' addresses: M. Kawahito (corresponding author), H. Komatsu, T. Moriyama, H. Inoue, and T. Nakatani, IBM Research Tokyo, NBF Toyosu Canal Front Building, 5-6-52 Toyosu, Koto-ku Tokyo, 135-8511 Japan; email: j125131@jp.ibm.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/09-ART13 \$15.00

DOI: <http://dx.doi.org/10.1145/2512431>

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 [Pinter and Pinter 1994; Pottenger and Eigenmann 1995; Sato 2001]. It is also useful for exploiting hardware-assist instructions, which are becoming more important as the rate of increase of processor frequencies is declining due to power and cooling limitations.

Traditional approaches for idiom recognition compare the input pattern against the idiom pattern for an exact match [Pinter and Pinter 1994; Pottenger and Eigenmann 1995; Sato 2001]. The state-of-the-art technique for idiom recognition is to apply loop canonicalization [Muchnick 1997] and common optimizations [Muchnick 1997] before using matching algorithms. However, even if we perform loop canonicalization, recognition will fail when an idiom pattern does not appear exactly as expected in the input pattern. The input pattern may include additional nodes. The order of the nodes in the input pattern may be different from an idiom pattern. For example, in Figure 1(a) and (c), the idiom cannot be recognized because the order of the nodes is different. In Figure 1(b), the idiom cannot be recognized because it has the additional node “store to ch”.

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 [Fu 1997], 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 three steps. If we succeed, we can convert the modified graph into faster code by using a hardware-assist instruction.

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 replicate “ch = a[i]” 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 during idiom recognition. For another example, we can move the node “i++” forward to match the idiom in Figure 1(c). We will explain these transformations later. As a result, our algorithm can convert many more candidates to faster code for the maximum use of hardware-assist instructions compared to previous approaches.

In addition, we use instruction simplification for the idiom recognition. The earlier version of this article [Kawahito et al. 2006] did not include this optimization, but there are some cases in which additional analysis can improve the performance of the converted code by transforming the complex instruction to a less complex instruction by instruction simplification when only a range of the value, such as positiveness or negativeness, is needed. For example, the method `compareTo` in the `String` class returns the difference of the two character values of the input strings if they have different characters [Sun Corp. 2006]. If it is known that the output is only used for comparisons with 0 (i.e., checking negative, 0, or positive), we do not need to compute the actual value. The method `compareTo` is often used for sorting strings, so there are a lot of these opportunities. This optimization can be important because this computation causes significant overhead on some architectures.

We implemented our new idiom recognition algorithm in the Java Just-In-Time (JIT) compiler that is part of the J9 Java Virtual Machine. For exploiting hardware-assist instructions on IBM System z [IBM Corp. 2013a; Siegel et al. 2004] 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

Exact match cannot convert these patterns.

Fig. 1. Limitations of exact match.

beyond a previous approach. The second one studies the performance improvements we can achieve over the previous approach. For the first experiment, we used the JCK [Sun Corp. 2013] 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 76% more loops in the JCK tests. On a z9 [IBM Corp. 2013a], we also observed significant average performance improvements of the XML parser by 54%, of SPECjvm98 by 1.9%, and of SPCjbb2000 by 4.4%, while the JIT compilation time increased by only 0.32% to 0.44%.

We also supported the same idioms for exploiting SIMD instructions on IBM System p. In Section 3.7.2, we explain how we implemented delimiter searches using vector instructions. We also describe some experiments on IBM System p.

1.1. Our Contributions

- A new idiom recognition approach.* Our two-phase idiom recognition algorithm can find variations of an idiom pattern in the input program more aggressively than previously known algorithms and automatically transform them into the idiom. As a result, it can take much greater advantage of any hardware-assist instructions.
- Instruction simplification for the idiom recognition.* This can reduce the overhead of computing an output value. If we find that we do not need an actual value for the output but only any value in a subrange, then we can assign a value in the subrange for the output. The code generation can generate faster code with this optimization.
- Exploitation of special hardware-assist instructions.* We demonstrate how these instructions can be exploited in commercial applications.

The rest of the article 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.

For library calls in Java, the IBM JIT compilers [Grcevski et al. 2004; Suganuma et al. 2004] can generate optimized code for such calls as `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 these libraries to use these instructions.

For 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 [Muchnick 1997]. This technique is widely used in optimizing compilers. For example, Aho et al. developed the system named *Twig*, which transforms a tree translation scheme into a code generator that combines a fast top-down tree-pattern matching algorithm with dynamic programming [Aho et al. 1989]. 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 [Suganuma et al. 2000]. Clark et al. proposed an approach [Clark et al. 2005] that extracts a specific instruction sequence from several basic blocks. They also performed code transformations, such as a kind of code motion, though this is still limited to an acyclic region. Acyclic approaches do not try to solve cyclic-specific problems, such as Figure 1(c). Superword-Level Parallelism (SLP) is an approach to exploit SIMD instructions [Larsen and Amarasinghe 2000] for optimizing a loop body. It unrolls a loop in advance, and then recognizes the vectorizable instructions in a basic block. This is suitable for a loop whose body consists of a single basic block. Shin et al. extends SLP in the presence of control flow [Shin et al. 2005], but this is still limited to an acyclic region. Olschanowsky et al. developed the idiom recognition [Olschanowsky et al. 2010] to improve the parallel applications by using hardware accelerators, such as GPUs and FPGAs. They defined seven idioms to utilize these hardware accelerators. He et al. used a similar approach [He et al. 2011]. They defined five idioms, which are similar to those of Olschanowsky.

The second family recognizes specific instruction sequences (which may include cycles) to parallelize numerical programs [Pinter and Pinter 1994; Pottenger and Eigenmann 1995; Sato 2001]. They compare the instruction sequence of the loop body with each predefined idiom. We call this an “exact match”, but this often fails to catch idioms when programmers slightly change a program. For example, this approach cannot recognize the examples in Figure 1. Metzger proposed a combination of idiom recognition and algorithm recognition [Metzger 1995]. This approach first replaces each idiom 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 an alternate implementation. This approach also relies on an exact match, and thus it still misses many opportunities.

To improve the effectiveness of idiom recognition, there has been a lot of research [Muchnick 1997] 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. These are effective in exposing specific patterns for idiom recognition. For Java, however, it is rare to find those loops that are candidates for loop distribution or loop fusion. One reason is that Java programmers tend to use many method calls, which makes 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 found that many graph transformations could not be performed because the input loops included one or more method calls. Another reason is that the multidimensional 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. This means we can only eliminate exception checks from the innermost loop by using the loop versioning technique, which limits the cases where loop optimizations can be applied.

It is also known that abstract interpretation techniques [Cousot and Cousot 1977; Leuschel 2004] or symbolic analysis techniques [Blume and Eigenmann 1994] can help improve the effectiveness of idiom recognition. Abstract interpretation techniques are feasible for JIT compilers. For example, we use an abstract interpretation technique [Inagaki et al. 2003] for software prefetching in our JIT compilers. In contrast, the

Fig. 2. Comparison of previous approaches and ours.

symbolic analysis techniques are powerful but more time consuming. For this reason, our baseline compiler uses 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 many traditional optimizations in advance, such as inlining, copy propagation, dead code elimination, code specialization, exception check elimination, and partial redundancy elimination. These techniques help find as many optimization candidates as possible.

3. OUR NEW ALGORITHM

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. After performing the loop

```

for (each inner-most loop  $l$ ) {
  Translate  $l$  to graph  $T$ ;
  for (each idiom graph  $P$  in the pre-defined list) {
    if (all of the nodes in  $P$  appear in the graph  $T$  &&           // Section 3.2
         $l$  is sufficiently iterated) {
      Perform our TE algorithm;                                     // Section 3.3
      if (  $P$  is embedded into  $T$  ) {
        create UD/DU chains;
        do partial peeling of the loop body;                       // Section 3.4
        do replicate the store nodes;
        do code motion;
        if (  $T$  matches  $P$  ) {
          Replace the loop with the pre-defined code pattern;       // Section 3.5
          break;
        }
      }
    }
  }
}
}
}
Perform instruction simplification;                                // Section 3.6
Code Generation                                                    // Section 3.7

```

Fig. 3. Pseudocode of our algorithm.

canonicalization and common optimizations, we find all of the innermost loops in the 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 innermost loop into one corresponding to the idiom graph using our three new graph transformation techniques. Then we perform exact matching as in the previous approaches. After the graph transformations, if the input graph is equal to the idiom graph, we can convert the input graph to the predefined faster code. When successful, we first transform an input loop to a special node (e.g., memcopy, memset, memcmp) at the Intermediate Language (IL) level, and then our system generates a code sequence corresponding to that node for each platform.

Figure 3 shows the pseudocode for our algorithm. First, we transform each loop to our graph representation. Next we apply the two prefilters 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 as described in Section 3.3 before attempting to match the idiom by applying the graph transformations described in Section 3.4. If a transformed graph matches an idiom, we can replace it with a special node corresponding to that idiom to generate a faster code sequence. With our algorithm, we can easily support a new idiom by adding an idiom graph and the corresponding code generation pattern without modifying the algorithm. Finally, instruction simplification for the idiom recognition as described in Section 3.6 reduces some of the overhead for several patterns.

Table I shows the currently supported idioms. These idioms are architecture independent. We use special hardware instructions both on IBM System z and IBM System p. More details will appear in Section 3.7.

3.1. Advantages of Topological Embedding

In this section, we briefly describe the advantages of the Topological Embedding (TE) algorithm [Fu 1997]. In this article, we use “idiom graph” or P for an idiom and “input

Table I. Supported Idioms, Use of a Booltable Node, Opportunities for Instruction Simplification, and Algorithm Transformation

Idiom Name	Description	Booltable	Simplification	Algorithm Transformation
findbytes	searching for delimiters	yes	yes	no
arraytranslate	converting character codes	yes	no	no
memcpy	copying memory	no	no	no
memset	filling memory	no	no	no
memcmp	comparing memory	no	no	no
compareTo	String.compareTo	no	yes	no
countDigits	count digits of integers	no	no	yes
intToString	converting integers to strings	no	no	yes

a) TE allows **any nodes to be included** between any two nodes

b) TE allows **a different order of nodes** in a cycle

Fig. 4. Advantages of topological embedding algorithm.

graph” or T for an input program. We consider the 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 the siblings is significant. In other words, we cannot reorder siblings. 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 its 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, that is, for each node α in P , the i th child of α from the left can be mapped to either the i th child c of $f(\alpha)$ or a descendant of c . The computational order of the TE algorithm is $O(|V_P||E_T| + |E_P|)$ [Fu 1997]. Here, V and E are nodes and edges, respectively. For detailed algorithms and pseudocode, please see the paper [Fu 1997].

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 Figure 4(b), the order of nodes is different. In a Strongly Connected Component (SCC), any node can be reached from any other node. According to Figure 4(a), TE allows

Fig. 5. Our input graph for our TE algorithm.

nodes to be included between any two nodes of the idiom graph. Thus, we can recognize this input program as a candidate.

3.1.1. Our Extension to TE Algorithm. Here are the main characteristics of our TE algorithm.

- We focused on the innermost loop as a candidate input graph for maximum gain with minimum overhead. The input loop consists of n -nodes and k -exits. It may include a header and/or footers for the loop, as shown in Figure 5.
- The original TE assumes that the left-to-right order of the 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 wild-card nodes, which match several opcodes (labels) in an input graph. Currently, we use two kinds of wild-card nodes. The first one is a variable. It matches any variable in the input graph. The second one is a booltable. It matches any number of comparisons of its operand against any constant. For example, it matches the two comparisons in Figure 8(c) and the seven comparisons in Figure 16(a).

3.2. Prefilters

There are two reasons to perform prefiltering. Avoiding 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 prefilters before our TE algorithm. The first prefilter checks that all the nodes in the idiom appear in the input 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 bit-vector is on. We compare the bit-vector of each idiom graph with that of the input graph to exclude those idioms which cannot be matched.

The second prefilter excludes rarely iterated loops if the hardware-assist instruction corresponding to the idiom has a large startup cost. For example, for the TRANSLATE AND TEST (TRT) instruction [IBM Corp. 2004] on IBM System z, we cannot estimate the actual search length, because it depends on the content of each input array. To predict the search length, we use runtime profile information. We compute the ratio of the frequency of the inner block over that of the outer block for each loop and exclude the rarely iterated loops from the candidates.

We prepared a threshold for each idiom. For example, the threshold for compareTo is 12, because Figure 31 showed that performance is improved if the length is larger than 12. Note that the startup costs for each special hardware-assist instruction vary based 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.

Fig. 6. An example that exploits the TRT instruction.

3.3. Finding the Candidate Graphs

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

- (1) Translate input Intermediate Language (IL) code into two graph representations, which consist of:
 - Abstract Syntax Trees (AST): The nodes are IL nodes (or wild-card nodes for idiom graphs). The edges are the operands.
 - Control-Flow Graph (CFG): The nodes are the same as in an AST. The edges show the control flows.
- (2) Find candidates among the leaf nodes in the abstract syntax trees [Muchnick 1997] from step (1).
- (3) Use the TE algorithm to walk through each AST from every leaf to its ancestor and find the candidate nodes. In this step, we check the commutative operands as required.
- (4) Use the TE algorithm to walk through the CFG from the exit to the entry while checking the relationships between all of the ancestors and descendants for each node found in step (3).
- (5) Extract the smallest subgraph that includes all of the nodes in the current idiom candidate.

3.3.1. Example: TRT Instruction. We use the following examples to clarify how our algorithm works to find the candidate input graphs. The TRANSLATE AND TEST (TRT) instruction [IBM Corp. 2004] on IBM System z 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, nonzero values signify the special characters. In this article, we call them “*delimiters*”.

Figure 6 shows an example exploiting the TRT instruction. The example in Figure 6(a) searches for 0×00 , $0 \times 0A$, and $0 \times 0D$ in a byte array. In this example,

Fig. 7. Idiom graph of the TRT instruction.

the result address should point to $0 \times 0D$ (carriage return). We can convert this example code to faster code using the TRT instruction as shown in Figure 6(b). We need to prepare the function table by setting nonzero values for the table entries of the delimiters. By using this conversion, performance can be improved tenfold, depending on the search length. Such loops are often found in text processing programs such as XML parsers. In actual programs, these multiple 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 7(a) and (b) show AST and CFG representations for the TRT instruction. It consists of nodes and two kinds of edges. Here, “baload” means “load byte from array” [Lindholm and Yellin 1996]. In the graph, there are two kinds of wild-card nodes, variables and the special node “booltable”. Variable nodes in the idiom match all of the variables in the input 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. Figure 7(c) shows the pseudocode corresponding to the idiom.

Figure 8 shows a motivating example of an input program. Step 1 of the algorithm described in Section 3.3 translates the input program (c) to the graph representations (a) and (b). In step 2, we find candidate leaf nodes by analyzing their ancestors. For example, since the parents of the variable “v1” in Figure 7(a) are “baload”, “iadd”, and “istore”, the variable “i” in Figure 8(a) becomes a candidate for “v1”. For each node in Figure 7(a) and (b), all of the children can be mapped to either children or descendants in Figure 8(a) and (b). Therefore, we can detect Figure 8 is a candidate.

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. In Figure 8, the nodes iadd and istore are placed at the first position, but they are placed at the last position in the idiom graph (Figure 7). (2) There is an additional node “store into the variable ‘ch’”. (3) There are multiple if-statements. As we mentioned in

Fig. 8. Motivating example of an input program.

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 wild-card 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 Section 3.3 may find graphs whose program patterns are different from the idioms, we need to transform the candidate input graphs to the idiom graphs.

Before graph transformations, we create Use-Def(UD)/Def-Use(DU) chains [Aho et al. 1986] to analyze the data dependencies among the variables. Then we tried to apply each of the following three graph transformations once and check whether the modified input graph matches the idiom graph: (1) partial peeling of a loop body, (2) replicating store nodes outside of loops, and (3) code motion.

Intuitively, the partial peeling replicates the region from the loop entry to the ideal loop entry outside of the loop in order to align the loop entry to the idiom. Forward code motion reorders the nodes to match the idiom graph. Store replication copies the store node outside of the loop to nullify the store node in the candidate input graph.

In this phase, we do not yet directly modify the input intermediate language code because it is difficult to undo those transformations, and because unneeded transformations may degrade performance. For example, if a store node is replicated with the technique of Section 3.4.2 but the loop cannot be transformed, then it will degrade the

```

P: Idiom graph, T: Input 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;  

}  

rightLoopEntry = 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 rightLoopEntry;


```

Fig. 9. Algorithm for partial peeling.

performance (the transformation from Figure 8 to Figure 12). Partial peeling and the forward code motion in Sections 3.4.1 and 3.4.3 will not improve the performance if the loop is not transformed. Therefore, in general, compilers usually do not perform such transformations.

Instead of modifying the input intermediate language code, 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, then we generate the compensation code and the special IL node corresponding to a hardware-assist instruction.

3.4.1. Partial Peeling of a Loop Body. The goal of this transformation is to align the loop entry point with the idiom. Figure 9 shows our algorithm. This algorithm first finds the right loop entry node that corresponds to the loop entry node of the idiom. This transformation replicates the region from the current loop entry to the right loop entry outside of the loop, and then it moves the entry point to match the right entry point.

Figure 10 shows the transformed result of Figure 8. In the example of Figure 8(b), the loop entry is the node “iadd”, but it should be the node “baload”. Thus, this transformation replicates the nodes “iadd” and “istore” outside of the loop and changes the loop entry to the node “baload”.

3.4.2. Replicating Store Nodes. This transformation is very unique for our idiom recognition. The example in Figure 8 includes an additional node, a store to the variable “ch”. As we mentioned in Figure 8, we are assuming that the variable “ch” will be used after the loop. In that case, we cannot ignore the store node. Previous approaches give up on that 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.

Figure 11 shows our algorithm for store replication. This transformation is similar to Partial Dead code Elimination (PDE) [Knoop et al. 1994b]. 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

Modified program:

```

byte ch;
i++;
while(true){
    ch = a[i];
    if (0x20 > ch || ch == '<') break;
    i++;
}

```

// This expression is replicated.
// Change the loop entry

Fig. 10. After performing partial peeling.

P: Idiom graph, *T*: Target graph

S = additional store node;
L = list of conditional branches that exit the target loop;
RHSe = right-hand side expression of *S*;
 return *AnalyzeStoreReplication*(the fall-through node of *S*);

```

AnalyzeStoreReplication(t) {
    while(true) {
        if (t is already analyzed)
            return SUCCESS;
        if (L contains t) {
            replicate S and RHSe into an exit edge of t;
            remove t from L;
            if (L is empty)
                return SUCCESS;
        }
        if (t breaks S || t breaks RHSe)
            return FAILURE;
        if (t has multiple successors) {
            anode = non fall-through node in this loop;
            if (AnalyzeStoreReplication(anode) == FAILURE)
                return FAILURE;
        }
        t = the fall-through node of t;
    }
}

```

Fig. 11. Algorithm for store replication.

Modified program:

```

byte ch;
i++;
while(true){
    ch = a[i];
    if (0x20 > ch || ch == '<') break;
    i++;
}
ch = a[i];
  
```

// This expression is replicated.

// Change the loop entry

// This expression is replicated.

Fig. 12. After store replication.

and the loop exit point. For moving memory access expressions in Java, Kawahito et al. discussed possible barriers and alias analysis in Kawahito et al. [2004].

Figure 12 shows the transformed result of Figure 10. 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 shown in Figure 12, the store replication itself degrades the performance of the program. Thus, we need to cancel such a transformation if the loop cannot be transformed to faster code.

Figure 13 shows the optimized result of the input program in Figure 8. Finally, we obtained the optimized program as shown in the pseudocode by converting the loop enclosed in the dashed box to TRT(), which is a faster code sequence that uses the TRT instruction. Because we introduced wild-card nodes into the topological embedding algorithm (which allows any node to be included), we can find compound “if-statements” or nested ones corresponding to the special node booltable.

3.4.3. Forward 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 have implemented only forward code motion, because the partial peeling covers some of the transformations required for backward code motion. Figure 14 shows our algorithm for this code motion. For each node t in the input graph, this algorithm first gets the idiom node p corresponding to

Pseudo-code of the Optimized Result

```

byte ch;
i++;
i = TRT(a, i, FunctionTable);
ch = a[i];

```

// This expression is replicated.
// This expression is replicated.

Fig. 13. The optimized results for the input program in Figure 8.

P: Idiom graph, *T*: Target graph

```

for (each target node t from end to start in T) {
  p = a pattern node corresponding to t;
  if (p is empty) return FAILURE;
  NextP = the next pattern node of p;
  NextT = a target node corresponding to NextP;
  Analyze whether we can move t immediately before NextT
  if (t can be moved to immediately before NextT)
    move t before NextT;
  else
    return FAILURE;
}
return SUCCESS;

```

Fig. 14. Algorithm for code motion.

t. It next gets the *NextT* corresponding to the next node of *p*. Finally, it tries to move *t* immediately before *NextT*. When every node in *T* is moved, we have successfully reordered the nodes to match the idiom graph.

Figure 15 shows another example to illustrate this transformation. In this example, we see that the nodes “iadd” and “istore” are not at their ideal locations but are between the nodes “baload” and “if-eq”. Note that there is a control dependence between nodes “if-eq” and “iadd, istore”. We use a form of the busy code motion algorithm [Knoop et al. 1994a] 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 nodes “iadd” and “istore” are moved

Fig. 15. Example of code motion.

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	0	0	1	1	0	1	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	1	0	0	0	0	0	0	0	0	0
0x30	0	0	0	0	0	0	0	0	0	0	0	1	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
⋮																
0xF0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

('1' means exit conditions)

Fig. 16. Complex if-statements.

after the node “if-eq” and outside of the loop, as shown in (b), and we can now convert the loop to the SRST instruction as shown in (c). Note that the SRST instruction is a single-delimiter version of the TRT instruction. We do not need to create a function table for the SRST instruction.

3.5. Generating Function Table

So far we have discussed how we recognized and transformed the input graphs into which the idioms graph can be topologically embedded. In this section, we describe the

Assumption 1: We already know the input 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.

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

$GEN(n)$: 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
    notTaken = taken;
    return ((the condition is met along the edge from  $m$  to  $n$ ) ? taken : notTaken)  $\cap$  In;
  } else return In;
}

```

Fig. 17. Algorithm for computing the exit conditions.

analysis required for generating the code patterns of a few idioms using booltable. On System z, we generate the TRT instruction for a booltable. As explained in Section 3.3.1, we need to prepare a function table for TRT as a 256-byte array to indicate which characters are delimiters. This analysis is necessary for creating the function table. In Figure 13(c), we used the parameter FunctionTable. In this section, we explain how we create this table.

Compound “if-statements” or nested ones are sometimes very complex, as shown in Figure 16. In this example, the node “load from array” ($t = a[i]$) appears twice, one before the loop, the other at the end of the loop. Our approach can successfully recognize this as a candidate to match the idiom in Figure 7(a) and (b). This is because our algorithm detects that both load nodes are the same and are matching the operand of the booltable in the idiom graph.

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 17. This is a kind of a value range analysis. This algorithm propagates possible value ranges forward for the input operand of the booltable to compute the exit conditions.

We explain this algorithm by using the example of Figure 7. For this example, the value range of t will be “−128 to 127” at “ $t = a[i]$ ”, because it is a byte array. Then along the true edge of “ $t < 0 \times 20$ ”, the value range will be “−128 to 31”. Along the false edge, the value range will be “32 to 127”. We used 256-byte or 65536-byte tables to represent these value ranges for 1-byte or 2-byte arrays, respectively. After performing the dataflow analysis, we will determine the exit conditions at the block “exit” point in Figure 7(b). Finally, we need to convert the signed value ranges (−128 to 127) to the unsigned value ranges (0 to 255). From this, we can compute the function table for the TRT instruction as shown in Figure 7(c).

3.6. Instruction Simplification for the Idiom Recognition

This section describes an important optimization after performing the idiom recognition. The earlier version of this article [Kawahito et al. 2006] does not include this optimization. At the end of idiom recognition (as seen in Figure 3), instruction simplification can transform a complex instruction to a less complex instruction when the exact value is not needed as the output for the following instructions. This will speed up the execution. For example, if only positiveness is needed as the output of the String.compareTo instruction by its following instructions, the CLC instruction is generated instead.

```

while(i < len){
    int diff = src1[i++] - src2[j++];
    if (diff != 0) break;
}

```

Fig. 18. Pseudocode of the core loop of the method `String.compareTo()`

a) Non-speculative variable	b) Speculative variable
// generated by Idiom Recognition result = <i>HWInst</i> ();	// generated by Idiom Recognition result = <i>HWInst</i> ();
status[id] = result ;	if (result < 0){ return NotFound; } else { return Found; }

Fig. 19. Examples of a nonspeculative or a speculative variable.

It analyzes all of the usages of the output of the reduced code. We apply this optimization to the idiom “compareTo” in Table I. If we find that we do not need an actual value for the output but only some value in a subrange, then we will be able to assign an arbitrary value in the subrange to the output. Figure 18 shows the pseudocode of the core loop of the method `String.compareTo`. Note that the actual code of `String.compareTo` includes the code when the lengths of the two strings are different. The output of this loop is the variable “diff”. We created the idiom named “compareTo” in Table I based on this loop. In this example, if the variable “diff” is used only for comparisons after `String.compareTo`, we do not need to compute the actual value of “diff”.

In this section, we explain how we reduced the computation costs with an example. We first describe a general algorithm, and then we describe an example of `String.compareTo`.

3.6.1. General Algorithm. We define two categories for a variable.

- Nonspeculative variable.* We cannot use an arbitrary value for this variable. Examples are a store operation or a computation. Figure 19(a) shows such an example. The variable **result** is the output of a hardware-assist instruction. It is stored into memory, so an actual value is needed. We call this a nonspeculative variable.
- Speculative variable.* We can use an arbitrary value in the subrange for this variable. One example is a comparison to a constant, such as “**result** < 0”. Figure 19(b) shows such an example. In this example, the variable **result** selects for two conditions: NotFound and Found. We do not need to compute an actual value for this case. If we know **result** is always positive, and if **result** is only used in this comparison, then we can use any positive number for **result**. We call this a speculative variable.

We perform the following four steps.

- (1) *Collect value ranges.* We collect value ranges for uses of the result of a complex instruction by using this algorithm. For each definition of a variable, we compute the union of the triple {*value range before this operation*, *value range after this operation*, *speculative*} at each use point. We set *speculative* to true if the use is a comparison to a constant value. Otherwise, we set it to false. We compute value ranges after this operation for each edge. For example, we compute two different value ranges for each edge of a conditional branch.

- (2) *Remove intermediate value ranges.* For two elements u and v in the set computed in step (1), we can eliminate u from the set, if u 's *speculative* is true, and if u 's *value range after this operation* is equal to v 's *value range before this operation*. The purpose of this step is to remove intermediate value ranges for nested if-statements.
- (3) *Write a representative value to the variable.* If the set computed in steps (1) and (2) is equal to one of these two sets, we can write a representative value into the variable. Here, an asterisk "*" means an arbitrary value.
 - $\{*, 0, *\}$ and $\{*, \text{nonzero}, \text{true}\}$: We can write zero or a representative nonzero value into the variable.
 - $\{*, 0, *\}$, $\{*, \text{smaller than zero}, \text{true}\}$, and $\{*, \text{greater than zero}, \text{true}\}$: We can write zero, a representative positive value, or a representative negative value into the variable.
- (4) *Replace the representative value with the Condition Code (CC).* If we can directly use the CC instead of the representative value in step (3), then we can replace the representative value with the CC.

Figure 20 shows the general instruction simplification algorithm for our idiom recognition. Note that step (4) is called from inside of step (3) to reduce the compilation time.

3.6.2. Example of CompareTo. In this section, we refine the previous algorithm to optimize the method `compareTo`. The method `compareTo` in the `String` class has to return the difference between the two character values of input strings if they are different characters [Sun Corp. 2006].

System z has several kinds of memory compare instructions [IBM Corp. 2004]. In Figure 18, we consider two instructions: CLC and CLCL. The CLC instruction compares two strings and stores one of the three states (equal, low, or high) into the condition code. The drawback of the CLC instruction is that we do not obtain the position of the differing character. In contrast, the CLCL instruction returns the position of the differing character in addition to the result of the CLC instruction. The drawback of the CLCL instruction is that it is much slower than the CLC instruction. Therefore, we should use the CLC rather than CLCL whenever possible.

Figure 21 shows the pseudocode of the result of applying the idiom recognition to Figure 18. There are two kinds of overhead we can reduce. The CLCL instruction in (1) is slower than a CLC instruction. Computing the difference in (2) is also slow. If it is known that the output is only used for comparisons with 0 (i.e., checking negative, 0, or positive), we do not need to compute the actual value. In this case, we can use any negative value for the negative subrange and any positive value for the positive subrange. Figure 22 shows our algorithm for analyzing the conditions of the instruction simplification. Basically, we check if the variable is used only for comparisons with 0. If we find that the variable is the source operand of a copy instruction, then we will also recursively analyze the destination operand of the copy instruction.

The method `compareTo` is often used for sorting strings. For this purpose, `compareTo` is usually used only for comparisons with 0, as in Figure 23. Since sorting strings is common in many real applications, this optimization is important.

Figure 24 shows the result of applying our optimization to Figure 21. This shows we can use the CLC instruction at (1). In addition, we can compute the output at (2) without loading anything from memory. We used -1 and 2 to represent negative and positive values, respectively. Our implementation of (2) exploits the fact that the condition code (CC) is 0, 1, or 2. As shown in Figure 24, we can rapidly compute these values without using any conditional branches.

For future work, we want to completely eliminate step (2) while guaranteeing that the CC retains the same value for all of the uses of the output. For example, in Figure 23,

```

Input  $s$ : A store operation whose source operand is the result of the complex instruction
 $set = \text{step1}(s)$ 
 $\text{step2}(set)$ ;
 $\text{step3}(set, s)$ ;

 $\text{step1}(s)$  { //  $s$  is the target store operation.
   $ret1 = \emptyset$ ;
  for (each  $u \in \text{uses of } s$ ) {
    if ( $u$  is a source operand of a copy operation) {
       $ret1 \cup= \text{step1}(\text{this copy operation})$ ;
    } else {
       $needValue = (u \text{ is a comparison to a constant}) ? \text{false} : \text{true}$ ;
      for (each  $e \in \text{edge}(s)$  from  $u$ )
         $ret1 \cup= \{\text{value range before the operation } u, \text{ value range for the edge } e, \text{ needValue}\}$ ;
    } }
  return  $ret1$ ;
}

 $\text{step2}(set)$  {
  for (each  $u \in set$ ) {
    if ("need of a value" in  $u$  is false) {
      for (each  $v \in set$ ) {
        if ("value range before this operation" in  $v ==$  "value range after this operation" in  $u \ \&\&$ 
             $u \neq v$ ) {
           $set = set - u$ ;
          break;
        } } } }

 $\text{step3}(set, s)$  {
  for (each  $u \in set$ ) {
    if ("need of a value" in  $u$  is false &&
        "value range after this operation" in  $u \supseteq$  value range of the source operand of  $s$ ) {
       $replace = \text{true}$ ;
      for (each  $v \in set$ ) {
        if ( $u \neq v$  &&
            ("value range after this operation" in  $v \cap$  value range of the source operand of  $s$ )  $\neq \emptyset$ ) {
           $replace = \text{false}$ ;
          break;
        } }
      if ( $replace$ ) {
         $newValue = \text{Representative value within the value range of the source operand of } s$ ;
        if (Computation cost of  $newValue <$  Current computation cost)
          Replace the source operand of  $s$  with  $newValue$ ;
         $\text{step4}(s)$ ;
        return true;          // Success
      } } }
  return false;          // Failure
}

 $\text{step4}(s)$  {
  if (The source operand of  $s$  is computed based on the result of CC) {
     $replace = \text{true}$ ;
    for (each  $u \in \text{uses of } s$ ) {
      if (CC is broken between  $s$  and  $u \parallel u$  cannot be replaced with an expression of using CC) {
         $replace = \text{false}$ ;
        break;
      } }
    if ( $replace$ ) {
      for (each  $u \in \text{uses of } s$ ) {
        Replace  $u$  with the expression of using CC;
        Remove  $s$ ;
      } } }
}

```

Fig. 20. General instruction simplification algorithm for our idiom recognition.

```

pos = CLCL(src1, src2, i, j, len);          — (1)
// pos has the position in src1 where the difference occurred.

if (pos < len){
    // Different character found.
    diff = (src1[pos] - src2[pos-i+j]);      — (2)
} else {
    // Two arrays are equal within len.
    diff = 0;
}

```

Fig. 21. Pseudocode for the result of applying idiom recognition to Figure 18.

```

// Input: v is a target variable we will analyze.
// Output: True, if v is used for “comparisons with 0”. False, otherwise.
AnalyzeAllUses(v){
    for (each u ∈ uses of v){ // using DU-chain
        if (u is the source operand of a copy){
            if (AnalyzeAllUses(the destination operand of the copy) == false) return false;
        } else if (u is not a source operand for “comparison with 0”){
            return false;
        }
    }
    return true;
}

```

Fig. 22. Algorithm for analyzing the condition of the instruction simplification.

```

for (;){
    do i++; while(i < end && c[i].compareTo(pivot) < 0);
    do j--; while(j > start && c[j].compareTo(pivot) > 0);
    if (j <= i) break;
    tmp = c[i]; // swap c[i] and c[j]
    c[i] = c[j];
    c[j] = tmp;
}

```

Fig. 23. Example of the core loop of a quicksort program.

```

CC = CLC(src1, src2, i, j, len);          — (1)
// Condition Code (CC) will have:
// 0 : Operands equal
// 1 : First operand low
// 2 : First operand high
diff = (CC == 0) ? 0 : (CC == 1) ? -1 : 2; — (2)

```

Actual assembly code of (2) on System z

IPM	R1	# R1 = PSW (Program Status Word)
SLL	R1, 2	# R1 <<= 2 (Set the sign flag if CC is 2)
SRA	R1, 30	# R1 >>= 30 (R1 has 0, 1, or -2)
LCR	R1, R1	# R1 = -R1 (R1 has 0, -1, or 2)

Fig. 24. Pseudocode of the result of applying the instruction simplification to Figure 21.

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	7E	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)

Fig. 25. Example of exploiting the TROT instruction.

the output of compareTo is immediately used by a compare instruction. In this case, we do not need to convert the CC to any particular value for the result.

3.7. Code Generation

In this section, we describe the generated code using the hardware-assist instructions on System z, the SIMD instructions on System p, and algorithm transformations on System z.

3.7.1. Generated Code Using Hardware-Assist Instructions. This section describes the generated code using hardware-assist instructions on the IBM System z for the idioms in Table I. 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 I on IBM System z, searching for delimiters. We describe the generated code for the second to fifth idioms in Table I.

Converting character codes. The IBM System z has the following four instructions for simple character conversions [IBM Corp. 2004]:

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 25 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 25(b) shows the function table. In this example, we assume that 0×80 signifies the exit condition. We choose an exit value in the range where the loop exits (0×80 through $0 \times FF$ and $0 \times 0D$ in this example). We also use

Fig. 26. VMX instructions corresponding to the TRT instruction of System z.

the wild-card node boottable to handle flexible if-statements, such as the example of the TRT instruction as shown in Figure 16.

String operations. We can convert loops for copying memory, for filling memory, and for comparing memory into special instructions on IBM System z. A loop copying memory can be converted to use the MOVE (MVC) instruction [IBM Corp. 2004]. A loop filling memory can be converted to use 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 [IBM Corp. 2004]. A loop for comparing memory can be converted to the COMPARE LOGICAL (CLC) or the COMPARE LOGICAL CHARACTER LONG (CLCL) instructions [IBM Corp. 2004].

3.7.2. Generated Code Using SIMD Instructions. Using the same idiom recognition framework, we also supported the same idioms described in Table I for IBM System p. As noted, we generate a code sequence corresponding to the special node on each platform. Only the code generation is different from that of IBM System z.

To begin with, we emulated the hardware-assist instructions of IBM System z by using the Vector Multimedia eXtension (VMX, also known as AltiVec or Velocity Engine) instructions [IBM Corp. 2005; Motorola Corp. 1999] that are available on some models of the IBM PowerPC processors [IBM Corp. 2013b]. 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 used the instruction set for “sixteen 8-bit values”.

As an example, we describe how we emulated delimiter searches by using VMX instructions in Figure 26. We converted 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 16(c). This implementation effectively evaluates the following if-statement for 16 characters in one step.

if ((BitVec[ch >> 3] << (ch & 7)) >= 0 × 80) break;

Fig. 27. Optimize the conversion of integers to strings.

3.7.3. Generated Code Using Algorithm Transformations. For converting integers to strings, we implemented the following two idioms.

- countDigits*. Count the digits of the integer using “divide by 10”.
- intToString*. 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 instead use a binary search as shown in Figure 27(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 it with a code sequence using two special instructions on the IBM System z. We use the CONVERT TO DECIMAL (CVD) and the UNPACK UNICODE (UNPKU) instructions [IBM Corp. 2004] as shown in Figure 27(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.

3.8. Adding a New Idiom

In this section, we show how we can add a new idiom. Our idiom recognition can support any idiom that is represented in such graphs as shown in Figure 5. When we want to add a new idiom, we just need to prepare a graph for this idiom and the respective code pattern that should be generated for the new idiom. Figure 28 shows an example of adding a new idiom and its optimized result using the “memory and” operation. Figure 28(a) and (b) show the AST and CFG representations of the idiom, respectively. Figure 28(c) shows the pseudocode for the idiom. As one can see, it performs an AND operation with two array elements and stores the result into an array with a specific length. System z has the NC instruction, which performs the same operation as Figure 28(c). Figure 28(d) shows the definition of a new idiom for (a) and (b). Figure 28(e) shows its optimized result. We just need to prepare (d) and (e) for this idiom. We transformed this input program into the intrinsic AndMem we defined. In the code generation phase, machine instructions will be generated for this intrinsic based on each architecture.

Fig. 28. Example of adding a new idiom and its optimized result.

In our current implementation, both idioms and code patterns are embedded in our compiler. We are planning to represent both of them in a configuration file in the future. In the code patterns, we can describe any code sequence, including vectorization, SIMDization, or algorithm transformation. We have already supported two idioms, countDigits and intToString in Table I, for algorithm transformations as shown in Figure 27. In addition, we already supported an idiom, findbytes, for SIMD instructions on the IBM System p as shown in Figure 26.

Regarding the graph transformations described in Section 3.4, they are also general transformations that are common across every idiom. They are derived from differences between exact matching and topological embedding. As we mentioned in Figure 1, TE can detect any candidate whose graph: (1) includes an additional node and/or (2) includes the nodes even though they appear in a different order from the idiom. Store replication copies the store node outside of the loop to nullify the store node in the candidate input graph. By using this transformation, all uses of the original stores are enclosed within the loop and thus the store can be ignored. Partial peeling and the

The AST is the same as that of the idiom AndMem, but the CFG is different. In this case, the compiler will perform our partial peeling to align the loop entry to the idiom. Figure 29(d) is after the partial peeling. Our idiom recognition will convert the dashed box in (d) to (e).

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) [Sun Corp. 2013] 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 when running the JCK.

To evaluate the performance improvements, we used microbenchmarks for J2SE class files, an IBM XML parser, SPECjvm98, and SPECjbb2000. For the XML parser, we measured thirteen different XML documents. 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 the 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. Use exact pattern matching loop recognition. This compares each IL node in a loop to a predefined template. If all of the IL nodes in the loop match the predefined template, it will convert the loop to faster code. In order to find as many candidates as possible, we first performed as many of the traditional optimizations as possible including 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. However, for delimiter searches, it only handles a single if-statement.

New idiom. Use the algorithm described in this article in addition to the baseline. We performed the pattern matching loop recognition in the baseline algorithm first and then applied our algorithm.

New idiom + simplification. Use our instruction simplification described in Section 3.6 in addition to the “new idiom”.

Disable all. Disable both the pattern matching loop recognition and our algorithm.

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

4.1. Coverage in the JCK API Tests

Figure 30 shows how many loops we converted on IBM System z for the JCK API tests, which invokes many variants of the methods in the J2SE class library. Because the class library is used frequently in Java programs, that coverage is very important. The JIT compiler tried to optimize all of the innermost loops (3,724,925 loops). The topological embedding found 29.5% of them, and our algorithm finally converted 28.4% 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 83% more candidates ($= (29.5/16.1) - 1$) and ultimately converted 76% more candidates ($= (28.4/16.1) - 1$). Our instruction simplification

Fig. 30. Coverage of our approach for 3,724,925 loops that the JIT compiler tried to optimize in JCK API tests.

Fig. 31. Performance improvements of two microbenchmarks (compareTo and readLine) for the J2SE class library on IBM System z. For readLine, we did not find any opportunities for instruction simplification.

successfully transformed 0.2% of all of the innermost loops. The label “failure” (1.1%) means that our TE algorithm found a candidate, but that our graph transformations failed to transform it.

We still have two areas for further improvements. We want to create new idioms to convert more noncandidates (from the remaining 70.5%). We can also create new graph transformations to convert some of the current failures (1.1%). We are investigating several transformation failures, but we have not yet found any cases transformable by the compilers. Those loops include additional nodes that have data dependences upon the values of an array. Thus, we cannot separate those nodes from the original loop by using loop distribution or code motion techniques.

4.2. Performance Improvements

Figure 31 shows the performance improvements of the microbenchmarks 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 the store nodes described in Section 3.4.2 are needed, respectively. For the method `compareTo`, we also measured the performance improvements of the instruction simplification described in Section 3.6. As can be seen, we obtained good performance improvements for these methods. This graph also shows that using a complex instruction reduces the performance if the length is short. Therefore, prefiltering for excluding such a rarely iterated loop is important.

In addition, the instruction simplification was very effective for the method `compareTo`. For short strings of characters, the idiom recognition reduces the performance

Fig. 32. Performance improvements of XML parser using special hardware-assist instructions on System z. We did not find any opportunities for the instruction simplification for the XML parser.

Fig. 33. Performance improvements of XML parser using VMX instructions on IBM System p. We did not find any opportunities for the instruction simplification for the XML parser.

for this method. However, the instruction simplification mitigates this performance degradation. For memory copy or memory initialization, we know the length to be processed before executing the core loop, because the length is given in the program. In contrast, for memory compare or memory search, it is very difficult to estimate the length to be processed, because it depends on the values in memory. For example, in Figure 18, we know the specified length “len”, but the actual processed length may be shorter, because this loop exits if any two array elements are different.

Figure 32 shows the performance improvements for the XML parser over our baseline on IBM System z. Because we did not find any opportunities for the instruction simplification for the XML parser, we did not measure the performance of “new idiom + simplification”. In Figure 32, the x -axis shows 13 XML documents. The labels refer to the file sizes. Our approach improves the 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. In summary, our approach improves performance by 54% on average and by up to 122% (2.22x). Since parsing XML documents is done quite often in Web applications, this result is very significant in the real world.

To see the effectiveness of our approach on IBM System p, we again measured the performance improvements of the XML parser. All the experiments were conducted on a BladeCenter JS20 (PowerPC 970FX 2.2 GHz with 1GB of RAM), and running Linux. Figure 33 shows the performance improvements for the XML parser over our baseline

Fig. 34. Average search length of the delimiter search loops replaced by our approach.

Fig. 35. Performance improvements of SPECjvm98.

on IBM System p. In summary, our approach improves performance by 11% on average and by up to 41%.

Figure 34 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 the performance for long data blocks, it may degrade the performance for very short data blocks because of its startup costs. Figure 34 shows the pattern of the performance differences in parsing the XML files in Figure 32.

Figure 35 and Figure 36 show performance improvements for SPECjvm98 and SPECjbb2000 on IBM System z, respectively. We did not see significant improvements in comparison to the XML parser results. This is because many hardware-assist instructions in the IBM System z are targeted at text processing. There are fewer opportunities in these benchmarks than in the XML parser.

The instruction simplification was essential for the “db” benchmark in SPECjvm98. This benchmark uses the method `compareTo` in a sort operation. As we mentioned in the explanation of Figure 31, converting this method harms performance for short strings. The instruction simplification avoided this drawback. It is also effective in SPECjbb2000. This benchmark also uses the method `compareTo` in a sort operation.

Fig. 36. Performance improvements of SPECjbb2000.

Table II. 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%

4.3. Compilation Time

We have two filters to reduce the compilation time by excluding:

- rarely executed methods;
- idioms unlikely to be matched.

Recent JIT compilers use multiple optimization levels [Suganuma et al. 2001], which are controlled by the hotness of each method. Our idiom recognition algorithm is performed only at the higher optimization levels.

As we mentioned in Figure 3, we exclude those idioms which are unlikely to be matched against the input loop to limit the extra compilation time. We can consider the nodes of an idiom, and if a graph is missing any of these nodes, we already know that no topological embedding exists. For each idiom and graph, we create a bit-vector whose bits represent the opcodes. We compare the bit-vector of every idiom graph with that of the input graph to exclude those idioms which are unlikely to be matched. This minimizes the number of candidate input graphs passed to the topological embedding algorithm. In our experiment, we excluded 90% of the idioms with this filter. If an idiom is more complex, this filter will more effectively exclude the unmatchable idioms. This is because a complex idiom has many characteristics that we can use in filtering. Therefore, we think that the compilation-time increase would change little 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 System z, as shown in Table II. 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. Note that the compilation-time of the instruction simplification is smaller than 0.001%.

5. CONCLUDING REMARKS

We devised 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 System z and VMX instructions on some models of the IBM System p. Our new approach uses a topological embedding algorithm to detect an idiom pattern from the input 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 framework has two key 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.

In addition, we proposed an instruction simplification for some specific idioms. This optimization analyzes all of the usages of the output of the reduced code from the idiom recognition. If we find that we do not need an actual value for the output but only a value in a subrange, then we can assign a certain value in the subrange to the output and thus improve the performance.

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 was to see how many more patterns we could detect over the previous approach. The second one was to see how much more performance improvement we could 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 SPECjbb2000. In summary, relative to a baseline implementation using exact pattern matching, our algorithm converted 76% more loops in the JCK tests. We also observed significant performance improvement of the XML parser by 54%, of SPECjvm98 by 1.9%, and of SPECjbb2000 by 4.4% on average on a z9. Finally, we observed that the JIT compilation times increase by only 0.32% to 0.44%. The instruction simplification contributed to improving the performance for the “db” benchmark in SPECjvm98 and SPECjbb2000.

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.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their helpful comments. 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, Erik Charlebois, Joran Siu, Pramod Ramarao, and Marcel Mitran of the IBM Toronto Laboratory for helpful discussions. We also thank IHCS for supplying support in the presentation of this article.

REFERENCES

- AHO, A. V., GANAPATHI, M., AND TJANG, S. W. K. 1989. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.* 11, 491–516.

- AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compiler Principles, Techniques, and Tools*. Addison-Wesley.
- BLUME, W. AND EIGENMANN, R. 1994. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In *Proceedings of the International Conference on Parallel Processing*. 233–238.
- CLARK, N., BLOME, J., CHU, M., MAHLKE, S., BILES, S., AND FLAUTNER, K. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, Washington, DC, 272–283.
- COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A united lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, New York, 238–252.
- FU, J. J. 1997. Directed graph pattern matching and topological embedding. *J. Algor.* 22, 2, 372–391.
- GRCEVSKI, N., KIELSTRA, A., STOODLEY, K., STOODLEY, M., AND SUNDARESAN, V. 2004. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the Virtual Machine Research and Tech Symposium*. 151–162.
- HE, J., SNAVELY, A., VAN DER WIJNGAART, R., AND FRUMKIN, M. 2011. Automatic recognition of performance idioms in scientific applications. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS'11)*. 118–127.
- IBM CORP. 2013. IBM mainframe. <http://www-03.ibm.com/systems/z/>.
- IBM CORP. 2013. IBM powerpc architecture. <http://www-03.ibm.com/technology/ges/semiconductor/power/powerpc.html>.
- IBM CORP. 2004. z/Architecture principles of operation (sa22-7832-03). IBM, New York. <http://www-01.ibm.com/support/docview.wss?uid=pub1sa22783203>.
- IBM CORP. 2005. Powerpc microprocessor family: Vector/simd multimedia extension technology programming environments manual. <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/FBFA164F824370F987256D6A006F424D>.
- INAGAKI, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 2003. Stride prefetching by dynamically inspecting objects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. ACM Press, New York, 269–277.
- KAWAHITO, M., KOMATSU, H., MORIYAMA, T., INOUE, H., AND NAKATANI, T. 2006. A new idiom recognition framework for exploiting hardware-assist instructions. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 382–393.
- KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2004. Partial redundancy elimination for access expressions by speculative code motion. *Softw. Pract. Exper.* 34, 11, 1065–1090.
- KNOOP, J., RUTHING, O., AND STEFFEN, B. 1994a. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst.* 16, 4, 1117–1155.
- KNOOP, J., RUTHING, O., AND STEFFEN, B. 1994b. Partial dead code elimination. In *Proceedings of the International Conference on Programming Language Design and Implementation*. ACM Press, New York, 147–158.
- LARSEN, S. AND AMARASINGHE, S. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, 145–156.
- LEUSCHEL, M. 2004. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.* 26, 3, 413–463.
- LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.
- METZGER, R. 1995. Automated recognition of parallel algorithms in scientific applications. In *Proceedings of the Workshop on Plan Recognition at International Joint Conference on Artificial Intelligence (IJCAI'95)*.
- MOTOROLA CORP. 1999. AltiVec technology programming interface manual. <http://www.freescale.com/les/32bit/doc/ref manual/ALTIVECPIM.pdf>.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Fransisco, CA.
- OLSHANOWSKY, C., SNAVELY, A., MESWANI, M., AND CARRINGTON, L. 2010. Pir: Pmac's idiom recognizer. In *Proceedings of the 39th International Conference Parallel Processing Workshops (ICPPW'10)*. 189–196.
- PINTER, S. S. AND PINTER, R. Y. 1994. Program optimization and parallelization using idioms. *ACM Trans. Program. Lang. Syst.* 16, 3, 305–327.
- POTTINGER, B. AND EIGENMANN, R. 1995. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing (ICS'95)*. ACM Press, New York, 444–448.

- SATO, H. 2001. Array form representation of idiom recognition system for numerical programs. In *Proceedings of the Conference on Array Processing Languages (APL'01)*. ACM Press, New York, 87–98.
- SHIN, J., HALL, M., AND CHAME, J. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*. IEEE Computer Society, Washington, DC, 165–175.
- SIEGEL, T. J., PFEFFER, E., AND MAGEE, J. A. 2004. The ibm eserver z990 microprocessor. *IBM J. Res. Dev.* 48, 3–4, 295–309.
- SUGANUMA, T., OGASAWARA, T., KAWACHIYA, K., TAKEUCHI, M., ISHIZAKI, K., KOSEKI, A., INAGAKI, T., YASUE, T., KAWAHITO, M., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 2004. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM J. Res. Devel.* 48, 5/6, 767–795.
- SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. 2000. Overview of the ibm java just-in-time compiler. *IBM Syst. J.* 39, 1, 175–193.
- SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2001. 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*. ACM Press, New York.
- SUN CORP. 2013. The java compatibility kit (jck). <https://jck.dev.java.net/>.
- SUN CORP. 2006. The api specification of the class java.lang.string. <http://java.sun.com/javase/6/docs/api/java/lang/String.html>.

Received September 2012; revised March 2013; accepted March 2013