An Object-Oriented Programming Model for Processing-in-Memory Computing in Java Language

Wanhong Huang, Tomoharu Ugawa

Processing-in-memory (PIM) emerges to alleviate performance and power efficiency bottlenecks caused by data movement between CPU and memory. UPMEM, which is a publicly available PIM implementation,the problems in its development are (1) no object-oriented support for PIM device code writing, (2) algorithms cannot be written in a single program, and (3) PIM devices are not transparent in algorithm writing. Consequently, PIM development is error-prone and low-productive, and PIM applications suffer from low maintainability, flexibility, and scalability. The main goals of our research are (1) to write algorithms in one program in a single object-oriented language and (2) to make the PIM device transparent in algorithm writing. To this end, we propose a programming model in which the objects in different PIM memory spaces are transparent to algorithms' writing. This model is based on the distributed object model. We designed a lightweight Java Virtual Machine (JVM) for the UPMEM processor and a Java library to implement this model. After that, we conducted a preliminary experiment on a PIM-compliant in-memory database application.

1 Introduction

In recent years, Processing-in-memory (PIM) has emerged as an innovative computer architecture to tackle the performance and energy efficiency bottlenecks caused by data movement in traditional computer architectures. Traditional processor-centered architectures necessitate data movement between memory and the processor's cache for computation, incurring performance and energy overhead. To mitigate this overhead, PIM places in-memory processors inside the memory. Each in-memory processor is coupled with a part of the physical memory. For the CPU, by delegating parts of computations to in-memory processors, data movements can be reduced.

A search tree is one of the applications that can benefit from PIM architectures [4]. In Figures 1 and 2, we illustrate the process of searching for a tree node with a key of 25 under a traditional computer architecture and a PIM architecture, respectively. In traditional architectures, searching in a large tree can cause numerous memory load requests triggered by cache misses, resulting in ex-

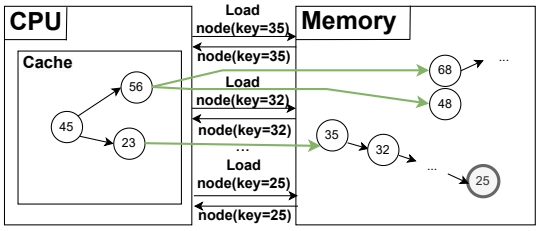

Fig. 1 Binary Search Tree Searching in Traditional Computer Architecture

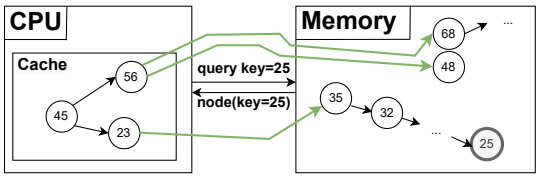

Fig. 2 Binary Search Tree Searching in PIM Architecture

Wanhong Huang, 東京大学情報理工学系研究科, Dept. of Information Science and Technology, The University of Tokyo.

Tomoharu ugawa, 東京大学情報理工学系研究科, Dept. of Information Science and Technology, The University of Tokyo.

```
class TreeNode{
  TreeNode left;
  TreeNode right;
  int leftDpuID;
  int rightDpuID;
  int searchInDpu(int key, int dpuId){
    if(dpuId == 0) return -1;
    copyToDpu(dpuId, 'request_key', key);
    getDPU(dpuId).execute();
    int result =
      copyFromDpu(dpuId, 'result');
    return result;
  }
  public int search(int key){
    /* Algorithm Logic */
    if(key == getKey()) return getValue();
    if(key < getValue()){
      if(getLeft() == null)
        return getLeft().search(key);
      return searchInDpu
        (key, leftDpuID);
    }
    if(key > getValue()){
      if(getRight() == null)
        return getRight().search(key);
      return searchInDpu
        (key, rightDpuID);
    }
  }
}
```

Fig. 3 BST search in PIM (CPU side).

```
int request_key;
int result;
struct tree_node* root;
int main(){
  search(root, request_key);
}
void search(struct tree_node* node, int
    key){
  if(!node) return -1;
  if(node->key == key)
    return node->value;
  if(node->left)
    return search(node->left, key);
  return search(node->right, key);
}
```

Fig. 4 BST search in PIM (in-memory processor side).

pensive time and energy consumption due to data movement. In the PIM architecture, by placing the root and a small number of nodes near it into the CPU's cache while placing other nodes in memory coupled with processors, the CPU can send a single query request and retrieve the result from memory, shifting much of the tree search operations involving numerous memory accesses to the memory side. This transition presents an opportunity to alleviate the performance and energy efficiency bottlenecks caused by data movement.

The development of software for the PIM, however, remains complex, low-productivity, and error-prone due to the absence of a suitable programming model. Presently, PIM software development requires separate programming for the CPU and the in-memory processors, with explicit communication between the CPU and the in-memory processors. Furthermore, the CPU side program must manage where the data stay.

Figures 3 and 4 shows the search algorithm of a binary search tree (BST) in a PIM architecture (for more details, see Section 2. 2). In this search algorithm, the CPU is initially responsible for searching over a small number of tree nodes. Once it reaches a node whose left or right child is `null`, it continues the search for the key inside an in-memory processor that possibly has the node with the key. Such an in-memory processor is identified by the `leftDpuID` or `rightDpuID`. The function `searchInDpu` is the helper function that sends the key to the in-memory processor, launches the search program shown in Figure 4 in the in-memory processor, and retrieves the search result. As shown above, the program must explicitly communicate with the separately developed program for the in-memory processors, and it must also maintain the ID of the in-memory processor that may have desired data.

To address these problems, we present a programming model for Java that enables the development of PIM in a single program without explicit communication logic, leveraging object-oriented features to handle the complexities of PIM development. In our proposed programming model, Java objects are stored in both the CPU and in-memory processors. The in-memory processor's objects are structured in a way that is compatible with

```
class TreeNode{
  TreeNode left;
  TreeNode right;
  public int search(int key){
    if(key == getKey()) return getVal();
    if(key < getKey()){
      if(getLeft() != null)
        return getLeft().search(key);
    }else{
     if(getRight() != null)
       return getRight().search(key);
    }
    return -1;
  }
}
```

Fig. 5 BST search under the proposed programming model.

the CPU objects. Figure 5 presents the search algorithm in the proposed model. In Figure 5, a child of a `TreeNode` can be an object of either the CPU or an in-memory processor. The search algorithm is written in the same way as it is in traditional architectures.

2 Background

2.1 UPMEM

In our work, we use a real-world PIM computer, UPMEM [3]. Figure 6 shows the structure of UPMEM. The UPMEM places multiple general-purpose processing units within a memory. These processors are referred to as `DRAM Processing Units` (DPUs). Each DPU is associated a DRAM chip, comprising the main memory of the computer.

A program running on the CPU can access the memory of a specific DPU by using a dedicated API, `copyToDpu` in Figure 3, provided by the vendor†1. The vendor also provides APIs to install a binary program to DPU and to execute it, which is the `execute` method in Figure 3.

2.2 A PIM Application and Its Programming Challenges

We explain the PIM-compliant BST program in Figures 3 and 4 that we have shown in Section 1.

This application distributes the nodes of a BST to the CPU and DPUs. The cache memory of the CPU stores the root and a small number of

†1 https://sdk.upmem.com/2023.2.0/

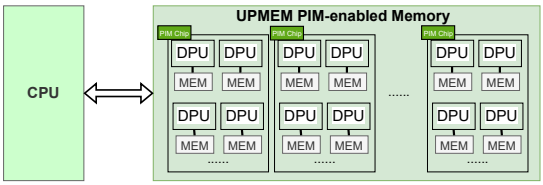

Fig. 6 UPMEM architecture

tree nodes near it, while a larger number of tree nodes are stored in PIM chips, with each of which DPUs are associated. This approach offers the advantages of reducing data movement for the search algorithm, as most search operations can be performed inside the DPU. The search algorithm finds the node with the given key either inside the CPU cache or within one of the DPU. In the former situation, the CPU does not need to access memory. In the latter situation, the CPU accesses memory twice; for sending a query and receiving the result.

In Figure 3, the `searchInDpu` method receives a query key and launches the search program in the specified DPU using the `execute` method. The `copyToDpu` method is responsible for copying the data to a specified global variable in a DPU, and the `copyFromDpu` method is responsible for copying the data back from a specified global variable in a DPU.

In typical PIM development, developers are responsible for writing both algorithm logic and communication logic on the CPU side. Developers must also write algorithm logic for the DPUs in C language separately. Additionally, as shown in Figure 3, when the `search` method reaches a CPU node without left or right nodes, developers face the challenge of continuing the key search in a DPU. More specifically, the developer needs to keep track of which DPU might hold the query key. This information is stored in the variables `leftDpuID` and `rightDpuID`.

3 Programming Model

3.1 Overview

The proposed programming model for PIM is based on a CPU-enteric distributed object model. Figure 7 shows an overview of the proposed model. In this model, each processor holds its objects. We call an object inside a DPU a *remote object*. This

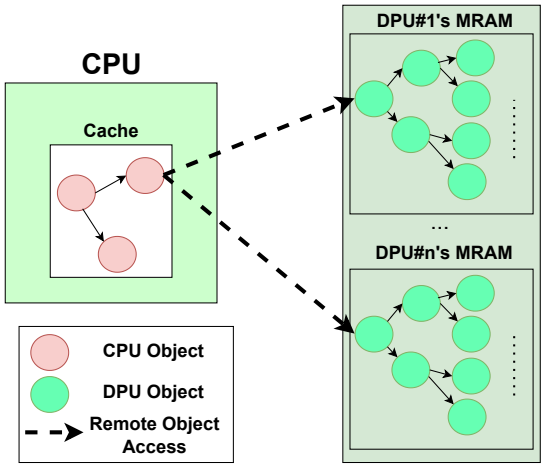

Fig. 7 model overview

model allows the CPU to invoke the methods of remote objects. On the contrary, a DPU accesses only the objects in its own memory space. In this model, any Java method can be executed on either the CPU or a DPU. A method will be executed on the processor where its holder object is located. When the method creates an object, the new object is created in the same memory space as the method holder object. For the first object in a DPU, we take a pragmatic approach: we explicitly create it with specifying the ID of the DPU.

3.2 Proxy

We introduce a special kind of objects on the CPU side, called a *proxy* to enable the CPU to invoke the methods of objects in the same manner as local objects. A proxy is an object that contains the DPU ID and the address of the remote object in the memory space where the remote object is located. A proxy also contains all the methods of the remote object. However, these methods are overwritten to delegate their execution to the DPU that holds the remote object.

3.3 Processor-Dependent Behaviors

In some cases, the programmer wants some methods to behave differently in the CPU and DPUs. We refer to such methods as *processor-dependent methods*.

An example is the method that creates a new node in the BST application, *createNode*. Figure 8

```
class DPUTreeNode extends TreeNode{
  @Override
  public TreeNode createNode
          (int key, int val)
  {return new DPUTreeNode(key, val);}
}
class CPUTreeNode extends TreeNode{
  int height;
  public CPUTreeNode(int key, int val,
       int height){
    super(key, val);
    this.height = height;
  }
  @Override
  public TreeNode createNode(int key, int
       val){
    if(height > 20)
        return UPMEM.createObject
          (allocateDPU(),
          DPUTreeNode.class, new
              Object[]{key, val});

    TreeNode.nodeAmount++;
    return new CPUTreeNode
      (key, val, height + 1);
  }
}
```

Fig. 8 `createNode` for CPU and DPU.

shows the different behaviors in CPU and DPUs. In a DPU, we want the *createNode* method to create a new node in its memory. In the CPU, we want it to create a new node in the appropriate location, in the CPU or in memory of a DPU, depending on the depth of the new node.

A processor-dependent method is implemented by using subclass mechanism. Figure 10 shows the class hierarchy of the tree node-related classes. For a class that has a *processor-dependent method*, we first define an abstract class (`TreeNode`) that has all methods but *processor-dependent methods*. By extending it, we define the classes with *processor-dependent methods* for the CPU (CPUTreeNode) and DPUs (DPUTreeNode). Note that the proxy class should be a subclass of the DPU class.

3.4 Example

In Figure 5, as shown earlier, the search method of the BST for PIM can be implemented in the same way as in a traditional architecture. Figures 8 and 9

```
class TreeNode{
  /* ... */
  public abstract TreeNode createNode(int
      key, int val);
  public void insert(){
    if(k < getKey()){
      if (getLeft() == null)
        setLeft(createNode(k, v));
      else
        getLeft().insert(k, v);
    }else{
      if (getRight() == null)
        setRight(createNode(k, v));
      else
        getRight().insert(k, v);
    }
  }
}
```

Fig. 9 `insert` **method of BST in the proposed model.**

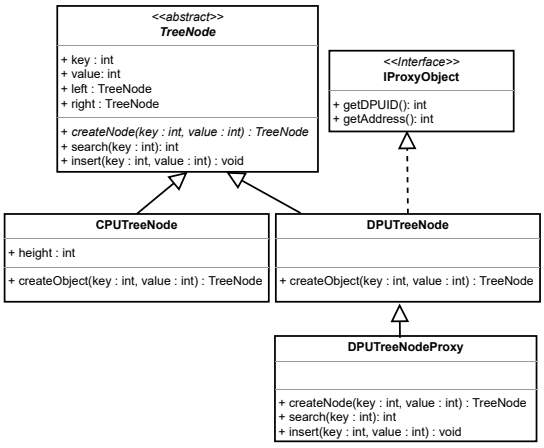

Fig. 10 Classes in the proposed model.

show the `createNode` and `insert` methods of the BST in the proposed programming model. In the `insert` method, it uses the `createNode` method to create a new node in the CPU or a DPU. Because this class has a processor-dependent method, `createNode`, we define it as an abstract method within the class. In addition, we define two classes, namely `CPUTreeNode` and `DPUTreeNode`, to provide implementations for it. The remaining methods, `insert` and `search` are implemented in the abstract class.

When the `createNode` method of a CPU object is invoked and the depth exceeds 20, the algorithm create a new DPU tree node by using the `createObject` API, which we introduce in Seciton 4. Otherwise, it creates a new CPU tree node. This allows most of the tree nodes to be stored inside the DPU. The `allocateDPU` method is a method that chooses a proper DPU to create a new tree node at the programmer's discretion. When the `createNode` of a DPU object is invoked, it creates an object inside the DPU where the object is located.

4 System Implementation

We implemented a prototype system to support the proposed programming model. The system comprises two main components: a Java virtual machine (JVM) for DPUs and a library for the Java VM running on the CPU.

This section demonstrates the implementation of two main components in this system: a JVM for DPUs and a library for the Java program running in the CPU.

4.1 In-Memory JVM

This system runs a lightweight JVM on the DPUs. We refer to it as `in-memory JVM`.

After launching an in-memory JVM, it identifies the method to be executed and its class by using the method and class addresses provided by the CPU through global variables. It also retrieves the arguments through global variables and creates a function frame for the method in the execution stack. Subsequently, a pure interpreter executes the method's bytecodes. On the return from this method, i.e., execution of a returning bytecode like `IRETURN`, the return value is stored in a dedicated global variable `return_value` and finishes execution of the PIM side program. The global variable `return_value` is read by the CPU.

The current JVM implementation is a subset of JVM, with an interpreter-based execution engine that is enough to execute the BST application.

4.2 Java Library

We implemented a Java library for the CPU to cooperate with the programming model. The library has the following three functions.

```
public class RPCHelper{
  /* ... */
  public int getClassAddress(int dpuID,
      String className){
    loadedClassesTable.get(dpuID,
        className);
  }
  public int getMethodAddress(int dpuID,
      String className, String signature){
    dpuMethodsTable.
        get(dpuID,className,signature);
  }
  public void invokeNonstaticMethod
  (int dpuID, String className, String
      signature, Object... args){
    int classAddress =
        getClassAddress(dpuID, className);
    int methodAddress = getMethodAddress
      (dpuID, className, signature);
    copyToDpu(dpuID, 'class_address',
        classAddress);
    copyToDpu(dpuID, 'method_address',
        methodAddress);
    sendArguments(dpuID, args);
    getDPU(dpuID).execute();
  }
  public int getIntReturnValue(){
    return copyFromDpu('return_value');
  }
}
public class DPUTreeNodeProxy extends
    DPUTreeNode{
  @Override
  public int search(int key){
    RPCHelper.invokeNonstaticMethod
      (getDpuID(), "TreeNode",
          "search(I):I", key);
    return RPCHelper.getIntReturnValue();
  }
}
```

Fig. 11 Remote procedure call in proxy Class.

4.2.1 Remote Procedure Call

A remote procedure call (RPC) may occur when a proxy object's method is invoked. Figure 11 presents the pseudo-code for the remote procedure call to the `search` method in the `TreeNode` class.

Every time the CPU makes a remote procedure call of the `search` method, the proxy uses a class name and a method descriptor to locate the specific method and class's addresses under the in-memory processor, along with the arguments that need for the method execution. The `RPCHelper` then obtains the class and the method's addresses under the DPU that contains this object. The CPU holds tables for each DPU: a loaded class table to look up the address of a given class, and a method table to look up the address from a pair of a method signature and a class.

The `RPCHelper` then transfer the class address and method addresses to the DPU global variables `class_address` and `method_address` by using `copyToDpu` method, and lunch the DPU for execution. As the search method has a return value with type `int`, it invokes `RPCHelper.getIntReturnValue()` method to copy the return value back from the DPU, and return this value.

4.2.2 Remote Object Creation

The `createObject` API allows us to create an object of a specified class in a given DPU. First, it sends the class of the object if the class is not yet present in the DPU's memory. Then the API generates an instance of the class in a byte array form and sends it to the DPU's memory. After writing the instance data into the DPU's memory, the library invokes the corresponding `init` method of the sent instance to initialize it in the DPU. After the object completes the initialization in the DPU, the API generates a proxy object at the CPU.

4.2.3 Class Loading and In-Memory Classes Management

The in-memory JVM needs to store Java class structures for executing Java methods. The library provides class loading utilities and loaded classes management for the in-memory JVM. The library is responsible for parsing Java class files, resolving symbols, i.e., method and field names, and creating their internal representation in memory of DPU. All class loading processes are performed by the CPU. The class and method addresses in DPUs are kept in the tables per DPU in the CPU's memory. They are used for RPC as described above.

5 Experimentation

The objective of this experimentation is to evaluate the impact of the proposed programming model on reducing data movement in the BST application under the PIM architecture.

In this experimentation, we used UPMEM, a

real-world PIM architecture computer. Its configuration is shown in Table 1. The JVM running on the CPU was the HotSpot VM of OpenJDK 17.0.1, configured with the following VM options.

```
-XX:+UnlockDiagnosticVMOptions
-XX:+PreserveFramePointer
-XX:+DumpPerfMapAtExit -Xmx131072m
```

5.1 Methodology

In our experiment, we constructed two types of BST. (1)CPU-BST: a BST that uses only the CPU, and (2)PIM-BST: a BST using PIM in the proposed programming model.

In the experiment, to construct a PIM-BST, we split a BST into different parts. We placed the root and the nodes close to it in addresses that are not associated with a DPU, while storing other parts in DPUs' memories. In PIM-BST, the number of nodes on the CPU side was set to 199,159, to fit within the CPU cache. The other nodes are distributed over DPUs so that each DPU can have subtrees whose roots are leaves of the tree in the CPU. We limited the number of nodes in a single DPU to 2,000,000. The number of nodes in the tree was 100,000,000. Under the limitation of the node capacity in a single DPU, we used 66 out of 2560 DPUs. Both the CPU-BST and PIM-BST we constructed were structurally identical and had identical key-value pairs.

Since data movement occurs when an Last Level Cache (LLC) miss occurs, we employed the LLC miss counts as approximations of data movement. We compared the pre-query LLC misses of the `search` methods between CPU-BST and PIM-BST. For each type of BST, we applied 500,000 queries. Note that the queries applied to CPU-BST and PIM-BST were identical.

5.2 Results

Table 2 presents a comparison of LLC missed per-query in the `search` method between CPU-BST and PIM-BST. We obtained approximate LLC miss counts using the `perf` utility in a sampling mode.

The result shows that, in terms of LLC *load* misses per query in the `search` method, PIM-BST outperformed CPU-BST by a factor of 0.13x, while PIM-BST outperformed CPU-BST by a factor of 0.29x in terms of LLC *total* misses. In terms of LLC *store* misses, PIM-BST exhibited 144x the count of CPU-BST. We think the LLC store misses occurred when the RPC mechanism prepared data to send to the DPU, including serialization of the arguments.

This experiment demonstrated that the proposed programming model reduced the LLC misses. Note that we only compared LLC messes, which do not include data movement for communications between CPU and DPU. Nevertheless, data movement for it can be negligible for a large tree because it is a constant while LLC misses per query increases as the height of the tree increases.

6 Discussion And Future Work

6.1 Performance

Though the data movement can be reduced by the proposed programming model under a PIM architecture, the regrettable outcome is that the execution time of Java methods inside the current in-memory JVM is significantly higher than the method execution time under the CPU. The performance of the current in-memory JVM is a bottleneck of the whole system. The low performance of the in-memory JVM can be attributed to two primary factors: (1) the DPU processor operates at a low clock frequency and (2) the in-memory JVM employs a pure interpreter to execute Java methods.

Our future direction to improve the performance of the in-memory JVM includes adopting a Just-In-Time (JIT) compiler. This reduces time consumption in the interpretation loop by compiling the method bytecodes into the native code of the DPU. In addition, enhancing the programming model and in-memory JVM with multithreading to enable parallel computations could enhance the overall system performance for certain applications. Furthermore, strategies such as lazy dispatch and batch dispatch have the potential to reduce the overall communication cost between the CPU and DPU.

6.2 Binary Size Limitation

Each DPU has a limited capacity to store binary programs. As the in-memory JVM's functionality increases, the size of binaries may exceed the DPU's storage capacity. To address this issue, strategies such as splitting the binary into different modules and loading them on demand could be employed.

Table 1 UPMEM configuration.

	Configuration
CPU	INTEL Xeon Silver 4215
PIM Memory	20 DDR4-2400 PIM modules and 160 GB PIM memory
DPU	2560 x @450 MHz
LLC Cache size	11 MB

Table 2 Comparison of Per-query LLC Misses in `search` method

Application	LLC load misses	LLC store misses	LLC total misses
CPU-BST	17.7	0.02	17.9
PIM-BST	2.31	2.88	5.19
PIM-BST/CPU-BST Ratio	0.13x	144x	0.29x

6.3 Static Method Dispatching

The prototype system for the proposed programming model currently supports only invocations of instance methods. An object instance allow us to locate a specific DPU for executing the method.

However, invocation of a static method is a more complicated situation. A static method invocation is not associated with an object reference, the system needs a strategy to determine which DPU should execute the method. One approach is to infer the execution location from the object reference inside the method arguments list. In our programming model, we forced all the object reference in the arguments list to be in the same space, because this programming model does not allow DPUs to access memory of other DPUs'. In the case where there is an object reference in the arguments list, the method must be executed in the processor that holds the object. In the condition that there are no object references in the arguments list, we must introduce a strategy to decide an DPU for executing the method.

7 Related Work

7.1 Tornado VM

Tornado VM [2] is a virtual machine (VM) designed for computations using different kinds of processors (a.k.a., heterogeneous computing) in the Java language. Tornado VM adopts an annotation-based approach to specify which methods can be delegated to a remote processor for execution. Inside the VM, a compilation-based approach is used to delegate computations to remote processors, utilizing OpenCL as the execution backend for remote processor execution. Using this approach, developers can write a single program to achieve high-performance computation using different kinds of processor.

Our work was inspired by the Tornado VM. However, ours has the following differences. (1) We use the in-memory JVM as the execution backend to provide object-oriented support for remote processors. (2) The remote call in our system is based on the proxy.

7.2 SYCL

SYCL [1] is a highly abstract specification for heterogeneous computing in the C++ language. Its primary objective is to enable developers to write a single C++ program that performs computations using different kinds of processor. The SYCL specification provides an execution model, a memory model, and a programming model. The execution model determines the execution order of method calls and where a method should be executed; the memory model describes how to access the memories of different processors and how to maintain data consistency during memory accesses; the programming model outlines how to use SYCL in C++ programming, including computation definition and data parallelization, methods scheduling, exception handling, and management of memory objects including buffers and images.

In our work, we adopt a concept similar to that of writing a single program for heterogeneous computing. However, our model is presently applicable exclusively to the PIM architecture and lacks parallelism and ordered organization of RPCs. On the other hand, employing a JVM as the execution backend enables our model to accommodate more kinds of memory objects.

8 Conclusion

This paper addressed the challenges associated with the development of PIM architectures. While PIM architectures offer benefits in mitigating performance and energy efficiency bottlenecks caused by data movement in computer systems, PIM application development is complex due to the absence of a proper programming model.

We provide a programming model for the PIM architecture. The model allows developers to write PIM software within a single program and eliminates the communication logic in the algorithm description. Through the implementation of a system prototype and its application to a PIM-compliant BST, we demonstrated that our programming model can reduce data movement under a real-world PIM architecture.

Our future work includes implementing the in-memory JVM, optimizing its performance, and minimizing communication costs to make our system better exploit the PIM architecture's performance.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Numbers JP19K11904 and JP22H03566.

References

[1] Alpay, A. and Heuveline, V.: SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL, *Proceedings of the International Workshop on OpenCL*, IWOCL '20, New York, NY, USA, Association for Computing Machinery, 2020.

[2] Clarkson, J., Fumero, J., Papadimitriou, M., Zakkak, F. S., Xekalaki, M., Kotselidis, C., and Luján, M.: Exploiting High-performance Heterogeneous Hardware for Java Programs Using Graal, *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, ACM, 2018, pp. 4:1–4:13.

[3] Gómez-Luna, J., Hajj, I. E., Fernandez, I., Giannoula, C., Oliveira, G. F., and Mutlu, O.: Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System, *IEEE Access*, Vol. 10(2022), pp. 52565–52608.

[4] Kang, H., Zhao, Y., Blelloch, G. E., Dhulipala, L., Gu, Y., McGuffey, C., and Gibbons, P. B.: PIM-Tree: A Skew-Resistant Index for Processing-in-Memory, *Proc. VLDB Endow.*, Vol. 16, No. 4(2022), pp. 946–958.