IVT: An Efficient Method for Sharing Subtype Polymorphic Objects

YU-PING WANG*, Tsinghua University, China
XU-QIANG HU, Tsinghua University, China
ZI-XIN ZOU, Tsinghua University, China
WENDE TAN, Tsinghua University, China
GANG TAN, The Pennsylvania State University, University Park, USA

Shared memory provides the fastest form of inter-process communication. Sharing polymorphic objects between different address spaces requires solving the issue of sharing pointers. In this paper, we propose a method, named Indexed Virtual Tables (IVT for short), to share polymorphic objects efficiently. On object construction, the virtual table pointers are replaced with indexes, which are used to find the actual virtual table pointers on dynamic dispatch. Only a few addition and load instructions are needed for both operations. Experimental results show that the IVT can outperform prior techniques on both object construction time and dynamic dispatch time. We also apply the proposed IVT technique to several practical scenarios, resulting in the improvement of overall performance.

CCS Concepts: · Software and its engineering → Cooperating communicating processes; Object oriented languages; Polymorphism.

Additional Key Words and Phrases: Shared memory management, Inter-process communication, Virtual table, Preprocessors & parsers

ACM Reference Format:
Yu-Ping Wang, Xu-Qiang Hu, Zi-Xin Zou, Wende Tan, and Gang Tan. 2019. IVT: An Efficient Method for Sharing Subtype Polymorphic Objects. Proc. ACM Program. Lang. 3, OOPSLA, Article 130 (October 2019), 22 pages. https://doi.org/10.1145/3360556

1 INTRODUCTION

Shared memory allows two or more processes to share a given region of memory and is the fastest form of inter-process communication (IPC) because the data does not need to be copied [Stevens 1992]. Compared with sockets, shared memory is considered less scalable, since the communicating processes must run on the same machine. The emergence of distributed shared memory (DSM)
techniques [Karimi and Sharifi 2006; Tanenbaum 1995] has lifted this limit. But there are still more issues to address.

The usage of pointers is one of them. Common IPC techniques aim to transmit plain byte stream; therefore serialization techniques [Herlihy and Liskov 1982] must be employed to transmit structured data. Pointers are serialized into the form of plain bytes before transmission. Although shared memory allows data to be shared directly, pointers cannot be shared, since they are meaningless to another process. This issue can be solved by manually replacing pointers with relative offsets [Boost C++ Libraries 2018a], but implicit pointers introduced by subtype polymorphism (i.e., virtual table pointers) are hard to share since they are transparent to developers.

Subtype polymorphism is a feature that is supported by many object-oriented languages. It allows objects of different classes to be called by the same API defined by a common base class and behave differently. Without causing misunderstanding, we use polymorphism to refer to subtype polymorphism in this paper.

In order to support polymorphism, most C++ compilers use virtual tables. By using virtual tables, each polymorphic object contains at least one pointer that points to the virtual table of its class type. Two objects of the same class point to the same virtual table. However, the address of a virtual table is different across different processes. Therefore, directly sharing polymorphic objects would lead to invalid virtual table pointers at processes other than the one that created the objects.

Hajj et al. [Hajj et al. 2017] provide two techniques to solve this problem. But they introduce performance degradation to object construction and dynamic dispatch, which makes them less suitable for scenarios where objects are constructed and accessed constantly.

In this paper, we propose an alternative solution, named Indexed Virtual Tables (IVT for short). The basic idea is similar to Bacon et al. [Bacon et al. 2002], although for a different context. They proposed an index-based object model for Java to improve the space and time performance of synchronization associated routines. Our IVT technique also replaces virtual table pointers with indexes, but is designed to share polymorphic objects efficiently. On object construction, virtual table pointers are replaced with indexes of a virtual table pointer array. Dynamic dispatch is performed by finding the actual virtual table pointer using the stored index. Only a few addition and load instructions are needed for both operations.

The main contributions of this paper include:

1. We propose a new technique for sharing polymorphic objects between processes, namely IVT.
2. We implement this technique, and compare it with two prior state-of-the-art techniques.
3. We perform a detailed performance evaluation using micro-benchmarks. The results show that IVT outperforms prior techniques on both object construction and dynamic dispatch.
4. We apply this technique to several scenarios, including XML transformation, facial recognition, and robotic middleware, and show significant performance improvements.

The rest of this paper is organized as follows. Section 2 motivates this work and analyses why sharing polymorphic objects is necessary. Section 3 describes the design and implementation of the IVT technique. Section 4 evaluates the IVT technique and compares the result with prior works. Section 5 discusses some usage concerns and Section 6 discusses related work. Finally, we conclude in Section 7.
2 MOTIVATION

2.1 Motivating Scenario

Our work is motivated by a surveillance scenario. Suppose a number of cameras are set up around a building we want to secure. The captured videos are sent to a central unit and a facial recognition algorithm is used to record the time when each person enters or leaves.

One solution is to run a number of threads within the same process. The facial features of every known person is loaded first, and each worker thread is responsible for handling the video data from one camera. However, putting all code into one address space is not a robust design as it is susceptible to software failures. In particular, facial recognition algorithms typically need a large amount of memory for performance, and exceptions can happen when there are not enough resources. Besides, because of hardware failure, a camera can generate hard-to-recognize images, which can trigger exceptions as well. When these exceptions are not well handled, the whole system crashes since all code is in one address space.

Another solution is to run a number of isolated processes and each worker process is responsible for handling the video data from one camera. A master process monitors the status of these processes. When an exception happens, only one worker process possibly crashes and the master process can detect the crash and restart its work with another worker process. Unfortunately, each worker process has to load the facial features by itself. This procedure consumes additional time and memory space. What makes things worse is, when a new person is added to the known list, every worker process has to stop and load the facial feature of the new person.

How about sharing the loaded facial features across these worker processes? The shared memory mechanism can be employed to realize this. Furthermore, if videos are also shared within shared memory, a worker process no longer needs to be tied to a specific camera, and load balancing among these processes can be performed in real time. Although a shared object corrupted by one process may cause failure in the other processes, the probability is much lower than the one in the solution of using multiple threads since not all data is shared. Unfortunately, in the OpenCV library [OpenCV team 2018], which is the most commonly used open-source computer vision library, the class that is used to express facial features contains virtual functions. A facial feature object is therefore a polymorphic object, and sharing polymorphic objects generally causes trouble.

2.2 Issue of Sharing Polymorphic Objects

Polymorphism is supported by many object-oriented languages. It allows programmers to manipulate a collection of objects with different types by using the API that their common base class defines, and objects can behave differently if derived classes provide different implementations of the same API.

Many modern C++ compilers realize polymorphism with virtual tables. Each polymorphic class is associated with a virtual table. A virtual table contains pointers to type information, and pointers to virtual functions. A virtual function is a member function annotated with the C++ keyword virtual, which enables this function to have different implementations in different derived classes.

Each object with polymorphism contains at least one virtual table pointer. Multiple virtual table pointers within one object only occurs in the case of virtual inheritance and multiple inheritance. When a virtual function is called, these virtual table pointers are used to find virtual tables, and then virtual function pointers. This process is called dynamic dispatch.

In our motivating scenario, a CascadeClassifier class is designed to abstract object detection algorithms in the OpenCV library. It supports different object detection algorithms by employing polymorphism. Different object detection classes inherit the CascadeClassifierImpl class, and implement their own algorithms. The CascadeClassifier class contains a CascadeClassifierImpl class.
pointer. In this way, the \textit{CascadeClassifier} class provides an interface that is independent of concrete algorithms.

However, since \textit{CascadeClassifierImpl} is a polymorphic class, issues would arise when its objects are shared across processes. It is because a virtual table is usually arranged at a particular address in the global region of a process by compilers, and virtual table pointers are native pointers; this makes virtual table pointers stored in an object meaningless to another process. Figure 1 shows an example. For the same reason, libraries supporting shared memory, such as Boost, explicitly forbid sharing polymorphic objects [Boost C++ Libraries 2018c].

![Fig. 1. Example showing the issue of sharing virtual table pointers. The virtual table pointer in $P_C$ is meaningless to $P_R$.](image)

Hajj et al. [Hajj et al. 2017] proposed two techniques to solve this issue, namely virtual table duplication and hashing-based dynamic dispatch. The first technique duplicates virtual tables in a fixed-address region of each process involved in sharing. The duplicated virtual tables make the native pointers to virtual tables reusable across processes, as shown in Figure 2(a). Following Hajj et al. [Hajj et al. 2017], we will refer to this technique as DVT (Duplicate Virtual Table). The second technique totally gets rid of virtual table pointers, and realizes dynamic dispatch by looking up in a global virtual table using a hash number stored in each object, as shown in Figure 2(b). Following Hajj et al. [Hajj et al. 2017], we will refer to this technique as GVT (Global Virtual Table).

Both of these techniques can enable sharing polymorphic objects, but they introduce performance degradation. The DVT technique introduces performance degradation to object construction. When an object is created, DVT first checks whether it is created within a shared memory region. This is done by looking up in a table (called the Range Table). This table records all shared memory regions. Then, if the object is created within a shared memory region, DVT needs to check whether the virtual table of the object has been duplicated and finds where the duplicated virtual table is. This is done by looking up in another table (called the DVT Look-up Table). This table records pairs of mangled class names and addresses of duplicated virtual tables. Besides, the most important
limitation of DVT is that it needs a fixed address region in each process. If any one of the involved processes cannot duplicate virtual tables at the same address as others do, the DVT technique cannot be realized. Although the virtual memory space provided by modern operating systems is large, it is still difficult to avoid conflicts completely, especially when there are multiple processes involved in sharing.

The GVT technique introduces performance degradation to dynamic dispatch. By using GVT, all polymorphic classes are transformed to non-polymorphic classes. When a virtual function is called, GVT first accesses the hash code of the class stored in the object. Then, a hash key is generated by combining the hash code and the mangled function name. The hash key is used to look up in the global virtual table.

Overall, neither DVT nor GVT meet the demand of our motivating scenario, where multiple processes are involved in sharing and objects are constructed and accessed continuously. To overcome the limitations of DVT and GVT, we propose a solution called Indexed Virtual Tables (IVT for short).

3 INDEXED VIRTUAL TABLES

3.1 Overview

The main idea of our IVT technique is to replace the virtual table pointers with indexes. Figure 3 shows how this technique works on the previous example in Figure 1. All processes maintain a virtual table pointer array (VTPA) in their own global region. This VTPA contains all virtual table pointers involved in sharing and it is constructed on process entry. In the object creation process $P_C$, after an object $obj_A$ of class A is constructed, its virtual table pointers are changed to the corresponding indexes in the VTPA. In the object receiving process $P_R$, the index is used to find the real virtual table pointer from VTPA. Note that there is one VTPA in each process. The meaning and order of the elements in these VTPAs are the same, but the values of the elements may differ. Therefore, the same index (e.g., 2) enables our system to find different virtual table pointers in $P_C$ (0xbe00) and $P_R$ (0xce00). As a result, object $obj_A$ can be used in both $P_C$ and $P_R$.

When using the IVT technique, recompilation is needed. The overall compilation flowchart is shown in Figure 4. Our IVT technique assumes that programmers know all information of the
classes that are involved in sharing. Thus, they can provide a simple specification of the involved classes. We will discuss this assumption in detail later in Section 5.

Fig. 4. The IVT compilation flowchart.

Each source file (e.g., src.cpp in Figure 4) is compiled into the corresponding object file (e.g., src.o) with IVT enabled and the user-define specification. We will explain this routine in Section 3.3. The specification itself is transformed into a source file (e.g., spec.cpp) with IVT and later compiled normally. We will explain this routine in Section 3.2. All object files are linked into the executable normally. This routine can be done automatically by changing the compiler and linker scripts.

For implementation convenience, we employ LLVM Clang [Lattner and Adve 2004; LLVM 2018] as the compiler framework. For better understanding, we will describe IVT transformations using LLVM-specific terminologies and patterns. But our proposed IVT technique can be implemented in other compiler frameworks.
3.2 Specification and VTPA

Figure 5 shows an example specification and the corresponding VTPA. The specification lists all classes whose objects may be shared; we call them *shared classes*. Typically an object of a class contains only one virtual table pointer, and therefore only one slot is needed for that pointer in the VTPA; an example is class A in Figure 5. However, when virtual inheritance (e.g., class B and C in Figure 5) or multiple inheritance (e.g., class D in Figure 5) is used, an object may contain multiple virtual table pointers, and therefore multiple slots are needed for those pointers in the VTPA.

![Fig. 5. Example of programmer annotated specification and the generated VTPA.](image)

During **process entry**, when global variables are initialized, the VTPA should be initialized with virtual table pointers. To get information about how to initialize the VTPA, during compilation our system looks for constructors in shared classes, and records the symbols of virtual table pointers. With the information of those symbols, we can generate a global function that initializes the VTPA with those symbols filled in. During static or dynamic linking, these symbols are mapped into actual addresses.

We next discuss how virtual table pointers are located in constructors of shared classes. In particular for LLVM IR, the following pattern in the constructor code is used to identify a virtual table pointer. Figure 6(b) shows an example. The gray part in Figure 6(b) is the symbol of a virtual table pointer.

- It is the first operand of a *store* instruction.
- The second operand of the *store* instruction is the result of a *bitcast* instruction.
- Its full name contains the mangled name of a shared class.

The process of identifying virtual table pointers is not affected by LLVM’s optimization since it happens before LLVM’s optimization passes. It can reliably identify all virtual table pointers in LLVM. After all IVT transformations are finished, LLVM can further optimize the IR code. For other compilers, the process of identifying virtual table pointers might be different, but in general it should pose no difficulty.

Note that the VTPA is not filled with concrete virtual table pointers until linking; therefore it is likely to be filled with different pointer values in different processes. Besides, the VTPA itself is also likely to be placed at different addresses in different processes. However, the order of virtual
table pointers in the VTPA is decided by the specification and therefore indexes of virtual table pointers are the same across processes.

### 3.3 Compiler Transformation

In order to replace virtual table pointers with indexes, two compiler transformations are implemented.

During **object construction**, virtual table pointers within the constructed object are replaced with their corresponding indexes in the VTPA. The mapping between virtual table pointers and indexes is decided by the specification and translated with the same rule as when the VTPA is generated. Note that if the class of the constructed object uses virtual inheritance or multiple inheritance, multiple virtual table pointers are replaced with their corresponding indexes.

Figure 6 shows an example of this transformation. Figure 6(b) is the LLVM IR code of a typical constructor. The highlighted part in Figure 6(c) shows the instrumented code that replaces the virtual table pointer with its index. Note that the index 0 in Figure 6(c) is decided by the specification in Figure 5 during compilation.

For better comparison, Figure 6(d) shows the pseudo code of a constructor function transformed by prior DVT technique. Some table look-up operations are performed before the actual constructor routine. Thus, we can expect that DVT introduces more overhead to object construction.

During **dynamic dispatch**, the index is loaded from an object and added to the start address of the global VTPA. The actual virtual table pointer is then loaded and further used to access the virtual table. Besides dynamic dispatch, there are other cases that may make use of the virtual table, such as type casting. In these cases, the process of converting from indexes to virtual table pointers is the same.

In LLVM IR, dynamic dispatch can be found by recognizing the following pattern.

- It is a *load* instruction.
- The name of its result starts with "%vtable[number]".
- Its operand address is the result of a *bitcast* instruction.
- The source type of the *bitcast* instruction is a pointer type that points to a shared class.

Such an instruction sequence is considered as loading the virtual table pointer of an object. It is then instrumented to add code that converts the index stored in the object into the actual virtual table pointer. Figure 7 shows an example. Figure 7(b) is the corresponding LLVM IR code of the C++ code in Figure 7(a). The highlighted part in Figure 7(c) shows the instrumented code that is responsible for converting the index into the corresponding virtual table pointer.

For better comparison, Figure 7(d) shows the pseudo code of a dynamic method transformed by the prior GVT technique. Combining two hash numbers, searching from a map or a hash table, and an extra call are introduced before the actual work of the method. Thus, we can expect that GVT introduces more overhead to dynamic dispatch.

Overall, the transformations described in Section 3.2 and Section 3.3 realize the steps described in Figure 4. From the example shown in Figure 6 and 7, we can see that the IVT technique adds no extra instructions to object construction, and adds only two *load* instructions and one *add* instruction to dynamic dispatch. Therefore, we expect that the performance degradation brought by IVT should be low.

### 3.4 Specification Generation

When using the proposed IVT technique, programmers need to write a specification about the shared classes. In our application studies in Section 4, this specification can be easily written since only a few polymorphic classes are involved in sharing.
When programmers are not able to provide a specification of shared classes, we further design a static-analysis tool that helps to generate a specification. This tool identifies all polymorphic classes in the input program and uses them to generate a specification, and we will show that the introduced overhead is still acceptable. To avoid confusion, we use IVT_Auto to refer to IVT that uses the automatically generated specification.

Our static-analysis tool is also useful when the input program is updated to a new version (e.g., when new features are added). The tool can be rerun on the new version to generate an updated specification. The specification may remain the same if no new classes are involved in sharing. In this case, IVT needs to recompile code that defines constructors and uses virtual functions. If new classes are involved in sharing, programmers can expand these new classes to the specification manually or with the help of the specification generation tool. Sharing processes that do not use
these new classes need not be recompiled. If a class that was involved with sharing is removed in the new version, programmers can leave a dummy class at its position so that the index remains the same. Sharing processes that do not use this class need not be recompiled.

In comparison, since DVT and GVT introduce overhead to different operations, it is claimed in [Hajj et al. 2017] that these two techniques can coexist and the programmer can choose which technique to use for efficiency; for that, a user specification is also needed to decide which technique should be used for each class.

### 3.5 Comparison and Limitations

Table 1 shows a comparison of our IVT technique with prior DVT and GVT techniques. The main advantages of IVT are performance and that there are no memory mapping conflicts. DVT adds less overhead on dynamic dispatch, but more overhead on object construction. GVT adds less overhead on object construction, but more overhead on dynamic dispatch. IVT adds no overhead on object construction and slight overhead on dynamic dispatch. Therefore, IVT is more suitable in scenarios in which both object construction and dynamic dispatch are frequently used.

Furthermore, DVT needs fixed address mappings to store duplicated virtual tables. Mapping a fixed address range across different processes may not succeed, because the address range may...
Table 1. Comparing IVT with prior DVT and GVT techniques.

| Item               | DVT                                                                 | GVT                                                                 | IVT                                                                 |
|--------------------|----------------------------------------------------------------------|----------------------------------------------------------------------|----------------------------------------------------------------------|
| Object construction| Look up in RT, look up in DLT, and initialize the object’s virtual table pointer(s). | Store a type hash value. The object no longer contains virtual table pointer(s). | Convert the object’s virtual table pointer(s) to index(s). |
| Dynamic dispatch   | No change. Duplicate the virtual table at the first time if lazy duplication employed. | Look up the function pointer in GVT using the hash value. | Find the virtual table pointer using the index. |
| Compiler transformation | Modify region mapping and unmapping functions. Transform constructors. | Add a hash value field to class definition. Transform constructors and virtual function definitions. | Transform constructors. Transform virtual table usage code. |
| Recompilation      | Need to recompile code that defines constructors. | Need to recompile code that defines constructors and virtual functions. Need to recompile code that use these types. | Need to recompile code that define constructors. Need to recompile code that use virtual functions. |
| Limitations        | Fixed address mapping is needed. | Need to resolve name conflicts when the hash values of two types conflict. | Need user-defined specification. |
| Overall comparison | More overhead on object construction. | Less overhead on object construction. | Less overhead on object construction. |
|                    | Less overhead on dynamic dispatch. | More overhead on dynamic dispatch. | Less overhead on dynamic dispatch. |
|                    | Less code to be recompiled. | No library code changed. | No conflicts need to be handled. No library code changed. |

have already been occupied. This is called a mapping conflict. GVT needs to resolve hashed name conflicts, although the probability of conflicts is low. IVT does not need fixed address mappings or hashing functions; therefore, there are no conflicts that need to be handled.

The main limitation of IVT is the assumption described in Section 3.1, which states that programmers need to know information of shared classes to write a specification. This specification can usually be easily written. We will discuss this in Section 5.

4 EVALUATION

In order to evaluate our IVT technique, we first conducted an evaluation using the same methodology as prior work [Hajj et al. 2017], and examined the performance of object construction and dynamic dispatch using synthetic micro-benchmarks; the results are presented in Section 4.1.

We further applied IVT to three application scenarios, and showed how IVT improved the overall performance; this is discussed in Section 4.2, 4.3, and 4.4.

The following experiments were performed on a machine with an Intel Core i5-4200M @ 2.5GHz processor (2 cores, 4 threads) and 8GB of memory. The operating system is 64bit Ubuntu 14.04 LTS with 4.4.0 kernel. The LLVM Clang that we use has version 7.0.0.
4.1 Micro-benchmarks

In this subsection, we used the same evaluation methodology as the work that proposed DVT and GVT, which used synthetic micro-benchmarks to evaluate the impact on two main features with respect to four parameters.

The two features are object construction time and dynamic dispatch time, and the four parameters are the number of constructed objects, the number of shared regions, the number of polymorphic classes, and the number of virtual functions per class.

Micro-benchmarks were generated with the four parameters of different values: \( n \) objects, \( r \) regions, \( c \) polymorphic classes, and \( f \) virtual functions per class. Each micro-benchmark first creates and maps \( r \) shared regions, and \( n / (r \times c) \) objects are constructed for each one of \( c \) classes in each region, resulting in \( n \) objects constructed overall. Then, for each of the \( n \) objects, each of the \( f \) virtual functions is called once, resulting in \( n \times f \) dynamic dispatches performed overall. The constructors are all empty and the virtual functions perform a 64-bit integer comparison and an addition. We ran each micro-benchmark for 10 times, and the average results are shown in the following figures. The time of a single run was about 5% above or below the average, and therefore we do not draw error bars to avoid clutter.

For a fair comparison, we re-implemented the DVT and GVT techniques, and performed measurements on the same machine. The results of DVT and GVT were not exactly the same as those reported in [Hajj et al. 2017], due to a different environment and/or a different implementation, but the results showed similar trends. Since all classes in the synthetic micro-benchmarks are polymorphic classes involved in sharing, we omit the result of IVT_Auto.

4.1.1 Object construction time. Figure 8 shows the results of object construction time.

![Fig. 8. Object construction time results.](image)

Figure 8(a) shows that the construction time per object does not change as the number of objects increases for all techniques. GVT and IVT do not introduce any overhead, but DVT introduces a constant overhead.
Figure 8(b) shows that the construction time per object remains unchanged as the number of regions increases when using GVT and IVT. DVT introduces an overhead logarithmic with respect to the number of regions. This is caused by look-up operations in the so-called range table (RT).

Figure 8(c) shows that the construction time per object remains unchanged as the number of polymorphic classes increases when using GVT and IVT. DVT introduces an overhead logarithmic with respect to the number of classes. This is caused by look up operations in the so-called DLT cache.

Figure 8(d) shows that the construction time per object does not change as the number of virtual function increases for all techniques. But DVT still introduces a constant overhead.

Overall, IVT and GVT do not introduce detectable overhead. However, the DVT technique introduces extra overhead to object construction time, which cannot be neglected in realistic settings. These results confirm the claims in the first row of Table 1.

4.1.2 Dynamic dispatch time. Figure 9 shows the results of dynamic dispatch time.

Figure 9(a) shows that the dynamic dispatch time per dispatch does not change as the number of objects increases for all techniques.

Figure 9(b) shows a similar result: the dynamic dispatch time per dispatch does not change as the number of regions increases for all techniques. GVT introduces a constant overhead in both cases, due to look up operations in the so-called global virtual table. The looking up overhead is constant and does not correlate with the number of objects or the number of regions.

Figure 9(c) shows that the dynamic dispatch time per dispatch does not change as the number of polymorphic classes increases for IVT and DVT. GVT introduces an overhead logarithmic with respect to the number of polymorphic classes. This is because the global virtual table is implemented with an ordered map of hash values. We have implemented another version of GVT using unordered map, a hash table data structure. The results are shown as the GVT_HASH series. As we can see, GVT_HASH introduces an overhead nearly constant as the number of polymorphic classes increases.
classes increases, but the overhead is higher than GVT when the number of classes is less than 4. Indeed, there may be many polymorphic classes used by a process. Therefore, GVT_HASH is potentially more efficient than GVT, but still introduces nearly an order of magnitude more overhead than the original dispatch time.

Figure 9(d) shows similar results as Figure 9(c); the amount of time spent on looking up the global virtual table is positively correlated with the number of virtual functions.

Overall, IVT and DVT do not introduce detectable overhead. However, the GVT technique and an modified GVT technique introduce extra overhead to dynamic dispatch. These results confirm the claims in the second row of Table 1.

From Figure 7, we can see that our proposed IVT technique should introduce some overhead to dynamic dispatch. At the LLVM IR level, IVT inserts two extra load instructions, one extra add instruction, and one extra inttoptr instruction to each dynamic dispatch. However, these IR instructions are translated to only two extra mov instructions at the x86 assembly level. Therefore, the overhead is small and hard to notice in Figure 9. To show the overhead, we further increase $f$, the number of virtual functions. The result is shown in Figure 10. Note that the vertical axis is not logarithmic, and the overhead is less than 1%.

![Fig. 10. Amplified dynamic dispatch time results.](image)

As we summarize the two parts of results, we can see that our proposed IVT technique combines the advantages of both DVT and GVT, and introduces little overhead to both object construction time and dynamic dispatch time.

4.2 Application Case Study 1: XML Transformation

This is the application described in [Hajj et al. 2017]. We repeated this application case study and applied our IVT technique. An open source framework named Apache Xalan-C++ [The Apache XML Project 2004] is used to transform XML documents into HTML web pages based on XSLT templates. This procedure can be divided into three major steps: parsing the XML document, compiling the XSLT templates, and the actual transform. Among these steps, parsing the XML documents dominates most of the transformation time. If the same XML document is transformed into HTML web pages of different styles based on different XSLT templates, the parsing can be reused. Therefore, we would like to modify the process by sharing the result of the parsed XML document, and transform it using different XSLT templates. However, the XalanParsedSource class that Xalan uses to express a parsed XML document is an abstract base class, and cannot be shared without proper handling.

In this experiment, five XML documents with different sizes (from 373KB to 30MB) are parsed and transformed into HTML web pages based on an XSLT template. Figure 11 shows the breakdown of the execution time of Xalan-C++ for the original version, along with the normalized time consumption by DVT, GVT, IVT, and IVT_Auto, respectively. As we can see in Figure 11, the "XML
parsing” time is minimal after using each technique. This is because the XML parsing result has been shared and only retrieving the parsed XML object from shared memory is necessary. By sharing the XML parsing result, the overall performance can be improved by 30% to 80%.

![Fig. 11. Xalan-C++ execution time breakdown and comparison.](image1)

Furthermore, if we focus on the “Transform to HTML” step, we can see the performance improvement by IVT more clearly. Figure 12 shows the execution time of the “Transform to HTML” step. For better comparison, all time are normalized with its original execution time; that is, it shows the performance overhead introduced by different techniques. As we can see, GVT introduces 50% to 80% overhead, and DVT introduces at least 10% overhead; relatively, IVT’s overhead is less than 3%, and it is up to 15% overhead when using the automatically generated specification. GVT introduces more overhead because dynamic dispatch is used more heavily than object construction. IVT introduces little overhead to both operations, and therefore performs the best. By using the automatically generated specification in IVT_Auto, more classes are included in the specification. These classes are not involved in sharing, but including them introduces overhead when object creation and dynamic dispatch happen. However, the overall performance of IVT_Auto is still better than DVT and GVT.

### 4.3 Application Case Study 2: Facial Recognition

This is our motivating scenario described in Section 2. We manually modified the `CascadeClassifier::load` method in OpenCV, to allocate objects inside shared memory. The OpenCV library that we use is version 3.4.0.
In this experiment, four files with different sizes (from 51KB to 2.6MB) and facial features were loaded and used to detect faces from images. The size of these images is 352x288, which is a commonly used standardized video format known as the Common Intermediate Format (CIF).

Figure 13 shows the breakdown of the execution time, along with the normalized time consumption by DVT, GVT, IVT, and IVT_Auto, respectively. In the “Load facial feature file” step, a CascadeClassifier object is created in shared memory, and is used to load a facial feature file; in the “Facial detection” step, the CascadeClassifier object in the shared memory is used to detect faces from an image. These steps are repeated for different facial feature files and images.

As we can see, the Load step takes more time when the size of facial feature size increases (up to 84.7% for a 2.6MB facial feature file). Similar to the application case study 1, there is nearly no time cost of the Load step for each technique because the facial feature object has been shared and only retrieval from shared memory is needed. Larger files indicate more complicated features and may bring more accurate recognition results. Therefore, we consider the performance improvement is notable.

Fig. 14. The performance overhead introduced to the “Facial detection” step.

Besides, if we focus on the “Facial detection” step, we can see the performance improvement by IVT more clearly. The result is shown in Figure 14. As we can see, IVT introduces almost no overhead, and IVT_Auto introduces less than 3% overhead; in contrast, DVT introduces up to 5% overhead and GVT introduces up to 11% overhead. If we compare the result with that in Section 4.2, the introduced overhead is lower. This is because less classes in OpenCV are polymorphic classes, and the facial detection algorithm is more time-consuming than XML transformation.
4.4 Application Case Study 3: Robotic Middleware

We also applied IVT to improving the performance of robotic middleware. ROS [Foundation 2018; Quigley et al. 2009] is one of the most commonly used robotic middleware. For better compatibility, ROS uses network sockets as its communication mechanism; consequently, transmitting large messages from a module to another may cause significant latency.

For example, if we want to capture 1920x1080 images from a camera and provide them to a facial recognition module and an object recognition module, the communication latency from the camera module to the recognition modules is over 20ms. This is disappointing since the recognition modules only take 25 ∼ 30ms to finish facial/object recognition with the help of modern GPU. Things get even worse if the images captured by the camera are also used by the Simultaneous Localization And Mapping (SLAM) module, the human body recognition module, and the gesture recognition module, etc.

To avoid the latency caused by sockets, one solution is to put all modules that involve large messages into the same process, and possibly make modules run as threads. In fact, ROS provides a mechanism called nodelet [ROS Wiki 2018] to help putting modules into the same process. The main drawback is the lack of fault isolation: when any of these modules malfunctions because of a bug, the whole system malfunctions. This is unacceptable for most systems that require high reliability.

Shared memory is promising to provide a zero-copy feature. Magnenat et al. [Magnenat 2018] designed a plugin for ROS (referred to as ETHZ later in this paper). Their idea is to serialize messages into shared memory, and de-serialize them to another process. However, the serialization costs a similar amount of time as copying.

In order to provide a zero-copy feature, a message should be an object that is constructed inside shared memory and directly accessed by another process. Different from the last two application cases, messages are constantly generated. Since the space of shared memory is limited, message objects need to be constructed and destructed constantly. Thus, synchronization among processes needs to be taken care of. Since the synchronization procedure is the same among different message types, it is reasonable to design a common base class to handle synchronization and at least one virtual function (e.g. the destruction function) is needed. Here is where we can apply IVT.

In this experiment, images are generated from one process, and sent to one or more other processes. There are two parameters for each test case: the image size, and the number of receiving processes. In each test case, the sending process creates an image, and sends (or shares) it to every receiving process. The metadata of each image records the time when it was created. When any receiving process receives an image, the creation time is extracted and the latency is calculated and recorded. The average latency is shown in Figure 15.

Figure 15(a) shows that, as the image size increases, the communication latency increases little when using IVT, but the latency significantly grows when using ROS and ETHZ. When using ROS, each message is copied at least four times. When using ETHZ, each message is serialized into a buffer allocated in shared memory, and de-serialized to the memory space of each receiver. As the image size increases, the copying and serializing operations become more expensive. However, when using IVT, images are constructed in shared memory, and directly shared and accessed from each receiving process, resulting in zero copying operations.

Figure 15(b) shows that as the number of receiving processes increases, the communication latency grows using all techniques. Note that the vertical axis is represented in a logarithmic scale. The communication latency grows much slower when using IVT than the other two techniques.

Note that when using IVT, the average latency for transmitting each image is about 0.1ms due to synchronization. During the life time of each image message, it is only constructed and destructed
Fig. 15. Comparison results of ROS [Quigley et al. 2009], ETHZ [Magnenat 2018] and our IVT technique.

(which is a dynamic dispatch) once. Therefore the overhead introduced by IVT/DVT/GVT is negligible in this case. But the results show that there is still potential for this kind of techniques to improve performance in application scenarios.

5 DISCUSSION

5.1 Concurrency

In the first two application case studies (XML transformation in Section 4.2 and facial recognition in Section 4.3), there is only one process that writes to shared memory; other processes just read from shared memory. Therefore, there are almost no concurrency problems.

However, accessing objects within shared memory can potentially cause concurrency issues. For example, in the third application case study (robotic middleware in Section 4.4), although there are still only one writer and multiple readers, messages in shared memory need to be constructed and destructed all the time. A message can be destructed after no readers need it any more. To handle this issue, we employ a lazy-free policy. When the writer fails to allocate memory space for a new message, it will try free some old messages that are not in use. Therefore, for each message, we maintained a reference count with `boost :: atomic < int32_t >`, a mutex with `boost :: interprocess :: interprocess_mutex`, and a double-linkage node header. In order to address the concurrency issue uniformly, we designed a `Message` base class, and a virtual destructor for the `Message` base class. The resulting virtual function pointers were then handled by IVT.

If an application has multiple writers, concurrency can be handled with mutexes. In this case, the performance improvement brought by IVT may become less obvious if the application extensively writes to shared memory. However, typically an application reads more often from shared memory; in this case, the performance improvement by IVT should be substantial as long as the read lock is well designed.

It should be noted that concurrency issues are also there when using multiple threads and those threads share memory. If an application uses multiple processes that relies on message passing instead of shared memory, heavy communication traffic caused messages can bring high performance overhead.
6 RELATED WORK

6.1 Inter-Process Communication

Our IVT technique provides an improvement to inter-process communication (IPC). IPC is a classic research topic [Druschel and Peterson 1993] and considered a core service provided by an operating system kernel.

There are mainly two kinds of IPC mechanisms. The first kind, such as socket and pipe, provides APIs for processes to read and write OS managed buffers; the second kind, such as shared memory and memory-mapped file, directly map the buffers to the virtual address spaces of processes. The second kind is more efficient, but lack of synchronization mechanism [Stevens 1992]. Programmers have to perform synchronization with the help of other APIs, such as futexes.

Both kinds of OS-level IPC mechanisms provide low-level APIs, and many libraries and middleware provide more friendly APIs based on them. ZeroMQ [Sústrik 2018] is an open-source communication middleware. It integrates many socket-based techniques and provides multiple APIs. It can achieve the near zero-copy feature by avoiding copying operations at the sender process, but it still has to copy the buffer at the receiver process. Message Passing Interface (MPI) [Gropp et al. 1999], CORBA [The Object Management Group 2018b], and Data Distribution Service (DDS) [The Object Management Group 2018a] are communication standards. MPI is designed for parallel computing, CORBA is designed for network communication, and DDS is designed for real-time communication. There are more research works that aim to improve performance of socket-based frameworks [Goglin and Moreaud 2013; Jin et al. 2005; Kurmann and Stricker 2003]. However, shared memory based frameworks starts to get attentions [Osttott et al. 2017].

Most of these middleware are message-oriented, and leave the problem of encapsulating an object to a message to serialization frameworks.

6.2 Serialization Methods

Serialization is a traditional technique [Herlihy and Liskov 1982] that transforms abstract data types into byte buffers for communication or persistent storage. Many frameworks, such as JSON [Bray 2017], Protocol Buffers [Varda 2018] and DDS [The Object Management Group 2018a], help programmers generate types from data format descriptions and generate corresponding serialization routines. In these frameworks, formal types such as string and vector are used to avoid using pointers. It is feasible to enable sharing these types by customizing the allocator. However, automatically serializing general data types is a difficult task.

Liu et al. [Liu et al. 2017] proposed a method called Type Based Marshalling that enabled automatically serializing pointer data. It can track the boundary of a pointed buffer with the help of Program Dependence Graph (PDG). With the boundary information, a pointer can be automatically serialized along with the buffer it points to. Although this work handle C style pointers nicely, it still need extension to cover C++ classes and has not paid attention to polymorphism.

Overall, serialization methods and frameworks enable object communication. But as we have shown in Section 4.4, serializing objects into shared memory is not an efficient solution to show the advantages of shared memory.

6.3 Object Models

Fundamentally, the implicit pointer issue when sharing polymorphic objects is caused by the virtual table object model [Ellis and Stroustrup 1990]. In order to support sharing polymorphic objects, other kinds of object models are worth noting.

Myers [Myers 1995] proposed an object model named bidirectional object layout. It could outperform C++ implementation on dynamic dispatch with less memory. However, more pointers
were used in the object layout which made it also not suitable for sharing polymorphic objects. Tip and Sweeney [Tip and Sweeney 1997] also aimed on performance. They developed an algorithm that could translate C++ class hierarchy into a more efficient class hierarchy specialization. They also employed more pointers and not suitable for sharing polymorphic objects. Bacon et al. [Bacon et al. 2002] described four different object models for Java class. Although they aimed to improve the space and time performance of synchronization associated routines, the so-called index-based object model is inspiring. The basic idea of replacing pointers with indexes is the same with our IVT technique. But we have carefully designed for the feature of C++ and integrated a compiler-level transformation to enable practical applications.

6.4 Polymorphic Objects in Shared Memory

Using polymorphic objects in shared memory is challenging. Commonly used libraries, such as Boost [Boost C++ Libraries 2018b] and Qt [Qt Documentation 2018] provide portable and object-oriented APIs to manage shared memory, and help to handle explicit pointer fields, but incapable of solving the issues of sharing polymorphic objects.

Hon [Hon 1994] discussed the issues with virtual addresses and concurrency control when using C++ objects in shared memory. His work is inspiring on concurrency control, but did not present practical solutions with respect to virtual tables. Burshteyn [Burshteyn 2014] presented a technique that stores a unique field in shared objects, and used it to find a dummy object to perform dynamic dispatch. However, this technique becomes complicated when handling multiple inheritance. We have shown that our IVT technique can support virtual inheritance and multiple inheritance with the help of a user defined specification (e.g. Figure 5). Hajj et al. [Hajj et al. 2017] presented two techniques named DVT and GVT, and evaluated the tradeoffs between them. These techniques are inspiring and useful in some practical applications. However, as we have claimed in Table 1 and confirmed in Section 4.1, the DVT technique still partially relies on fixed address memory mappings, and the GVT technique introduces overhead to dynamic dispatch. Our IVT technique can combine their advantages and avoid the disadvantages at the same time.

The development of non-volatile memory (NVM) brings new attention to sharing objects. Atлас [Chakrabarti et al. 2014] focused on guaranteeing failure atomicity. Makalu [Bhandari et al. 2016] focused on enabling fast recovery. Berryhill et al. [Berryhill et al. 2015] focused on concurrent reliability and achieving linearizability. CoMerge [Doudali and Gavrilovska 2017] focused on data sharing rather than sharing objects. These systems have not paid attention on polymorphism, and it would be interesting to combine these techniques with the IVT technique to enable polymorphism within the persistent memory programming model.

The emergence of distributed shared memory (DSM) also extends the application area of sharing objects. Schöttner et al. [Schöttner et al. 2000] designed a DSM system that enabled sharing Java objects. They have carefully designed the system for the Java language. Richie et al. [Richie et al. 2017] focused on the C++ programming model for DSM systems. Rheindt et al. [Rheindt et al. 2018] focused on the latency caused by synchronization. It is one of our future work to apply our IVT technique to DSM systems.

7 CONCLUSION

In this paper, we present the Indexed Virtual Tables (IVT) technique that enables directly sharing of polymorphic objects across processes. The main idea is to replace virtual table pointers with indexes. Indexes are used to find the actual virtual table pointers from a global virtual table pointer array. We implemented the IVT technique using the LLVM Clang compiler framework.

Evaluation results show that the proposed IVT technique brings minimal runtime overhead to both object construction and dynamic dispatch; it outperforms prior techniques including DVT
and GVT. We also applied the IVT technique to three application scenarios and showed that the overall application performance can be improved by directly sharing polymorphic objects.

ACKNOWLEDGEMENTS

The authors deeply thank the anonymous reviewers for their constructive comments. This work was supported by the Natural Science Foundation of China (Project Number 61872210).

REFERENCES

David F. Bacon, Stephen J. Fink, and David Grove. 2002. Space- and Time-Efficient Implementation of the Java Object Model. In ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings. 111–132. https://doi.org/10.1007/3-540-47993-7_5

Ryan Berryhill, Wojciech M. Golab, and Mahesh T. Tripunitara. 2015. Robust Shared Objects for Non-Volatile Main Memory. In 19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France. 20:1–20:17. https://doi.org/10.4230/LIPIcs.OPODIS.2015.20

Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: fast recoverable allocation of non-volatile memory. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. 677–694. https://doi.org/10.1145/2983990.2984019

Boost C++ Libraries. 2018a. Class Template offset_ptr. http://www.boost.org/doc/libs/1_67_0/doc/html/boost/interprocess/offset_ptr.html.

Boost C++ Libraries. 2018b. Managed Memory Segments. http://www.boost.org/doc/libs/1_67_0/doc/html/interprocess/managed_memory_segments.html.

Boost C++ Libraries. 2018c. Sharing memory between processes. https://www.boost.org/doc/libs/1_67_0/doc/html/interprocess/sharablememorybetweenprocesses.html.

Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259 (2017), 1–16. https://doi.org/10.17487/RFC8259

B. Burshteyn. 2014. Method and System for Accessing C++ Objects in Shared Memory.

Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. 433–452. https://doi.org/10.1145/2660193.2660224

Thaleia Dimitra Doudali and Ada Gavrilovska. 2017. CoMerge: toward efficient data placement in shared heterogeneous memory systems. In Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017. 251–261. https://doi.org/10.1145/3132402.3132418

Peter Druschel and Larry L. Peterson. 1993. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993. 189–202. https://doi.org/10.1145/166819.166834

Margaret A. Ellis and Bjarne Stroustrup. 1990. The Annotated C++ Reference Manual. Addison-Wesley.

Open Source Robotic Foundation. 2018. ROS, Robot operating system. http://www.ros.org/.

Brice Goglin and Stéphanie Moreaud. 2013. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. J. Parallel Distrib. Comput. 73, 2 (2013), 176–188. https://doi.org/10.1016/j.jpdc.2012.09.016

W. Gropp, E. Lusk, A. Skjellum, and R. Thakur. 1999. Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press. https://books.google.ae/books?id=DFT1ngEACAAJ

Izzat El Hajj, Thomas B. Jablin, Dejan S. Milojicic, and Wen-Mei W. Hwu. 2017. SAVI objects: sharing and virtuality incorporated. PACMPL 1, OOPSLA (2017), 45:1–45:24. https://doi.org/10.1145/3133869

Maurice Herlihy and Barbara Liskov. 1982. A Value Transmission Method for Abstract Data Types. ACM Trans. Program. Lang. Syst. 4, 4 (1982), 527–551. https://doi.org/10.1145/69622.357182

Lenny Hon. 1994. Using objects in shared memory for C++ application. In Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 3, 1994, Toronto, Ontario, Canada. 29. https://doi.org/10.1145/782214

Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. 2005. LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster. In 34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway. 184–191. https://doi.org/10.1109/ICPP.2005.48

Kamran Karimi and Mohsen Sharifi. 2006. Transparent Distributed Programming under Linux. In 20th Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2006), 14-17 May 2006, St. John's, Newfoundland, Canada. 13. https://doi.org/10.1109/HPCS.2006.50

Proc. ACM Program. Lang., Vol. 3, No. OOPSLA, Article 130. Publication date: October 2019.
Christian Kurmann and Thomas Stricker. 2003. Zero-Copy for CORBA - Efficient Communication for Distributed Object Middleware. In 12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003), 22-24 June 2003, Seattle, WA, USA. 4–13. https://doi.org/10.1109/HPDC.2003.1210011

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. 75–88. https://doi.org/10.1109/CGO.2004.1281665

Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. 2359–2371. https://doi.org/10.1145/3133956.3134066

LLVM. 2018. Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org/.

S. Magnenat. 2018. ethzasl_message_transport. https://github.com/ethz-asl/ros-message-transport.

Andrew C. Myers. 1995. Bidirectional Object Layout for Separate Compilation. In OOPSLA’95, Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Austin, Texas, USA, October 15-19, 1995. 124–139. https://doi.org/10.1145/217838.217849

OpenCV team. 2018. OpenCV - Official Site. http://opencv.org/.

M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. 2009. ROS: an open-source Robot Operating System. In ICRA Workshop on Open Source Software.

Sven Rheindt, Andreas Schenk, Akshay Srivatsa, Thomas Wild, and Andreas Herkersdorf. 2018. CaCAO: Complex and Compositional Atomic Operations for NoC-Based Manycore Platforms. In Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings. 139–152. https://doi.org/10.1007/978-3-319-77610-1_11

Qt Documentation. 2018. QSharedMemory Class. http://doc.qt.io/qt-5/qsharedmemory.html.

W. Richard Stevens. 1992. Advanced Programming in the UNIX Environment. Addison-Wesley.

The Apache XML Project. 2004. Xalan-C++ version 1.10. http://xml.apache.org/xalan-c/.

The Object Management Group. 2018a. About the Data Distribution Service Specification Version 1.4. http://www.omg.org/spec/DDD/1.4/.

The Object Management Group. 2018b. Common Object Request Broker Architecture (CORBA/IIOP). 3.3 edition. http://www.omg.org/spec/CORBA/3.3/.

Frank Tip and Peter F. Sweeney. 1997. Class Hierarchy Specialization. In Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA ’97), Atlanta, Georgia, October 5-9, 1997. 271–285. https://doi.org/10.1145/263698.263748

K. Varda. 2018. Protocol buffers: Google’s data interchange format. https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.