SAVI Objects: Sharing and Virtuality Incorporated

IZZAT EL HAJJ, University of Illinois at Urbana-Champaign, USA and Hewlett Packard Labs, USA
THOMAS B. JABLIN, University of Illinois at Urbana-Champaign, USA and MulticoreWare Inc, USA
DEJAN MILOJICIC, Hewlett Packard Labs, USA
WEN-MEI HWU, University of Illinois at Urbana-Champaign, USA

Direct sharing and storing of memory objects allows high-performance and low-overhead collaboration between parallel processes or application workflows with loosely coupled programs. However, sharing of objects is hindered by the inability to use subtype polymorphism which is common in object-oriented programming languages. That is because implementations of subtype polymorphism in modern compilers rely on using virtual tables stored at process-specific locations, which makes objects unusable in processes other than the creating process.

In this paper, we present SAVI Objects, objects with Sharing and Virtuality Incorporated. SAVI Objects support subtype polymorphism but can still be shared across processes and stored in persistent data structures. We propose two different techniques to implement SAVI Objects and evaluate the tradeoffs between them. The first technique is *virtual table duplication* which adheres to the virtual-table-based implementation of subtype polymorphism, but duplicates virtual tables for shared objects to fixed memory addresses associated with each shared memory region. The second technique is *hashing-based dynamic dispatch* which re-implements subtype polymorphism using hashing-based look-ups to a global virtual table.

Our results show that SAVI Objects enable direct sharing and storing of memory objects that use subtype polymorphism by adding modest overhead costs to object construction and dynamic dispatch time. SAVI Objects thus enable faster inter-process communication, improving the overall performance of production applications that share polymorphic objects.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Polymorphism**; **Concurrent programming structures**; **Memory management**;

Additional Key Words and Phrases: Managed data structures, Shared memory regions, Inter-process communication, Dynamic dispatch, Devirtualization

ACM Reference Format:
Izzat El Hajj, Thomas B. Jablin, Dejan Milojicic, and Wen-mei Hwu. 2017. SAVI Objects: Sharing and Virtuality Incorporated. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 45 (October 2017), 24 pages. https://doi.org/10.1145/3133869

1 INTRODUCTION

Sharing memory objects across processes or storing them to be used by future programs is important in modern parallel and collaborative workloads. A common technique for sharing (and storing)

Authors' addresses: Izzat El Hajj, University of Illinois at Urbana-Champaign, 1308 W Main St, Urbana, IL, 61801, USA , Hewlett Packard Labs, USA, elhajj2@illinois.edu; Thomas B. Jablin, University of Illinois at Urbana-Champaign, 1308 W Main St, Urbana, IL, 61801, USA , MulticoreWare Inc, USA, jablin@illinois.edu; Dejan Milojicic, Hewlett Packard Labs, 3000 Hanover St, Palo Alto, CA, 94304, USA, dejan.milojicic@hpe.com; Wen-mei Hwu, University of Illinois at Urbana-Champaign, 1308 W Main St, Urbana, IL, 61801, USA, w-hwu@illinois.edu.

2475-1421/2017/10-ART45
https://doi.org/10.1145/3133869

objects is serializing them to special intermediate formats such as XML, JSON, YAML, or Protocol Buffers [Varda 2008]. Such formats provide an architecture- and language-neutral representation of data achieving the most portability, however, their performance overhead makes them unnecessarily burdensome in situations where architecture- and language-neutrality are not of concern. In such situations, direct sharing of objects via managed memory techniques effectively enables faster inter-process communication. Such memory management techniques have also received recent attention from non-volatile memory programming models centered around persistent in-memory data-structures [Bhandari et al. 2016; Chakrabarti et al. 2014; Coburn et al. 2011; Volos et al. 2011]. This renewed interest is in part due to the fact that load-store domains will become very large, sharing will be much more common, and long latencies will be exposed [Asanovic 2014; Bresniker et al. 2015; Narayanan and Hodson 2012], therefore efficient object sharing will be critical for application performance.

A primary challenge with direct sharing of objects is ensuring the validity of pointers across virtual address spaces. Existing managed memory techniques deal with explicit data pointers by fixing the mapping of managed memory segments or using offset pointers that are independent of virtual address mapping. However, these techniques can only handle plain old data (POD) types; they do not deal with implicit pointers introduced by subtype polymorphism which is common in object oriented programming languages.

Subtype polymorphism enables the creation of class hierarchies where super classes establish APIs with varying implementations in each subclass. Subtype polymorphism makes it easier to deal with heterogeneous collections of objects efficiently. Although most language specifications do not specify an implementation for subtype polymorphism [Lippman 1996], most compilers including GCC, LLVM, and MSVC++ use virtual tables.

In a virtual-table-based implementation of subtype polymorphism, each polymorphic type has a unique virtual table containing pointers to all polymorphic functions, also called virtual functions in C++ terminology. Every object of a type that supports subtype polymorphism will contain a pointer to that type's virtual table. Within a process, two objects of the same type will have the same virtual table pointer, but this is not true for objects in different processes. In different processes, virtual tables of the same type are likely stored at different locations because different programs have different collections of types, and therefore different placements for their virtual tables. Even two processes executing the same program may store a type's virtual table at different addresses due to address space layout randomization. Consequently, when a process creates an object, the virtual table pointer it stores in that object is not usable by other processes which store their virtual tables elsewhere, which makes these objects not shareable across processes. For this reason, state-of-the-art libraries for managed shared memory such as Boost [Boost C++ Libraries 2015a] forbid the use of subtype polymorphism inside shared data structures [Boost C++ Libraries 2015b].

To surmount this limitation, we propose SAVI[1] Objects, objects with Sharing and Virtuality Incorporated. SAVI Objects are designed to support subtype polymorphism for shared in-memory data-structures while mitigating the impact on the costs of polymorphic object construction and polymorphic function dispatch. Our initial implementations target the C++ programming language, but similar subtype polymorphic languages can also be supported using the same techniques. In this paper, we present and evaluate two low-level implementations of SAVI Objects: virtual table duplication and hashing-based dynamic dispatch.

In *virtual table duplication*, an object of a specific type constructed in a shared memory region is initialized with a virtual table pointer that points to a duplicate virtual table of that type located at

[1]Pronounced like *savvy*

a fixed virtual address associated with that region. Any process mapping that region, duplicates its virtual table for that type to the same fixed location. All processes can then use traditional dynamic dispatch through the duplicate virtual table.

In *hashing-based dynamic dispatch*, virtual table pointers are replaced with hash values of the object's type that are universally unique with high probability. Dynamic dispatch is then performed by using the hash value of the type and that of the function being called to look up a function pointer in a global virtual table that is unique to the process.

We make the following contributions:

(1) We propose two novel techniques for inter-process subtype polymorphism: virtual table duplication and hashing-based dynamic dispatch.
(2) We implement these two techniques in a real system and analyze the tradeoffs between them.
(3) We conduct a detailed performance evaluation of the two techniques using microbenchmarks that help isolate their cost in different scenarios.
(4) We apply these techniques to a production object serialization application that uses polymorphic objects, thereby avoiding traditional inter-process communication overhead and improving performance significantly.

The rest of this paper is organized as follows. Section 2 motivates the problem while providing background information on subtype polymorphism and object sharing. Section 3 describes the design and implementation of virtual table duplication and Section 4 describes that of hashing-based dynamic dispatch. Section 5 evaluates the two techniques using microbenchmarks as well as a production object serialization application. Section 6 reviews related work and Section 7 concludes.

2 BACKGROUND

2.1 Subtype Polymorphism

Subtype polymorphism is a feature in many object-oriented languages that permits super classes (or base classes) in a class hierarchy to define high-level APIs that are implemented by different subclass (or derived classes). Subtype polymorphism enables programs to provide different implementations for the same API, and also enables clean manipulation of heterogeneous collections of objects. In each case, subtype polymorphism helps to simplify software structure, reduce code size, and enhance code readability.

Many modern compilers implement subtype polymorphism using virtual tables. A *virtual table (VT)* is a table of virtual functions associated with a particular type. A *virtual function* is a polymorphic member function of a class that often has different implementations for different derived classes, and invocations of that function should call the version of the function corresponding to the dynamic type of the invoking object. In some languages such as Java, all functions are virtual by default, whereas in other languages such as C++, virtual functions must be specified explicitly using the *virtual* qualifier.

Fig. 1 shows a C++ example of a type `A` that uses subtype polymorphism by explicitly declaring functions `bar` and `baz` as virtual. In this example, a VT is created for the type `A` that stores a function pointer to each virtual function declared in `A`, namely `bar` and `baz`; function `foo` is not included because it is not declared virtual. Every object constructed of type `A` contains, in addition to the explicitly declared fields `x` and `y`, a hidden field that stores a pointer to `A`'s VT. For the type `B` which inherits from `A`, the object layout is appended with the additional fields declared in `B`, namely `z`, and the VT layout of `B` is appended with the additional virtual functions declared in `B`, namely `qux`. Implementations are actually more complicated in the case of multiple inheritance, but this simplification is sufficient for the purpose of this paper.

```
class A {
  int x;
  float y;
  void foo();
  virtual void bar();
  virtual void baz();
};
class B : public A {
  int z;
  virtual void baz();
  virtual void qux();
};
```

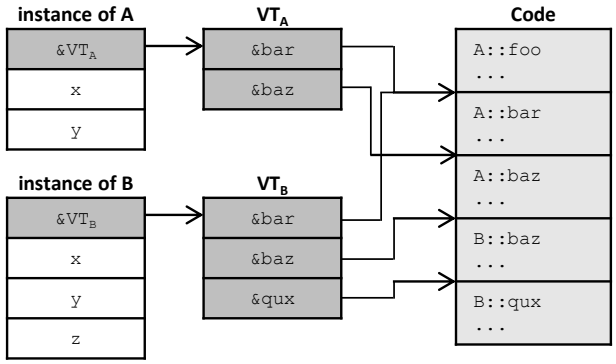

Fig. 1. Layout of objects in a virtual table (VT) based implementation of subtype polymorphism

When a virtual function is called on an object, the VT pointer is loaded from the object, the function pointer is obtained from the VT at its fixed offset, and then the function is invoked. This process is called *dynamic dispatch*. In the example in Fig. 1, if an object pointed to by a pointer of type `A*` invokes the function `baz`, then the VT pointer is loaded from that object, the second function pointer `&baz` is loaded from the VT, and then the function is invoked via that pointer. Thus, regardless of whether the dynamic type of the object was `A` or `B`, the correct `&baz` pointer will be loaded and invoked.

2.2 Incompatibility with Sharing of Objects

VTs are typically stored as statically initialized data in a binary. Because different programs, or different versions of the same program, use different sets of types, the ordering and placement of the VTs for those types in different binaries will vary. Even the same program, if compiled with different optimizations, could result in a different overall layout, thereby different offsets for the VTs in the binary. Moreover, even if processes execute the same program using the same dynamically linked binaries, they could end up having different locations for their VTs due to address space layout randomization (ASLR).

As a result, one must assume that VTs of the same type can be placed at different locations in different processes and that polymorphic objects containing pointers to those VTs are unusable outside the process that creates them. This makes using polymorphic objects inside shared data structures infeasible. An example of this situation is shown in Fig. 2. In this example, process P_c creates an object of type `A` inside a shared region, and initializes the object's VT pointer to `0xbead` which is the location of `A`'s VT in process P_c. When the object is later shared with a different process P_u which places `A`'s VT at a different address `0xbeef`, the VT pointer to `0xbead` in process P_u becomes a dangling pointer and causes P_u to execute junk code or to crash when a dynamic dispatch takes place.

For the aforementioned reasons, shared data structures have not been able to use subtype polymorphism in main-stream programming systems. For example, Boost, the state-of-the art C++ library for handling shared memory and memory-mapped files, says in its documentation regarding this issue: *"This problem is very difficult to solve, since each process needs a different virtual table pointer and the object that contains that pointer is shared across many processes. Even if we map the mapped region in the same address in every process, the virtual table can be in a different address in every process. To enable virtual functions for objects shared between processes, deep compiler changes are needed and virtual functions would suffer a performance hit"* [Boost C++ Libraries 2015b]. The

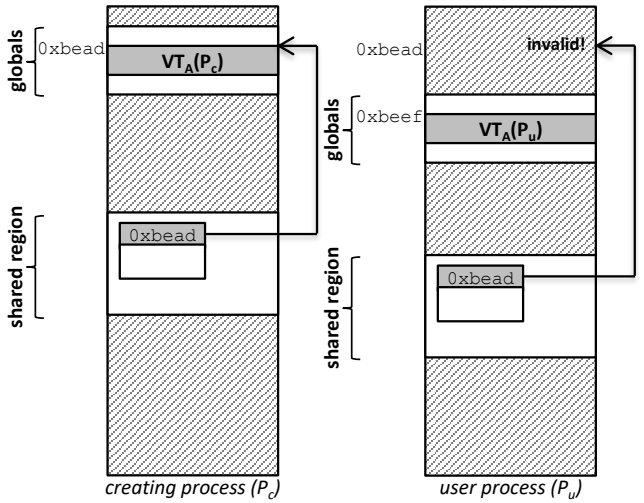

Fig. 2. Sharing of objects with subtype polymorphism: VT pointers are unusable by other than the creating process

objective of this paper is to investigate alternative designs for enabling virtual functions in objects shared between processes, to explore the depth of compiler changes needed, and to evaluate the resulting performance implications.

It is noteworthy that this problem is one instance of a more general problem which is sharing objects containing pointers to global data such as function pointers and string constants. In these other cases, the pointers to global data are stored in the objects explicitly by the programmer in the source code. Therefore, it is possible for programmers to workaround these issues at the source code level. However, VT pointers are different in that they are not created explicitly, but rather they are created implicitly by the compiler as a byproduct of its implementation of subtype polymorphism. For this reason, a compiler solution is necessary to handle VT pointers which is why we focus on them in this paper.

2.3 Data Structure Sharing Mechanism

For the rest of this paper, we assume that data structures are shared across processes through mapping of memory regions (or segments) containing those data structures in and out of virtual address spaces. Regions can be mapped concurrently by multiple processes. Thus *sharing* throughout the paper refers to both sharing between multiple simultaneously executing processes via shared memory segments, or storing objects to be used by later executing processes via memory-mapped files.

3 VIRTUAL TABLE DUPLICATION

3.1 Overview

The first proposed solution for enabling sharing of objects with subtype polymorphism is *virtual table duplication*. In this solution, when a polymorphic object is allocated in a shared region, the VT of the object's type is duplicated to a fixed virtual address specific to that region. This duplicate is known as the Duplicate Virtual Table (DVT). The VT pointer of the allocated object is set to be the location of the DVT instead of the original VT. All processes using the shared region also duplicate

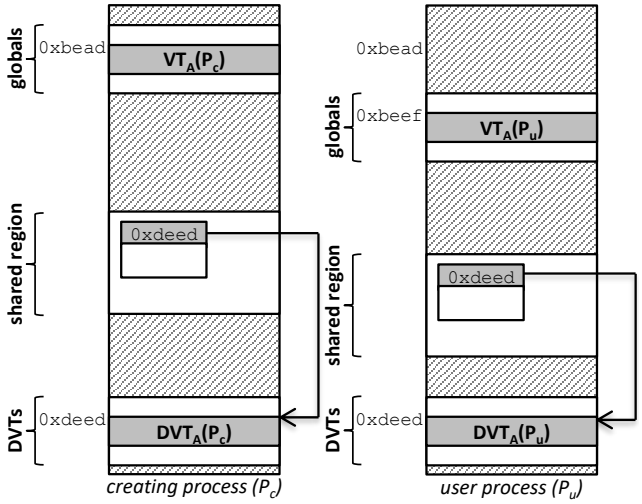

Fig. 3. Overview of virtual table duplication

their VTs to the same fixed virtual address location. This makes the objects usable in all processes. The duplication of VTs happens lazily for reasons that are explained later in this section.

Fig. 3 shows how this technique impacts the example in Fig. 2. In this example, the creating process P_c, upon constructing the object of type `A` in the shared region, allocates a DVT for type `A` at the address `0xdeed` and uses `0xdeed` instead of `0xbead` to initialize the object. In the user process P_u, the DVT for the type `A` is also allocated at address `0xdeed`, thereby making the object also usable in P_u. Note that the DVTs of P_c and P_u are not themselves shared; each process duplicates its own local VT to ensure that the VT contains the correct state for that process.

3.2 Implementation Details

To accomplish the actions just described, multiple supporting look-up data structures are needed. We begin by describing these data structures then go into the steps taken at each relevant event to realize this technique.

Examples of the look-up data structures used throughout this technique is shown in Fig. 4. We define these data structures as follows and explain how and why they are initialized and used later as we describe the various steps.

- *Range Table (RT)*: The RT is a data structure that is local to a process that takes a virtual memory address and returns the region within which that address is located in that process.
- *VT Look-up Table (VLT)*: The VLT is a data structure that is local to a process that takes a string with the mangled name of a type and returns the address of that type's VT in that process.
- *DVT Look-up Table (DLT)*: The DLT is a data structure that is stored inside a shared region with support for concurrent access. It contains, for each polymorphic type having objects stored in that region, a mapping between the mangled names of those types and the fixed virtual addresses of those types' DVTs. That is, given a string with a mangled type name, it returns the address of that type's DVT, or given the address of a DVT, it returns a string with a mangled type name.

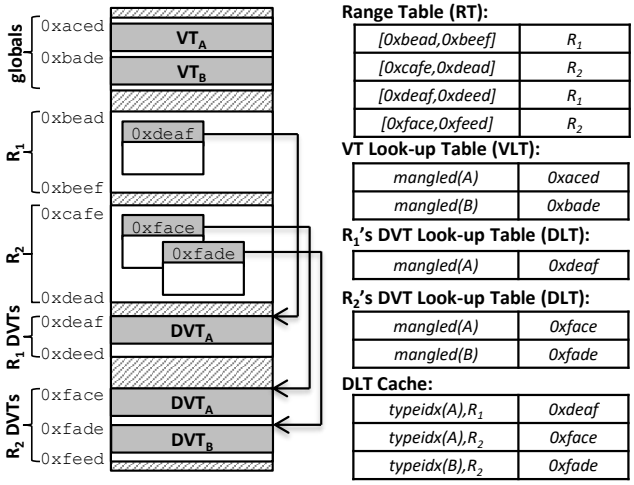

Fig. 4. Look-up data structures used in virtual table duplication

- *DLT Cache*: The DLT Cache is a data structure local to a process that takes a region and a type index (not a string) and returns the address of the DVT of that type for that region.

On ***process entry***, the RT and VLT are initialized. The RT is initialized to be empty of ranges, meaning that the entire virtual address range does not point to any shared region. The VLT is initialized by inspecting the symbols of the program's static binary and dynamically linked binaries and extracting those symbols corresponding to VTs.

Another step performed at process entry is the creation of a dispatch fault handler. A *dispatch fault handler* is a segmentation fault handler that is used to perform lazy duplication of a type's VT on the first dynamic dispatch on an object of that type in a specific region. The details of how it is triggered and what it does will be explained later in this section.

On ***region mapping***, the region being mapped is registered in the RT. Moreover, the region's DLT is scanned to identify where in virtual memory all the region's DVTs must be mapped. Those mappings are then reserved and registered in the RT as well.

In the example in Fig. 4, we show the contents of the RT after mapping the two regions R_1 and R_2 into the process. The RT records that R_1 has two virtual address ranges associated with it: `0xbead` through `0xbeef` (for the region itself) and `0xdeaf` through `0xdeed` (for the region's DVTs). It also contains similar entries for R_2. The figure also shows the DLTs of the two regions.

Although memory is reserved for the DVTs, the duplication does not happen yet. Instead, the reserved pages are protected such that the first access to a DVT triggers a dispatch fault handler to duplicate that table. There are several reasons why the duplication is done lazily and not when the region is mapped. One reason is that not all types in the region may exist in the program, or they may exist but not be used in a specific run. Thus, we do not want to attempt to duplicate a VT that is non-existent or unused. Another reason is that new types may be introduced in a region by another concurrent process after the region is mapped. The VTs for these new types will be missed if duplication happens on region mapping.

On ***region unmapping***, the region is removed from the RT and the memory pages reserved for that region's DVTs are released.

On ***object construction***, the location where the object has been allocated is used to look up the corresponding region of allocation in the RT. If the object is not allocated in a shared region, the

object's original VT pointer is kept and nothing further needs to be done. If the object is allocated in a shared region, the region's DLT is used to look up the address of the region's DVT corresponding to the type of the object being constructed. The obtained DVT address is then used to initialize the object's VT pointer. There can be multiple pointers to initialize in the case of multiple inheritance which is supported.

If an address is not found, this indicates that it is the first time an object of this type is allocated in this region. Therefore, new memory is allocated to store the DVT for that type and the DLT is updated accordingly. The memory for the DVT is just reserved and protected, while the actual duplication happens lazily when the first dynamic dispatch takes place. We over-allocate memory for the DVT to allow it to grow if more polymorphic functions are added to future versions of the class. DVTs must be allocated at page granularity so that they can be independently protected for lazy duplication.

Because the DLT stores mangled type names, DLT look-ups must perform string comparisons which can be very expensive if done whenever an object is constructed. A faster way to perform these look-ups is by using the type indexes provided by the C++ runtime. However, these type indexes are only meaningful within the same process and cannot be used across processes. That is why the DLT Cache is used. The first time an object of a specific polymorphic type is constructed in a specific region, a string-based look-up is used on the DLT. The result is then entered into the DLT Cache to be used by all subsequent look-ups.

The reason the RT is needed by the constructor is that the constructor is agnostic to the region of allocation. Maintaining this agnosticism is needed for preserving separation of concerns between allocation and initialization of objects, which is important for applications that wish to do their own memory management. Without the RT, pervasive code changes would be needed to pass the region information to the constructor and all code that uses the class would need to be recompiled.

On ***dynamic dispatch***, since the layout of the object and representation of the VT has not changed, nothing special needs to be done for recurrent dynamic dispatches. Only the first dynamic dispatch per type per region (i.e., the first access to a DVT) will trigger the dispatch fault handler to perform lazy duplication to initialize that DVT. The handler does the following. First, it looks up the faulting virtual address in the RT to identify the region it corresponds to. Next, it looks up the faulting address in the region's DLT to identify the type name. Finally, it looks up the address of the type's original VT in the VLT, and copies the original VT to the faulting address while removing the protection so that future dispatches can proceed normally.

3.3 Compiler Transformation

Minimal compiler changes are needed to implement virtual table duplication. The actions performed on process entry are performed when the SAVI runtime library is loaded without needing any compiler transformations. Registering/unregistering of regions requires modifying the region management library (we use Boost) to invoke the proper routines when the region is mapped/unmapped. The only compiler transformation we perform to make an object a SAVI Object is inserting code into the constructor which calls a runtime function to obtain the DVT pointer and then sets the object's VT pointer accordingly. As a result, the only code that needs to be recompiled is the code that defines constructors of polymorphic types that use SAVI Objects. If the region management library is header-only, then code that maps/unmaps regions also needs to be recompiled.

These transformations are transparent to programmers and there are no changes to the programming model. Programmers simply need to annotate which classes they would like to use SAVI Objects with (i.e., those involved in sharing) so that the compiler does not transform all polymorphic types unnecessarily.

3.4 Limitations

The limitation of this technique is when the fixed virtual address of the DVTs desired by a region conflicts with another memory mapping. Randomization of the virtual address assigned to the DVTs could minimize the chance of this situation occurring, but cannot eliminate it completely. In the case of a conflict, the region cannot be mapped into the virtual address space until the conflicting mapping is unmapped. Recent OS techniques have also been proposed that could also be useful in addressing this issue by providing processes with the ability to sustain multiple mappings [El Hajj et al. 2016; Litton et al. 2016]. Otherwise, the data structure will need to be (de)serialized using traditional approaches.

While it is true that we have simply replaced the problem of fixed VT mapping with that of fixed DVT mapping, the implications of latter are much less severe. VT collision means that the program binary itself cannot be mapped into the virtual address space for the process to execute, and avoiding such collision would require all compilers to coordinate to agree where the VT for every possible type should be placed. On the other hand, DVT collision means that only a specific memory region cannot be mapped into the virtual address space, however the process itself can still execute and can recover from such collision using the techniques mentioned earlier.

Note that this technique does not require the regions themselves to be mapped to fixed virtual address locations, just their DVTs. Whether or not the regions need to be mapped to fixed virtual addresses depends on whether the programmer uses absolute or relative data pointers, which is independent of whether or not the objects are polymorphic and therefore not relevant to SAVI Objects. The reason we do not use offset pointers for the VTs themselves is that, in this technique, we try not to deviate from the traditional way objects are laid out and the way dynamic dispatch is implemented to minimize compiler changes and the scope of code that needs to be recompiled.

4 HASHING-BASED DYNAMIC DISPATCH

4.1 Overview

The second proposed solution for enabling sharing of objects with subtype polymorphism is *hashing-based dynamic dispatch*. In this solution, we change the way polymorphic objects are laid out and the way dynamic dispatch is implemented. An object no longer stores a pointer to its type's VT. Instead, it stores a 64-bit hash value of the type's name that is universally unique with high probability. That is, there is very low probability (see Section 4.4) that two class names hash to the same 64-bit value. To perform dynamic dispatch, the hash value of the type is combined with a hash value of the function being dispatched, and together used to look up an entry in a *global virtual table (GVT)* which contains pointers to all the polymorphic functions in the process.

Fig. 5 illustrates how this technique impacts the example in Fig. 2. Instead of storing a pointer in the constructed object, the creating process P_c stores a hash value of the type `A` designated as $hash(A)$. When the polymorphic function `bar` is invoked in any process, the hash value of `bar` for that process, designated as $hash(bar)$, is used along with $hash(A)$ to look up the location of `A::bar` in that process' GVT.

4.2 Implementation Details

To accomplish the actions just described, the following steps are taken at each relevant event to realize this technique.

On ***process entry***, the GVT is constructed by inspecting the symbols of the program's static binary and dynamically linked binaries. The GVT takes as input the hash value of a type and that of a function and returns the pointer to that function for that type.

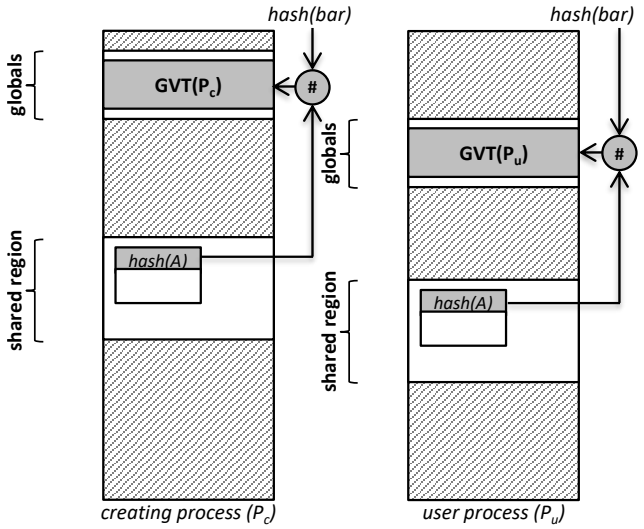

Fig. 5. Overview of hashing-based dynamic dispatch

On ***object construction***, the hash value of the type is stored in the polymorphic object instead of the VT pointer. The type's hash value is known statically so there is no hash value computation overhead.

On ***dynamic dispatch***, the hash value of the type is loaded from the object and combined with the hash value of the function to look up the location of the desired function in the GVT. The function obtained is then invoked.

4.3 Compiler Transformation

Although the steps for this technique may seem simpler than those of virtual table duplication, the compiler transformations needed to achieve these steps are more involved. However, because the transformed code does not use virtual tables and makes no assumptions about the object's implicit layout, the transformations can be performed at the source-to-source level without the need for deep compiler changes. The source-to-source transformation is described in the rest of this subsection, and an example of the transformation is shown in Fig. 6.

All virtual function declarations and definitions are stripped from their `virtual` qualifiers which effectively removes the implicit VT pointer from the class. Instead, a field is inserted in the topmost class in the hierarchy which uses virtual functions (see `_h` variable in `class A` in Fig. 6(b)), and this field is initialized with the hash value of the type by the constructors. There can be multiple fields to initialize in the case of multiple inheritance which is supported.

The topmost declaration or definition of a virtual function in the class hierarchy has its definition replaced with a member function that performs a GVT look-up and an invocation of the obtained function (see functions `A::foo` and `B::bar`). The definitions of the function in the lower classes of the hierarchy are all removed. As a result, any original dynamic dispatch of the function in source code that uses the class will become a static dispatch of this look-up function without the need for that source code to be transformed (the code will still need to be recompiled).

All definitions of virtual functions throughout the class hierarchy are replaced with declarations of global friend functions. This is shown in the definitions of `B::foo`, `B::bar`, `C::foo`, and `C::bar`. However, notice that `A::foo` does not have a corresponding friend function because it is a pure

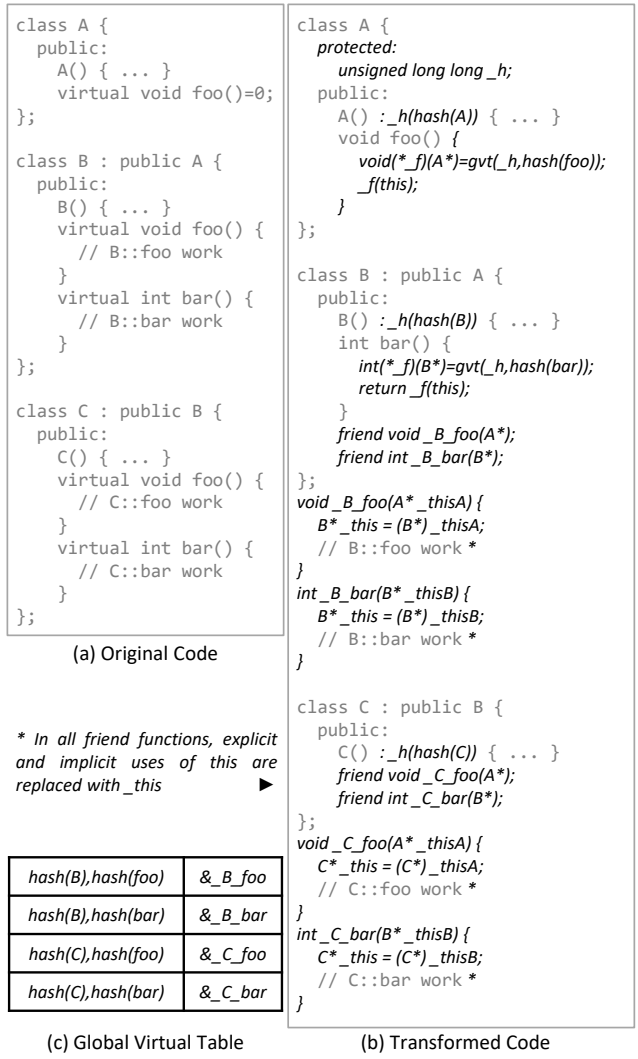

Fig. 6. Example code transformation of hashing-based dynamic dispatch

virtual function declaration without a body. The friend functions take an additional parameter for the `this` object. The type of that parameter must be a pointer to the topmost class which declares the virtual function in the original code and is responsible for the look-up. The bodies of the friend functions cast that parameter to the correct type and store it in the `_this` variable, then execute the original code of the function replacing explicit and implicit uses of `this` with `_this`.

As a result of these compiler transformations, the code that needs to be recompiled is code which defines constructors and virtual functions of polymorphic classes that use SAVI Objects as well as code which uses these classes.

The initialization of the GVT is performed on process entry when the SAVI runtime library is loaded without needing any compiler transformation. Fig. 6(c) shows an example of the contents of the GVT for the example program. One optimization we perform is to use special names for the

Table 1. Comparing the two solutions

Item	Virtual Table Duplication	Hashing-based Dynamic Dispatch
Object construction	Look up the region in the range table, look up the type's DVT address in the region's DLT, and initialize the object's VT pointer(s) accordingly.	Store a type hash value instead of a VT pointer in the object.
Dynamic dispatch	No change. Only the first dispatch triggers the dispatch fault handler to perform lazy duplication.	Compute an additional hash value and look up the function pointer in a GVT instead of a VT.
Compiler transformation and code changes	Transform constructors. Modify region management library functions for mapping and unmapping regions.	Add field for class hash value. Transform constructors and virtual function declarations and definitions.
Recompilation	Need to recompile code the defines constructors of client polymorphic types.	Need to recompile code the defines constructors and virtual functions of client polymorphic types, and code that uses these types.
Limitations	Need to resolve mapping conflicts or fall back on traditional (de)serialization when regions require conflicting addresses for their DVTs.	Need to resolve name conflicts when two types in a single program have conflicting hash values.
Overall comparison of advantages	Incurs less overhead on dynamic dispatch. Less code to be recompiled.	Incurs less overhead on object construction. Conflicts detectable at link time.

friend functions that indicate that they were originally virtual so that they can be identified when the GVT is constructed. Otherwise, all functions would conservatively need to be included in the GVT which would result in it being unnecessarily large and degrade performance.

As with virtual table duplication, these transformations are transparent to programmers and there are no changes to the programming model. Programmers simply need to annotate which classes they would like to use SAVI Objects with (i.e., those involved in sharing) so that the compiler does not transform all polymorphic types unnecessarily.

4.4 Limitations

The limitation of this technique is when two types in the program hash to the same hash value. Through proper choice of the universal hash function, it can be shown that 5 billion polymorphic types would be needed to generate such a collision for a 64-bit hash value, and that a program with 6 thousand polymorphic types would have a probability of collision of 10^{-12} [Wikipedia 2017]. In the rare case that a collision should happen, the collision can be detected at link time and the programmer can recover by renaming one of the colliding types. Name collisions at link time are an existing issue that programmers already deal with. This technique has only extended the scope of collisions from the type name to the type hash value.

4.5 Summary and Comparison of the Two Solutions

Table 1 shows a summary of the two techniques with a qualitative comparison of important issues. The advantages of virtual table duplication are that it adds less overhead to dynamic dispatch and that it performs fewer compiler changes, requiring less code to be recompiled. On the other hand, the advantages of hashing-based dynamic dispatch are that it adds less overhead to object construction and that conflicts are detectable at link-time.

Keep in mind that the overhead to object construction and dynamic dispatch only applies to polymorphic types so non-polymorphic types are not affected. Furthermore, the two techniques are coexistent with each other and with regular polymorphic type implementations, so it is not required for all polymorphic types in a program to use the same technique, as long as a derived class uses the same technique as its base class. Thus polymorphic types that do not participate

in any sharing can use regular polymorphic type implementations instead of SAVI Objects and are also not affected. For polymorphic types involved in sharing, the programmer can choose which technique to use, so virtual table duplication can be used on polymorphic types that are dispatch heavy, while hashing-based dynamic dispatch can be used on polymorphic types that are construction heavy. If an object subscribes to a polymorphic type involved in sharing but is itself not shared, then the overhead from virtual table duplication will become negligible, while the overhead from hashing-based dynamic dispatch will remain as though it were shared.

5 EVALUATION

The evaluation is divided into two parts. In the first part (Section 5.1), we use synthetic microbenchmarks that isolate the performance of the two main features impacted by our technique: construction of polymorphic objects and dynamic dispatch of polymorphic functions. In the second part (Section 5.2), we use Apache Xalan-C++ [The Apache XML Project 2004b], a production XML serialization framework, to show how SAVI Objects enable programs to share subtype polymorphic objects to achieve better performance.

We use Clang [Lattner and Adve 2004] version 3.8.0 as a compiler framework and Boost [Boost C++ Libraries 2015a] as the region management library. We evaluate our results on a machine with an Intel Core i7 950 CPU (3.07GHz) and 24GB of memory.

Throughout this section, in the figures and text, we refer to virtual table duplication as *duplication* and to hashing-based dynamic dispatch as *hashing* for brevity.

5.1 Microbenchmarks

In this section, we use synthetic microbenchmarks to evaluate the impact of our techniques on construction time of polymorphic objects and dynamic dispatch time of polymorphic functions with respect to four properties of the program: the data set size, the number of regions used by the program, the number of polymorphic types, and the number of polymorphic functions per type. The synthetic microbenchmarks are designed as follows. A microbenchmark is configured with n objects, r regions, t polymorphic types, and f polymorphic functions per type. It starts by creating and mapping r shared regions into the process address space. In each region, it constructs $n/(r * t)$ objects for each of the t types, resulting in a total of n objects constructed across all regions. Then, for each of the n objects, it calls each of the f polymorphic function once resulting in $n * f$ total dispatches. The constructors are all empty and the virtual functions perform an integer comparison and an addition.

5.1.1 Construction Time Scalability. Fig. 7 shows the scalability of construction time for regular polymorphic objects, SAVI Objects that use duplication, and SAVI Objects that use hashing. It is evident from all graphs that only duplication suffers a penalty at object construction time, whereas hashing performs on par with regular objects. That is because at construction time, hashing replaces the VT pointer store with a type hash value store but does not do anything extra, whereas duplication performs two additional look-ups to the range table and DLT to locate the DVT.

The absolute overhead of duplication varies between 24ns and 120ns more than the baseline which is around 8ns for regular objects and for hashing. Although this may seem like a large relative overhead, recall that the constructors are empty to isolate the absolute value of the overhead. However, in practice, constructors perform varying amounts of work which amortizes the overhead as will be shown in Section 5.2

Fig. 7(a) shows that the construction time per object does not change for either technique as the number of objects allocated increases. This result indicates that the overhead will remain the

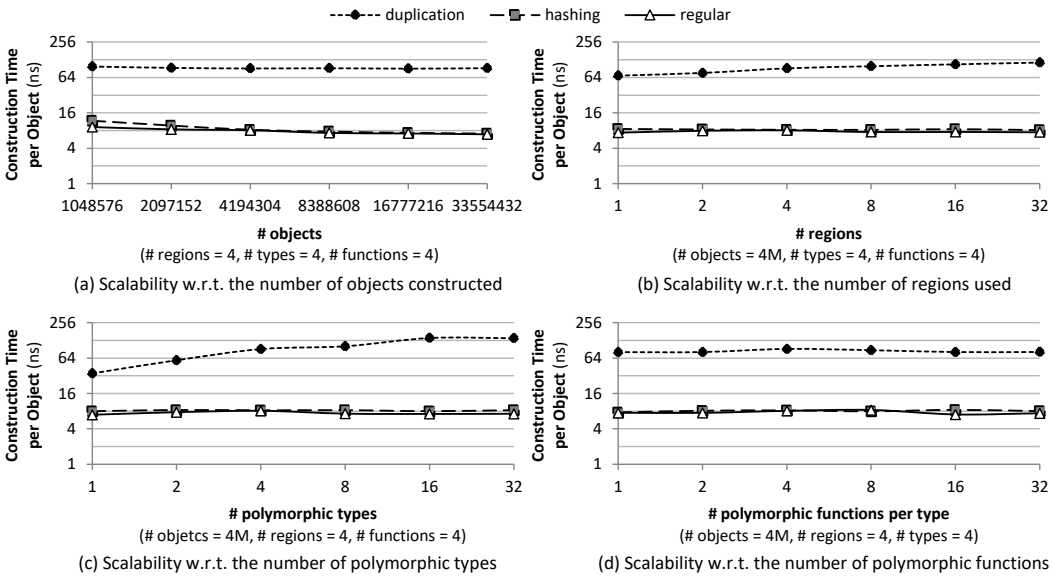

Fig. 7. Construction time scalability results

same as the dataset size increases such that the program will not suffer from any degradation in efficiency that will make the construction overhead dominate performance at scale.

Fig. 7(b) shows that the construction time per object increases for duplication as the number of shared regions used by the program increases. This increase comes from the range table look-up that is needed to identify the region within which an object is allocated because the range table look-up is logarithmic with respect to the number of regions simultaneously used by the program. In practice, we do not expect the same program to use a lot of regions simultaneously so this increase should not be a major issue.

Fig. 7(c) shows that the construction time per object increases for duplication as the number of polymorphic types increases. This increase comes from the DLT cache look-up because accessing the DLT cache is logarithmic with respect to the number of types it contains. However, while the total number of types in a program may be large, the DLT cache only contains polymorphic types which have had objects of the type constructed in a shared region by the current process which is expected to be much smaller in practice. Moreover, we currently do not perform any cache eviction, but doing so may help further limit the DLT cache access latency. This optimization is left as future work.

Fig. 7(d) shows that the construction time per object does not change for duplication as the number of polymorphic functions per type increases. The number of polymorphic functions only makes the size of the VTs and DVTs larger but does not impact any of the additional look-ups that need to be performed at construction time.

5.1.2 Dynamic Dispatch Time Scalability. Fig. 8 shows the scalability of dynamic dispatch time for regular polymorphic objects, SAVI Objects that use duplication, and SAVI Objects that use hashing. It is evident from all graphs that only hashing suffers a penalty at dynamic dispatch time, whereas duplication performs on par with regular objects. That is because duplication performs dynamic dispatch in the same way that regular objects do, whereas hashing requires a GVT look-up which incurs additional overhead. The exception for duplication is the first dispatch per type per

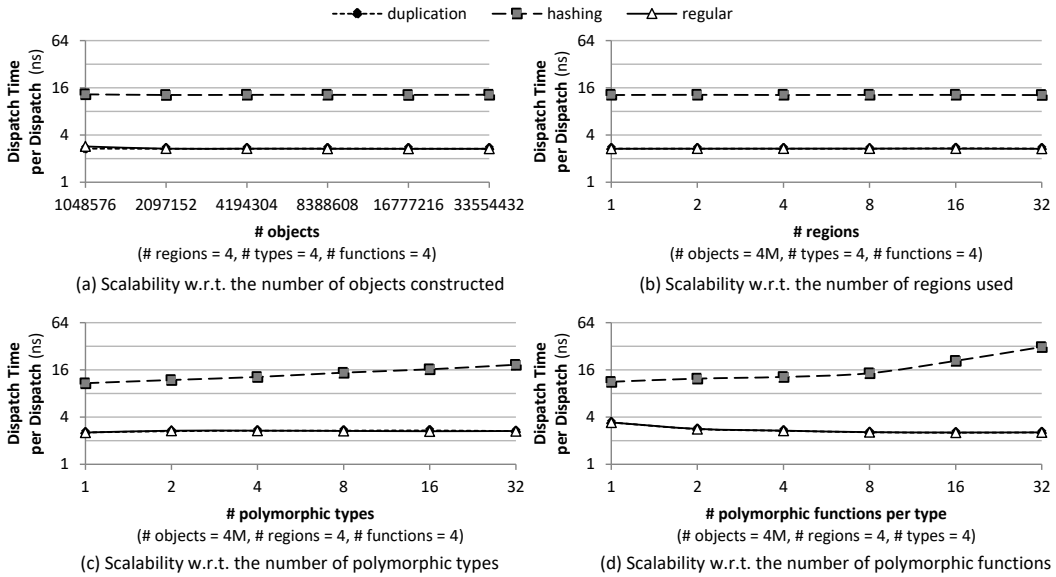

Fig. 8. Dynamic dispatch time scalability results

region which triggers the dispatch fault handler which takes around 2.2μs, however this overhead gets amortized.

The absolute overhead of hashing varies between 8ns and 30ns more than the baseline which is around 3ns for regular objects and for duplication. Although this may seem like a large relative overhead, recall that the polymorphic functions are nearly empty to isolate the absolute value of the overhead. However, in practice, functions may perform a lot more work which amortizes the overhead as will be shown in Section 5.2.

Fig. 8(a) shows that the dynamic dispatch time per dispatch does not change for either technique as the number of objects allocated increases. As in the previous section, this result demonstrates that efficiency does not degrade and that dynamic dispatch overhead will not dominate performance at scale. Fig. 8(b) shows that the dynamic dispatch time per dispatch does not change as the number of regions in the program changes.

Fig. 8(c) and Fig. 8(d) show that the dynamic dispatch time per dispatch for the hashing technique increases with both the number of polymorphic types and the number of polymorphic functions per type. That is because with hashing, a GVT look-up is performed on every dynamic dispatch. The GVT is currently implemented as a map of the hash values which makes the look-up logarithmic in the total number of functions in the GVT, thus making it logarithmic in the number of polymorphic types and number of polymorphic functions per type in the program. To further optimize the look-up, a hash table can be used instead of a map and a custom hash function can be computed on program entry that minimizes collisions. This will make the dynamic dispatch time shorter and with constant complexity at the expense of adding additional computation at program entry to compute a hash function. This optimization is left as future work.

Comparing the two techniques, we observe that with duplication, construction time increases while dynamic dispatch is free, whereas with hashing, dynamic dispatch time increases while construction is free. This observation reflects an important difference in the techniques' strengths and weaknesses. For applications that build a data structure once and use it many times, it is

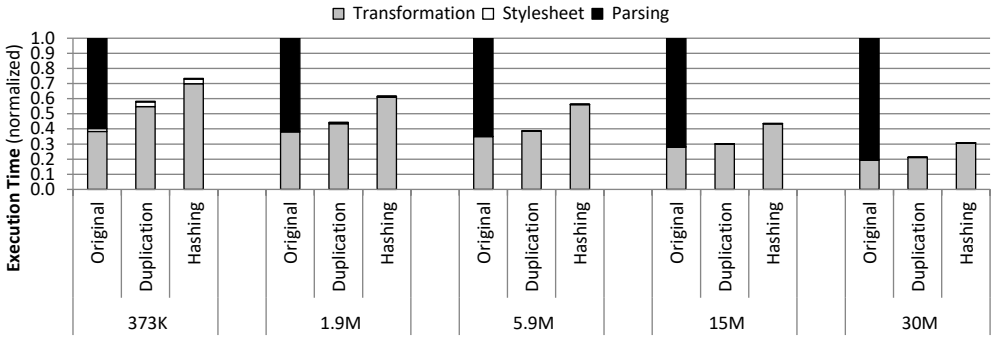

Fig. 9. Xalan-C++ execution time breakdown

expected that duplication is better. On the other hand, for applications that frequently create new objects, it is expected that hashing is better.

5.2 Application Case Study: Apache Xalan-C++

Apache Xalan-C++ [The Apache XML Project 2004b] is an open source framework that parses XML documents and transforms them into other formats, such as HTML or text, based on a XSLT stylesheet. The program is divided into three major steps: parsing the XML file, compiling the XSLT stylesheet, and applying the transformation.

Because the parsing step tends to dominate performance, the Xalan-C++ usage patterns advise users to parse the XML source once and reuse the parsed data structure when the same source is being transformed multiple times [The Apache XML Project 2004a]. The code cleanly separates the parsing step from the transformation in order to facilitate this kind of reuse. However, this reuse is confined to a single process because the data structure which stores the parsed XML file is rich with polymorphic objects which makes it not amenable to sharing with other processes or storing as-is for future processes to use.

To address this issue, we modify Xalan-C++ to store the parsed data structure in a shared memory region, and we transform the types that are polymorphic using SAVI Objects to make the sharing of the objects feasible. Modifying Xalan-C++ did not require pervasive changes to the source code because the application is designed to allow users to define their own custom memory allocators. We thus defined our own memory allocator which used Boost managed memory regions for allocation. Since the objects in Xalan-C++ use explicit data pointers, we used Boost regions with fixed-address mapping to handle those data pointers. However, SAVI Objects themselves do not require fixed mapping of regions to work, just DVTs, as explained in Section 3.4. In the resulting Xalan-C++ implementation, the first process that parses an XML file stores the parsed data structure using SAVI Objects in a memory region, and later processes that need to use the file can skip the parsing step entirely by reusing the SAVI Objects in the region. It is noteworthy that this application uses some polymorphic types that are defined inside dynamically linked libraries and that are involved in lineages with multiple inheritance, indicating our techniques already support dynamic linking and multiple inheritance. The dynamically linked libraries are part of the application source code so they are recompiled with the application to support SAVI Objects.

Fig. 9 shows the breakdown of the execution time of Xalan-C++ for the original version, and the two SAVI Objects versions using duplication and hashing. The breakdown is shown across increasing XML file sizes ranging from 373kB to 30MB for the same stylesheet. As shown in the graph, the parsing step dominates the performance, and consumes an increasingly large fraction

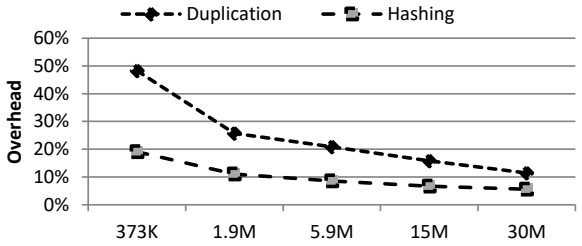

Fig. 10. Xalan-C++ parsing overhead for the first run

of the execution time as the size of the input XML file increases (up to 80% for the 30MB input). The compilation of the stylesheet takes up a negligible portion of the time, and the remaining time is spent performing the transformation. The total execution time of these runs ranges from tens of microseconds to a few seconds which can be significant in latency-sensitive services, and the fraction consumed by parsing is only expected to grow for even larger runs.

The results demonstrate that using SAVI Objects has enabled us to successfully eliminate the long time spent in parsing, replacing it with a simple mapping of the region containing the shared data structure. The speedup goes up to 5× for the largest input size for duplication which is the better technique in this application.

Comparing duplication and hashing, we find that duplication consistently outperforms hashing in the transformation step. That is because the transformation mainly operates on the existing objects thereby performing a lot of dynamic dispatch and little object construction. Thus, it makes sense that duplication outperforms hashing because duplication adds less overhead to dynamic dispatch than hashing does.

However, the weakness of duplication is that it adds more overhead to object construction than hashing does. Fig. 10 shows the overhead added to the parsing step of the first run when the SAVI Objects are being constructed in the shared region for the first time. Duplication incurs significantly more overhead than hashing does because the parsing step frequently performs object construction but seldom dispatches polymorphic function. This again demonstrates the tradeoff between the two techniques: duplication incurs a higher penalty at construction time than hashing in return for a lower penalty at dynamic dispatch time. However, in both cases, the overhead gets amortized as the size of the data structure increases.

Finally, we note that the overheads incurred from both techniques are quite reasonable as the size of the input increases. For the transformation step in Fig. 9 which is heavy on dynamic dispatch, the overhead of duplication is almost negligible for the large dataset and the overhead of hashing is around 50% which is reasonable compared to the savings we get from object sharing. For the parsing step in Fig. 10 which is heavy on object construction, the overhead of duplication drops to 11% for the large dataset and the overhead of hashing is just 6%. This observation verifies the points mentioned in Section 5.1 about the overheads getting amortized in practice.

6 RELATED WORK

6.1 Implementations of Dynamic Dispatch

A survey of techniques for implementing dynamic dispatch under subtype polymorphism has been done by Driesen et al. [Driesen et al. 1995]. These techniques aim at optimizing performance, whereas the techniques we propose also target enabling object sharing. Virtual tables are discussed, which our virtual table duplication technique augments to enable sharing. Techniques similar to

our hashing-based dynamic dispatch are also discussed, but they are not favored because they do not achieve the best performance or space efficiency in their evaluation context where object sharing is not considered.

Driesen & Hölzle [Driesen and Hölzle 1996] evaluate the overhead of dynamic dispatch in C++. Many works propose compiler techniques to mitigate dynamic dispatch overhead. Some techniques [Aigner and Hölzle 1996; Bacon 1997; Bacon and Sweeney 1996; Calder and Grunwald 1994; Dean et al. 1995; Detlefs and Agesen 1999; Fernandez 1995; Grove et al. 1995; Holzle and Ungar 1994; Ishizaki et al. 2000; Pande and Ryder 1996; Porat et al. 1996; Sundaresan et al. 2000; Wanderman-Milne and Li 2014] perform *devirtualization* which includes a set of analyses and transformations to statically resolve dynamic dispatches or replace them with checked static dispatches, thereby improving performance by exposing more inlining opportunities and more predictable branches. SAVI Objects can benefit from such transformations as well. While these techniques reduce the reliance on virtual tables, they do not eliminate them entirely so SAVI Objects are still needed for sharing. Others [Porat et al. 1996; Zendra et al. 1997] do eliminate virtual tables entirely, however they require whole program visibility which prevents separate compilation and is not always feasible in practice. A number of recent works aim at making whole-program optimization and link-time optimization more effective [Doeraene and Schlatter 2016; Johnson et al. 2017a; Sathyanathan et al. 2017] or at improving call graph analyses [Johnson et al. 2017b; Petrashko et al. 2016; Tan et al. 2017; Tip and Palsberg 2000] which can increase the scope and precision of devirtualization optimizations.

Hardware techniques have also been proposed to accelerate dynamic dispatch. Roth et al. [Roth et al. 1999] improve virtual function call target prediction by identifying instruction sequences that perform dynamic dispatch and using a small hardware engine to prefetch the target. Kim et al. [Kim et al. 2007] propose hardware support to perform devirtualization dynamically. The virtual function cache [Pentecost and Stratton 2015] makes dynamic dispatch faster by caching virtual table entries. Such hardware optimizations are also applicable for accelerating dynamic dispatch using SAVI Objects.

Other architecture optimizations have recently been proposed specifically targeting polymorphism in dynamic scripting languages. Typed Architectures [Kim et al. 2017] perform dynamic type checking implicitly in hardware and introduce polymorphic instructions. ShortCut [Choi et al. 2017] employs optimizations to bypass the dispatcher that performs type comparisons under most conditions. Dot et al. [Dot et al. 2017] remove type checks by performing hardware profiling of object types.

A lot of work has been done on security implications of how dynamic dispatch is implemented [Borchert and Spinczyk 2016; Bounov et al. 2016; Dewey and Giffin 2012; Elsabagh et al. 2017; Gawlik and Holz 2014; Haller et al. 2015; Jang et al. 2014; Miller et al. 2014; Prakash et al. 2015; Sarbinowski et al. 2016; Tice et al. 2014; Zhang et al. 2015, 2016; Zixiang et al. 2016]. Our work is concerned with object sharing, but security implications would be interesting to study.

6.2 Shared and Managed Data Structures

Boost [Boost C++ Libraries 2015a] is a state-of-the-art C++ library which supports sharing data structures between processes using shared memory segments or memory-mapped files. Interfaces are provided for creating managed memory objects that can be mapped into the address space of multiple processes concurrently or separated in time. However, subtype polymorphism is not currently handled [Boost C++ Libraries 2015b]. We demonstrate that SAVI Objects can be used to fill the gap.

A number of recent programming models for byte-addressed non-volatile memory such as Mnemosyne [Volos et al. 2011], NV-Heaps [Coburn et al. 2011], and others [Bhandari et al. 2016;

Chakrabarti et al. 2014] are based on managed persistent regions which are very similar to those used in Boost for volatile memory. The main focus of these works is guaranteeing failure atomicity of shared persistent data structures. Failure-atomicity and subtype polymorphism are orthogonal issues. SAVI Objects are expected to be enablers for emerging persistent memory programming models to support subtype polymorphism as they are for existing transient ones.

Representation of pointers in shared data structures has been dealt with in different ways. Offset pointers store offsets relative to the base address of a shared region instead of native pointers, but they are cumbersome for programmers to use. Fixed-address mapping of regions enables the use of native pointers, but creates the need to resolve mapping conflicts. OS techniques [El Hajj et al. 2016; Litton et al. 2016] have been proposed to address mapping conflicts by allowing processes to sustain multiple sets of mappings. Offset pointers and fixed-address mapping are both concerned with explicit data pointers and do not solve the problem of implicit virtual table pointers. Both treatments of data pointers can be used alongside SAVI Objects as discussed in Section 3.4.

There are a number of object serialization formats and frameworks that enable sharing of complex data structures across processes such as JSON [Crockford 2006], YAML [Ben-Kiki et al. 2005], XML [Bray et al. 1998], Protocol Buffers [Varda 2008], and Apache Avro [Apache Avro 2012]. Shared regions avoid the overheads of serialization and de-serialization when architecture- and language-neutrality are not needed, and SAVI Objects further widen the applicability of shared regions by enabling the use of subtype polymorphism in shared objects.

6.3 Sharing of Subtype Polymorphic Objects

The problem of sharing subtype polymorphic objects has been studied before. Hon [Hon 1994] presents a scheme for adding concurrency control to C++ objects in shared memory regions, and suggests that the original virtual tables be mapped to fixed addresses. We have discussed in Section 2.2 that this is not a practical solution because layouts of code binaries vary as programs evolve or across different programs, and because of ASLR. Vitillo [Vitillo 2013] suggests fixed mapping of dynamically linked binaries containing virtual tables. This approach still does not work if the layouts of the binaries vary, raises security concerns for bypassing ASLR, and exposes the DLL Hell [Pietrek 2000] problem. SAVI Objects do not require fixed mapping of code binaries in either technique, and virtual table duplication only requires fixed mapping of some data regions in processes that use them (different regions using the same type have different DVTs for that type that can go in different locations) which is more practical to resolve than DLL Hell. O++ [Biliris et al. 1993] is a database programming language on top of C++ that implements persistent polymorphic objects by reinitializing VT pointers when objects are loaded before they are used. This approach requires knowing the dynamic types of loaded objects beforehand which is not always practical, and does not support sharing of objects simultaneously by concurrent processes. E [Richardson and Carey 1989] and Burshteyn [Burshteyn 2014] suggest storing a unique field in the object which is similar to our hashing-based dynamic dispatch technique. E [Richardson and Carey 1989] uses this field to look up the VT for the type then looks up the virtual function in the VT, while Burshteyn [Burshteyn 2014] uses this field to look up a dummy object in the heap that acts as a proxy for performing the dynamic dispatch though the dummy object's VT pointer. Our technique looks up the dispatched function in a global virtual table directly. None of these works evaluate the performance implications of their suggested techniques.

Sharing subtype polymorphic objects between processor and accelerator address spaces in heterogeneous systems faces the same problems as sharing between the virtual address spaces of different processes. Ishizaki et al. [Ishizaki et al. 2015] perform devirtualization on GPUs when possible, but do not focus on sharing. Rootbeer [Pratt-Szeliga et al. 2012] handles dynamic dispatch on GPUs using a switch statement and a derived type id. This approach is similar to those discussed

earlier [Porat et al. 1996; Zendra et al. 1997] and has the same limitation; it requires whole program visibility which prevents separate compilation and is not always practical. Concord [Barik et al. 2014] supports sharing of subtype polymorphic objects between CPU and GPU virtual address spaces by duplicating virtual tables into shared virtual memory and including metadata to translated function pointers of the creator address space to those of the user address space. Our virtual table duplication does not share DVTs across virtual address spaces. It places DVTs at the same location in each address space, but DVTs in different address spaces contain different function pointers that are correct in those address space, therefore not requiring special checks during dynamic dispatch or visibility of derived types during compilation. Other suggested approaches [Yan et al. 2015; Zhou et al. 2015] have included using the CPU VT pointer to determine the GPU VT pointer, or placing the VT in a shared non-coherent region at the same address but having different contents on each device. The latter approach is similar to out virtual table duplication approach. Supporting sharing between processor and accelerator address spaces is expected to become increasingly important as the introduction of shared virtual memory enables closer collaboration between CPUs and accelerators [Chang et al. 2017; Gómez-Luna et al. 2017; Sun et al. 2016].

7 CONCLUSION

In this paper, we present SAVI Objects, an implementation of subtype polymorphism that supports sharing of objects across processes. We present and evaluate two alternative techniques for implementing SAVI Objects in C++. The first technique, virtual table duplication, duplicates virtual tables to a fixed virtual address associated with a shared region across all processes using that region. The second technique, hashing-based dynamic dispatch, performs dynamic dispatch using hashing-based look-ups to a global virtual table replacing the use of traditional virtual table pointers.

Our evaluation of the two techniques shows that in return for modest overhead costs added to object construction time and dynamic dispatch time, SAVI Objects enable programs to share objects that could not have been shared otherwise due to the presence of subtype polymorphism, thereby avoiding traditional inter-process communication in production applications and improving performance significantly. We also analyze the tradeoffs between the two techniques, showing that virtual table duplication is most suitable for programs heavy on dynamic dispatch, whereas hashing-based dynamic dispatch is most suitable for programs heavy on object construction.

ACKNOWLEDGMENTS

This work is supported by Hewlett Packard Labs and the Blue Waters PAID Use of Accelerators project (NSF OCI 07-25070 490595).

REFERENCES

Gerald Aigner and Urs Hölzle. 1996. Eliminating virtual function calls in C++ programs. In *European conference on object-oriented programming*. Springer, 142–166.

Apache Avro. 2012. Apache Avro. (2012). https://avro.apache.org

Krste Asanovic. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FASTâĂŹ14)*. USENIX Association, Santa Clara, CA, USA. https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote

David Francis Bacon. 1997. *Fast and effective optimization of statically typed object-oriented languages.* University of California, Berkeley.

David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 324–341. DOI: http://dx.doi.org/10.1145/236337.236371

Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T. Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. 2014. Efficient Mapping of Irregular C++ Applications to Integrated GPUs. In *Proceedings of Annual*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14). ACM, New York, NY, USA, Article 33, 11 pages. DOI: http://dx.doi.org/10.1145/2544137.2544165

Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2005. YAML Ain't Markup Language (YAMLâĎć) Version 1.1. *yaml. org, Tech. Rep* (2005).

Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J 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*. ACM, 677–694.

Alexandros Biliris, Shaul Dar, and Narain H. Gehani. 1993. Making C++ objects persistent: The hidden pointers. *Software: Practice and Experience* 23, 12 (1993), 1285–1303.

Boost C++ Libraries. 2015a. Managed Memory Segments. (2015). http://www.boost.org/doc/libs/1_61_0/doc/html/interprocess/managed_memory_segments.html

Boost C++ Libraries. 2015b. Sharing memory between processes – Virtuality Forbidden. (2015). http://www.boost.org/doc/libs/1_47_0/doc/html/interprocess/sharedmemorybetweenprocesses.html

Christoph Borchert and Olaf Spinczyk. 2016. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 37–43.

Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving.. In *Network and Distributed System Security Symposium (NDSS)*.

Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1998. Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210* 16 (1998), 16.

K. M. Bresniker, S. Singhal, and R. S. Williams. 2015. Adapting to Thrive in a New Economy of Memory Abundance. *Computer* 48, 12 (Dec 2015), 44–53. DOI: http://dx.doi.org/10.1109/MC.2015.368

B. Burshteyn. 2014. Method and system for accessing c++ objects in shared memory. (Feb. 6 2014). https://www.google.com/patents/US20140040566 US Patent App. 13/956,595.

Brad Calder and Dirk Grunwald. 1994. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 397–408.

Dhruva R. Chakrabarti, Hans-J. 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 '14)*. ACM, New York, NY, USA, 433–452. DOI: http://dx.doi.org/10.1145/2660193.2660224

Li-Wen Chang, Juan Gómez-Luna, Izzat El Hajj, Sitao Huang, Deming Chen, and Wen-mei Hwu. 2017. Collaborative Computing for Heterogeneous Integrated Systems. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 385–388.

Jiho Choi, Thomas Shull, Maria J Garzaran, and Josep Torrellas. 2017. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 494–506.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. DOI: http://dx.doi.org/10.1145/1950365.1950380

Douglas Crockford. 2006. The application/json media type for javascript object notation (json). (2006).

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.

David Detlefs and Ole Agesen. 1999. Inlining of Virtual Methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer-Verlag, 258–278.

David Dewey and Jonathon T Giffin. 2012. Static detection of C++ vtable escape vulnerabilities in binary code.. In *Network and Distributed System Security Symposium (NDSS)*.

Sébastien Doeraene and Tobias Schlatter. 2016. Parallel incremental whole-program optimizations for Scala. js. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 59–73.

Gem Dot, Alejandro Martínez, and Antonio González. 2017. Removing checks in dynamically typed languages through efficient profiling. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 257–268.

Karel Driesen and Urs Hölzle. 1996. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 306–323. DOI: http://dx.doi.org/10.1145/236337.236369

Karel Driesen, Urs Hölzle, and Jan Vitek. 1995. Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*. Springer, 253–282.

Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 353–368.

Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. 2017. Strict Virtual Call Integrity Checking for C++ Binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 140–154.

Mary F Fernandez. 1995. *Simple and effective link-time optimization of Modula-3 programs*. Vol. 30. ACM.

Robert Gawlik and Thorsten Holz. 2014. Towards automated integrity protection of C++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 396–405.

Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor Garcia-Flores, S de Gonzalo, T Jablin, Antonio J Pena, and WM Hwu. 2017. Chai: collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE*.

David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 1995. Profile-guided Receiver Class Prediction. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. ACM, New York, NY, USA, 108–123. DOI: http://dx.doi.org/10.1145/217838.217848

Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 341–350.

Urs Holzle and David M Ungar. 1994. *Adaptive optimization for SELF: reconciling high performance with exploratory programming*. Number 1520. Department of Computer Science, Stanford University.

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*. IBM Press, 29.

Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and optimizing java 8 programs for gpu execution. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 419–431.

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A study of devirtualization techniques for a Java Just-In-Time compiler. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 294–310.

Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks.. In *NDSS*.

Nick P. Johnson, Jordan Fix, Stephen R. Beard, Taewook Oh, Thomas B. Jablin, and David I. August. 2017b. A Collaborative Dependence Analysis Framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 148–159. http://dl.acm.org/citation.cfm?id=3049832.3049849

Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017a. ThinLTO: scalable and incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 111–121.

Channoh Kim, Jaehyeok Kim, Sungmin Kim, Dooyoung Kim, Namho Kim, Gitae Na, Young H Oh, Hyeon Gyu Cho, and Jae W Lee. 2017. Typed Architectures: Architectural Support for Lightweight Scripting. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 77–90.

Hyesoon Kim, José A Joao, Onur Mutlu, Chang Joo Lee, Yale N Patt, and Robert Cohn. 2007. VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 424–435.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

Stanley B Lippman. 1996. *Inside the C++ object model*. Vol. 242. Addison-Wesley Reading.

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight contexts: an OS abstraction for safety and performance. In *Proceedings of OSDIâĂŹ16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 49.

Matthew R Miller, Kenneth D Johnson, and Timothy William Burrell. 2014. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities. (March 25 2014). US Patent 8,683,583.

Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. DOI: http://dx.doi.org/10.1145/2150976.2151018

Hemant D Pande and Barbara G Ryder. 1996. Data-flow-based virtual function resolution. In *International Static Analysis Symposium*. Springer, 238–254.

Lillian Pentecost and John Stratton. 2015. Accelerating dynamically typed languages with a virtual function cache. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*. ACM, 3.

Dmitry Petrashko, Vlad Ureche, Ondřej Lhoták, and Martin Odersky. 2016. Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 394–409.

Matt Pietrek. 2000. Metadata in. NET-Avoiding DLL Hell: Introducing Application Metadata in the Microsoft. NET Framework. *MSDN Magazine* (2000), 42–55.

Sara Porat, , David Bernstein, , Yaroslav Fedorov, , Joseph Rodrigue, , and Eran Yahav. 1996. Compiler Optimization of C++ Virtual Function Calls. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*.

Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.. In *NDSS*.

Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. 2012. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 375–380.

Joel E Richardson and Michael J Carey. 1989. Persistence in the E language: Issues and implementation. *Softw., Pract. Exper.* 19, 12 (1989), 1115–1150.

Amir Roth, Andreas Moshovos, and Gurindar S Sohi. 1999. Improving virtual function call target prediction via dependence-based pre-computation. In *Proceedings of the 13th international conference on Supercomputing*. ACM, 356–364.

Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, New York, NY, USA, 448–459. DOI: http://dx.doi.org/10.1145/2991079.2991121

Patrick W Sathyanathan, Wenlei He, and Ten H Tzen. 2017. Incremental whole program optimization and compilation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 221–232.

Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–10.

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 264–280. DOI: http://dx.doi.org/10.1145/353171.353189

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 278–291.

The Apache XML Project. 2004a. Xalan-C++ Basic usage patterns. (2004). https://xalan.apache.org/old/xalan-c/usagepatterns.html

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

Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. 941–955.

Frank Tip and Jens Palsberg. 2000. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 281–293. DOI: http://dx.doi.org/10.1145/353171.353190

Kenton Varda. 2008. Protocol buffers: Google's data interchange format. (2008). https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html

Roberto Agostino Vitillo. 2013. Sharing C++ objects in Linux. (2013). http://www.slideshare.net/RobertoAgostinoVitil/sharing-objects2011

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. DOI: http://dx.doi.org/10.1145/1950365.1950379

Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.

Wikipedia. 2017. Birthday attack. (2017). https://en.wikipedia.org/wiki/Birthday_attack

Shoumeng Yan, Sai Luo, Xiaocheng Zhou, Ying Gao, Hu Chen, and Bratin Saha. 2015. Sharing virtual functions in a shared virtual memory between heterogeneous processors of a computing platform. (March 31 2015). US Patent 8,997,113.

Olivier Zendra, Dominique Colnet, and Suzanne Collin. 1997. Efficient Dynamic Dispatch Without Virtual Function Tables: The SmallEiffel Compiler. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 125–141. DOI: http://dx.doi.org/10.1145/263698.263728

Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Defending virtual function tablesâĂŹ integrity. In *Symposium on Network and Distributed System Security (NDSS)*. 8–11.

Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *Network and Distributed System Security Symposium (NDSS)*.

Xiaocheng Zhou, Shoumeng Yan, Ying Gao, Hu Chen, Peinan Zhang, Mohan Rajagopalan, Avi Mendelson, and Bratin Saha. 2015. Language level support for shared virtual memory. (March 31 2015). US Patent 8,997,114.

Wang Zixiang, Shan Chun, Xue Jingfeng, and Sun Shiyouhu Changzhen. 2016. Research on the Defense Method of Vtable Hijacking. *International Journal of Security and Its Applications* 10, 11 (2016), 267–280.