Type Information Elimination from Objects on Architectures with Tagged Pointers Support

Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján

Abstract—Implementations of object-oriented programming languages associate type information with each object to perform various runtime tasks such as dynamic dispatch, type introspection, and reflection. A common means of storing such relation is by inserting a pointer to the associated type information into every object. Such an approach, however, introduces memory and performance overheads when compared with non-object-oriented languages. Recent 64-bit computer architectures have added support for *tagged pointers* by ignoring a number of bits - *tag* - of memory addresses during memory access operations and utilize them for other purposes; mainly security. This paper presents the first investigation into how this hardware support can be exploited by a Java Virtual Machine to remove type information from objects. Moreover, we propose novel hardware extensions to the address generation and load-store units to achieve low-overhead type information retrieval and tagged object pointers compression-decompression. The evaluation has been conducted after integrating the Maxine VM and the ZSim microarchitectural simulator. The results, across all the DaCapo benchmark suite, pseudo-SPECjbb2005, SLAMBench and GraphChi-PR executed to completion, show up to 26 and 10 percent geometric mean heap space savings, up to 50 and 12 percent geometric mean dynamic DRAM energy reduction, and up to 49 and 3 percent geometric mean execution time reduction with no significant performance regressions.

Index Terms—Runtime environments, high-level language architectures, simulation

1 Introduction

Managed runtime environments are extensively used in many computing domains ranging from mobile devices to cloud servers. Managed object-oriented languages have been employed not only in application and middleware domains but also in system programming for the development of research prototypes such as the Maxine *Virtual Machine* (VM) [1], [2], Jikes RVM [3], and the Singularity OS [4].

Implementations of managed object-oriented languages associate object type information with each object by inserting a pointer to type information into every object as part of its object header. However such an approach, prevalent for most object-oriented languages, increases memory utilization (and can introduce performance overheads) when compared with non-object-oriented languages.

At the same time, modern 64-bit computer architectures have added support for *tagged pointers*. In such architectures, a number of bits - *tag* - of memory addresses are ignored during memory access operations and utilized for other purposes; mainly security. Furthermore, they provide less than 64-bit addressable memory space, leaving a number of bits in object pointers for useful purposes.

In this paper, we present the first investigation into how tagged pointers in 64-bit architectures can be exploited by an object-oriented language implementation to remove a pointer to type information from objects. In addition, we propose novel hardware extensions to the address generation and load-store units to achieve low-overhead type information retrieval and tagged object pointers compression-decompression, respectively. In other words, we explore a *Hardware* (HW)/*Software* (SW) co-designed technique in the context of the Java programming language, although the proposed technique is applicable to other managed and unmanaged object-oriented languages.

The key contributions of this paper are:

- *A technique of type information elimination* from object headers on architectures with tagged pointers support. We propose both a SW-only along with a HW/SW co-designed solution that yields the best results.
- *Novel backward-compatible HW extensions* (1) to the *Address Generation Unit* (AGU) to efficiently retrieve type information for objects with type information elimination enabled (optimized) and for objects without (un-optimized), and (2) to the *Load-Store Unit* (LSU) to efficiently handle compression and decompression of tagged object pointers.
- *Demonstration of a novel experimental platform for HW/SW co-design space exploration* on the basis of the state-of-the-art Maxine VM, the ZSim [5] microarchitectural simulator, and the McPAT [6] power, area, and timing modeling framework. Parts of this experimental platform laid the foundation for the MaxSim [7] platform with more general and advanced features.

- *The authors are with the School of Computer Science, University of Manchester, Manchester M13 9PL, United Kingdom. E-mail: {andrey.rodchenko, christos.kotselidis, andy.nisbet, antoniu.pop, mikel.lujan}@manchester.ac.uk.*

Manuscript received 9 Jan. 2017; revised 15 May 2017; accepted 23 May 2017. Date of publication 28 June 2017; date of current version 19 Dec. 2017. (Corresponding author: Andrey Rodchenko.) Recommended for acceptance by O. Plata. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2709739

TABLE 1
Glossary of Terminology

Acronym	Full name	Description
CI	Class Information	The entity that entails the type information of an object. CI normally contains some metadata for dynamic dispatch (virtual method table), classification (pointer to supertype), object layout description (size, fields information), and other implementation-dependent metadata.
CIP	Class Information Pointer	A raw pointer to (address of) the CI. Typically stored in extra words preserved for each object before its content.
CID	Class Identifier	A compact non-negative integer identifier of a CI.

- *The evaluation of the proposed technique* in the context of the introduced platform against the DaCapo-9.12-bach [8] benchmark suite, pseudo-SPECjbb2005 [9], SLAMBench [10] and GraphChi-PR [11] on several HW models achieving up to 26 and 10 percent geometric mean heap space savings, up to 50 and 12 percent geometric mean *Dynamic Random-Access Memory* (DRAM) dynamic energy reduction, and up to 49 and 3 percent geometric mean execution time reduction with no significant regressions in these characteristics.

The paper is organized as follows: Section 2 presents a survey of how type information is associated with objects in modern *Java Virtual Machine* (JVM) implementations, as well as how tagged pointer support works in modern architectures. Section 3 details the proposed technique along with the accompanying changes to the JVM implementation. Section 4 describes the proposed architectural support for retrieval of type information with extensions to the AGU and for tagged pointers load-decompression and store-compression with extensions to the LSU. Sections 5 and 6 describe the experimental framework and methodology used in this work along with the obtained results, respectively. Finally, Section 7 discusses previous work on type information elimination, while Section 8 summarizes our work.

2 Background

Next, we explain how type information is typically associated with objects in a number of modern industrial-strength and research JVMs. Although we focus on type information elimination from object headers in the context of the JVM implementation and the Java language, it is important to mention that the proposed ideas are applicable to C++ and other non-managed and managed object-oriented languages with runtime type information associated with objects. Hereafter, we will use the terms *Class Information* (CI), *Class Information Pointer* (CIP), and *Class Identifier* (CID), whose descriptions are presented in Table 1. Furthermore, we provide an overview of the latest computer architectures that support tagged pointers and succinctly how they implement this support.

2.1 Association of Objects with Class Information in JVMs

Fig. 1 presents the relationship between an object, its pointer, and its associated CI. When a new object is allocated on the heap, its pointer is stored on the stack. When an object tests if it is an instance of some class, type introspection happens via data stored in the associated CI, which is referenced by the CIP stored in the object. Aside from the CIP and the fields of the object's class, an object reserves some extra space for special *miscellaneous data* (MISC) that can be associated with it during its lifetime.

The small survey below concerns only 64-bit JVMs since, to the best of our knowledge, there are no modern 32-bit systems that support tagged pointers. The layouts of object headers, to be discussed below, are presented in Fig. 2.

HotSpot. On 64-bit systems, the first eight bytes of an object are dedicated to the "mark word". This word is multi-purpose and is currently used for hashing, locking, and *Garbage Collection* (GC) information. The second eight bytes of an object contain the "klass pointer" (essentially a CIP), which can be reduced to four bytes when the flag `-XX:+UseCompressedClassPointers` is used (enabled by default in OpenJDK 8 [12]).

Zing. Azul's Zing uses only eight bytes as an object header, four of which are dedicated to a compressed CIP [13].

Maxine. The meta-circular research Maxine VM, written mostly in Java, uses two words as an object header in the default object layout scheme. The first eight bytes contain the reference to the "hub" (essentially a CIP) or a forwarding pointer during copying GC. The second eight bytes are dedicated to the MISC word responsible for object locking and hashing.

Jikes. The meta-circular 64-bit Jikes RVM (which is also written mostly in Java), like Maxine, has a two-word object header layout. The main difference is that a forwarding pointer is stored in the second word.

Fields of objects in the described VMs are located at positive offsets after their headers (by default). Although it is possible to lay out an object in memory in fragments [14], in all other sections, without loss of generality, we assume that objects are allocated in contiguous blocks of memory.

2.2 Architectural Support for Tagged Pointers

The shift from 32-bit computing to 64-bit has started from server and desktop deployments and continued to embedded computing after the introduction of the ARM 64-bit processor family. Current 64-bit architectures, normally, provide less than 64-bit addressable memory space leaving a number of bits of an address unused. Dealing with these unused bits is architecture-dependent, and the following paragraphs describe how several modern architectures handle them.

AArch64. ARM's latest 64-bit AArch64 architecture provides support for tagged pointers. Virtual address tagging in AArch64 is enabled by setting the `Top Byte Ignore` field in the `TCR_ELn` control register. In this case, the high eight bits are ignored during addressing and can be utilized by the developers in an unmandated way. However, hints for exploitation in object-oriented languages are given in the associated programmer's guide [15]. The most recent `ARMv8.3-A` version of the architecture [16] features pointer authentication to prevent unauthorized memory accesses and associated exploits [17].

Sparc M7. Oracle's Sparc M7 architecture also provides tagged pointers support. Sparc M7 supports virtual address masking, allowing the use of 8, 16, 24 or 32-bit metadata.

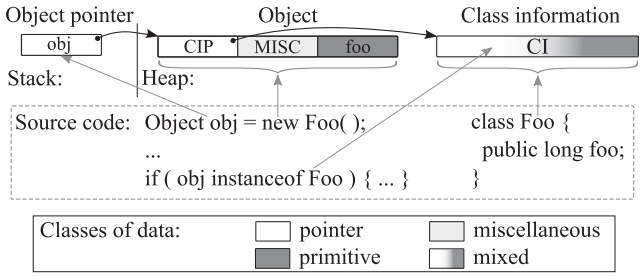


Fig. 1. Object and class information association in VMs.

This metadata is, consequently, ignored by the underlying HW during addressing [18], [19].

x86-64. In current Intel's and AMD's x86-64 architectures virtual addressing is limited to 48 bits, while the high 16 bits of the virtual address are required to replicate bit 47. Consequently, tagged addressing is not supported on such architectures [20] (Section 3.3.7.1). However, the property that 16 bits are not effectively used can be utilized during simulation, which is described in Section 5. In future generations of processors, virtual addressing will be limited to 57 bits [21].

From the architectural and JVM descriptions, we notice that the majority of production JVMs use CIP compression in order to minimize the memory footprint of objects. At the same time, modern off-the-shelf CPUs offer tagged pointer support making them ideal candidates for storing CIDs in tags, thus eliminating CIPs from frequently allocated objects.

3 Class Information Handling via Tagged Pointers

This section describes how tagged pointers can be utilized to associate object pointers with CIs, what additional data structures are required in a VM, and how objects with eliminated CIPs can be handled by a VM.

3.1 Considerations on CIP Placement Inside an Object and Reuse of CIP Location

The main benefit of CIP elimination from objects is memory space saving, so any reuse of the CIP location should be disabled to take advantage of CIP elimination. Among the VMs described in Section 2, only the Maxine VM reuses it, as a forwarding pointer is stored in the CIP location of an object during GC. The way to disable the reuse of the CIP location in the Maxine VM will be described in Section 5.2.

The other important factor for benefiting from CIP elimination is CIP placement inside the object and the memory management mechanism used in the VM. If free memory chunks are managed in the way of linked lists of fixed-size blocks and utilization of memory blocks of the size of CIP is high, then the technique can be oblivious to CIP placement.

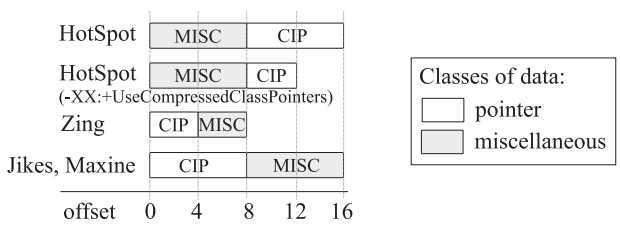


Fig. 2. Layout of object headers in various 64-bit JVMs.

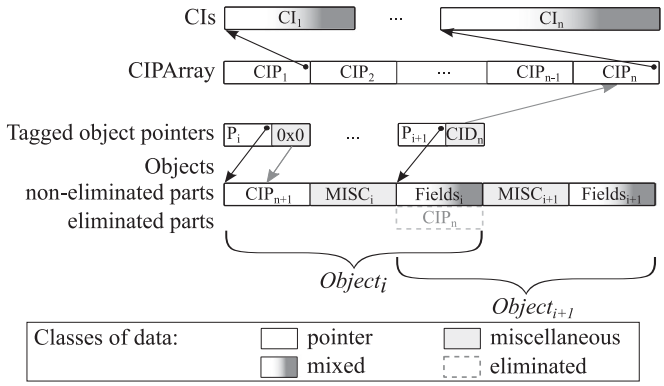


Fig. 3. Scheme of encoding CIDs in tagged pointers and CIP elimination.

However, if objects are allocated in thread-local allocation buffers by returning and post-incrementing a pointer to free memory space by the size of a recently allocated object,[1] then a CIP should be placed on the boundary of the allocated block of memory for an object. Among the VMs described in Section 2, only the HotSpot VM does not meet this requirement. However, as the implementations of other VMs show, there are no fundamental restrictions on the CIP placement, and it can be placed at the beginning of the allocated block of memory for an object. In all other sections, without loss of generality, we assume that CIP is located at the beginning of the allocated block of memory for an object.

3.2 Encoding CIDs in Tagged Pointers

The number of CIDs that can be encoded in a tagged pointer depends on the number of bits dedicated for that purpose. The proposed technique can utilize a variable number of tag bits from 0 to n. If all tag bits are used for security or addressing purposes and no bits are left for encoding CIDs, a VM, implementing the proposed technique, does not have to be re-implemented. In other words, the proposed technique can be added to existing VMs without breaking backward compatibility.

For n bits dedicated for storing CIDs in a tag, the range of CIDs is $[0; 2^n - 1]$ where `UNSPECIFIED_CID` $= 0$ represents any CI. When an object is allocated, the pointer is tagged by its respective CID. If the CID is not equal to `UNSPECIFIED_CID`, its CIP is not stored alongside the object (i.e., in the heap). Instead, the CID is directly encoded in the object pointer. The CI for specified CIDs can be stored directly in an array with a fixed element size, or alternatively, CIPs can be stored in the array. We use the second option because CIs have variable sizes while the cost of performing array modifications with CIPs is lower than in the first option.

The scheme of encoding CIDs in tagged pointers and enabling CIP elimination is depicted in Fig. 3. There are two tagged object pointers in Fig. 3: $\mathtt{P}_i$ and $\mathtt{P}_{i+1}$ pointing to $Object_i$ and $Object_{i+1}$, respectively. Pointer $\mathtt{P}_i$ has `UNSPECIFIED_CID` (0x0) in the tag indicating that $\mathtt{CIP}_{n+1}$ is stored in the object, while pointer $\mathtt{P}_{i+1}$ has $\mathtt{CID}_n$ in the tag indicating that $\mathtt{CIP}_n$ is stored in `CIPArray`, and so it is eliminated from $Object_{i+1}$. In this case, $\mathtt{P}_{i+1}$ points to the memory location where the beginning of the object would

1. This technique is also known as a bump pointer allocation, and it is widely used in JVM implementations.

have been if $\texttt{CIP}_n$ was not eliminated. Thus, the offsets to the same fields in objects of the same class (or superclass) with and without eliminated CIPs will stay the same.

3.3 CIPs Retrieval from Tagged Pointers

From the above description, the retrieval of a CIP from a given tagged object pointer (`objectAddress`) can be performed via the method `retrieveCIP` shown in Listing 1. Depending on the available bit extraction instructions of a given architecture and whether the available bits are contiguous, the method `extractCID` can result in a different number of instructions. For the rest of the paper, we assume that CID bits are contiguous high-order bits of the tag, and tag bits are contiguous high-order bits of the tagged pointer. Thus, we can use only one instruction for `extractCID`; a shift operation.

A check of whether the extracted CID is unspecified is performed. If `cid` is unspecified, CIP is loaded by an object address at a constant offset `CIP_OFFSET` in one instruction. Else, if `cid` is specified, the CIP is loaded from `CIPArray` with one instruction. In our experiments we used `CIP_OFFSET` $= 0$. On architectures without predication, two jumps (conditional and unconditional) will be emitted.

The code in Listing 1 is compiled to seven static instructions of 25 bytes size with an average dynamic execution height of 5.5 instructions on x86-64. By contrast, the JVMs described in Section 2 utilize just one instruction (load by `objectAddress` at offset `CIP_OFFSET`) of three bytes size on x86-64 to retrieve a CIP. Regarding storing a CIP in an object, the similar code snippet is used with the exception of omitting the store to `CIPArray` if `cid` is not unspecified.

Listing 1. CIP retrieval algorithm from tagged pointers.

```
// Array of Class Information Pointers
CIP_t CIPArray[ 1 << CID_BITS_NUM ];
// Retrieving CIP
CIP_t retrieveCIP( Address_t objectAddress) {
  CID_t cid = extractCID( objectAddress);
  CIP_t res;
  if ( cid == UNSPECIFIED_CID ) {
    res = * (( CIP_t *) (objectAddress + CIP_OFFSET));
  } else {
    res = CIPArray[ cid ];
  }
  return res;
}
```

3.4 Heap Traversal During Copying GC

When CIPs are eliminated from objects, certain changes to the copying GC schemes have to be applied. The problem is that traversal of copied objects in the copying GC schemes is performed via pointer arithmetic (untagged pointers), and CIPs in the objects are used to find the references inside the copied objects and their sizes. Thus, when CIPs are eliminated, the association of copied objects with their CIs should be maintained via an additional data structure. It is important to note that our proposed changes to the copying GC schemes do not require additional memory allocation.

The necessary changes are introduced in the context of the serial stop-the-world semi-space copying GC which employs Cheney's algorithm [22]. Modern generational GC algorithms employ a copying scheme based on Cheney's breadth-first copying GC scheme for frequent young generation collections, and the proposed changes can be applied to a wider spectrum of copying collectors (including parallel).

3.4.1 Cheney's Scheme Overview and Its Reliance on CI

Overview. In Cheney's scheme, objects are allocated in the "from-space", and upon a GC invocation, live objects are copied to the "to-space". Upon completion, the two spaces are swapped, and allocation continues in the "from-space". During a GC invocation, all threads stop at a safepoint where an initial set of live heap references can be retrieved. The initial set of live objects (typically includes objects whose references reside on the stack, in thread-local variables, etc.) is known as *root set*. When objects from the root set are copied to the "to-space", the GC threads start traversing the objects in the "to-space" in order to copy all the objects referenced by them. This iterative process is repeated until all live objects are copied from the "from-space". Essentially it is a breadth-first traversal of all live objects.

Reliance on CI. As the GC thread starts scanning the copied objects of the "to-space", it usually uses the CIs in order to find both the reference maps of the processed objects as well as their sizes in order to continue its traversal on subsequent objects. In the proposed CIP elimination scheme, the GC thread which traverses the "to-space" objects does not have CIPs in order to extract CIs during execution. Hence, a mechanism to maintain the mapping between copied objects and their CIs is required.

3.4.2 Proposed Changes

Overview. We propose a scheme in which CIDs of the copied objects are stored in the "dead space" created upon the evacuation of live objects from the "from-space". The cells of memory in the "from-space" which contained copied live objects will be referred to as *containers*. The outline of the proposed scheme is depicted in Fig. 4.[2]

Copying Objects from the "from-Space". When objects are copied from the "from-space" to the "to-space", a tagged forwarding pointer is installed in the MISC words of the evacuated objects in the "from-space". The role of the forwarding pointer during GC is to indicate whether the object has been evacuated and where its new location is. Containers are used to store CIDs of evacuated objects until they are full (e.g., $[\texttt{CID}_0;\texttt{CID}_n]$ in $Container_0$), forming a singly linked list (e.g., $Container_0[\texttt{CID}_0;\texttt{CID}_n] \rightarrow Container_1[\texttt{CID}_{n+1}; ...] \rightarrow ...$). Furthermore, the space from an evacuated object should meet *a minimum necessary container size requirement* to be used in the list: the space should be enough to accommodate a tagged forwarding pointer, an untagged pointer to the next container, and at least one CID.

Traversing Objects in the "to-Space". When GC roots are copied from the "from-space" to the "to-space", the GC thread starts traversing linearly the objects that have just been copied there. As described before, the pointers to these

2. Scales of "from-space" and "to-space" are different, so the sizes of $Container_0$ and $Object_0$ are equal.

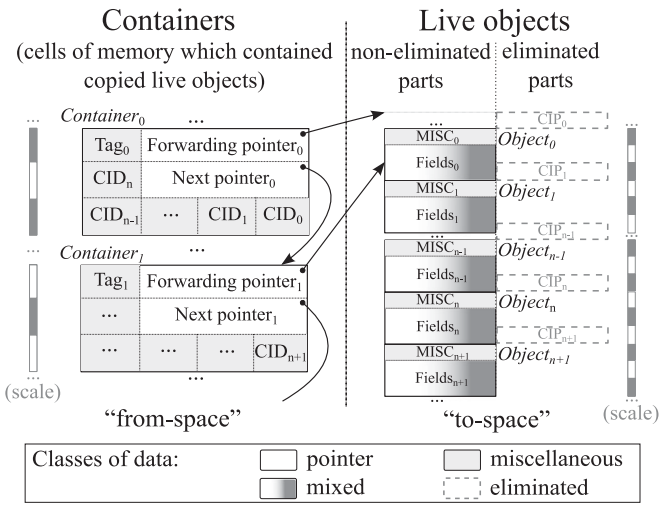


Fig. 4. List of CIDs in the "from-space" during GC representing list of copied objects in "to-space".

objects are calculated via pointer arithmetic during traversal and, therefore, the GC thread has to retrieve their CIs by reading the CIDs from the list of CIDs in the "from-space". In order to achieve that, the GC thread maintains a CID list iterator, which consists of an address to the current container and an offset inside it. When a GC thread traverses an object in the "to-space", it reads its CID using this iterator. By iterating the objects of the "to-space" and the list of CIDs synchronously we can associate currently traversed objects and their CIDs.

Traversing the List of CIDs in the "from-Space". The containers are traversed with untagged pointers. Initially, when the first container ($Container_0$) is traversed, the offset of the iterator is reset to the size of the container minus the size of the CID. The size of the container can be found by using the CID extracted from $\texttt{Tag}_0$. After the initial offset is set to the offset of $\texttt{CID}_0$, the iterator will start traversing the CIDs backward by decrementing the offset every time by CID size. When the CID list iterator points to $\texttt{CID}_{n-1}$, which is a constant offset from the beginning of a container, we know that we reached the point where there is only one $\texttt{CID}_n$ left in the container. Therefore, when $Object_n$ is traversed, the iterator offset will be set to the offset of $\texttt{CID}_n$. Since the next container pointer $\texttt{Next pointer}_0$ is untagged, we have space to save $\texttt{CID}_n$. When $Object_{n+1}$ is traversed, we move to $Container_1$, the iterator offset is set to the offset of $\texttt{CID}_{n+1}$, and the process is repeated for that container.

Special Cases. Regarding the first container, when GC is triggered, we use the free space from the "from-space", if it is equal to or greater than the minimum necessary container size. In order to guarantee that there will be enough space to store the list of CIDs, we account for the space needed conservatively for GC during allocation. Small objects, not meeting the minimum necessary container size requirement, are counted during allocation, and their total number multiplied by the CID size is subtracted from the maximum heap space occupied before triggering a GC. Such small objects are quite rare in practice and do not reduce significantly the effective heap usage. For instance, in 64-bit VMs in which objects are 8-byte aligned and the size of the MISC word is 8 bytes (all the VMs described in Section 2 with the exception of Zing), only objects that contain just one MISC word do not meet the minimum necessary container size requirement (e.g., objects of class `java.lang.Object`).

To summarize, the benefits of the proposed scheme are: (1) re-use of existing "dead space" requiring no off-heap memory allocation to save CIDs, (2) exploitation of temporal locality of cache lines related to dead objects, (3) low-overhead traversal, by employing a light-weight heuristic during object evacuation which utilizes only large evacuated objects for containers.

4 Architectural Support

4.1 CIP Retrieval

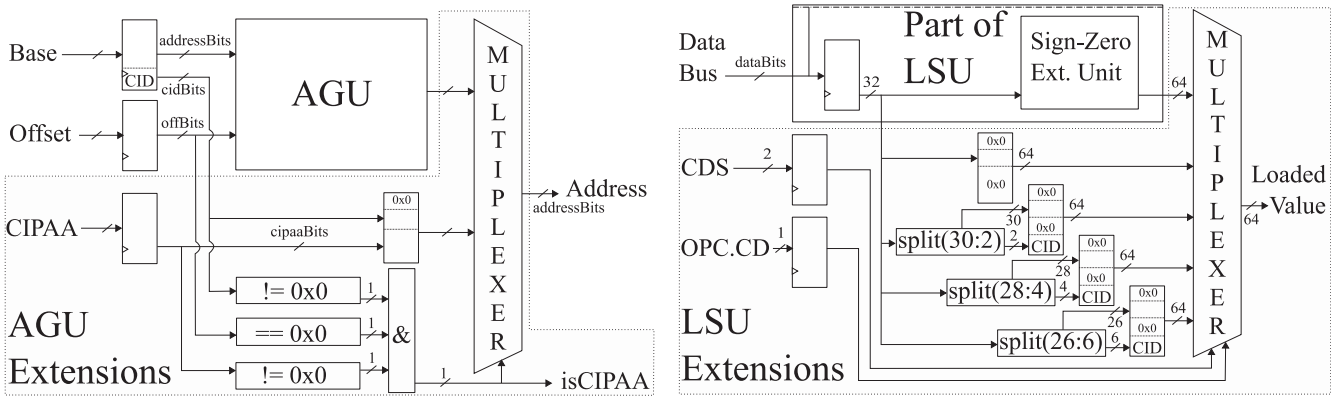
In Section 3 we have described the CIP retrieval from tagged pointers (Listing 1). As shown in Section 6, the SW-only scheme of CIP retrieval from tagged pointers can lead to degradation of the execution time due to the extra latency introduced and code footprint increase. To avoid performance regressions, we propose novel architectural support to accelerate CIP retrieval, as well as to minimize the modifications needed to adopt the CIP elimination scheme in existing VMs.

The scheme relies on the property that memory accesses to object fields are performed by using a two-or-more operand addressing mode (in the form of `[Base + Offset]`).[3] Consequently, CIPs should always be accessed by addresses in the form of `[objectAddress + CIP_OFFSET]`. If more than two operands are used during addressing, one of them should represent the base address while the rest should represent the offset. This pattern, assuming it is preserved by the VM, can be identified by HW and handled accordingly depending on the CID value encoded in the tag.

The goal of the HW extension is to perform the CIP retrieval presented in Listing 1 simply by executing a single load instruction at an object address plus a constant offset `CIP_OFFSET`, as it happens in the VMs described in Section 2. The aforementioned address pattern can be identified by the *Address Generation Unit* (AGU) in the proposed AGU extension functionality presented in Fig. 5a. In this scheme, we present the functional block of an AGU of a given processor, similarly to [23]. The input to the AGU is represented by two operands: `Base` and `Offset`. The extended AGU has an extra operand, *Class Information Pointer Array Address* (`CIPAA`), which is stored in a non-frequently changed control register. The control register has the same value for all AGUs, in case a core has several AGUs. The `CIPAA` control register holds the address of `CIPArray` and it is defined by the VM. Furthermore, it is required to be aligned to its size ($= 2^n$) in order to calculate the effective address of the retrieved CIP location just by combining the `CID` at an offset shifted by $log_2(\texttt{sizeof(CIP\_t)})$ bits and with zeros before this offset.

The output address depends on the *"is Class Information Pointer Array Access"* (`isCIPAA`) signal. Since both addresses from the AGU block and the extension (combiner) can be generated in parallel, the proposed scheme adds no significant delays to the address generation. The `CIPAA` is stored in a special purpose control register and can be read only by the AGU. When the `CIPAA` is zero, the extended address

3. This addressing mode is supported by the majority of architectures, and this property has to be preserved by the VM.



(a) Extensions to Address Generation Unit (AGU) for CIPs retrieval

(b) Extensions to Load-Store Unit (LSU) for object pointers decompression

Fig. 5. Architectural support for CIPs retrieval and object pointers compression-decompression.

generation can be fully deactivated. When the `CID` is equal to `UNSPECIFIED_CID` or the `Offset` is different from `CIP_OFFSET`, then generation happens in the unextended way. Both `UNSPECIFIED_CID` and `CIP_OFFSET` are represented by all zeroes in our scheme for AGU extension simplicity, although they can be set by the control register as well.

4.2 Tagged Pointers Compression-Decompression

To reduce the memory footprint of objects, 64-bit VMs apply the object pointers compression optimization [24]. Depending on the heap size and object alignment, the possible values of object pointers can be represented by 32 or fewer bits, and in this case any 64-bit object pointer can be stored in a 32-bit location. An untagged object pointer compression and decompression can require shift and/or add operations or no extra operations depending on the heap size and base address of the heap. However, tagged pointers compression can require extra bit manipulation instructions to gather and scatter tag and non-tag object pointer bits. To avoid this overhead, we propose the following *Load-Store Unit* (LSU) extension.

The goal of the proposed LSU extension is to perform scattering of 32 loaded bits of a compressed tagged object pointer during load operation to a 64-bit register and gathering them during reverse store operation in a limited number of ways efficiently. Fig. 5b presents the necessary extensions to the LSU for load-decompress operation and the part of the LSU responsible for the sign-zero extension. The proposed LSU extension is activated by an opcode of the memory access instruction (`OPC.CD`) indicating that compression-decompression should be performed.

The extended LSU has an extra operand, *Compression-Decompression Selector* (`CDS`), which is a non-frequently changed control register. The `CDS` defines which bits are compressed-decompressed and it can have four states (s0-s3) depending on the heap sizes and used tag bits. In our scheme, these states are: (s0) 32 GB and 0 bits, (s1) 8 GB and 2 bits, (s2) 2 GB and 4 bits, and (s3) 512 MB and 6 bits. In the presented scheme, addresses of objects are required to be 8-byte aligned. State (s0) is needed to be able to reach the maximum heap size without code recompilation when compressed pointers are used. Before changing the `CDS` state to another state with fewer tag bits in use, objects, whose pointers are utilizing the bits which will no longer fit into compressed pointers, must be handled to release them. This handling means restoring the eliminated CIPs of such objects and untagging their pointers. A larger number of states and different object alignments can be supported by extending the `CDS` and the scheme. The reverse store-compress operation is performed likewise but is not shown in the scheme for simplicity.

4.3 ISA Modifications

The control of the proposed AGU extension can be performed by the introduction and modification of a dedicated control register CR_n. If CR_n is set to zero, the extra functionality of the AGU will be switched off. The functionality is enabled when CR_n is set to a valid aligned `CIPAA`. When a tagged address in the form of `[CID:Base + CIP_OFFSET]` is passed to the extended AGU, the generated memory access will be [`CIPAA` | (`CID` $\ll log_2($`sizeof(CIP_t)`$)$)]. If such an address is generated during a store operation, it will be executed as a NOP instruction. The `CIPAA` stored in CR_n should be maintained by the VM and new CIPs should be stored in `CIPArray`. If `CID` is unspecified or the offset is different from `CIP_OFFSET`, then the address generation will happen in the unextended way.

The control of the proposed LSU extension can be performed by the introduction and modification of a 2-bit dedicated control register CR_{n+1} to support 4 compression-decompression ways as described above. If the instruction set provides load-pointer and store-pointer instructions, LSU extensions can be activated by them and no extra instructions are necessary; for instance, PowerPC has such instructions to support *Technology Independent Machine Interface* [25]. However, if load-pointer and store-pointer instructions are not available, the extension can be activated by extra opcodes for memory-access operations:

```
ld32.cd Reg64, [ Base + Offset ]; // load-decompress
st32.cd [ Base + Offset ], Reg64; // store-compress
```

5 Experimental Platform and Methodology

As described in Section 2, architectures with tagged pointers support may have different tag sizes, cache hierarchies, and

other parameters. Moreover, these architectures are not supported by state-of-the-art research JVMs. To make our study more general, and to utilize research JVMs while evaluating the proposed HW-assisted CIP elimination technique, we opted for a simulation-based approach.

5.1 ZSim Simulator

ZSim is the simulator of choice since it provides high accuracy and fast simulation speed ($\approx$ 10 MIPS in our experiments). It is capable of running managed workloads and required minor modifications to run Maxine VM. Alternative options, such as the Sniper [26], [27] simulator that runs with JikesRVM, or the full-system gem5 [28] simulator, were considered but abandoned due to a number of limitations. Regarding the Jikes RVM running on top of the Sniper simulator [29], [30], it is only possible to run it in a 32-bit mode, while gem5 has a relatively low simulation speed.

ZSim, being a simulator, has a number of declared idealizations and limitations, among which are: not fully supported prefetchers, an ideal branch target buffer, not supported loop stream detector, micro-sequenced instructions and other components of the microarchitecture (described in [5]). With the exception of prefetchers, these limitations and idealizations do not directly affect the evaluation of the proposed CIP elimination technique. However as shown in [31] Fig. 3, the impact of the not modeled HW prefetchers on the performance of the DaCapo benchmarks, which are used in this work, is insignificant.

5.1.1 Extensions

ZSim is a user-level x86-64 simulator with an OOO-core model of the Westmere microarchitecture. Despite the fact that current x86-64 architectures do not support tagged pointers (Section 2), we implemented this extra functionality in ZSim (in a similar way as in [7]). On x86-64, the high 16 bits of an address are not used and, therefore, can be utilized by software that runs on top of ZSim. When memory accesses with tagged addresses are simulated by ZSim, they are untagged and executed in an ordinary way during Pin [32] translation. However, the tags are delegated to the core simulation model making possible to simulate tagged pointers and evaluate the proposed AGU extensions. Setting the CR_n is performed via a magic NOP instruction. The proposed AGU extensions are functionally modeled, as described in Section 4, assuming there is no extra latency overhead introduced by them.

The VM of choice (Maxine VM) does not support compressed object pointers, so we designed a technique to model them. The Maxine VM provides an easy way of changing the layout of objects, and we implemented a version where the memory footprint of the primitive data types is expanded two times, but the memory footprint of the object references remains the same. During simulation, memory accesses to the heap are mapped to a memory address space of half size where the objects would have been located if they were shrunk by two times. Special adjustments were also made to correctly simulate object copying and initialization with zeros, which in Maxine VM are executed in a number of loops. The addresses of these loops are reported to ZSim so that only every second iteration is simulated. This technique is a subcase of the more general address space morphing technique [7], with a simulation error of less than 1 percent in geometric mean on the DaCapo benchmarks.

TABLE 2
ZSim Configurations

Name	4C	4CA	4CAL	1C	1CA	1CAL	$1C_V$
Cores	x86-64 Nehalem OOO core at 2.66 GHz						
	4			**1**			
AGU ext.	-	+	+	-	+	+	-
LSU ext.	-	-	+	-	-	+	-
L1I caches	32 KB, 4-way, LRU, 3-cycle latency						
L1D caches	32 KB, 8-way, LRU, 4-cycle latency						
L2 caches	256 KB, 8-way, LRU, 6-cycle latency						
L3 cache	16-way, hashed, 30-cycle latency						
	8 MB			2 MB			8 MB
Memory controller	1, 3 DDR3 channels, 47-cycle latency						
DRAM	3 GB, DDR3-1066, 1 GB DIMM per channel						

With this technique, no other changes in the simulator were made to simulate LSU extensions, as we assume that: (1) load-decompress and store-compress operations do not introduce extra overhead in comparison to ordinary load and store operations; (2) introducing two extra opcodes does not significantly change the code footprint; (3) Compression-Decompression Selector changes infrequently.

The power and energy estimation model using McPAT was integrated from the Sniper simulator [33] for the same microarchitecture simulated by ZSim in order to perform energy estimations (in a similar way as in [7]). The proposed extensions to AGU and LSU were not added in the power estimation model as the energy overhead of these functional units extensions is significantly less than the energy savings from the reduction of memory traffic to DRAM and *Last-Level Cache* (LLC).

5.1.2 Configurations

Table 2 details the seven hardware configurations used for the evaluation. The latencies associated with the levels of memory hierarchy in the table do not include latencies associated with the lower levels, so for instance L2 data hit will be $(4 + 6 =)$ 10 cycles. The memory controller latency is in core clocks.

Configuration `4C` models a 4-core Intel Nehalem CPU. Configuration `1C` models a 1-core Intel Nehalem CPU with a quarter of the available LLC. Configuration $\texttt{1C}_V$ models a 1-core Intel Nehalem CPU with one online core. Configurations `4CA` and `1CA` represent the proposed extensions to the AGU of configurations `4C` and `1C` respectively. Configurations `4CAL` and `1CAL` represent the proposed extensions to both the AGU and the LSU of configurations `4C` and `1C` respectively. Configurations `4C`, `4CA`, `4CAL`, `1C`, `1CA`, and `1CAL` were used in the evaluation part, while configurations `4C` and $\texttt{1C}_V$ were used for validation against the real platform. Configurations `1C`, `1CA` and `1CAL` were selected to simulate the scenario when only a quarter of the available resources is available to the workload (if LLC could be partitioned, adding pressure to the caches). Finally, all configurations support 16-bit pointer tags.

5.2 Maxine VM

Maxine VM was selected as the JVM for our experiments. In contrast to Jikes RVM, Maxine VM is a 64-bit VM supporting x86-64, is compatible with OpenJDK, and is capable of running the full set of the DaCapo-9.12-bach benchmarks. The benefits of using Maxine VM are its modularity and its powerful co-designed integrated debugging support (Maxine Inspector) leading to research productivity. The modules (called schemes) are accessed by scheme interfaces making possible to switch easily between implementations for heap operations (GC and memory allocation), object layouts, monitors, references, etc. As these modules are not optimized across the boundaries in contrast to production VMs, research productivity comes at the expense of performance. The observed performance difference is less than 2x on the DaCapo-9.12-bach benchmarks against the state-of-the-art OpenJDK [12].[4]

The main direct effect of the proposed CIP elimination technique is the reduction of the memory footprint of the objects. In all VMs supporting the same object layout, the memory space savings related to the reduced memory footprint of objects are expected to be the same as in Maxine VM.

In our experiments, we used a non-generational semi-space heap scheme of a constant 2 GB size with a stop-the-world copying GC, described in Section 3. This is the default and stable GC scheme in the Maxine VM. In the context of the single-core configurations that we use, this scheme can be optimal for throughput. Finally, modern generational GC algorithms employ a copying scheme based on Cheney's breadth-first copying GC scheme for frequent young generation collections, and the proposed changes can be applied to a wider spectrum of copying collectors.

As we use a constant 2 GB heap size, Compression-Decompression Selector was in state (s2), when compressed object pointers optimization was enabled. In this state, a loaded 32-bit compressed pointer value is split into 28 and 4 bits, the former shifted by 3 bits and the latter placed in the high-order bits of the 64-bit register as CID.

5.2.1 Extensions

An extensible communication interface, between Maxine VM and ZSim, was based on magic NOP operations (holding extra semantics for ZSim and Maxine VM but executed as NOPs on x86-64) and Protocol Buffers [35] for serializable data (such as profiling data, configuration data, etc.). A tagging scheme was added which assigns CIDs to tags of object pointers in the VM (in a similar way as in [7]). CIP elimination from object headers in the heap was implemented by modifying object memory allocation and GC as discussed in Section 3.

Originally, during GC, a forwarding pointer was stored in the CIP word of an object in Maxine VM. We modified Maxine VM to store the forwarding pointers in the MISC words instead of the eliminated CIP words. Furthermore, we reserve bit 47 of the MISC word for forwarding pointer indication (when it is set to one), constraining the range of heap addresses to have bit 47 set to zero. The decision to reserve bit 47 of the MISC word has been taken after considering the least collateral effect to the VM functionality. In this case, the result is a 2× reduction of the possible concurrent threads from 2^{16} to 2^{15} used in thin locking [36].

Originally, in Maxine VM, a null pointer check is performed by loading the value of the CIP (object pointer plus offset zero). If the pointer is null, an exception will be raised, otherwise the CIP would be loaded. If the CIP is stored alongside the object, a null check can have a positive prefetching effect. In a case of CIP elimination from object headers, CIPs would be loaded from `CIPArray`. We decided to change the offset of the null pointer check to eight, which points to the MISC word, which is the first word in the object when CIP is eliminated. Thus, positive prefetching effects of null checks are preserved.

TABLE 3
Maxine VM Configurations

Name	Description
`B`	**B**aseline Maxine VM.
`E`	`B` with CIP **E**limination for ZSim configurations without extra HW support.
`EA`	`B` with CIP **E**limination using 16-bit CIDs in tagged pointers for ZSim configurations with **A**GU extensions.
`C`	`B` with **C**ompressed object pointers.
`EAL`	`C` with CIP **E**limination using static profile and 4-bit CIDs in tagged pointers for ZSim configurations with **A**GU and **L**SU extensions.

5.2.2 Configurations

We experimented with five Maxine VM configurations described in Table 3. Later in the paper, we will be using pairs of ZSim and Maxine VM configurations in order to evaluate the combinations of different configurations (e.g., `4C-B`).

5.3 Benchmarks

We used two widely acknowledged and two emerging benchmarks in our work:

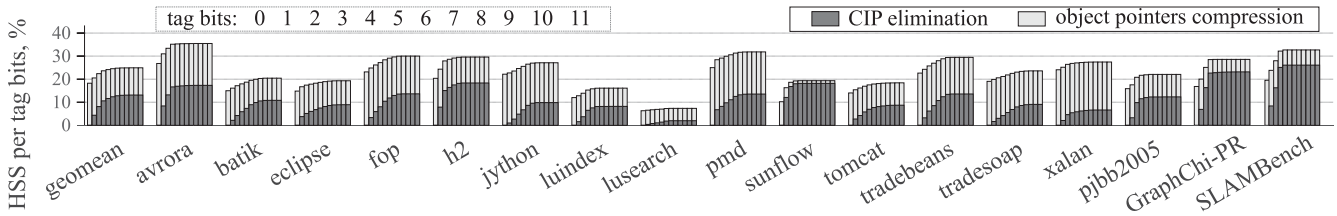
- *DaCapo-9.12-bach* [8] suite covers a representative number of server and desktop applications.
- *pjbb2005* [9] is a version of the SPECjbb2005 [37] benchmark with a fixed workload.
- *GraphChi-PR* [11] is a PageRank algorithm [38] running on top of the disk-based system for graph analytics GraphChi.
- *SLAMBench* [10] is a Java version of computer vision benchmark for simultaneous localization and mapping which implements the KinectFusion algorithm [39].

5.4 Validation

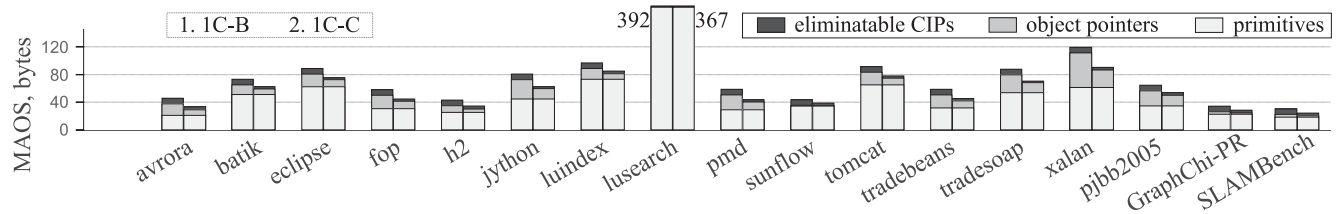
With respect to ZSim, the execution times were validated against the real system with switched-off prefetchers,[5] and the following expected inconsistencies were found. First, the execution times of `eclipse` and `tradesoap` on the $1C_V$-`B` configuration were more that two times greater than on the

4. Against both the C2 and Graal [34] compilers.

5. We disabled the prefetchers from the real hardware in order to achieve a fair comparison since ZSim does not fully support hardware prefetchers.



(a) Estimation of Heap Space Savings (HSS) per available tag bits for CIDs storage



(b) Mean Allocated Object Size (MAOS)

Fig. 6. Estimation of heap space savings and mean allocated object size for different configurations.

real system. Second, the execution times of `avrora` on the `1C`$_V$`-B` and `4C-B` configurations were more than three times less than on the real system. These inconsistencies are associated with the following declared ZSim limitations: (1) different thread scheduling algorithms used in the real system and in ZSim, and (2) the inability of ZSim to simulate non-user-level code. The geometric mean difference in execution time between the real and simulated platforms for `1C`$_V$`-B` and `4C-B` configurations was measured to be 10 percent.

5.5 Experimental Methodology

In our experiments, we used a constant heap size of 2 GB, where only 1 GB is used in the semi-space scheme. Although VMs can support variable heap sizes, the variable heap size will only directly affect the amount of GC and heap resize work. We did separate evaluations of execution times of GC, as one of the positive effects of the proposed technique is the reduction in the amount of GC work to be performed. Thus, the presented data allow making analytical estimations of the proposed technique for other heap sizes.

Each experiment has been run 10 times, and whiskers represent 95 percent confidence intervals. The variance for some of the tests can be up to 5 percent due to the dynamic nature of the VM or related to nondeterminism aspects in GC, JIT compilation, threads scheduling, and other factors. ZSim is a DBT-based execution-driven simulator, so the simulation is not deterministic. Assuming normal distribution, when the whisker crosses zero in a chart showing relative changes, we are less than 95 percent confident whether a result is positive or negative.

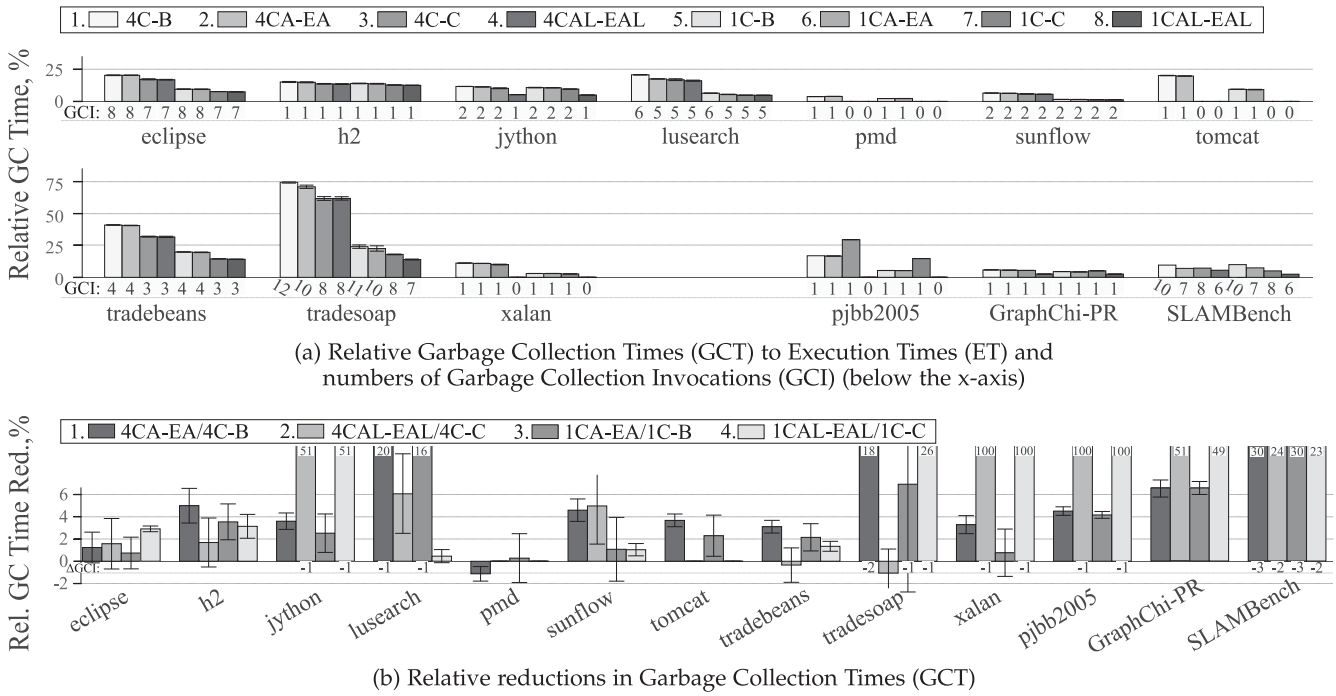
6 Experimental Results

6.1 Heap Space Savings

The main benefit of CIP elimination from object headers is heap space savings. Since the number of tag bits can vary on different architectures, and some of the bits can be utilized for other purposes, it is important to explore how much heap space savings can be achieved per available tag bits for CIDs storage. *Heap Space Savings* (HSS) per available tag bits are estimated on the two baseline configurations, `1C-B` and `1C-C`, by collecting profiling information on the number of all *Allocated Objects* (AO) for each class $AO(c)$, where c is class, and on the total allocated *Heap Space Volume* (HSV). We sort $AO(c)$ in decreasing order and we get the *Sorted Allocated Objects* $SAO(i)$ sequence. Finally, we estimate how much $HSS_{\texttt{C}}(n)$ can be gained for configuration `C` using n tag bits available for CIDs storage by using the following formula:

$$HSS_{\texttt{C}}(n) = \frac{HSV_{\texttt{C}} - \texttt{sizeof(CIP\_t)} * \sum_{i=0}^{2^n-2} SAO_{\texttt{C}}(i)}{HSV_{\texttt{1C-B}}} * 100\%.$$

In this estimation, we assume that the $AO(c)$ distribution can be perfectly predicted dynamically or it is known statically from previous runs. Estimated heap space savings per tag bits for `1C-B` (bottom dark gray bar segments) and `1C-C` (full stacked bars) configurations are presented in Fig. 6a. As depicted in the figure, the number of bits for CIP elimination required to reach heap space savings close to maximum can vary for different workloads, and 8 bits are sufficient to gain 99 percent of possible heap space savings on selected benchmarks, while 11 bits are enough to gain 100 percent. These savings can lead to a proportional reduction in GC times and cache misses due to a reduced memory footprint. Heap space savings inversely correlate with *mean allocated object size* as can be seen in Fig. 6b. Mean allocated object size is shown for two configurations: `1C-B` (left full stacked bar) and `1C-C` (right full stacked bar). Thus, the bottom segments represent mean sizes of the primitive parts of the objects, the two upper segments represent mean sizes of the pointers in the objects, and the uppermost segments represent mean sizes of the CIPs that can be eliminated from the objects with the technique proposed in the paper. On configurations with CIP elimination and proposed HW extensions, `1CA-EA` and `1CAL-EAL`, we were able to achieve up to 26 percent (`SLAMBench`) and 10 percent geometric mean heap space savings. It can be also observed in Fig. 6a that the effect of object pointers compression on heap space savings (the first bar for each test) is more significant than the effect of CIP elimination (the last dark gray bottom segment for each test) for all tests with the exception of `sunflow`, `GraphChi-PR`, and `SLAMBench`.



(a) Relative Garbage Collection Times (GCT) to Execution Times (ET) and numbers of Garbage Collection Invocations (GCI) (below the x-axis)

(b) Relative reductions in Garbage Collection Times (GCT)

Fig. 7. Changes in GC times and numbers of GC invocations for various configurations.

6.2 Effects of CIP Elimination on GC

First, we examine how much time is spent in GC invocations relative to the execution time for configurations with and without CIP elimination. Relative GC times to execution times are presented in Fig. 7a. Since GC never happens on `avrora`, `batik`, `fop`, and `luindex` on the tested configurations, the results for these tests are not shown in the figure. When no space is left in the "from-space", a *Garbage Collection Invocation* (GCI) occurs that copies live objects to the "to-space". The number of GCIs during execution of each benchmark for each configuration is shown below each bar. Configurations with (`*-E*`) and without (`*-B`, `*-C`) CIP elimination are depicted adjacently. The evaluation is done pairwise `4CA-EA` against `4C-B`, `4CAL-EAL` against `4C-C`, etc. We will use the following notation hereafter: `4CA-EA/4C-B` means `4CA-EA` compared against `4C-B`. It can be seen that CIP elimination leads to reduction in the number of GC invocations and relative GC times to execution times for `jython` (`*CAL-EAL/*C-C`), `lusearch` (`*CA-EA/*C-B`), `tradesoap` (`*CA-EA/*C-B`, `1CAL-EAL/1C-C`), `xalan` (`*CAL-EAL/*C-C`), `pjbb2005` (`*CAL-EAL/*C-C`), and `SLAMBench` (`*CA-EA/*C-B`, `*CAL-EAL/*C-C`). For all subsequent figures the deltas in garbage collection invocations for compared configurations (`△GCI`) not equal to zeros will be shown below the bars.

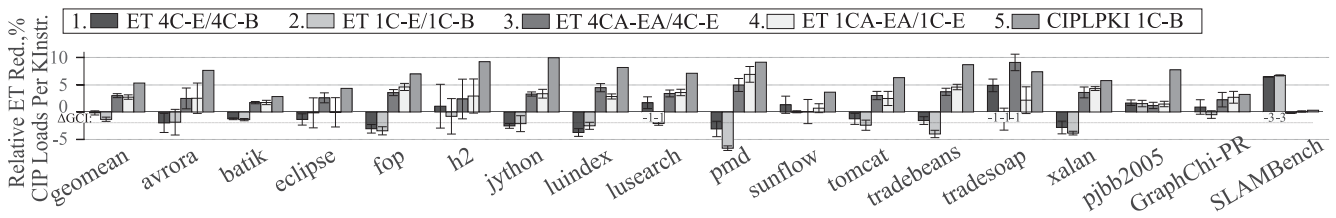
Second, we investigate how GC time is affected as a result of CIP elimination. The relative reductions in GC times are presented in Fig. 7b. When the number of garbage collection invocations is reduced as a result of CIPs elimination, the GC times reductions are above 16 percent (`lusearch 1CA-EA/1C-B`). When `△GCI` is zero, the GC times are reduced for the majority of tests with no increases above 3 percent (`tradesoap 4CAL-EAL/4C-C`). In two cases (`GraphChi-PR *CAL-EAL/*C-C`), the reduction in GC time is approximately 50 percent, which is explained by the high dynamism in the number of live objects, as was shown by Nguyen et al. in Fig. 2 [40].

The presented data provide evidence that the maintenance and traversal of the list of CIDs during copying GC (described in Section 3) do not introduce significant overhead and do not overweigh the performance gains. Provided that the set of live objects is the same during copying GC, the gains are due to less memory footprint of the copied objects.

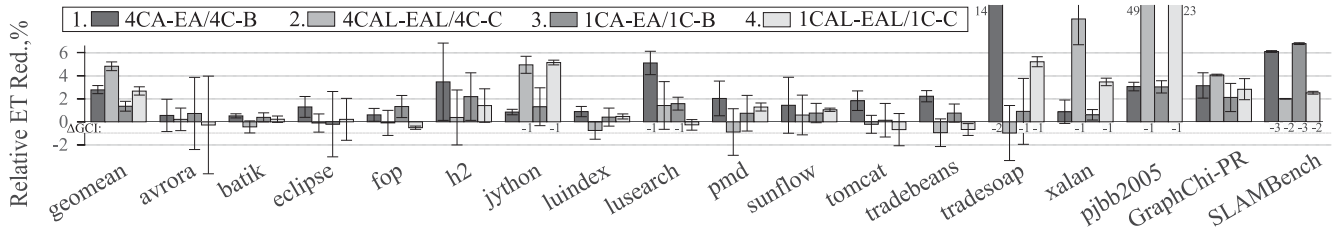
6.3 Effect of CIP Elimination on Execution Time for Configurations without HW Extensions

In this experiment, we investigate the effect of the SW-only CIP elimination, without the proposed HW extensions, on execution time. We evaluate the following configurations: `4C-E/4C-B`, and `1C-E/1C-B`. We observe 0 and 1 percent geometric mean execution time degradations for the aforementioned configurations respectively, which are shown in Fig. 8a (the first two series). Execution time reductions when `△GCI` is zero are observed for `pjbb2005` and `SLAMBench`. On `pjbb2005` reductions are 1.7 percent for `4C-E/4C-B` and 1.6 percent for `1C-E/1C-B`, and on `SLAMBench` reductions are 6.5 percent for `4C-E/4C-B` and 6.8 percent for `1C-E/1C-B`. Our observations correspond with the fact, that `pjbb2005` was classified as "cache-miss-intensive", while `DaCapo-9.12` as "non-cache-miss-intensive" by Inoue and Nakatani [41]. The performance degradation for `DaCapo-9.12` is observed because the performance overhead during the SW-only CIP retrieval from tagged pointers is not covered by the gains due to cache miss reductions.

We validate this hypothesis by measuring the percentage of *CIP Loads Per Kilo-Instruction* (CIPLPKI) on the `1C-B` configuration, which is shown as the fifth (the last) series in Fig. 8a with a geometric mean value of 5.4 CIPLPKI. As discussed in Section 3, SW-only CIP retrieval from tagged pointers has the dynamic execution height of 5.5 instructions in our implementation. Thus, the estimated overhead of CIP



(a) Relative reductions in Execution Times (ET) for various configurations and CIP Loads Per Kilo-Instruction (CIPLPKI) for 1C-B



(b) Relative reductions in Execution Times (ET) for various configurations

Fig. 8. Relative changes in Execution Times (ET) for various configurations.

access on the `1C-E` configuration will be 24.3 extra instructions per kilo-instruction. We test our estimation by evaluating configurations with enabled CIP elimination with the proposed AGU extensions against the SW-only CIP elimination: `4CA-EA/4C-E`, and `1CA-EA/1C-E`, which are the third and the fourth series in Fig. 8a. The relative geometric mean execution time reductions for these configurations are 3.1 and 2.8 percent respectively, which are consistent with our estimation and motivate the usage of the AGU extension.

The data for the SW-only CIP elimination for configurations with compressed pointers is not presented in the paper. The reason behind that is the high estimated performance overhead, since loads of compressed object pointers happen approximately 6 times more frequently than CIP loads in geometric mean for `1C-C` configuration, with the geometric mean rate of 32.4 loads of compressed object pointers per kilo-instruction. A few modern architectures, like Intel Haswell and AMD Excavator, provide advanced instructions for bit manipulation, such as PDEP and PEXT [20], that can be used for tagged pointers compression and decompression in one instruction. However, these instructions have a high latency of 3 cycles [42]. We estimated that the overhead of compressed pointers decompression using these instructions is 32.4 extra instructions and 97.2 extra execution cycles per kilo-instruction. This significant overhead motivates the usage of the LSU extension.

6.4 Effect of CIP Elimination on Execution Time for Configurations with HW Extensions

In this experiment, we investigate the effect of CIP elimination for configurations with the proposed AGU and LSU extensions. We compare the following configurations: `4CA-EA/4C-B`, `4CAL-EAL/4C-C`, `1CA-EA/1C-B`, and `1CAL-EAL/1C-C`. The relative reductions in execution time for these configurations are shown in Fig. 8b, with geometric mean values of 2.9, 5.0, 1.4, and 2.8 percent respectively. The maximum reductions are 13.6, 49.1, 6.9, and 22.6 percent respectively, with no degradations on single tests below 4 percent.

6.5 Reduction in Cache Misses

When the CIPs are eliminated from the object headers, they are densely located in `CIPArray`, and the memory footprint of the allocated objects in the heap is smaller. Both factors lead to a decrease in cache misses. We test our hypothesis by comparing the following configurations: `4CA-EA/4C-B`, `4CAL-EAL/4C-C`, `1CA-EA/1C-B`, and `1CAL-EAL/1C-C`. We observed significant relative reductions in *L3 Cache Misses Per Kilo-Instruction* (L3CMPKI) which are shown in Fig. 9b, with geometric mean values of 12.7, 10.0, 12.4, and 9.2 percent respectively. These values are correlated with the heap space savings estimations presented in Fig. 6a, which are 13.3 percent for `1C-B`(11 bits)/`1C-B`(0 bits) and 7.2 percent for `1C-C`(4 bits)/`1C-C`(0 bits).

Relative reductions in *L2 Cache Misses Per Kilo-Instruction* (L2CMPKI) are more moderate compared to L3CMPKI. As shown in Fig. 9a, the geometric mean values are 8.2, 4.4, 8.5, and 5.8 percent for respective configurations. The relative increase in L2CMPKI in `pjbb2005` for `4CAL-EAL/4C-C` and `1CAL-EAL/1C-C` configurations is related to the complete elimination of GC invocations on configurations with CIP elimination, `4CAL-EAL` and `1CAL-EAL`. Furthermore, GC can have different cache miss characteristics from the workload and can have a positive effect on the locality of copied objects.

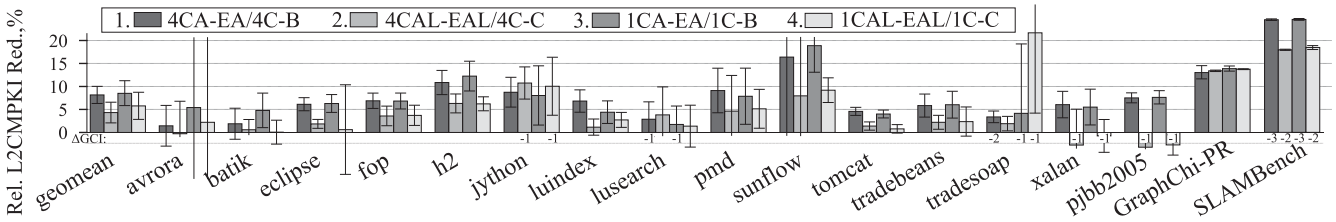
6.6 Reduction in Dynamic Energy

Reduction in cache misses leads to less DRAM and L3 cache traffic which is consequently translated to *Dynamic Energy* (DE) reductions in these components. We test our hypothesis on the same configurations as before. The relative reductions in *DRAM Dynamic Energy* (DRAMDE) are shown in Fig. 10b. The geometric mean values of 12.9, 11.1, 12.3, and 10.6 percent are correlated with L3CMPKI reductions for the respective configurations. Maximum relative DRAMDE reductions reach up to 27.6, 50.1, 27.0, 41.5 percent respectively.

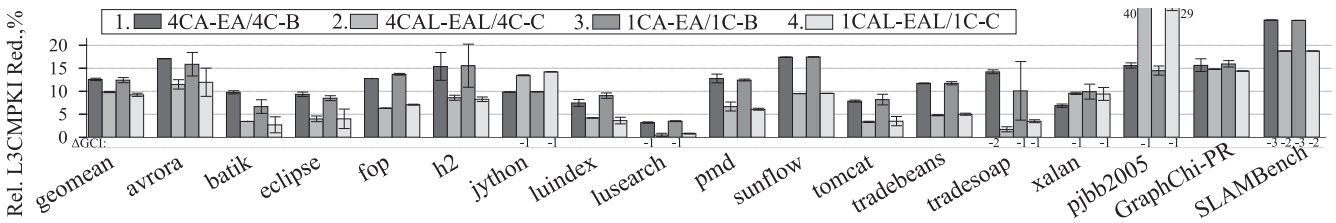
Finally, relative reductions in *L3 Dynamic Energy* (L3DE) are shown in Fig. 10a, with geometric mean values of 8.6, 5.3, 9.3, and 7.6 percent respectively.

7 Related Work

From a historical perspective, Steele [43] proposed using contiguous memory regions for LISP, so that only variables of a certain data type can be in a given region. Using

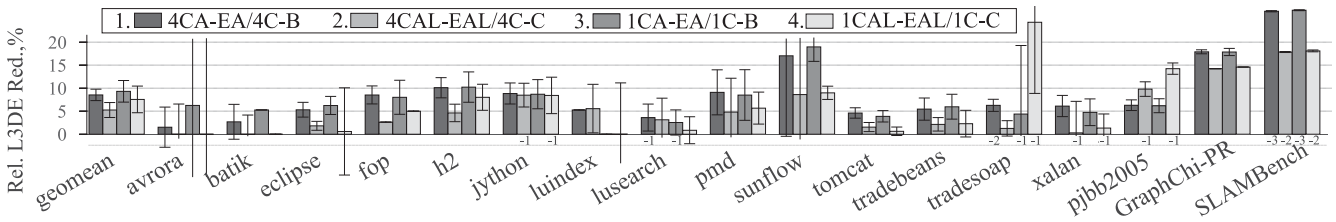


(a) Relative reductions in L2 Cache Misses Per Kilo-Instruction (L2CMPKI) for various configurations

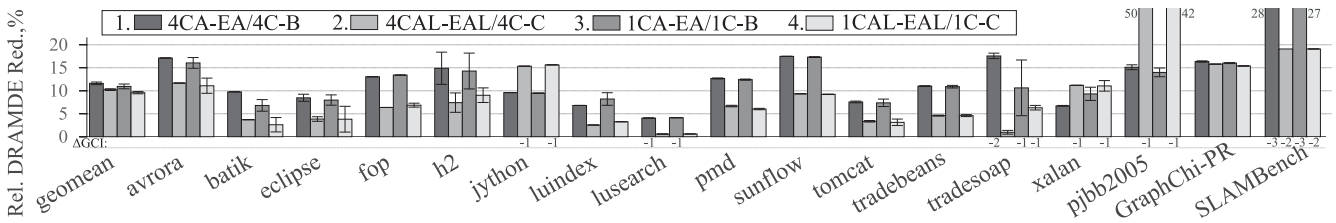


(b) Relative reductions in L3 Cache Misses Per Kilo-Instruction (L3CMPKI) for various configurations

Fig. 9. Relative reductions in cache misses per kilo-instruction for various configurations.



(a) Relative reductions in L3 Dynamic Energy (L3DE) for various configurations



(b) Relative reductions in DRAM Dynamic Energy (DRAMDE) for various configurations

Fig. 10. Relative reductions in dynamic energy for various configurations.

segmentation and enforcing data structures alignment, Dybvig et al. [44] investigated a hybrid technique utilizing also the least significant bits of the address for implicit typing for the Scheme language. Continuing taking advantage of the contiguous virtual addresses for implicit typing, Bacon et al. [51] used this technique for compression of object headers in the context of 32-bit Jikes RVM. They gained most of the space savings by eliminating the thin lock word. The implicit typing scheme via virtual addressing was later explored in the context of the 64-bit Jikes RVM on PowerPC by Venstermans et al. [45] to remove the object header completely.

In contrast to these related systems, our approach using object typing via tagged pointers does not bind objects to contiguous memory regions. Thus, our proposed technique can facilitate additional optimizations such as data transformations and object fusing [46] and objects alignment and collocation as proposed by Inoue and Nakatani [41].

Using tags for storing type information has been used in many computer systems [47], [48], [49]. However, the main use case of tagged pointers is capability-based security which also requires tagged memory. A generalized hardware support for tag processing has been recently proposed by Dhawan et al. [50], which can be extended for performance purposes by the HW extensions introduced in this work.

8 Conclusions

We have demonstrated a novel and open-source experimental platform for HW/SW co-design research, based on the Maxine VM, ZSim, and McPAT, which laid the foundation for the MaxSim [7] platform. By using this platform, we have thoroughly evaluated a proposed technique of CIP elimination by encoding CIDs in tagged pointers. Although this technique significantly reduces memory footprint, we have shown that retrieving CIPs from object pointer tags without extra HW support can degrade performance. To address performance issues, we have proposed novel hardware extensions to the AGU removing the performance degradations associated with CIP retrieval from tagged pointers and to the LSU for efficient load-decompression and store-compression of tagged object pointers.

In addition, the experimental results for the proposed HW/SW co-designed technique achieve significant heap space savings, dynamic energy reductions, and performance improvements without significant regressions. Although the proposed technique has been researched in the context of a research JVM implementation, it can be widely applicable to other object-oriented languages and managed runtimes.

Acknowledgments

This work is partially supported by EPSRC grants PAMELA EP/K008730/1, AnyScale EP/L000725/1, and EU Horizon 2020 ACTiCLOUD 732366 grant. A. Rodchenko is funded by a Microsoft Research PhD Scholarship, A. Pop is funded by a Royal Academy of Engineering Research Fellowship, and M. Luján is funded by a Royal Society University Research Fellowship.

References

[1] C. Wimmer, M. Haupt, M. L. van de Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, Java," *ACM Trans. Archit. Code Optimization*, vol. 9, no. 4, pp. 30:1–20:24, Jan. 2013.

[2] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján, "Heterogeneous managed runtime systems: A computer vision case study," in *Proc. 13th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2017, pp. 74–82.

[3] B. Alpern, et al., "Implementing Jalapeño in Java," in *Proc. 14th ACM SIGPLAN Conf. Object-Oriented Program., Syst. Languages Appl.*, 1999, pp. 314–324.

[4] J. Larus and G. Hunt, "The singularity system," *Commun. ACM*, vol. 53, no. 8, pp. 72–79, Aug. 2010.

[5] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 475–486.

[6] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optimization*, vol. 10, no. 1, pp. 5:1–5:29, Apr. 2013.

[7] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Luján, "MaxSim: A simulation platform for managed applications," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, 2017, pp. 141–151.

[8] S. M. Blackburn, et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Languages Appl.*, 2006, pp. 169–190.

[9] pjbb2005, 2005. [Online]. Available: http://users.cecs.anu.edu.au/steveb/research/research-infrastructure/pjbb2005 Last Accessed on: May 12, 2017.

[10] L. Nardi, et al., "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2015, pp. 5783–5790.

[11] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[12] OpenJDK, 2017. [Online]. Available: http://openjdk.java.net, Last Accessed on: May 12, 2017.

[13] C. Click, "Cliff Click's blog: Biased locking," 2010. [Online]. Available: http://www.cliffc.org/blog/2010/01/09/biased-locking/ Last Accessed on: May 12, 2017.

[14] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 1999, pp. 13–24.

[15] ARM Cortex-A series programmer's guide for ARMv8-A, 2015. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf, Last Accessed on: May 12, 2017.

[16] D. Brash, "ARMv8-A architecture–2016 additions," 2016. [Online]. Available: https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions, Last Accessed on: May 12, 2017.

[17] Pointer authentication on ARMv8.3, 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, Last Accessed on: May 12, 2017.

[18] S. Phillips, "M7: Next generation SPARC," 2014. [Online]. Available: http://www.oracle.com/us/products/servers-storage/servers/sparc-enterpr ise/migration/m7-next-gen-sparc-presentation-2326292.html, Last Accessed on: May 12, 2017.

[19] K. Aingaran, et al., "M7: Oracle's next-generation Sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar. 2015.

[20] Intel 64 and IA-32 architectures software developers manual. volume 1: Basic architecture, 2011. [Online]. Available: http://download.intel.com/design/processor/manuals/253665.pdf, Last Accessed on: May 12, 2017.

[21] 5-level paging and 5-level EPT white paper, 2016. [Online]. Available: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, Last Accessed on: May 12, 2017.

[22] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, no. 11, pp. 677–678, Nov. 1970.

[23] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar, "A 4-GHZ 130-nm address generation unit with 32-bit sparse-tree adder core," *IEEE J. Solid-State Circuits*, vol. 38, no. 5, pp. 689–695, May 2003.

[24] A. R. Adl-Tabatabai, et al., "Improving 64-bit Java IPF performance by compressing heap references," in *Proc. Int. Symp. Code Generation Optimization*, 2004, pp. 100–110.

[25] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. San Mateo, CA, USA: Morgan Kaufmann, 2005.

[26] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2011, pp. 52:1–52:12.

[27] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optimization*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[28] N. Binkert, et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[29] Jikes–Sniper page in Sniper online documentation, 2014. [Online]. Available: http://snipersim.org/w/Jikes, Last Accessed on: May 12, 2017.

[30] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, "Cooperative cache scrubbing," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation*, 2014, pp. 15–26.

[31] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 308–319.

[32] C.-K. Luk, et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2005, pp. 190–200.

[33] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout, "Power-aware multi-core simulation for early design stage hardware/software co-optimization," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 3–12.

[34] OpenJDK: Graal project, 2016. [Online]. Available: http://openjdk.java.net/projects/graal/, Last Accessed on: May 12, 2017.

[35] Protocol Buffers - Google's data interchange format (ver. 2.6.1), 2014. [Online]. Available: https://developers.google.com/protocol-buffers/, Last Accessed on: May 12, 2017.

[36] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin locks: Featherweight synchronization for Java," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 1998, pp. 258–268.

[37] SPECjbb2005, 2005. [Online]. Available: http://www.spec.org/jbb2005/, Last Accessed on: May 12, 2017.

[38] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the Web," in *Proc. 7th Int. World Wide Web Conf.*, 1998, pp. 161–172.

[39] R. A. Newcombe, et al., "KinectFusion: Real-time dense surface mapping and tracking," in *Proc. 10th IEEE Int. Symp. Mixed Augmented Reality*, 2011, pp. 127–136.

[40] K. Nguyen, et al., "Yak: A high-performance big-data-friendly garbage collector," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 349–365.

[41] H. Inoue and T. Nakatani, "Identifying the sources of cache misses in Java programs without relying on hardware counters," in *Proc. Int. Symp. Memory Manage.*, 2012, pp. 133–142.

[42] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," 2016. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf, Last Accessed on: May 12, 2017.
[43] G. Steele, *Data Representation in PDP-10 MacLISP*. Cambridge, MA, USA: Massachusetts Inst. Technol., Artif. Intell. Laboratory, 1977.
[44] R. K. Dybvig, D. Eby, and C. Bruggeman, "Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages," Indiana Univ., Bloomington, IN, USA, Tech. Rep. #400, 1994.
[45] K. Venstermans, L. Eeckhout, and K. De Bosschere, "Java object header elimination for reduced memory consumption in 64-bit virtual machines," *ACM Trans. Archit. Code Optimization*, vol. 4, no. 3, pp. 17:1–17:30, Sep. 2007.
[46] C. Wimmer and H. Mössenbösck, "Automatic feedback-directed object fusing," *ACM Trans. Archit. Code Optimization*, vol. 7, no. 2, pp. 7:1–7:35, Oct. 2010.
[47] E. I. Organick, *Computer System Organization: The B5700/B6700 Series (ACM Monograph Series)*. New York, NY, USA: Academic, 1973.
[48] B. Babayan, "E2K technology and implementation," in *Proc. 6th Int. Euro-Par Conf. Parallel Process.*, 2000, pp. 18–21.
[49] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM system/38 support for capability-based addressing," in *Proc. 8th Annu. Int. Symp. Comput. Archit.*, 1981, pp. 341–348.
[50] U. Dhawan, et al., "Architectural support for software-defined metadata processing," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 487–502.
[51] D. F. Bacon, S. J. Fink, and D. Grove, "Space- and time-efficient implementation of the Java object model," in *Proc. 16th European Conf. Object-Oriented Program. (ECOOP)*, 2002, pp. 111–132.

Andrey Rodchenko received the BSc and MSc degrees in applied mathematics and physics from the Moscow Institute of Physics and Technology, in 2005 and 2007, respectively. He is currently working toward the PhD degree in the Advanced Processor Technologies Group, School of Computer Science, University of Manchester. From 2007 until 2013, he was a software engineer at Optimizing Technologies (start-up company) and a senior software engineer at Intel. His research interests include optimizing compilation, dynamic binary translation, runtime systems, computer architecture, and hardware/software co-design.

Christos Kotselidis is a lecturer in the School of Computer Science, University of Manchester working on hardware/software co-designed virtual machines. He has worked as a principal member of technical staff at Oracle Labs and as a senior research scientist at Intel Labs. During his industry experience, he has worked across the entire stack of computing including chip design, microarchitecture research, hardware/software co-designed CPUs, compilers, virtual machines, and garbage collection. Finally, he authors more than 20 refereed papers and eight patents.

Andy Nisbet received the BSc degree in physics with electronic engineering, in 1988 and the PhD degree in electrical & electronic engineering from Manchester University, in 1993. He was a postdoctoral researcher with the University of Manchester prior to lecturing appointments in computer science at Trinity College, Dublin, and the Manchester Metropolitan University, United Kingdom. He is currently a research fellow working on low-power virtualization for many-core in the Advanced Processor Technologies Group, School of Computer Science, University of Manchester. His research interests embrace managed runtimes, compilation, and novel computational accelerators.

Antoniu Pop received the MSc degree from Ecole Nationale Superieure d'Informatique et Mathematiques Appliquees de Grenoble, in 2004 and the PhD degree from MINES ParisTech, in 2011. He is a lecturer and Royal Academy of Engineering Research Fellow in the School of Computer Science, University of Manchester. His research interests include the challenges of the many-core revolution, in particular focusing on high productivity parallel programming models, performance analysis, and dynamic optimization. He conducts research on the OpenStream programming language (http://openstream.info/) and on performance analysis in Aftermath (http://www.aftermath-tracing.com/).

Mikel Luján received the PhD degree in computer science from the University of Manchester. He is a Royal Society University Research Fellow in the School of Computer Science, University of Manchester. His research interests include managed runtime environments and virtualization, manycore architectures, and application-specific systems and optimizations.