Analytics with Smart Arrays: Adaptive and Efficient Language-Independent Data

Iraklis Psaroudakis
Oracle Labs
iraklis.psaroudakis@oracle.com

Stefan Kaestle
Oracle Labs
stefan.kaestle@oracle.com

Matthias Grimmer*
contact@matthiasgrimmer.com

Daniel Goodman
Oracle Labs
daniel.goodman@oracle.com

Jean-Pierre Lozi
Oracle Labs
jean-pierre.lozi@oracle.com

Tim Harris*
tim.harris@gmail.com

ABSTRACT

This paper introduces *smart arrays*, an abstraction for providing adaptive and efficient language-independent data storage. Their *smart functionalities* include NUMA-aware data placement across sockets and bit compression. We show how our single C++ implementation can be used efficiently from both native C++ and compiled Java code. We experimentally evaluate smart arrays on a diverse set of C++ and Java analytics workloads. Further, we show how their smart functionalities affect performance and lead to differences in hardware resource demands on multicore machines, motivating the need for adaptivity. We observe that smart arrays can significantly decrease the memory space requirements of analytics workloads, and improve their performance by up to 4×. Smart arrays are the first step towards general *smart collections* with various smart functionalities that enable the consumption of hardware resources to be traded-off against one another.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Shared memory algorithms*; • **Software and its engineering → General programming languages**; *Virtual machines*; • **Computing methodologies** → *Parallel computing methodologies*;

KEYWORDS

Data structures, language interoperability, resource trade-offs, adaptivity, multicore, NUMA, compression, graph analytics

ACM Reference Format:
Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. 2018. Analytics with Smart Arrays: Adaptive and Efficient Language-Independent Data. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190514

*Work conducted while employed at Oracle Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '18, April 23–26, 2018, Porto, Portugal
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190514

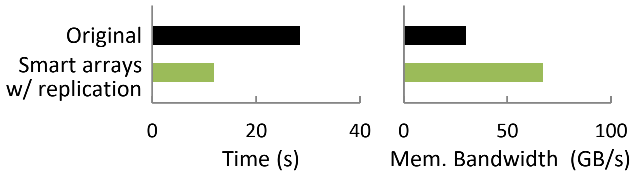


Figure 1: Smart arrays with replication improve PGX's PageRank performance & memory bandwidth utilization by more than 2× on a 2-socket machine with 8-core Xeon CPUs.

1 INTRODUCTION

Early implementations of big-data analytics frameworks had a reputation for being slow. Recurring issues included costly transfers of data between disk and main memory, inefficient data representations during processing, and excessive garbage collection activity in managed languages. Many of these issues have been addressed, e.g., keeping data in memory [41, 44, 65], and using compact data representations with storage outside the garbage collected heap [40, 64]. A consequence of this progress is that analytics workloads are increasingly limited by simple bottlenecks within the machine, e.g., saturating the rate at which data can be transferred from memory into a CPU, saturating the interconnect between CPUs, or saturating a core's functional units [4, 10, 25, 45].

The challenge in achieving further performance improvements is to identify the bottleneck resources and use them more productively. In this paper, we introduce *smart arrays* as a way to tackle this challenge. Different implementations, or *smart functionalities*, of the same smart array interface provide different trade-offs between the use of different resources. Selecting between these functionalities, either manually or automatically, allows a given workload to be mapped to hardware with different resource characteristics.

We showcase the real-world effect of a smart functionality in Figure 1, where we evaluate a popular graph analytics algorithm, PageRank, with PGX on a NUMA machine.[1] The performance of the original implementation is limited by the interconnect's bandwidth. The smart functionality of replicating smart arrays across the sockets of the machine can fully exploit the memory bandwidth of the sockets and localize accesses to memory to reduce pressure

[1]We introduce PGX and NUMA in §2, and describe the full experiments and machine details in §5.

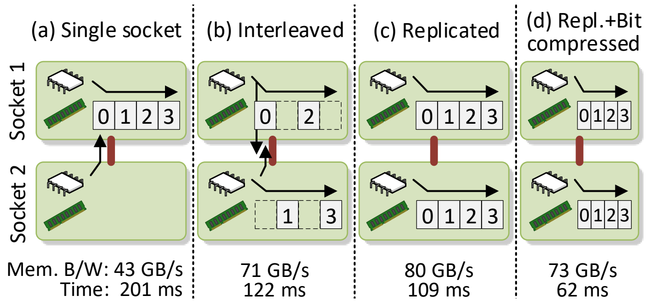


Figure 2: Parallel array aggregation with various smart functionalities on a 2-socket machine with 18-core Xeon CPUs.

on the interconnect. This improves the overall performance by more than 2×.

Our implementation of smart arrays supports additional smart functionalities to express resource trade-offs: multiple data placement options within a NUMA machine and bit compression of the array's contents. Figure 2 illustrates these smart functionalities for a parallel summation of an array on a 2-socket NUMA machine. When the array is placed on a single socket with accesses coming from threads on both sockets (Figure 2a), the bottleneck is the socket's memory bandwidth. When the array is interleaved across the machine's sockets (Figure 2b), we use both sockets' memory bandwidth to decrease the execution time, and the bottleneck is the interconnect. If the array is replicated across sockets using more memory space (Figure 2c), we localize memory access and hence remove the interconnect as a bottleneck to further decrease the execution time. Finally, we can use memory bandwidth more productively by compressing the array's contents (Figure 2d) to pass more elements through the same memory bandwidth, achieving the best performance.

Moreover, our smart arrays provide language-independent access to their contents and smart functionalities. Our prototype is implemented in C++, but can be accessed from workloads written in C++ or in Java. Even without exploiting the smart functionalities, the performance achieved from Java workloads is competitive with Java's built-in array types. Figure 3 illustrates the performance of a simple single-threaded workload. The top two bars show the array aggregation implemented natively in C++ and Java using their built-in array types. Then, we use arrays allocated in C++ and accessed via Java Native Interface (JNI) [30, 34] calls, using

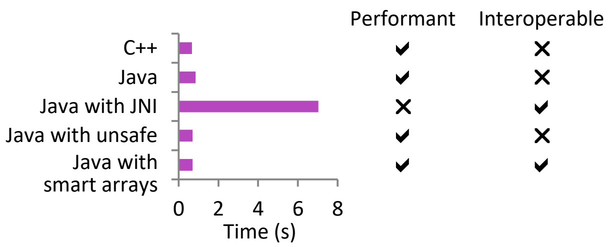


Figure 3: Single-threaded aggregation with C++, Java, Java accessing the native arrays via JNI, unsafe, and smart arrays.

`sun.misc.Unsafe` [36], and using our smart arrays. JNI is interoperable, meaning that we would not need to re-implement our smart functionalities for Java, but the performance is poor. Unsafe is fast but would require us to re-implement our smart functionalities in Java. Finally, our smart arrays are both fast and interoperable, letting us make the different smart functionalities implemented in C++ available to Java without re-implementation.

Contributions. Our main contributions are:

- *Language interoperability.* Smart arrays are implemented once in C++ and can be used efficiently in multiple programming languages, e.g., C++ and Java (see §3).
- *Smart functionalities.* Smart arrays support different trade-offs between the use of hardware resources, e.g., different NUMA-aware data placements and bit compression (see §4). We experimentally show how these smart functionalities can significantly decrease the memory space requirements of analytics workloads, and improve their performance by up to 4× (see §5).
- *Adaptivity.* Our analysis motivates the need for dynamically adapting smart functionalities to the system and the workload. We show an algorithm for this (see §6).

Finally, we consider smart arrays as the first fundamental building step towards general *smart collections.* We envision smart collections that are accessible through interfaces, such as arrays, sets, maps, by multiple programming languages without needing re-implementation, and their smart functionalities can be automatically selected at runtime (see §7).

2 BACKGROUND

In this section, we give a brief overview of modern NUMA machines (see §2.1) and the frameworks that we use for our implementation: the Callisto run-time system we build upon (see §2.2), the PGX graph analytics system used for our evaluation (see §2.3), and the GraalVM compilation framework (see §2.4).

2.1 Modern NUMA Machines

Modern machines consist of interconnected sockets of multi-core processors. Memory is decentralized, and attached to each socket in a cache-coherent non-uniform memory access (ccNUMA) architecture [28, 44]. Figure 4 shows a typical machine with two sockets, each containing an 8-core Xeon CPU with two hyper-threads per core. Each core has a L1-I/D and L2 cache. Each CPU has a shared L3 last-level cache. Each socket has four 32 GB DDR4 DIMMs, and the two sockets are connected with Intel QuickPath Interconnect (QPI) 16 GB/s links [23].

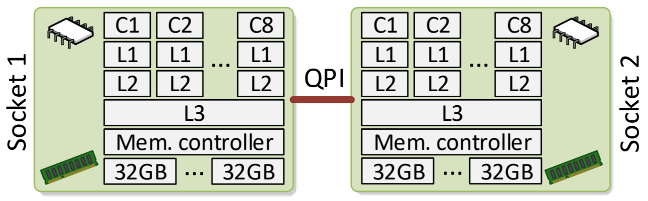


Figure 4: A 2 socket machine using 8-core Xeon CPUs.

Although NUMA topologies can vary, e.g., by the number of sockets, processors, memory, and interconnects, there are a few common fundamental performance characteristics [4, 45]: (*i*) remote memory accesses are slower than local accesses, (*ii*) the bandwidth to a socket's memory and interconnect can be separately saturated, and (*iii*) the bandwidth of an interconnect is often much lower than a socket's local memory bandwidth.

Performance-critical applications need to be *NUMA-aware* by using OS facilities to control the placement of data and of threads [4, 45]. On Linux, the default data placement policy is to physically allocate a virtual memory page on the socket on which the thread that first touches it is running on (after raising a page-fault) [28]. Other policies include explicitly pinning pages on sockets and interleaving pages in a round-robin fashion across sockets (see Figure 2).

2.2 Callisto-RTS

The Callisto runtime system (RTS) [22] is a C++ runtime system that supports parallel loops with dynamic distribution of loop iterations between worker threads. This provides a similar programming model to dynamically scheduled loops in OpenMP, with the difference that the work distribution techniques permit more fine-grained scalable distribution of work, even on an 8-socket machine with 1024 hardware threads [22].

In this paper, we use a prototype of Callisto-RTS which includes a basic Java library to express loops. The loop body is written as a Java lambda function. Each such loop executes over a pool of Java worker threads which make JNI calls from Java to C++ each time the worker requires a new batch of loop iterations; the fast-path distribution of work between threads occurs in C++, and the use of JNI is designed to pass only scalar values, avoiding cases which are typically costly with JNI.

2.3 PGX

PGX [38, 50] is a fast, parallel, in-memory graph analytics framework written in Java. For this paper and our evaluation of smart arrays, we use a prototype of PGX. The inner loops of graph analytics algorithms such as PageRank are written in parallel loops and scheduled using Callisto-RTS.

PGX serves as a main use-case for our work. In in-memory graph analytics systems, main memory is typically a precious resource given the large memory consumption of graphs. Further, the performance of graph analytics queries is often limited by memory bandwidth, motivating the need to optimize its use.

2.4 GraalVM

The Graal Virtual Machine (GraalVM) is an environment for compiling and running multi-language applications. It is a modified version of the Java HotSpot VM [35] and reuses all its components including the garbage collectors and the interpreter. However, it adds Graal [12, 13, 52], a novel dynamic compiler implemented in Java, and Truffle [61–63], a framework for building high-performance language implementations (see Figure 5). A Truffle language implementation is an interpreter, e.g., of an abstract syntax tree or bytecode, that uses Graal to dynamically compile guest language applications to optimized machine code.

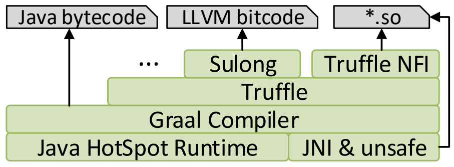


Figure 5: The GraalVM for multi-language applications.

The GraalVM consists of Truffle language implementations for JavaScript, Python, R, and Ruby. Pertinent to our work in this paper, GraalVM also includes Sulong, a Truffle implementation of LLVM bitcode [47]. Sulong uses LLVM front ends such as clang [54], to compile source languages, e.g., C/C++, to LLVM bitcode and interprets it on the GraalVM.

GraalVM guest languages can use Truffle's Native Function Interface (NFI) to access native libraries. When Sulong uses NFI, it aligns data allocations using the same layout as in executables produced by static compilers such as GCC [17], so that the native function can directly operate on allocations provided by Sulong as they match the platform's application binary interface.

The GraalVM features cross-language interoperability that enables efficient interaction between code in different languages [20]. The compiler can optimize applications across language boundaries without overhead. In this work, we show how to use the GraalVM to create language-independent smart arrays and execute their smart functionalities efficiently across languages.

3 LANGUAGE INTEROPERABILITY

We introduce smart arrays and explain our approach to language-independent data structures. The motivating example of Figure 2 illustrates the main problems we address: different implementations of the same abstract data type can have significantly different performance characteristics. There are trade-offs involving the consumption of various hardware resources, e.g., memory bandwidth and space. Programmers need to choose the specific implementation that fits the target hardware, workload, inputs, and system activity. Moreover, different scenarios may require these trade-offs to be made in different programming languages. Smart arrays aim to solve these problems. In this section, we focus on how we support access to smart arrays from C++ and from Java. Then, we turn to the implementation of *smart functionalities* that provide the different trade-offs between hardware resources (see §4), perform an experimental evaluation (see §5), present how to adaptively select smart functionalities (see §6), and finally discuss expanding to additional smart collections and functionalities (see §7).

Figure 6 illustrates our overall approach: the underlying data structure and trade-offs are implemented once in C++ (right-hand side), and exposed to different languages via thin per-language wrappers (left-hand side). In this section, we describe the underlying implementation in C++ (see §3.1), and then the techniques we use to make this available efficiently to Java (see §3.2).

3.1 Smart Arrays in C++

We implement the core functionality of smart arrays and their smart functionalities in C++ within Callisto-RTS. This approach

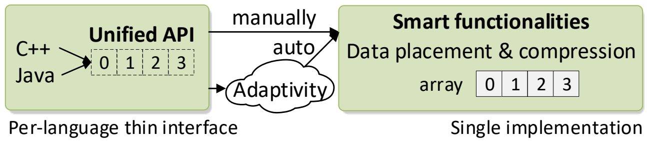


Figure 6: Access to a smart array.

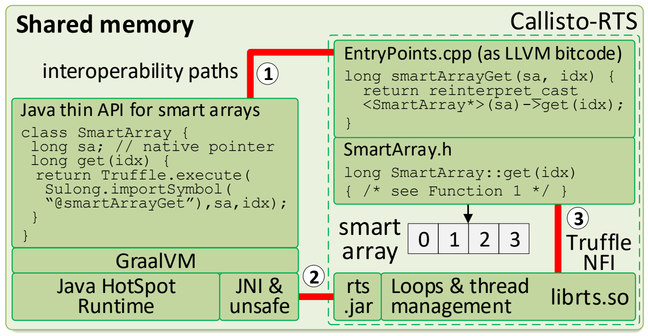
provides a number of advantages: (*i*) in C++ we can control the memory layout of the smart arrays by interfacing with the OS (e.g., by making system calls for NUMA-aware data placement), (*ii*) by careful design of the Java-C++ interface, we can use GraalVM to inline the C++ implementation into other languages and to optimize it alongside user code, and (*iii*) by having a single implementation we avoid re-implementing functionality for multiple languages while still enabling multi-language workloads.

3.2 Exposing Smart Arrays to Java

Figure 7 shows conceptually how we expose the C++ implementation to Java. In addition, the figure depicts the three different ways in which the native world of Callisto-RTS (see §2.2) interacts with the managed world of Java.

The first interoperability path is central to the efficient interoperability between C++ smart arrays and Java. This is the fastest interoperability path, and is made available by the GraalVM to enable access to our smart arrays for any GraalVM guest language, including Java. Through this path, we exploit the ability of the GraalVM to optimize and compile the LLVM bitcode of our smart functionalities together with the code of the guest language. More specifically, we expose entry point functions to the unified API of smart arrays (see §4.3). The entry points are compiled with clang into LLVM bitcode. Sulong executes the bitcode on top of the GraalVM (see §2.4). These entry points can be seamlessly used by guest languages running on top of GraalVM.

For a more user-friendly experience, we provide a per-language thin API layer that mirrors our unified API. This is shown in Figure 7 for the case of Java with a simplified example. The purpose is to hide the GraalVM API and make accessing our entry points more convenient. Note that no smart functionality is re-implemented in Java. E.g., the `SmartArray::get()` function incorporates our C++ logic for potential replicas and bit decompression. The function is



```
class SmartArray {
 long sa; // native pointer
 long get(idx) {
  return Truffle.execute(
   Sulong.importSymbol(
   "@smartArrayGet"),sa,idx);
 }
}
```

```
long smartArrayGet(sa, idx) {
  return reinterpret_cast
  <SmartArray*>(sa)->get(idx);
}
```

```
long SmartArray::get(idx)
{ /* see Function 1 */ }
```

Figure 7: Smart arrays in C++ exposed to Java.

exposed as an entry point that is compiled into LLVM bitcode and finally executed by Sulong. The GraalVM executes the user's Java code, the Java thin API, including the smart array functionality (C++ code running with Sulong), and eventually dynamically optimizes and compiles this multi-language application.

There are two additional interoperability paths that we use for accessing preexisting components used by our smart arrays. The second interoperability path is via JNI and unsafe methods. This path exists for any Java application but JNI is slow for array accesses and unsafe is not interoperable (see §1). We use this path to access Callisto-RTS's native functionality for parallel loop scheduling (see §2.2). The third interoperability path is Truffle's NFI. This is the slowest path as NFI, similar to JNI, needs pre- and post-processing. It is used to call into precompiled native libraries.

4 SMART FUNCTIONALITIES

In this section, we describe the smart functionalities our smart arrays currently support: NUMA-aware data placement and bit compression. We then describe our unified API for allocating and accessing smart arrays.

4.1 NUMA-aware Data Placement

Smart arrays support various NUMA-aware placements that, as we show in our experiments (see §5), need to be adapted to the workload and system (see §6):

- **OS default**. For NUMA-agnostic applications, or applications that do not need to specify a data placement, we support the default OS data placement policy. Depending on how the array is initialized, its physical location may vary from one socket, e.g., if one thread initializes the array, to random distribution across sockets, e.g., if multiple threads initialize the array.
- **Single socket**. The array's memory pages are physically allocated on a specified socket. This placement can be beneficial or detrimental depending on the relative bandwidths, and maximum compute capability of the processors. In some cases the speedup of the threads local to the data can outweigh the slow down of the remote threads.
- **Interleaved**. The array's memory pages are physically allocated across the sockets in a round-robin fashion. This can be a good default option to distribute memory space and local/remote memory accesses across sockets, but there can be a bandwidth bottleneck on interconnects.
- **Replicated**. One replica placed on each socket. A conceptual example is shown in Figure 8a. This placement can be the most performant solution for read-only or read-mostly workloads, such as analytics, since each thread has fast local accesses to an array's replica, but replication comes at the cost of a higher memory footprint and additional initialization time for replicas.

4.2 Bit Compression

Bit compression is a light-weight compression technique that is popular for many analytics workloads such as column-store database systems [43, 59]. Bit compression uses less than 64 bits for storing integers that require fewer bits. By packing the required bits

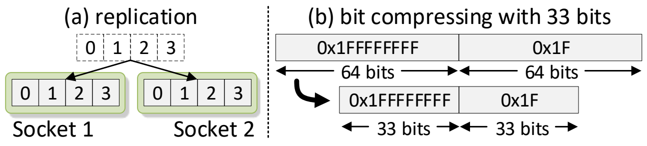


Figure 8: Examples of (a) a smart array replicated across sockets, and (b) bit compressing a smart array with 33 bits.

consecutively across 64-bit words, bit compression can pack the same number of integers into a smaller memory space than the one required for storing the uncompressed 64-bit integers. Figure 8b shows an example of compressing an array with two elements into a bit-compressed array using 33 bits per element. The number of bits used per element is the minimum number of bits required to store the largest element in the array. In future we plan to explore additional compression techniques and the ability to dynamically select the correct technique (see §7).

The primary advantages of bit compression are that it decreases the dataset's memory space requirements, and increases the number of values per second that can be loaded through a given bandwidth. The disadvantage is that it increases the CPU instruction footprint since each processed element needs to be compressed when initialized and be decompressed to 64 bits before the CPU is able to work with it. The additional instructions may hurt performance in comparison to using uncompressed elements (see §5). This additional work can be hidden when iterating sequentially over a bit-compressed array that has a memory bandwidth bottleneck, resulting in faster performance for the compressed array (see Figure 2 and §5). The performance improvement of these scan operations comes from needing to transfer less data through the same bandwidth-restricted memory channel.

Our implementation is based on logically chunking the elements of a bit-compressed array into chunks of 64 numbers. This ensures that the beginning of the first and the end of the last number of the chunk are aligned to 64-bit words for all cases of bit compression from 1 bit to 64 bits. As such, we can effortlessly execute the same compression and decompression logic across chunks. For the sake of simplicity, we focus on unsigned integers, but the concept and our unified API (see §4.3) can be extended to support signed integers.

Function 1 shows the logic of the getter of a smart array compressed with BITS number of bits. BITS is a C++ class template parameter, so there are 64 classes (see §4.3) allowing much of the arithmetic operations to be evaluated at compile time. The function does preparatory work to find the correct chunk index (line 1), the chunk's starting word in the array (lines 2-3), the corresponding chunk's starting bit and word (lines 4-5), the requested index's starting word in the array (line 6), and the mask to be used for extraction (line 7). If the requested element lies wholly in a 64-bit word (line 8), it is extracted with a shift and a mask (line 9). If the element lies between two words (line 10), its two parts are extracted and are combined to return the element (line 11). Our functions assume little-endian encoding, as used by Intel processors.

Function 2 shows the initialization logic. After the same preparatory work as the getter, the function calculates whether the element needs to be split across two words (line 2). The function initializes the element for each replica if the array is replicated (line 3). If the element wholly fits in the first word, its value is set (line 4). If it spills over to the next word (line 5), its second part is set in the next word (line 6).

A thread-safe variant of the function can be implemented using atomic compare-and-swap instructions or using locks, e.g., one per chunk. As we focus on read-only analytics workloads, we do not introduce these synchronization overheads for array accesses in our functions. In cases of concurrent read and write accesses the user of the smart arrays needs to synchronize the accesses.

Function 1 `BitCompressedArray::get(index, replica)`

```
 1: chunk ⟵ index / 64
 2: wordsPerChunk ⟵ BITS
 3: chunkStart ⟵ chunk * wordsPerChunk
 4: bitInChunk ⟵ (index % 64) * BITS
 5: bitInWord ⟵ bitInChunk % 64
 6: word ⟵ chunkStart + (bitInChunk / 64)
 7: mask ⟵ (1 << BITS) - 1
 8: if bitInWord + BITS <= 64 then
 9:  return (replica[word] >> bitInWord) & mask
10: else
11:  return ((replica[word] >> bitInWord) |
       (replica[word+1] << (64-bitInWord))) & mask
```

Function 2 `BitCompressedArray::init(index, value)`

```
1: /* ... same as lines 2-8 of Function 1 ... */
2: word2 ⟵ chunkStart + ((bitInChunk + BITS) / 64)
3: for replica = 0 to replicas do
4:  data[replica][word] = (data[replica][word] &
     ~(mask<<bitInWord))|(value<<bitInWord)
5:  if word != word2 then
6:   data[replica][word2] = (data[replica][word] &
      ~(mask>>(64-bitInWord)))|(value>>(64-bitInWord))
```

Function 3 `BitCompressedArray::unpack(chunk,replica,out)`

```
chunkStart ⟵ chunk * wordsPerChunk
word ⟵ chunkStart
value ⟵ replica[word]
bitInWord ⟵ 0
for i = 0 to 64 do
 if bitInWord + BITS < 64 then
  out[i] = (value >> bitInWord) & mask
  bitInWord += BITS
 else if bitInWord + BITS == 64 then
  out[i] = (value >> bitInWord) & mask
  bitInWord = 0
  word++
  value = replica[word]
 else
  nextWord = word + 1
  nextWordValue = replica[nextWord]
  out[i] = mask & ((value >> bitInWord) |
    (nextWordValue << (64-bitInWord)))
  bitInWord = (bitInWord + BITS) - 64
  word = nextWord
  value = nextWordValue
```

Additionally, in order to optimize scans, which are significant operations in analytics workloads, we support a function that can unpack a whole chunk [43, 59]. Function 3 shows the unpack logic, which condenses consecutive getter operations for a complete chunk of a replica and outputs the 64 numbers of the chunk to a given output buffer. After a similar preparatory work (lines 1-4) as in Function 1, we start iterating over the chunk's elements (line 5). For every element, we decide if it is wholly within the current word (line 6). If it is, we output it (line 7) and continue to the next element (line 8). If the current element also finishes the current word (line 9), we again output it (line 10), reset the bit index in the current word (line 11), and continue to the next word (lines 12-13). If the current element crosses over to the next word (line 14), we make up the element from its two parts across the words and output it (lines 15-17), continuing on to the next element (lines 18-20). The main loop of the function can be manually or automatically (by the compiler) unrolled to avoid the branches and permit compile-time derivation of the constants used.

4.3 Unified API

Figure 9 shows the smart arrays' C++ classes and their API. The `SmartArray` class is an abstract class holding the basic properties that signify whether the smart array is replicated, interleaved or pinned to a single socket (data placements cannot be combined), and the number of bits with which it is bit compressed. If the `SmartArray` is `replicated`, the `replicas` array holds a pointer per socket. Each pointer points to the replica allocated on the corresponding socket. If replication is not enabled, there is a single replica in the `replicas` array. The `allocate()` static function creates a new smart array using the concrete sub-classes depending on the bit compression, and allocates the replica(s) considering the given data placement parameters. The `getReplica()` function returns the replica corresponding to the socket of the calling thread. The remaining functions correspond to the pseudo code shown in Functions 1-3.

The concrete sub-classes of `SmartArray` correspond to all cases of bit compression with a number of bits 1-64, as explained in see §4.2. We specialize the cases of bit compression with 32 and 64 bits as they directly map to native integers. Consequently, they can be implemented with simplified getter, initialization, and unpack functions that do not require shifting and masking.

Iterator model. In addition to the random access API of the smart array class, we create a forward iterator for efficient scans (shown in Figure 9 as well). This makes it possible to hide replica selection and the unpacking of the compressed elements. `SmartArrayIterator` is an abstract class holding a pointer to the referenced smart array, the target replica, and the current index of the iterator. A new iterator can be created by calling the `allocate()` static function. The `allocate()` function sets the target replica by calling the given `SmartArray`'s `getReplica()` function to get the replica that corresponds to the socket of the calling thread, and finally constructs and returns one of the concrete sub-classes depending on the bit compression of the underlying smart array. In C++ the iterator is allocated slightly differently when compiled into LLVM bitcode for use from GraalVM: we use Sulong's API to allocate the iterator transparently in GraalVM's heap, to give GraalVM the chance

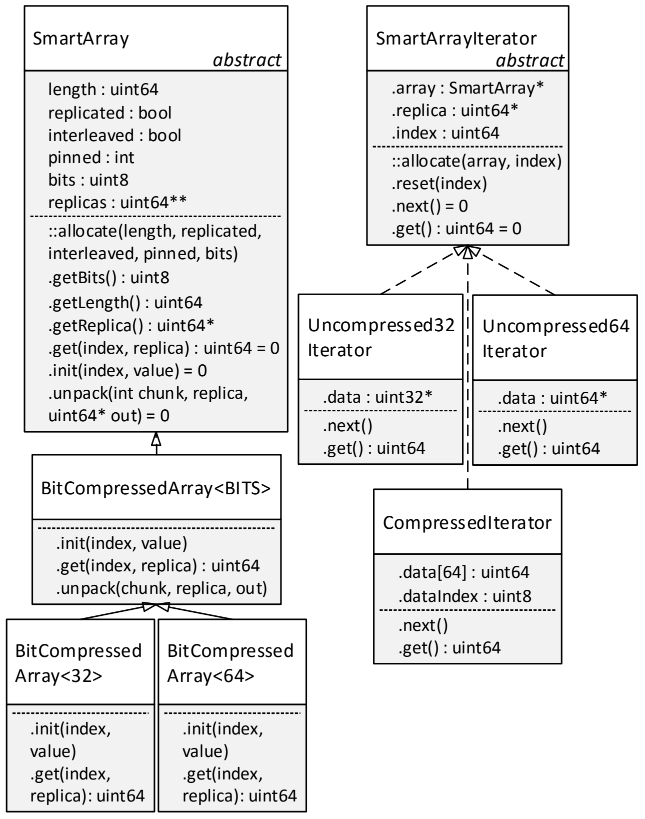


Figure 9: UML diagram of the smart array and iterator API.

to additionally optimize the allocation when compiling the user's code that uses our iterator API. The `reset()` function resets the current index to what is given as the argument. The `next()` function moves to the next index. The `get()` function gets the element corresponding to the current index.

The `SmartArrayIterator` has three concrete subclasses. Two correspond to the uncompressed cases with 32 and 64 bits per element, for which we have specialized versions using 32-bit and 64-bit integers directly. The third corresponds to all other cases of bit compression. The `CompressedIterator` holds a buffer for unpacking elements. When the `next()` function moves to the next chunk, it calls the smart array's `unpack()` function to fetch the next 64 elements into the buffer. The `get()` function returns the element from the buffer corresponding to the current index.

Java thin API. As explained in §3.2, we provide a thin API to hide the GraalVM API calls to the entry points of our unified API. Figure 6 shows a simplified example of our Java wrapper class for the SmartArray. The wrapper class stores the pointer to the native object of the SmartArray. The native pointer is given to the entry point functions.

We provide entry points and wrapper classes only for the two abstract classes of our unified API: `SmartArray` and `SmartArray-Iterator`. Furthermore, the entry points and wrapper functions have an additional version where the user can pass the number of bits with which the smart array is bit-compressed. Depending on

Function 4 `aggregate()` example in both C++ and Java

```
// C++
it = SmartArrayIterator::allocate(smartArray, 0);
for (long i=0; i < smartArray.getLength(); i++) {
 sum += it->get();
 it->next();
}

// Java
it = new SmartArrayIterator(smartArray, 0);
long bits = GraalVM.profile(smartArray.getBits());
for (long i=0; i < smartArray.getLength(); i++) {
 sum += it.get(bits);
 it.next(bits);
}
```

the number of bits, the entry point branches off and redirects to the function of the correct sub-class, thus avoiding the overhead of the virtual dispatch and dispensing us from the need to provide separate entry points to the sub-classes. Moreover, GraalVM can avoid the branching in the entry points by profiling the number of bits during the interpreted runs and considering it as fixed during optimization and when applying just-in-time compilation [61].

Example. Function 4 shows what the final experience looks like for a programmer with a simple example of an aggregation of a smart array in C++ and Java. The example uses an iterator since the aggregation scans the smart array.

The C++ example uses the abstract SmartArrayIterator class, but can immediately use a concrete sub-class depending on the number of bits with which the smart array is bit-compressed in order to avoid any virtual dispatch overhead.

The Java function is very similar to the C++ function. It is executed with the GraalVM. We use the versions of the thin API's functions that receive the number of bits. We explicitly profile the number of bits to ensure that GraalVM considers the number of bits fixed during compilation, and incorporate the final code of the `get()` and `next()` functions of the concrete sub-class, avoiding any virtual dispatch or branching overhead. E.g., if the smart array is bit-compressed with 33 bits, the `next()` function unpacks every 64 elements immediately with the code of the `BitCompressedArray<33>::unpack()` function. If the smart array is uncompressed with 64 bits, then the `get()` and `next()` functions are so simple that compiled code simply increases a pointer at every iteration of the loop without needing to allocate anything for the iterator.

Our aggregation experiments (see §5.1) parallelize the example running multiple instances of the single-threaded code through Callisto-RTS's parallel for (see §2.2). Each thread in Callisto-RTS executes a batch of work represented as a range of array indices. The `index` argument of the iterator's contructor is used to initialize the starting element of each thread's batch, and then the iterator's index is incremented using `next()` inside the loop batch.

5 EXPERIMENTAL EVALUATION

First, we describe our experimental configuration. Then, we present several experiments with simple aggregations that show various aspects of our smart arrays (see §5.1). Finally, we experiment with graph analytics workloads (see §5.2).

Experimental configuration. We use a prototype built on top of Callisto-RTS for loop parallelism and scheduling, where we add our smart arrays and expose our unified API for C++ and Java. By default, threads used by Callisto-RTS are pinned and do not move during execution. In all experiments, we use all available hardware thread contexts.

For all experiments, we perform 5 warm-up iterations and we ensure that Java code is compiled. Performance metrics are gathered from Linux and hardware counters via Intel PCM [58]. Results are averages of 10 iterations. Standard deviation is always <5%. Measured time does not include initialization time, as it is not the focus of this paper. Moreover, in many cases, e.g., in PGX, initialization time can be hidden behind the data loading's I/O bottleneck (see §6). Table 1 shows the used machines' characteristics. NUMA performance characteristics, such as local and inter-socket latencies and peak memory bandwidths, are measured with the Intel MLC [56]. We disable Linux's AutoNUMA [8, 49] page migration facility, as we are interested in evaluating data placements (see §4.1) separately and AutoNUMA requires several iterations to stabilize its final data placement.

Table 1: Oracle X5-2 machines' characteristics [39]. Note the difference in the remote memory bandwidth between the sockets of the machines.

Machine	2×8-core Xeon	2×18-core Xeon
CPU	E5-2630v3 (Haswell)	E5-2699v3 (Haswell)
Clock rate	2.4 GHz	2.3 GHz
Memory/socket	128 GB	192 GB
Local latency	77 ns	85 ns
Remote latency	130 ns	132 ns
Local B/W	49.3 GB/s	43.8 GB/s
Remote B/W	8 GB/s	26.8 GB/s
Total local B/W	98.6 GB/s	87.6 GB/s

5.1 Aggregations

The aggregation experiments are a custom benchmark implemented both in C++ (compiled with GCC [17]) and in Java. The dataset consists of two 4 GB arrays of 64-bit integers (~500 million elements). The workload is a parallel aggregation of the two arrays: `sum += a1[i] + a2[i]`. This workload is motivated by database analytics workloads, as it can represent the summation of two columns, and by the popular STREAM benchmark [32] that involves aggregating two arrays, to saturate memory bandwidth. The aggregation is expressed as a parallel for with Callisto-RTS, each thread calculating a local sum and atomically incrementing a global sum variable at the end of each loop batch. We evaluate different cases of bit compression. For each case, the arrays are initialized with integers using the following formula: `a[i] = (i+random(0,1,2)) & ((1<<bits)-1)`. This formula makes sure the integers are slightly random and in the $[0, 2^{bits})$ range. Due to the single-thread initialization, the "first-touch" OS default policy results in a single socket placement. Figure 10 shows the results of the experiments.

8-core machine. Due to the linear scans, the hardware prefetchers can saturate the memory bandwidth. We first focus on the 32-bit and 64-bit cases. The single socket placement exploits the socket's memory bandwidth. The interleaved placement is worse, since the limited memory bandwidth of the interconnect, which consists of a single QPI, is lower than a socket's bandwidth. The replicated placement is the best, as it can exploit the memory bandwidth of both sockets, reducing the time by 2×.

On this machine, bit compression is advantageous for interleaved placements where the compression allows more data to be passed through the low bandwidth QPI link. For the single socket and replicated cases compression hurts performance because the processors cannot saturate the sockets' memory bandwidth any more due to the additional CPU load needed for unpacking.

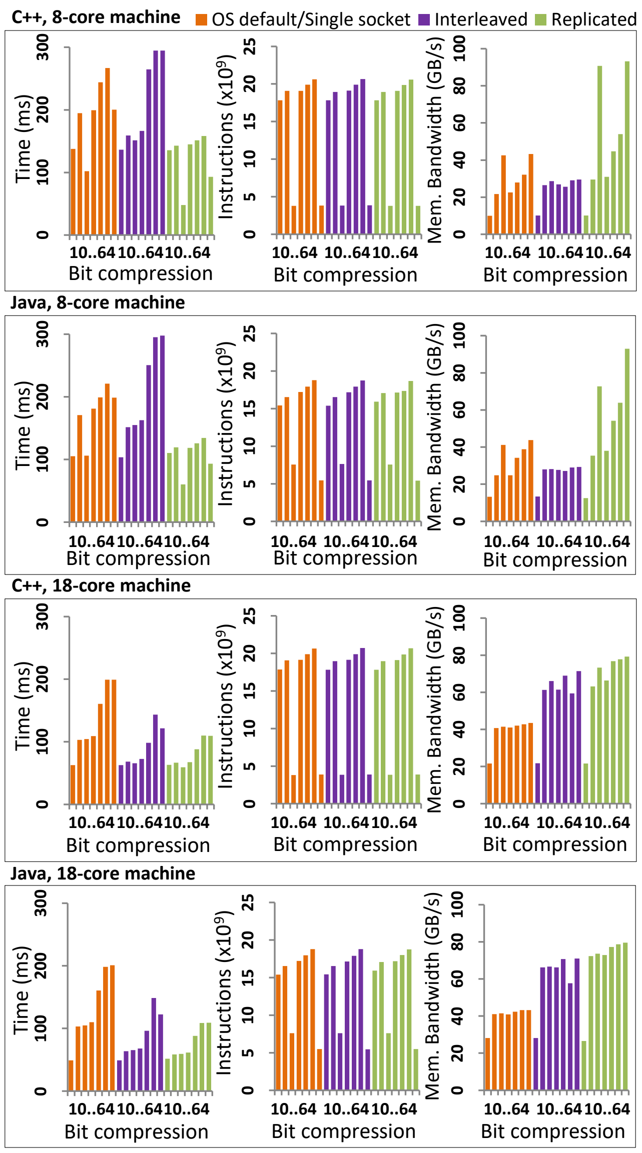


Figure 10: Aggregating two arrays with different cases of bit compression (10, 31, 32, 33, 50, 63, 64), different data placements (OS default / single socket, interleaved, replicated), for C++ and Java, on the 8- and 18-core NUMA machines.

18-core machine. The are two main differences on this machine. First, this machine has a much higher interconnect bandwidth as it has 3 QPI links [24]. This renders interleaving better than the single socket placement. Replication only slightly improves the performance of interleaving.

The second difference is that the 18 cores benefit from compression for all memory placements despite the additional CPU load. Bit compression performs as well as, or slightly better than, the 32-bit case, and much better than the typical 64-bit uncompressed case. The reason is that a smaller volume of memory needs to be passed through the same memory bandwidth, thus execution time is reduced. Bit compression can reduce the time by up to 4× for the default OS data placement, or by up to around 2× for the other data placements, compared to the 64-bit uncompressed case. What is more, at the same time it achieves to reduce the datasets' memory space requirements.

Language interoperability. An important achievement is that the performance of the Java application is generally as good as that of the C++ application. This means that we can practically use our smart arrays efficiently across programming languages without re-implementing the core smart functionalities. Naturally there are still small performance differences between the C++ and Java applications due to the different environments and compilers.

5.2 Graph Analytics Workloads

We use a prototype based on PGX (see §2.3). Graph data is stored in compressed sparse row (CSR) format [37, 51]. Each vertex has a 32-bit ID. A 32-bit edge array concatenates the neighborhood lists of all vertices, i.e., their edges (forward edges in case of directed graphs), using vertex IDs, in ascending order. Another 64-bit array `begin` holds array indices which point to the beginning of the neighborhood list of the vertices. Edges have 64-bit IDs. Two other similar arrays `redge` and `rbegin` hold the reverse edges for directed graphs. Additional arrays may be needed to store vertex and edge properties, as well as for some analytics algorithms and their output [50]. Arrays that may require more than Java's 32-bit maximum array length, including edges and their properties, are stored off-heap [36, 37].

In the following graphs, the "original" data placement refers to the algorithms using the initial on-heap and off-heap arrays, without any smart arrays. The remaining data placements use our smart arrays. We note that arrays are initialized in a multi-threaded way, and thus the execution time of the original and OS default placements varies between the execution time of the single socket and the interleaved data placements.

Degree Centrality. The degree centrality algorithm sums up the out- and in-degrees, i.e., the number of forward and reverse edges, respectively, for each vertex. For each vertex, the algorithm subtracts two consecutive values from the `begin` and `rbegin` arrays to calculate the degrees [50], and stores the sum of the degrees in the output array. We use a large custom graph of 1.5 billion vertices and

3 random edges per vertex. This is a good case for our approach as in the case of bit compression, 33 bits are required to encode edge IDs. The results of our evaluation are shown in Figure 11.

The implications are similar to the aggregation experiments, as the algorithm has also a highly streaming behaviour. On the 8-core machine, replication outperforms other placements as it can exploit the local memory bandwidth of both sockets. Note that the memory bandwidth is not as saturated as in the aggregation experiments, since the algorithm also writes intensively to the output array, which we interleave across sockets in all experiments to ensure a fair comparison. With replication, bit compression is slightly worse than the uncompressed case, but provides a performance boost for all other placements. On the 18-core machine, the implications are also similar to the aggregation experiments. Interleaving is better than the original, OS default, and single socket variations, while replication gives a slight further improvement in performance. Bit compression can still exploit the memory bandwidth sufficiently and further improves performance.

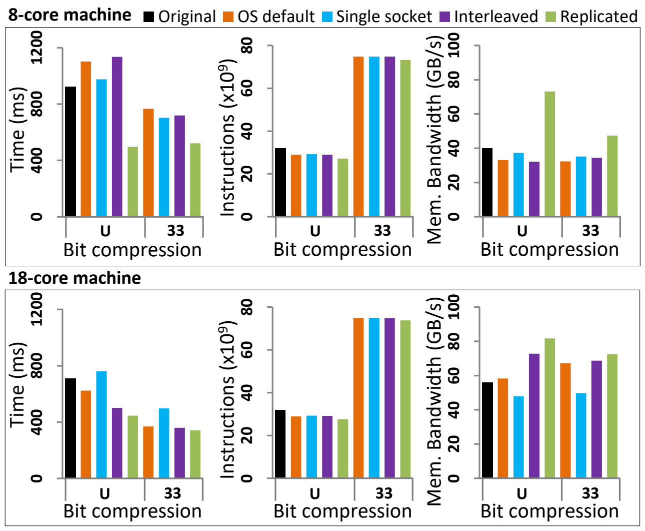


Figure 11: The degree centrality algorithm with different cases of bit compression ("U"/uncompressed, 33 bits) and data placements (original, OS default, single socket, interleaved, replicated), on the 8- and 18-core NUMA machines.

PageRank. Here, we evaluate a more complex and real-world graph analytics algorithm: PageRank [50], which consists of several iterations that calculate and refine the ranks of the vertices until a convergence condition is satisfied. In an iteration, the algorithm loops over the vertices. For each vertex, it loops over the reverse edges to incorporate the neighbours' ranks into the vertex's rank. The algorithm uses the `rbegin` and `redge` arrays, plus two additional vertex property 64-bit arrays: one for the ranks, represented as double-precision floating point numbers, and one for the vertices' out-degrees. By default, the vertex properties are allocated off-heap and are interleaved. The data placement variations apply to all arrays except for the output array, which is always interleaved.

We evaluate PageRank on a graph representing Twitter users and their followers [27], consisting of around 42 million vertices and 1.5 billion edges. PageRank is executed with a damping factor of 0.85 and it converges when the sum of the rank differences from the previous iteration is less than 10^{-3}; it takes the algorithm 15 iterations to meet this requirement. The results are shown in Figure 12 for different cases of bit compression. "U" signifies that we use uncompressed 64-bit and 32-bit arrays as their bits were originally. "V" signifies that we compress the `begin` and `rbegin` arrays with the least number of bits required (31 bits), and the out-degrees vertex property array with 22 bits. "V+E" signifies that we additionally compress the `edge` and `redge` arrays with the least number of bits required (26 bits).

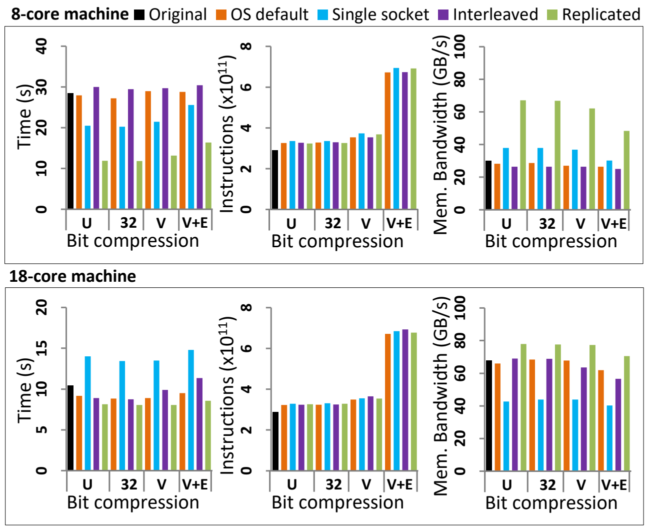


Figure 12: The PageRank algorithm with different cases of bit compression & data placements on the NUMA machines.

The implications are similar to previous experiments. On the 8-core machine, the single socket bandwidth is higher than the bandwidth of the original, OS default, and interleaved data placements, which are constrained by the limited interconnect bandwidth, while on the 18-core machine the interconnect does not limit the achieved bandwidth substantially. As far as replication is concerned, on the 8-core machine it can improve performance by up to 2× compared to the other data placements, while on the 18-core machine it is marginally better than the other data placements. Bit compressing the vertex and vertex property arrays does not have a significant impact on performance, as shown by the measured time and instructions, because PageRank is dominated by the loop over the edges, which accounts for the majority of the runtime and accesses. Thus, we see the real effect of bit compression when we further bit compress the edges. Bit compressing the edges significantly increases the CPU load and generally increases the runtime on the 8-core machine. On the 18-core machine the impact on time can be minimal, e.g., with replicated arrays. Bit compression does not improve performance since the memory bandwidth bottleneck cannot hide the increased CPU load for decompression. By calculating the memory space using the formula: $2{\cdot}bits_{edges}{\cdot}V + 2{\cdot}bits_{vertices}{\cdot}E + bits_{degrees}{\cdot}V + 64{\cdot}V$, where we capture the sizes of `begin`, `rbegin`, `edge`, `redge`, and the vertex property arrays for out-degrees and ranks, we can calculate that variation "V+E" reduces memory space requirements by around 21% over the uncompressed case.

6 ADAPTIVITY

As we observe in the experimental evaluation, depending on the machine, the algorithm, and the input data, the cost, benefit, and availability of the optimizations can vary. For example in the aggregation experiments (see §5.1), for some workloads and data placements on the 18-core machine, there is enough spare compute to benefit from bit compression while on the 8-core machine there is not. Table 2 describes the trade-offs we are seeking to balance.

In this section, we describe and evaluate our approach towards automating the decision about which configuration to use. We show that for our current test set we succeeded in picking the best configuration in 94% of cases, we are within 0.2% of the optimum configuration on average, and we are 11.7% better than the best statically chosen configuration.

To develop this technique we followed the approach we used in Pandia [18], where from a small number of workload measurements, the different configurations' resource needs can be predicted, and select the best configuration for each scenario.

The selection is based on three inputs. First, a specification of the machine containing the size of the system memory, the maximum bandwidth between components and the maximum compute available on each core. Second, a specification of performance characteristics of the arrays such as the costs of accessing a compressed data item. This is derived from performance counters and is specific to the array and the machine, but not the workload. Finally, information collected from hardware performance counters describing the memory, bandwidth, and processor utilization of the workload. We collect the latter information from previous runs from repeated invocation of the same workload, or from previous iterations of an iterative workload, e.g., PageRank iterating to convergence. Alternatively, one could collect workload information from early batches of a loop over the array, and restructure the array on the fly.

Table 2: Trade-offs of smart functionalities.

Technique	Advantages	Disadvantages
Bit Compression	• Smaller memory footprint. • Less memory bandwidth.	• Extra CPU load per access.
Replication	• Less interconnect traffic. • Spreads load evenly across all memory channels.	• More memory footprint. • Time initializing replicas. • Only for read-only data.
Interleaved	• Effective use of bidirectional interconnect. • Load on memory approximately equal across banks.	• May leave memory bandwidth unused as threads stall on interconnect transfers.
Single socket	• Increase in speed on the local socket can outweigh the loss of performance elsewhere.	• Only advantageous if the memory bandwidth is much higher than the interconnect bandwidth.

The configuration selection is broken into 2 steps. First, we use the decision diagrams of Figure 13 to select a candidate for uncompressed data placement and, if possible, for compressed data placement (see §6.1). Then, we use analytics based on ideas explored in Pandia [18] to determine which candidates to use (see §6.2).

The configuration used when collecting the initial workload information is flexible. Here, we use an uncompressed interleaved placement with an equal number of threads on each core. Interleaving both provides symmetry in the execution, and as the interconnect links on many processors are independent in each direction, the bandwidth available to perform the restructuring of the memory is effectively doubled, so reducing the time to change data placement if restructuring on the fly is implemented.

6.1 Step 1: Select Placement Candidates

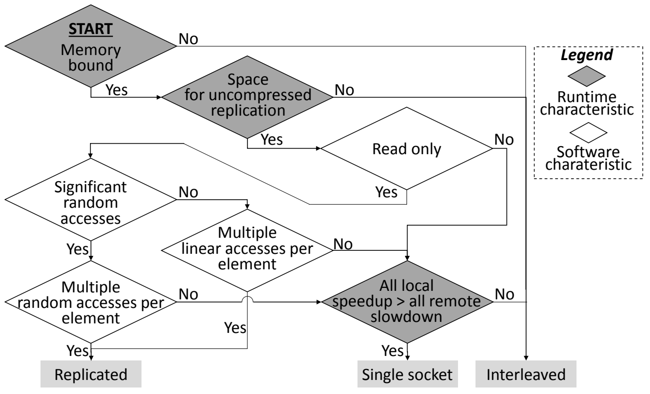
To select the placements we use the flow diagrams shown in Figure 13. The key difference between the two decision diagrams is the use of compression. Choosing a placement for compression requires some of the tests to be moved forward in order to determine if compression is possible before considering which data placement to use. For example, every access requires a number of words to be loaded, making random accesses more expensive than with uncompressed data. While most of the elements in the decision diagram are self-explanatory, we will consider some of them before looking at how to decide between the two candidates. Decisions are split into two categories, "software characteristics", that are based on information provided by the programmer such as numbers of iterations or if the accesses are read-only, and "runtime characteristics" which are based on measurements of the workload.

Space for replication. Replicating arrays, or single socket allocation, requires enough memory on each socket. There are versions of this test for both the compressed and uncompressed data as compression can make replication possible where uncompressed data would not fit otherwise.

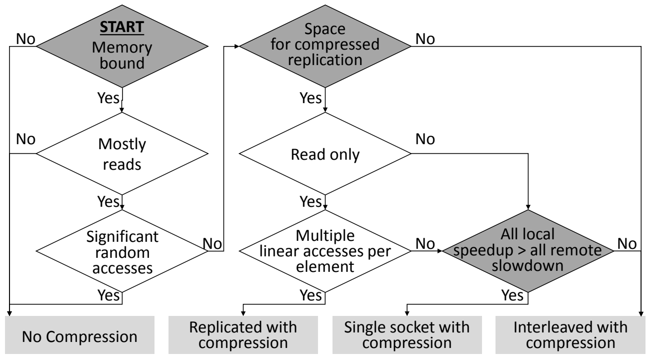
Multiple accesses per element. There is a time cost to initialize replicated data and sufficient accesses are required to amortize this cost. The bounds for this are machine-specific and vary depending on whether the accesses are random or linear. In certain cases, e.g., in PGX or databases, the initialization cost can be hidden behind the data loading's I/O bottleneck [19, 25].

Significant random accesses. If a loop contains many random accesses, then the additional latency cost may affect the point at which replication is worthwhile. This is a machine-specific bound.

All local speedup > all remote slowdown. For some workloads on some architectures, it is better to keep all data on a single socket. This strategy works when the ratio between remote and local access bandwidth is very high. In this case, the speedup for some threads performing only local accesses may outweigh the slowdown of the threads performing remote accesses. To determine if this is the case on a two-socket machine, we perform the following calculations:


START
Memory bound
No
Yes
Space for uncompressed replication
No
Yes
Read only
No
Yes
Legend
Runtime characteristic
Software charateristic
Significant random accesses
No
Yes
Multiple linear accesses per element
No
Multiple random accesses per element
No
Yes
All local speedup > all remote slowdown
No
Yes
Yes
Replicated
Single socket
Interleaved

a) Candidate selection for uncompressed placement.


START
Memory bound
No
Yes
Mostly reads
No
Yes
Significant random accesses
No
Yes
Space for compressed replication
No
Yes
Read only
No
Yes
Multiple linear accesses per element
No
Yes
All local speedup > all remote slowdown
No
Yes
No Compression
Replicated with compression
Single socket with compression
Interleaved with compression

b) Candidate selection for compressed placement.

Figure 13: Flow diagrams to select configuration candidates.

First, we calculate how quickly a socket could compute if relieved of any memory limitations. We use the notion of execution rate (exec) to represent the instructions executed per time unit. Frequency scaling makes instructions per cycle (IPC) an inappropriate metric. We define:

$$\text{improvement}_{\text{exec}} = \text{exec}_{\text{max}}/\text{exec}_{\text{current}}$$

Second, we use the used and available bandwidth (bw) between sockets and to main memory to calculate how fast the local socket could compute with all local accesses assuming that the remote socket is saturating the interconnect link. To account for bandwidth lost due to latency, the bandwidth values taken from the machine description are scaled to the maximum bandwidth used by the workload during measurement. For example, if we achieved 90% utilization of the link that is a bottleneck, the maximum performance of all links are scaled to 90% to reflect the maximum possible utilization. We define:

$$\text{improvement}_{\text{bw}} = \frac{\text{bw}_{\text{max memory}} - \text{bw}_{\text{max interconnect}}}{\text{bw}_{\text{current memory}}}$$

We take the minimum of these two improvements as the maximum speedup of the local socket, $\text{speedup}_{\text{local}}$.

Finally, we calculate the maximum speedup of the remote socket with all remote accesses. We would expect this value to be less than 1, indicating a slowdown.

$$\text{speedup}_{\text{remote}} = \text{bw}_{\text{max interconnect}}/\text{bw}_{\text{current memory}}$$

If the average of the local and remote speedup is greater than 1, then having the data on a single socket it beneficial.

6.2 Step 2: Whether to Use Bit Compression

After selecting our placement candidates in see §6.1, the first step in deciding whether to use the candidate with compression or the candidate without, is to add to the profile of the compression candidate the additional compute that is required to perform the compression. In addition to the current compute rate, we need to know the number of accesses per second (#accesses), and the cost per access resulting from the extra CPU load that need to be executed (cost). The cost of decompression varies with the compression ratio, since the number of values that can be extracted per instruction changes.

$$\text{exec}_{\text{compressed}} = \text{exec}_{\text{current}} + \text{\#accesses} \cdot \text{cost}$$

The reduction in bandwidth is calculated in a similar fashion, using the compression ratio (r) $(0 \ldots 1]$ of the compressed and the uncompressed size of the elements ($\text{elem}_{\text{size}}$).

$$\text{bw}_{\text{compressed}} = \text{bw}_{\text{current memory}} - \text{\#accesses} \cdot (1 - \text{r}) \cdot \text{elem}_{\text{size}}$$

Using these computed values for the compressed case and the measured values for the uncompressed case, we estimate each placement's speedup. For each placement, we compute the ratio of the maximum compute rate relative to the current rate. This way, we obtain each candidate's speedup if the workload is not memory-bound. Next, for each socket we compute the ratio of the maximum memory bandwidth for each candidate placement relative to the current bandwidth. This gives the socket speedup assuming the workload is not compute-bound. Finally, for each socket, we take the minimum of their two ratios as the socket's estimated speedup and average these for the configurations' estimated speedup. We then choose the configuration predicted to be the fastest.

6.3 Evaluation

We evaluate our adaptivity for the aggregation and degree centrality experiments (see §5). We evaluate the two steps separately before looking at overall accuracy. With step 1, we follow the decision diagram for every bit count, benchmark, and hardware combination to determine that both the compressed and uncompressed data placements are correct. We also do this under the assumption that there is insufficient memory for uncompressed replication and finally assuming insufficient memory for the compressed replication. The correct placements were chosen in 62 of the 64 cases. The two failures were for 10 bit aggregations with compression in Java. These were slightly faster with interleaving than with replication.

For the second step, for every placement, benchmark, bit count, and hardware combination we evaluate if compression should be used. This results in 96 combinations. The selection was correct in 86 of the combinations. When an incorrect decision was made, the average performance was 4.8% worse than the best choice, the median performance was 1.6% worse, and overall the performance was 6.4% better than the best static configuration.

Finally comparing the complete process the correct placement was chosen 30 times out of 32. The average performance of the selected configuration for each benchmark and hardware configuration pairing was 0.2% worse than the optimal configuration for that pairing, and 11.7% better than the best static configuration. Fixing the configuration on a per machine basis matches the adaptivity choice on the 8-core machine, but on the 18-core machine the adaptive choice is 1% better than a static choice.

Limitations. The first limitation of our adaptivity is that hardware is complex and the counters only provide an overview of the system state. For most workloads the metrics are acceptable, but it is possible to construct a workload that concentrates on less well provisioned instructions such as square root. Such workloads would appear to have lots of spare CPU so would speed up from extra bandwidth, but in reality would quickly hit the CPU limit. Furthermore, we only take into account latency by preferentially picking the lowest latency data placement. However if the workload is not only limited by bandwidth, but by the round trip time for memory access, we may not get the expected speedup. Finally, our adaptivity is not yet extended to multiple smart arrays, such as those used in our PageRank experiments. Despite these limitations, because hardware is designed to work well in the general case and because we are looking for the best configuration, not a prediction of how good that configuration is, we believe the metrics presented here can be effective.

7 EXTENDING TO SMART COLLECTIONS

Smart arrays are the first step towards more general smart collections with various adaptive smart functionalities, that are accessible through simple interfaces, such as arrays, sets, or maps by multiple programming languages without re-implementation. Our envisioned smart collections are depicted in Figure 14. The figure is a superset of the smart arrays depicted in Figure 6, as smart collections are an extension of the capabilities already demonstrated with smart arrays in this paper. In the rest of this section, we describe how each capability of the smart arrays will be extended to support smart collections.

Language interoperability. Similar to smart arrays, smart collections are implemented once in C++ and are accessible via Sulong by the GraalVM's guest languages such as C++, Java, R, JavaScript etc. To support each additional language, a per-language thin interface is needed, similar to our implemented thin interface for Java (see Figure 7), to connect to the entry points of the unified API.

Collections. Smart collections include sets, bags, and maps. For each smart collection, there is a simple interface to access it in the unified API. For example, a map has an interface to access the keys and a key's associated values. A smart collection can have different data layout implementations.

Similar to smart functionalities, different data layouts support different trade-offs between the use of hardware resources and performance. For example, we can readily use smart arrays to implement data layouts for sets, bags, and maps, by encoding binary trees into arrays, where accessing individual elements can require up to $\log_2 n$ non-local accesses (where n is the size of the collection). To trade size against performance we can use hashing instead of trees to index the smart arrays. This provides $O(1)$ access times on average and data locality on hash collisions [9, 11].

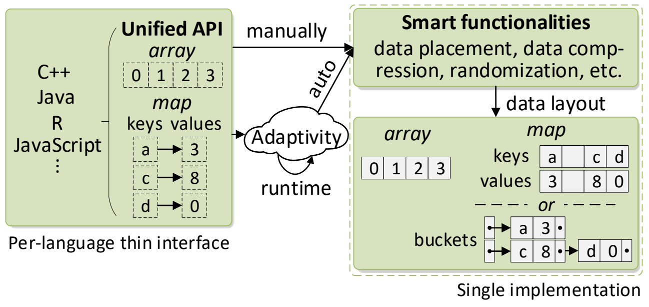


Figure 14: Smart collections.

Smart functionalities. We plan to explore more smart functionalities in addition to NUMA-aware data placement and bit compression. In the context of NUMA machines, we plan to support randomization, a fine-grained index-remapping of a collection's elements'. This kind of permutation ensures that "hot" nearby data items are mapped to storage on different locations served by different memory channels, thus reducing hot-spots in the memory systems if one memory channel becomes saturated before others.

We also intend to extend data placement techniques with partitioning data across the available threads based on domain specific knowledge, similar to [25, 45]. Moreover, we can investigate alternative compression techniques that can achieve higher compression rates on different categories of data [9], such as dictionary encoding, run-length encoding, etc. Furthermore, we can add synchronization support to smart collections in order to support both read and write concurrent workloads.

Alternative unified API. In the case of bit compression, the iterator API has to test whether a new chunk needs unpacking. This generates a large number of branch stalls, which are not evaluated speculatively and increase CPU load (see §5). We plan an alternative unified API for languages that support user-defined lambdas. This API will provide a bounded `map()` interface accepting a lambda and a range to apply it over. In comparison to the iterator API, the map interface can further improve performance as it does not stall on the branches because it is able to remove many of them, and to speculatively execute the lambda in the remaining cases.

Adaptivity. Our adaptivity mechanism will be extended to enable a more dynamic adaptation between alternative implementations at runtime, e.g., by considering the changes in the system load as other workloads start and finish, or the changes in utilization of main memory. It will re-apply its adaptivity workflow to select a potentially new set of smart functionalities and data layouts for multiple smart collections. This process can consider the concurrent workloads of all supported languages on each smart collection.

8 RELATED WORK

Cross-language interoperability. For Java, the most prominent Foreign Function Interfaces (FFIs) are JNI [30], Native Access [55], or Compiled Native Interface[16]. To ease integration of native code in Java, some code generators [2, 46] use annotations to generate FFI code from C/C++ so that programmers avoid writing this code. However, classic FFI approaches define a specific interface between only two languages. With GraalVM, we support transparent and efficient interoperability across all of its supported languages.

Furthermore FFIs often treat the native code as a black box introducing a compilation barrier that can hurt performance. To overcome this, Stepanian *et al.* [53] inline native functions in Java using a just-in-time compiler, which substantially reduces the overhead of JNI calls. Loading native code into the GraalVM via Sulong removes this compilation boundary [20, 21], so the compiler can inline the complete application including our Java thin API and smart functionalities.

NUMA replication and placement. Optimizations to data placement, including replication, have been proposed to increase cache and NUMA locality, or optimize the use of interconnects. Calciu *et al.* [6] propose a way to turn regular data structures into concurrent NUMA data structures with per-node replication. Their main focus is providing linearizable and fast concurrent access. For graphs, Wei *et al.* [57] improve cache locality by storing vertices that will be frequently accessed close to each other. Dashti *et al.* [10] use a combination of page co-location, interleaving, replication and thread clustering to avoid interconnect bandwidth saturation. We similarly support replication, but without requiring modifications to the OS. Lepers *et al.* [29] optimize placement on asymmetric NUMA architectures. AutoNUMA [8, 49] tries to run threads close to memory pages they access, by moving threads and migrating pages, but does not provide replication.

Bit compression and approximation. In-memory databases are known to use bit compression similar to ours, combined with dictionary encoding. Wilhalm *et al.* [59, 60] use SIMD for fast scanning of bit-compressed tables in SAP HANA. Polychroniou *et al.* [42, 43] show how to use SIMD to efficiently interleave bits of bit-compressed data to speed up selection scans. SIMD can be applied to our bit compression as well, however, previous related work is not straightforward to apply efficiently as their focus is on values smaller than 32 bits. Other compression schemes are feasible as well, e.g., to drop some least-significant bits in applications where full precision for floats is not needed. This has been done at the cache level to reduce the physical size of data caches [33].

Compression of large graphs. Boldi *et al.* [5] propose compression algorithms for large Web graphs. Khandelwal *et al.* [26] propose ZipG, a distributed, memory-efficient graph store capable of executing queries directly on a compressed representation of the graph. Maass *et al.*[31] propose Mosaic, a graph-processing engine that uses a space-efficient representation, Hilbert-ordered tiles, which has a good compression ratio while avoiding the overhead of decoding edge sets at runtime. Such graph-specific compression techniques use domain-specific knowledge and can be typically combined with the generic bit compression.

Adaptivity and self-tuning. Kaestle *et al.* [25] optimize data placement and selectively enable replication at compile-time, based on annotations or static code analysis. In contrast, our adaptivity uses runtime profiling and avoids overheads from replication if the workload does not benefit from it. Psaroudakis *et al.* [44, 45] adaptively move or repartition database tables, and selectively enable task stealing, but do not consider replication. Eastep *et al.* use machine learning to pick implementations for algorithms [14], and adapt lock algorithms at runtime [15]. We focus on adapting data, e.g., their placement and compression, at runtime.

PetaBricks [1] propose a programming language whose compiler automatically picks algorithms, parallelization techniques and data distributions. Our approach does not require a specific programming language. Self-tuning configuration parameters of databases [7] and frameworks such as Hadoop [3, 48] have also been proposed. Nevertheless, self-tuning work focuses on indexes, query optimization and execution rather than on NUMA-aware data placements.

9 CONCLUSIONS

In the era of big data, analytics workloads need to identify the hardware resource bottlenecks in modern machines and use the trade-offs between the hardware resources more productively. We introduce smart arrays as a novel language-independent abstraction that adaptively enables various smart functionalities to toggle the trade-offs between the use of hardware resources. Smart arrays are implemented once in C++ and are accessible by both C++ and compiled Java code efficiently. We implement smart functionalities for NUMA-aware data placement and bit compression of the smart arrays. We experimentally evaluate smart arrays with various analytics workloads, including real-world graph analytics workloads. We observe that smart arrays can significantly decrease the memory space requirements of analytics workloads, and improve their performance by up to 4×. Finally, smart arrays are the first step towards our envisioned general smart collections that are accessible through simple interfaces by multiple programming languages without re-implementation, and their smart functionalities can be adapted at runtime.

ACKNOWLEDGMENTS

The authors would like to thank our shepherd, Steven Hand, the anonymous reviewers, and Chris Seaton for his initial technical assistance. We would also like to thank Lucas Braun, Davide Basilio Bartolini, Vlad Haprian, Vasileios Trigonakis, and Alex Kogan for proofreading.

REFERENCES

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: a Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 38–49. https://doi.org/10.1145/1542476.1542481

[2] David M Beazley. 1996. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*. 129–139.

[3] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (2016), 1470–1483. https://doi.org/10.1109/TPDS.2015.2449299

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=2002181.2002182

[5] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, 595–602. https://doi.org/10.1145/988672.988752

[6] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[7] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 3–14. http://dl.acm.org/citation.cfm?id=1325851.1325856

[8] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. (March 2012). http://lwn.net/Articles/488709/.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[10] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: a Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 381–394. https://doi.org/10.1145/2451116.2451157

[11] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[12] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, 187–193. https://doi.org/10.1145/2647508.2647521

[13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, 1–10. https://doi.org/10.1145/2542142.2542143

[14] Jonathan Eastep, David Wingate, and Anant Agarwal. 2011. Smart Data Structures: an Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, 11–20. https://doi.org/10.1145/1998582.1998587

[15] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. 2010. Smartlocks: lock Acquisition Scheduling for Self-aware Synchronization. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC '10)*. ACM, 215–224. https://doi.org/10.1145/1809049.1809079

[16] GNU. 2017. Compiled Native Interface. (June 2017). https://gcc.gnu.org/onlinedocs/gcc-6.4.0/gcj/About-CNI.html.

[17] GNU. 2017. The GNU Compiler Collection (GCC). (Oct. 2017). https://gcc.gnu.org/.

[18] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 254–269. https://doi.org/10.1145/3064176.3064177

[19] Jim Gray and Franco Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. ACM, 395–398. https://doi.org/10.1145/38713.38755

[20] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[21] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, 1–13. https://doi.org/10.1145/2724525.2728790

[22] Tim Harris and Stefan Kaestle. 2015. Callisto-RTS: Fine-grain Parallel Loops. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 45–56. http://dl.acm.org/citation.cfm?id=2813767.2813771

[23] Intel. 2009. An introduction to the Intel QuickPath Interconnect. (Jan. 2009). http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[24] Intel. 2015. Intel Xeon Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual. (June 2015). https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html.

[25] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, 263–276. http://dl.acm.org/citation.cfm?id=2813767.2813787

[26] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. ZipG: a Memory-efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, 1149–1164. https://doi.org/10.1145/3035918.3064012

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[28] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages. https://doi.org/10.1145/2508834.2513149

[29] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[30] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[31] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, 527–543. https://doi.org/10.1145/3064176.3064191

[32] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25. http://www.cs.virginia.edu/stream/.

[33] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a Cache for Approximate Computing. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, 50–61. https://doi.org/10.1145/2830772.2830790

[34] Oracle. 2017. Java Native Interface (JNI) Specification. (Oct. 2017). http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.

[35] Oracle. 2017. Open Java Development Kit (JDK) - HotSpot virtual machine. (Oct. 2017). http://openjdk.java.net/groups/hotspot/.

[36] Oracle. 2017. Open Java Development Kit (JDK) - Unsafe. (Oct. 2017). http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/sun/misc/Unsafe.java.

[37] Oracle. 2017. PGX 2.5.0 Documentation - PGX Memory Consumption. (Oct. 2017). https://docs.oracle.com/cd/E56133_01/latest/reference/memory.html.

[38] Oracle. 2017. PGX website. (Oct. 2017). http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics/overview/index.html.

[39] Oracle. 2017. X5-2 server documentation. (Oct. 2017). http://docs.oracle.com/cd/E41059_01/.

[40] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. Project Snowflake: Non-blocking Safe Manual Memory Management in .NET. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 95 (Oct. 2017), 25 pages. https://doi.org/10.1145/3141879

[41] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquín, and Donald Kossmann. 2017. Fast Scans on Key-value Stores. *Proc. VLDB Endow.* 10, 11 (2017), 1526–1537. https://doi.org/10.14778/3137628.3137659

[42] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1493–1508. https://doi.org/10.1145/2723372.2747645

[43] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware (DaMoN'15)*. ACM, Article 9, 6 pages. https://doi.org/10.1145/2771937.2771943

[44] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1442–1453. https://doi.org/10.14778/2824032.2824043

[45] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware Data Placement and Task Scheduling for Analytical Workloads in Main-memory Column-stores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 37–48. https://doi.org/10.14778/3015274.3015275

[46] John Reppy and Chunyan Song. 2006. Application-specific Foreign-interface Generation. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. ACM, 49–58. https://doi.org/10.1145/1173706.1173714

[47] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of Workshop on Virtual*

Machines and Intermediate Languages (VMIL '16). 6–15. https://doi.org/10.1145/2998415.2998416

[48] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. 2009. Atune-IL: an Instrumentation Language for Auto-tuning Parallel Applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*. Springer-Verlag, 9–20. https://doi.org/10.1007/978-3-642-03869-3_5

[49] Lee T. Schermerhorn. 2007. A matter of hygiene: automatic page migration for Linux. (2007). https://linux.org.au/conf/2007/talk/197.html.

[50] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using Domain-specific Languages for Analytic Graph Databases. *Proc. VLDB Endow.* 9, 13 (2016), 1257–1268. https://doi.org/10.14778/3007263.3007265

[51] Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. 2014. Fast In-Memory Triangle Listing for Large Real-World Graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis (SNAKDD'14)*. ACM, Article 2, 9 pages. https://doi.org/10.1145/2659480.2659494

[52] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 165, 10 pages. https://doi.org/10.1145/2544137.2544157

[53] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. 2005. Inlining Java Native Calls at Runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, 121–131. https://doi.org/10.1145/1064979.1064997

[54] The LLVM Developer Group. 2017. The LLVM Compiler Infrastructure. (Oct. 2017). http://www.llvm.org/.

[55] Todd Fast, Timothy Wall, L. C. et al. 2018. Java Native Access. (Feb. 2018). https://github.com/java-native-access/jna/blob/master/README.md.

[56] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Patrick Lu, Blazej Filipiak, and Sri Sakthivelu. 2017. Intel Memory Latency Checker (MLC). (Oct. 2017). http://www.intel.com/software/mlc.

[57] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 1813–1828. https://doi.org/10.1145/2882903.2915220

[58] Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2017. Intel Performance Counter Monitor (PCM). (Oct. 2017). http://www.intel.com/software/pcm.

[59] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing database column scans with complex predicates. In *Proceedings of the 4th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, ADMS*, Vol. 13. 1–12.

[60] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 385–394. https://doi.org/10.14778/1687627.1687671

[61] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[62] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, 187–204. https://doi.org/10.1145/2509578.2509581

[63] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. ACM, 73–82. https://doi.org/10.1145/2384577.2384587

[64] Reynold Xin and Josh Rosen. 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. (April 2015). Blog post, accessed 18 October, 2017. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[65] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664