# UCARE Research

Ray Andrew

January 7, 2020

## Abstract

The abstract text goes here.

# Contents

## 0.1 Todos

### 0.1.1 03 November 2019

- Add markOop memory layout
- Upload docs into github
- Test dedup System.copyarray

## 0.2 Meeting Notes

#### 0.2.0.1 09 October 2019

- Try the Kafka sytem inside Eclipse OpenJ9 test whether parameter class-datashare is working or not
- IBM Java Multitenancy
- See implementation of native function (arrayCopy) and try see how the copy the array and look into the content
- No matplotlib and try to change it GNUPlot

#### 0.2.0.2 22 October 2019

- Create our own SLABs, 32 Nodes, 7 Workloads
- Check if object is unmodified
- Understand what is Klass, MarkOop, Array looks like on memory layout
- send me string dedup paper, read it and we discuss it next tuesday

#### 0.2.0.3 30 October 2019

- 1 more week to implement arraycopy to check whether we can call slab allocation directly (basically mmap)
- Mark class memory layout
- Upload the documents
- Check implementation of String Deduplication in OpenJ9
- Continuing other bugs

#### 0.2.0.4 05 November 2019

- Using jmap to dump the JVM, see this link : Jmap
- do sample program to allocate one array of ints (size 10) and run into two or three JVMs, dump that array
- check what is different, what is the same
- differences in object layout

#### 0.2.0.5 17 December 2019

- Write why those KSM works (a summary) (1-2 pages)

#### 0.2.0.6 07 January 2020

- we only share ‘all pages‘ not the ‘real data‘
- how can we be more specific -> to save the real data instead of whole heap?
- we need to go deeper into ‘old generation‘ itself and find the real content

- questions
  - 1 instance of systems -> running a workload -> we need to know at some point X objects
  - and at Y systems -> how many objects?
  - what is the intersection between X and Y
  - e.q =
  - 2 running replicas run workloads -> how many objects that have the same data?
- steps
  - using JVMTI in the callbacks -> to iterate all objects in Heap
  - for each one of objects we will try to compare to another JVMs
  - we want to check the objects between two objects
  - given two instances -> looking at the content -> are they same?

## 0.3 Summaries

### 0.3.1 Memory Deduplication

#### 0.3.1.1 Introduction

##### 0.3.1.1.1 Virtualization

Virtualization is used throughout the world and one of growing trend for hosting services within data-centers. [1] Virtualization uses dynamic resource allocation and migration techniques. Memory over-commit is one of commonly used technique to facilitate server consolidation where the total memory size for all running VMs exceeds actual pyshical memory of hypervisor.

**There are two improvements that have been done to overcome this issue:**

1. Demand paging -> pages are swapped back and forth to alleviate memory requirements of VMs
2. Memory ballooning -> inflated to force the machine to relinquish pages according to it's local memory management policy
3. Exploit redundancy of memory content

##### 0.3.1.1.2 Content Based Page Sharing (CPBS)

One of major feature that has been implemented throughout hypervisors around the world. Another essential feature of **CPBS** is **CPBS** able to perform transparently in the hypervisor layer and doesn't require any modification to guest OS / process.

**Classified into two categories:**

1. in-band sharing -> same page detection and merging in the I/O path (mostly disk access path)
2. out-band sharing -> periodically scans memory to identify and merge shareable pages

Out-of-band sharing techniques can potientially take advantage of complete system memory to identify identical memory pages. Out-of-band techniques such as KSM, usually **oscillate between a sleep period and a scan period**

#### 0.3.1.2 Kernel Samepage Merging (KSM)

##### 0.3.1.2.1 Introduction

Created by RedHat, Inc. in this paper [2] in team consists of Andrea Arcangeli, Izik Eidus, Chris Wright.

**The main goal is to:** share equal anonymous memory across different processes and in turn also across different KVM virtual machines.

One thing to remember is: **KSM is not only for KVM virtual machines, all processes that has equal anonymous memory can be shared through KSM daemon**

#### 0.3.1.2.2 Algorithm

**KSM daemon uses two global red-black comparisons tree for whole memory pages:**

1. **Stable tree** contains already shared pages with write-protected
2. **Unstable tree** contains only pages that are not shared yet but tracked with KSM (without write-protected)

To reduce the number of false negative from the unstable tree lookups, a checksum is used to insert into the unstable tree only pages whose checksum didn't change recently.

**KSM deduplication process:**

1. Calculate checksum of page and compare to its last checksum
   - if match : become candidate page, continue to compare page with pages in stable trees
   - else : volatile page
2. *Every* candidate page is compared with pages in stable tree
   - if match : candidate page will be merged and shared with matched KSM page -> end
   - else : search to unstable tree
3. *Every* candidate page is compared with pages in unstable tree
   - if match : matched page will be removed from unstable tree and merged to stable tree with write-protected
   - else : candidate page is inserted into the unstable tree (as leaf node)
4. Each scan, unstable tree needs to be reconstructed to get correct node position. this must be done because of pages in unstable tree are not write-protected -> content might be modified during the scan round
5. If multiple pages with same content are detected, one of pages is selected as KSM page, and merge other duplicate pages
6. Original space of duplicate pages are reclaimed and saved

**Charts:** Can be seen through this figure below

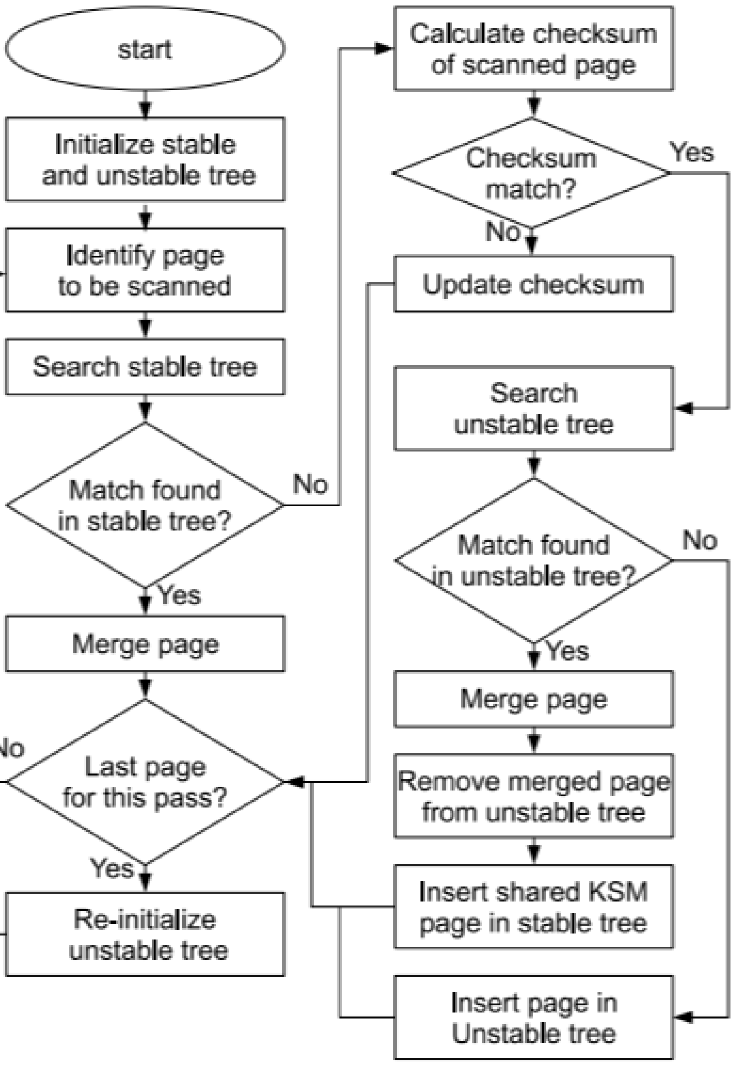

start
Initialize stable and unstable tree
Identify page to be scanned
Search stable tree
Match found in stable tree?
No
Yes
Merge page
Last page for this pass?
No
Yes
Re-initialize unstable tree
Calculate checksum of scanned page
Checksum match?
Yes
No
Update checksum
Search unstable tree
Match found in unstable tree?
No
Yes
Merge page
Remove merged page from unstable tree
Insert shared KSM page in stable tree
Insert page in Unstable tree

Figure 1: KSM Algorithm

#### 0.3.1.2.3 API

KSM is a Linux kernel thread that run independently on demand when virtual areas is registered as mergeable.

```
#include <sys/mman.h>

int madvise(void *addr, size_t length, int advice);
```

**int advice:**

- **MADV_MERGEABLE** (since Linux 2.6.32)
  Enable Kernel Samepage Merging (KSM) for the pages in the range specified by addr and length. The kernel regularly scans those areas of user memory that have been marked as mergeable, looking for pages with identical content. These are replaced by a single write-protected page (which is automatically copied if a process later wants to update the content of the page). KSM merges only private anonymous pages (see mmap(2)).

  The KSM feature is intended for applications that generate many instances of the same data (e.g., virtualization systems such as KVM). It can consume a lot of processing power; use with care. See the Linux kernel source file Documentation/admin-guide/mm/ksm.rst for more details.

  The **MADV_MERGEABLE** and **MADV_UNMERGEABLE** operations are available only if the kernel was configured with **CONFIG_KSM**.
- **MADV_UNMERGEABLE** (since Linux 2.6.32)
  Undo the effect of an earlier **MADV_MERGEABLE** operation on the specified address range; KSM unmerges whatever pages it had merged in the address range specified by addr and length.

**Caveats:**

- KSM is possible to scan all anonymous page but it should be wasteful where we cannot find equal anonymous pages.
- To keep track of the pages, KSM uses **Slab Allocation**.
  The number of slab allocation is linearly increasing with the size of registered virtual areas.

**The KSM behavior can be controlled through sysfs at /sys/kernel/mm/ksm/:**

- **run**
  - 1 for running
  - 0 for stop
  - 2 for stop ksmd and unmerge all pages currently merged, but leave mergeable areas registered for next run
- **pages_to_scan** -> how many pages to scan before ksmd go to sleep, e.g. 100
- **sleep_millisecs** -> how many milliseconds ksmd should sleep before next scan, e.g. 20
- **merge_across_nodes** -> specifies if pages from different NUMA nodes can be merged
  - 0 -> ksm merges only pages which physically reside in the memory area of same NUMA node. That brings lower latency to access of shared pages. Systems with more nodes, at significant NUMA distances, are likely to benefit from the lower latency

  - 1 (default) -> Smaller systems, which need to minimize memory usage, are likely to benefit from the greater sharing

- **use_zero_pages** -> specifies whether empty pages (i.e. allocated pages that only contain zeroes) should be treated specially
  - 0, normal behavior
  - 1, empty pages are merged with the kernel zero page(s) instead of with each other as it would happen normally. This can improve the performance on architectures with coloured zero pages, depending on the workload. Care should be taken when enabling this setting, as it can potentially degrade the performance of KSM for some workloads, for example if the checksums of pages candidate for merging match the checksum of an empty page
- **max_page_sharing** -> Maximum sharing allowed for each KSM page, to avoid high latency for virtual memory operations that involve traversal of virtual mappings that share the KSM page
- **stable_node_chains_prune_millisecs** -> specifies how frequently KSM checks the metadata of the pages that hit the deduplication limit for stale information

**The KSM effectiveness can be seen through sysfs at /sys/kernel/mm/ksm/ with this parameter:**

- **pages_shared** -> how many shared pages are being used
- **pages_sharing** -> how many more sites are sharing them i.e. how much saved
- **pages_unshared** -> how many pages unique but repeatedly checked for merging
- **pages_volatile** -> how many pages changing too fast to be placed in a tree
- **full_scans** -> how many times all mergeable areas have been scanned
- **stable_node_chains** -> the number of KSM pages that hit the **max_page_sharing** limit
- **stable_node_dups** -> number of duplicated KSM pages

#### 0.3.1.2.4 Sum up

1. KSM is to increase memory density
2. KSM is generating shared pages by merging equal pages, and in turn it is making free memory available allowing to run more virtual machines or applications on the same system, than otherwise would be possible without KSM
3. A high ratio of **pages_sharing** to **pages_shared** indicates good sharing, but a high ratio of **pages_unshared** to **pages_sharing** indicates

wasted effort. **pages_volatile** embraces several different kinds of activity, but a high proportion there would also indicate poor use of madvise **MADV_MERGEABLE**.

### 0.3.2 JVM Hotspot

#### 0.3.2.1 Oop Hierarchy

```
typedef class oopDesc*                                  oop;
typedef class   instanceOopDesc*                instanceOop;
typedef class   arrayOopDesc*                      arrayOop;
typedef class     objArrayOopDesc*              objArrayOop;
typedef class     typeArrayOopDesc*            typeArrayOop;
```

#### 0.3.2.2 Oop Memory Layout

1. narrowKlass = int, sizeof = 4
2. `const int HeapWordSize = 8;`
3. `const int LogBytesPerLong = 3;`
4. `const int BytesPerLong = 1 << LogBytesPerLong;`
5. `const int HeapWordsPerLong = BytesPerLong / HeapWordSize;`
6. Instance Oops for Object

| Type | Name | 64bit | | Compressed | |
|---|---|---|---|---|---|
| | | Offset | Size | Offset | Size |
| Header | mark *ptr | 0 | 8 | 0 | 8 |
| | klass *ptr | 8 | 8 | 8 | 4 |
| Fields | first field | 16 | n | 12 | n |
| | ...fields | 16 + n | ... | 12 + n | ... |

7. Instance Oops for Array

| Type | Name | 64bit | | Compressed | |
|---|---|---|---|---|---|
| | | Offset | Size | Offset | Size |
| Header | mark *ptr | 0 | 8 | 0 | 8 |
| | klass *ptr | 8 | 8 | 8 | 4 |
| Fields | length | 16 | 4 | 12 | 4 |
| | first field | 24 | n | 16 | n |
| | ...fields | 24 + n | ... | 16 + n | ... |

8. Klass

| Data type | Name | 64bit Offset | 64bit Size | Compressed Offset | Compressed Size |
|---|---|---|---|---|---|
| pointer | vtable *ptr | 0 | 8 | 0 | 8 |
| int | _layout_helper | 8 | 4 | 8 | 4 |

9. Array Klass

| Data type | Name | 64bit Offset | 64bit Size | Compressed Offset | Compressed Size |
|---|---|---|---|---|---|
| pointer | vtable *ptr | 0 | 8 | 0 | 8 |
| int | _layout_helper | 8 | 4 | 8 | 4 |

10. Mark Normal Object

| Name | 64bit Offset | 64bit Size | Name | Compressed Offset | Compressed Size |
|---|---|---|---|---|---|
| unused | 0 | 25 | unused | 0 | 25 |
| hash | 25 | 31 | hash | 25 | 31 |
| unused | 56 | 1 | unused | 56 | 1 |
| age | 57 | 4 | age | 57 | 4 |
| biased_lock | 61 | 1 | biased_lock | 61 | 1 |
| lock | 62 | 2 | lock | 62 | 2 |

11. Mark Biased Object

| Name | 64bit Offset | 64bit Size | Name | Compressed Offset | Compressed Size |
|---|---|---|---|---|---|
| JavaThread* | 0 | 54 | JavaThread* | 0 | 5 |
| epoch | 54 | 2 | epoch | 54 | 2 |
| unused | 56 | 1 | unused | 56 | 1 |
| age | 57 | 4 | age | 57 | 4 |
| biased_lock | 61 | 1 | biased_lock | 61 | 1 |
| lock | 62 | 2 | lock | 62 | 2 |

12. Mark CMS Promoted Object

| Name | 64bit |  | Name | Compressed |  |
|---|---|---|---|---|---|
|  | Offset | Size |  | Offset | Size |
| PromotedObject* | 0 | 61 | narrowOop | 0 | 32 |
|  |  |  | unused | 32 | 24 |
|  |  |  | cms_free | 56 | 1 |
|  |  |  | unused | 57 | 4 |
| promo_bits | 61 | 3 | promo_bits | 61 | 3 |

13. Mark CMS Free Object

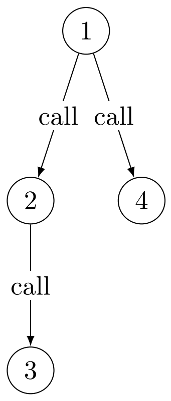
| Name | 64bit |  | Name | Compressed |  |
|---|---|---|---|---|---|
|  | Offset | Size |  | Offset | Size |
| size | 0 | 64 | unused | 0 | 21 |
|  |  |  | size | 21 | 35 |
|  |  |  | cms_free | 56 | 1 |
|  |  |  | unused | 57 | 7 |

#### 0.3.2.3 JVM Alignment

```
// Signed variants of alignment helpers.  There are two versions
↪  of each, a macro
// for use in places like enum definitions that require
↪  compile-time constant
// expressions and a function for all other places so as to get
↪  type checking.

// Using '(what) & ~align_mask(alignment)' to align 'what' down
↪  is broken when
// 'alignment' is an unsigned int and 'what' is a wider type. The
↪  & operation
// will widen the inverted mask, and not sign extend it, leading
↪  to a mask with
// zeros in the most significant bits. The use of
↪  align_mask_widened() solves
// this problem.
#define align_mask(alignment) ((alignment) - 1)
#define widen_to_type_of(what, type_carrier) (true ? (what) :
↪  (type_carrier))
#define align_mask_widened(alignment, type_carrier)
↪  widen_to_type_of(align_mask(alignment), (type_carrier))

#define align_down_(size, alignment) ((size) &
↪  ~align_mask_widened((alignment), (size)))
```

```
#define align_up_(size, alignment) (align_down_((size) +
↪  align_mask(alignment), (alignment)))

#define is_aligned_(size, alignment) (((size) &
↪  align_mask(alignment)) == 0)

// Temporary declaration until this file has been restructured.
template <typename T>
bool is_power_of_2_t(T x) {
  return (x != T(0)) && ((x & (x - 1)) == T(0));
}

// Helpers to align sizes and check for alignment

template <typename T, typename A>
inline T align_up(T size, A alignment) {
  assert(is_power_of_2_t(alignment), "must be a power of 2: "
  ↪  UINT64_FORMAT, (uint64_t)alignment);

  T ret = align_up_(size, alignment);
  assert(is_aligned_(ret, alignment), "must be aligned: "
  ↪  UINT64_FORMAT, (uint64_t)ret);

  return ret;
}

template <typename T, typename A>
inline T align_down(T size, A alignment) {
  assert(is_power_of_2_t(alignment), "must be a power of 2: "
  ↪  UINT64_FORMAT, (uint64_t)alignment);

  T ret = align_down_(size, alignment);
  assert(is_aligned_(ret, alignment), "must be aligned: "
  ↪  UINT64_FORMAT, (uint64_t)ret);

  return ret;
}

template <typename T, typename A>
inline bool is_aligned(T size, A alignment) {
  assert(is_power_of_2_t(alignment), "must be a power of 2: "
  ↪  UINT64_FORMAT, (uint64_t)alignment);

  return is_aligned_(size, alignment);
}

template <typename T>
inline T align_object_offset(T offset) {
  return align_up(offset, HeapWordsPerLong);
}
```

#### 0.3.2.4 Array Allocation



1. `void OptoRuntime::new_array_C(Klass* array_type, int len, JavaThread *thread)`
2. `typeArrayOop oopFactory::new_typeArray(BasicType type, int length, TRAPS)`
3. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
4. `objArrayOop oopFactory::new_objArray(Klass* klass, int length, TRAPS)`

#### 0.3.2.5 Array Copy Mechanism

- hotspot/share/prism/jvm.cpp

```
JVM_ENTRY(void, JVM_ArrayCopy(JNIEnv *env, jclass ignored,
↪  jobject src, jint src_pos, jobject dst, jint dst_pos, jint
↪  length))
  JVMWrapper("JVM_ArrayCopy");
  // Check if we have null pointers
  if (src == NULL || dst == NULL) {
    THROW(vmSymbols::java_lang_NullPointerException());
  }

  arrayOop s = arrayOop(JNIHandles::resolve_non_null(src));
  arrayOop d = arrayOop(JNIHandles::resolve_non_null(dst));
  assert(oopDesc::is_oop(s), "JVM_ArrayCopy: src not an oop");
  assert(oopDesc::is_oop(d), "JVM_ArrayCopy: dst not an oop");
  // Do copy
  s->klass()->copy_array(s, src_pos, d, dst_pos, length,
  ↪  thread);
JVM_END
```

- hotspot/share/oops/arrayOopDesc.hpp
  based on the subsubsection 0.3.2.1, array oopDesc can be both typeArrayOopDesc or objArrayOopDesc

```
class arrayOopDesc : public oopDesc { ... }

// so basically oopDesc have attribute method call klass()
Klass* oopDesc::klass() const {
  if (UseCompressedClassPointers) {
    return
    ↪  Klass::decode_klass_not_null(_metadata._compressed_klass);
  } else {
    return _metadata._klass;
  }
}
```

- hotspot/share/oops/typeArrayKlass.hpp

```
void TypeArrayKlass::copy_array(arrayOop s, int src_pos,
↪  arrayOop d, int dst_pos, int length, TRAPS) {
  assert(s->is_typeArray(), "must be type array");

  // Check destination type.
  if (!d->is_typeArray()) {
    ResourceMark rm(THREAD);
    stringStream ss;
    if (d->is_objArray()) {
      ss.print("arraycopy: type mismatch: can not copy %s[]
      ↪  into object array[]",
      ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()]);
    } else {
      ss.print("arraycopy: destination type %s is not an
      ↪  array", d->klass()->external_name());
    }

    THROW_MSG(vmSymbols::java_lang_ArrayStoreException(),
    ↪  ss.as_string());
  }

  if (element_type() !=
  ↪  TypeArrayKlass::cast(d->klass())->element_type()) {
    ResourceMark rm(THREAD);
    stringStream ss;
    ss.print("arraycopy: type mismatch: can not copy %s[] into
    ↪  %s[]",

    ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],

    ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()]);
    THROW_MSG(vmSymbols::java_lang_ArrayStoreException(),
    ↪  ss.as_string());
  }
```

```
// Check if all offsets and lengths are non negative.
if (src_pos < 0 || dst_pos < 0 || length < 0) {
  // Pass specific exception reason.
  ResourceMark rm(THREAD);
  stringStream ss;
  if (src_pos < 0) {
    ss.print("arraycopy: source index %d out of bounds for
    ↪  %s[%d]", src_pos,
    ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],
    ↪  s->length());
  } else if (dst_pos < 0) {
    ss.print("arraycopy: destination index %d out of bounds
    ↪  for %s[%d]", dst_pos,
    ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()],
    ↪  d->length());
  } else {
    ss.print("arraycopy: length %d is negative", length);
  }

  ↪  THROW_MSG(vmSymbols::java_lang_ArrayIndexOutOfBoundsException(),
  ↪  ss.as_string());
}

// Check if the ranges are valid
if ((((unsigned int) length + (unsigned int) src_pos) >
↪  (unsigned int) s->length()) ||
(((unsigned int) length + (unsigned int) dst_pos) > (unsigned
↪  int) d->length())) {

  // Pass specific exception reason.
  ResourceMark rm(THREAD);
  stringStream ss;
  if (((unsigned int) length + (unsigned int) src_pos) >
  ↪  (unsigned int) s->length()) {
    ss.print("arraycopy: last source index %u out of bounds
    ↪  for %s[%d]", (unsigned int) length + (unsigned int)
    ↪  src_pos,
    ↪  type2name_tab[ArrayKlass::cast(s->klass())->element_type()],
    ↪  s->length());
  } else {
    ss.print("arraycopy: last destination index %u out of
    ↪  bounds for %s[%d]", (unsigned int) length + (unsigned
    ↪  int) dst_pos,
    ↪  type2name_tab[ArrayKlass::cast(d->klass())->element_type()],
    ↪  d->length());
  }
```

```
    ↪  THROW_MSG(vmSymbols::java_lang_ArrayIndexOutOfBoundsException(),
    ↪  ss.as_string());
  }


  // Check zero copy
  if (length == 0)
    return;

  // This is an attempt to make the copy_array fast.
  int l2es = log2_element_size();
  size_t src_offset =
  ↪  arrayOopDesc::base_offset_in_bytes(element_type()) +
  ↪  ((size_t)src_pos << l2es);
  size_t dst_offset =
  ↪  arrayOopDesc::base_offset_in_bytes(element_type()) +
  ↪  ((size_t)dst_pos << l2es);
  ArrayAccess<ARRAYCOPY_ATOMIC>::arraycopy<void>(s, src_offset,
  ↪  d, dst_offset, (size_t)length << l2es);
}
```

- hotspot/share/oops/access.hpp

```
// Helper for array access.
template <DecoratorSet decorators = INTERNAL_EMPTY>
class ArrayAccess: public HeapAccess<IS_ARRAY | decorators> {
  typedef HeapAccess<IS_ARRAY | decorators> AccessT;
  public:
  template <typename T>
  static inline void arraycopy(arrayOop src_obj, size_t
  ↪  src_offset_in_bytes, arrayOop dst_obj, size_t
  ↪  dst_offset_in_bytes, size_t length) {
    AccessT::arraycopy(src_obj, src_offset_in_bytes,
    ↪  reinterpret_cast<const T*>(NULL), dst_obj,
    ↪  dst_offset_in_bytes, reinterpret_cast<T*>(NULL),
    ↪  length);
  }

  template <typename T>
  static inline void arraycopy_to_native(arrayOop src_obj,
  ↪  size_t src_offset_in_bytes, T* dst, size_t length) {
    AccessT::arraycopy(src_obj, src_offset_in_bytes,
    ↪  reinterpret_cast<const T*>(NULL), NULL, 0, dst,
    ↪  length);
  }

  template <typename T>
  static inline void arraycopy_from_native(const T* src,
  ↪  arrayOop dst_obj, size_t dst_offset_in_bytes, size_t
  ↪  length) {
    AccessT::arraycopy(NULL, 0, src, dst_obj,
    ↪  dst_offset_in_bytes, reinterpret_cast<T*>(NULL),
    ↪  length);
  }

  static inline bool oop_arraycopy(arrayOop src_obj, size_t
  ↪  src_offset_in_bytes, arrayOop dst_obj, size_t
  ↪  dst_offset_in_bytes, size_t length) {
    return AccessT::oop_arraycopy(src_obj, src_offset_in_bytes,
    ↪  reinterpret_cast<const HeapWord*>(NULL), dst_obj,
    ↪  dst_offset_in_bytes, reinterpret_cast<HeapWord*>(NULL),
    ↪  length);
  }

  template <typename T>
  static inline bool oop_arraycopy_raw(T* src, T* dst, size_t
  ↪  length) {
    return AccessT::oop_arraycopy(NULL, 0, src, NULL, 0, dst,
    ↪  length);
  }

};
```

- hotspot/share/oops/access.hpp

```
template <DecoratorSet decorators = INTERNAL_EMPTY>
class Access: public AllStatic {
  // ...


  protected:
  template <typename T>
  static inline bool oop_arraycopy(arrayOop src_obj, size_t
  ↪  src_offset_in_bytes, const T* src_raw, arrayOop dst_obj,
  ↪  size_t dst_offset_in_bytes, T* dst_raw, size_t length) {
    verify_decorators<ARRAYCOPY_DECORATOR_MASK | IN_HEAP |
    ↪  AS_DECORATOR_MASK | IS_ARRAY |
    ↪  IS_DEST_UNINITIALIZED>();
    return AccessInternal::arraycopy<decorators |
    ↪  INTERNAL_VALUE_IS_OOP>(src_obj, src_offset_in_bytes,
    ↪  src_raw, dst_obj, dst_offset_in_bytes, dst_raw,
    ↪  length);
  }

  template <typename T>
  static inline void arraycopy(arrayOop src_obj, size_t
  ↪  src_offset_in_bytes, const T* src_raw, arrayOop dst_obj,
  ↪  size_t dst_offset_in_bytes, T* dst_raw, size_t length) {
    verify_decorators<ARRAYCOPY_DECORATOR_MASK | IN_HEAP |
    ↪  AS_DECORATOR_MASK | IS_ARRAY>();
    AccessInternal::arraycopy<decorators>(src_obj,
    ↪  src_offset_in_bytes, src_raw, dst_obj,
    ↪  dst_offset_in_bytes, dst_raw, length);
  }
}
```

- hotspot/share/oops/accessBackend.inline.hpp

```
class RawAccessBarrierArrayCopy: public AllStatic {
  template<typename T> struct IsHeapWordSized: public
  ↪  IntegralConstant<bool, sizeof(T) == HeapWordSize> { };
public:
  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    HasDecorator<decorators,
    ↪  INTERNAL_VALUE_IS_OOP>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
          arrayOop dst_obj, size_t dst_offset_in_bytes, T*
          ↪  dst_raw,
          size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    // We do not check for ARRAYCOPY_ATOMIC for oops, because
    ↪  they are unconditionally always atomic.
    if (HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value) {
      AccessInternal::arraycopy_arrayof_conjoint_oops(src_raw,
      ↪  dst_raw, length);
    } else {
      typedef typename HeapOopType<decorators>::type OopType;

      ↪  AccessInternal::arraycopy_conjoint_oops(reinterpret_cast<OopType*>(src_raw),
      ↪  reinterpret_cast<OopType*>(dst_raw), length);
    }
  }

  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
          arrayOop dst_obj, size_t dst_offset_in_bytes, T*
          ↪  dst_raw,
          size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    AccessInternal::arraycopy_arrayof_conjoint(src_raw,
    ↪  dst_raw, length);
  }

  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
    ↪  IsHeapWordSized<T>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
          arrayOop dst_obj, size_t dst_offset_in_bytes, T*
          ↪  dst_raw,
          size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    // There is only a disjoint optimization for word
    ↪  granularity copying
    if (HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value) {
      AccessInternal::arraycopy_disjoint_words_atomic(src_raw,
      ↪  dst_raw, length);
    } else {
      AccessInternal::arraycopy_disjoint_words(src_raw,
      ↪  dst_raw, length);
    }
  }

  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    !(HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
    ↪  IsHeapWordSized<T>::value) &&
    !HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value &&
    !HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
          arrayOop dst_obj, size_t dst_offset_in_bytes, T*
          ↪  dst_raw,
          size_t length) {
```

```
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    AccessInternal::arraycopy_conjoint(src_raw, dst_raw,
    ↪  length);
  }

  template <DecoratorSet decorators, typename T>
  static inline typename EnableIf<
    !HasDecorator<decorators, INTERNAL_VALUE_IS_OOP>::value &&
    !(HasDecorator<decorators, ARRAYCOPY_DISJOINT>::value &&
    ↪  IsHeapWordSized<T>::value) &&
    !HasDecorator<decorators, ARRAYCOPY_ARRAYOF>::value &&
    HasDecorator<decorators, ARRAYCOPY_ATOMIC>::value>::type
  arraycopy(arrayOop src_obj, size_t src_offset_in_bytes, T*
  ↪  src_raw,
            arrayOop dst_obj, size_t dst_offset_in_bytes, T*
            ↪  dst_raw,
            size_t length) {
    src_raw = arrayOopDesc::obj_offset_to_raw(src_obj,
    ↪  src_offset_in_bytes, src_raw);
    dst_raw = arrayOopDesc::obj_offset_to_raw(dst_obj,
    ↪  dst_offset_in_bytes, dst_raw);

    AccessInternal::arraycopy_conjoint_atomic(src_raw, dst_raw,
    ↪  length);
  }
};
```

- hotspot/share/oops/accessBackend.cpp

```
// These forward copying calls to Copy without exposing the
↪  Copy type in headers unnecessarily

void arraycopy_arrayof_conjoint_oops(void* src, void* dst,
↪  size_t length) {
  Copy::arrayof_conjoint_oops(reinterpret_cast<HeapWord*>(src),
  ↪  reinterpret_cast<HeapWord*>(dst), length);
}

void arraycopy_conjoint_oops(oop* src, oop* dst, size_t length)
↪  {
  Copy::conjoint_oops_atomic(src, dst, length);
}

void arraycopy_conjoint_oops(narrowOop* src, narrowOop* dst,
↪  size_t length) {
  Copy::conjoint_oops_atomic(src, dst, length);
```

```
}

void arraycopy_disjoint_words(void* src, void* dst, size_t
↪  length) {
  Copy::disjoint_words(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

void arraycopy_disjoint_words_atomic(void* src, void* dst,
↪  size_t length) {
  Copy::disjoint_words_atomic(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_conjoint<jboolean>(jboolean* src, jboolean* dst,
↪  size_t length) {
  Copy::conjoint_jbytes(reinterpret_cast<jbyte*>(src),
   ↪  reinterpret_cast<jbyte*>(dst), length);
}

template<>
void arraycopy_conjoint<jbyte>(jbyte* src, jbyte* dst, size_t
↪  length) {
  Copy::conjoint_jbytes(src, dst, length);
}

template<>
void arraycopy_conjoint<jchar>(jchar* src, jchar* dst, size_t
↪  length) {
  Copy::conjoint_jshorts_atomic(reinterpret_cast<jshort*>(src),
   ↪  reinterpret_cast<jshort*>(dst), length);
}

template<>
void arraycopy_conjoint<jshort>(jshort* src, jshort* dst,
↪  size_t length) {
  Copy::conjoint_jshorts_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint<jint>(jint* src, jint* dst, size_t
↪  length) {
  Copy::conjoint_jints_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint<jfloat>(jfloat* src, jfloat* dst,
↪  size_t length) {
  Copy::conjoint_jints_atomic(reinterpret_cast<jint*>(src),
   ↪  reinterpret_cast<jint*>(dst), length);
}

template<>
void arraycopy_conjoint<jlong>(jlong* src, jlong* dst, size_t
↪  length) {
  Copy::conjoint_jlongs_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint<jdouble>(jdouble* src, jdouble* dst,
↪  size_t length) {
  Copy::conjoint_jlongs_atomic(reinterpret_cast<jlong*>(src),
   ↪  reinterpret_cast<jlong*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jbyte>(jbyte* src, jbyte* dst,
↪  size_t length) {

   ↪  Copy::arrayof_conjoint_jbytes(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jshort>(jshort* src, jshort*
↪  dst, size_t length) {

   ↪  Copy::arrayof_conjoint_jshorts(reinterpret_cast<HeapWord*>(src),
  reinterpret_cast<HeapWord*>(dst),
  length);
}

template<>
void arraycopy_arrayof_conjoint<jint>(jint* src, jint* dst,
↪  size_t length) {

   ↪  Copy::arrayof_conjoint_jints(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}

template<>
void arraycopy_arrayof_conjoint<jlong>(jlong* src, jlong* dst,
↪  size_t length) {

   ↪  Copy::arrayof_conjoint_jlongs(reinterpret_cast<HeapWord*>(src),
   ↪  reinterpret_cast<HeapWord*>(dst), length);
}
```

```
template<>
void arraycopy_conjoint<void>(void* src, void* dst, size_t
↪  length) {
  Copy::conjoint_jbytes(reinterpret_cast<jbyte*>(src),
  ↪  reinterpret_cast<jbyte*>(dst), length);
}

template<>
void arraycopy_conjoint_atomic<jbyte>(jbyte* src, jbyte* dst,
↪  size_t length) {
  Copy::conjoint_jbytes_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jshort>(jshort* src, jshort*
↪  dst, size_t length) {
  Copy::conjoint_jshorts_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jint>(jint* src, jint* dst,
↪  size_t length) {
  Copy::conjoint_jints_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<jlong>(jlong* src, jlong* dst,
↪  size_t length) {
  Copy::conjoint_jlongs_atomic(src, dst, length);
}

template<>
void arraycopy_conjoint_atomic<void>(void* src, void* dst,
↪  size_t length) {
  Copy::conjoint_memory_atomic(src, dst, length);
}
```

- hotspot/share/utilities/copy.hpp

```
class Copy : AllStatic {
 public:
  // Block copy methods have four attributes.  We don't define
  ↪  all possibilities.
  //   alignment: aligned to BytesPerLong
  //   arrayof:   arraycopy operation with both operands
  ↪  aligned on the same
  //              boundary as the first element of an array of
  ↪  the copy unit.
  //              This is currently a HeapWord boundary on all
  ↪  platforms, except
```

```
//               for long and double arrays, which are aligned
↪  on an 8-byte
//               boundary on all platforms.
//               arraycopy operations are implicitly atomic on
↪  each array element.
//   overlap:    disjoint or conjoint.
//   copy unit: bytes or words (i.e., HeapWords) or oops
↪  (i.e., pointers).
//   atomicity: atomic or non-atomic on the copy unit.
//
// Names are constructed thusly:
//
//     [ 'aligned_' | 'arrayof_' ]
//     ('conjoint_' | 'disjoint_')
//     ('words' | 'bytes' | 'jshorts' | 'jints' | 'jlongs' |
↪  'oops')
//     [ '_atomic' ]
//
// Except in the arrayof case, whatever the alignment is, we
↪  assume we can copy
// whole alignment units.  E.g., if BytesPerLong is 2x word
↪  alignment, an odd
// count may copy an extra word.  In the arrayof case, we are
↪  allowed to copy
// only the number of copy units specified.
//
// All callees check count for 0.
//

// HeapWords

// Word-aligned words,    conjoint, not atomic on each word
static void conjoint_words(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  assert_params_ok(from, to, HeapWordSize);
  pd_conjoint_words(from, to, count);
}

// Word-aligned words,    disjoint, not atomic on each word
static void disjoint_words(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  assert_params_ok(from, to, HeapWordSize);
  assert_disjoint(from, to, count);
  pd_disjoint_words(from, to, count);
}

// Word-aligned words,    disjoint, atomic on each word
static void disjoint_words_atomic(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, HeapWordSize);
  assert_disjoint(from, to, count);
  pd_disjoint_words_atomic(from, to, count);
}

// Object-aligned words,  conjoint, not atomic on each word
static void aligned_conjoint_words(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_aligned(from, to);
  pd_aligned_conjoint_words(from, to, count);
}

// Object-aligned words,  disjoint, not atomic on each word
static void aligned_disjoint_words(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_aligned(from, to);
  assert_disjoint(from, to, count);
  pd_aligned_disjoint_words(from, to, count);
}

// bytes, jshorts, jints, jlongs, oops

// bytes,                 conjoint, not atomic on each byte
↪  (not that it matters)
static void conjoint_jbytes(const void* from, void* to,
↪  size_t count) {
  pd_conjoint_bytes(from, to, count);
}

// bytes,                 conjoint, atomic on each byte (not
↪  that it matters)
static void conjoint_jbytes_atomic(const void* from, void*
↪  to, size_t count) {
  pd_conjoint_bytes(from, to, count);
}

// jshorts,               conjoint, atomic on each jshort
static void conjoint_jshorts_atomic(const jshort* from,
↪  jshort* to, size_t count) {
  assert_params_ok(from, to, BytesPerShort);
  pd_conjoint_jshorts_atomic(from, to, count);
}

// jints,                 conjoint, atomic on each jint
static void conjoint_jints_atomic(const jint* from, jint* to,
↪  size_t count) {
  assert_params_ok(from, to, BytesPerInt);
  pd_conjoint_jints_atomic(from, to, count);
}

// jlongs,                conjoint, atomic on each jlong
```

```
static void conjoint_jlongs_atomic(const jlong* from, jlong*
↪  to, size_t count) {
  assert_params_ok(from, to, BytesPerLong);
  pd_conjoint_jlongs_atomic(from, to, count);
}

// oops,                  conjoint, atomic on each oop
static void conjoint_oops_atomic(const oop* from, oop* to,
↪  size_t count) {
  assert_params_ok(from, to, BytesPerHeapOop);
  pd_conjoint_oops_atomic(from, to, count);
}

// overloaded for UseCompressedOops
static void conjoint_oops_atomic(const narrowOop* from,
↪  narrowOop* to, size_t count) {
  assert(sizeof(narrowOop) == sizeof(jint), "this cast is
  ↪  wrong");
  assert_params_ok(from, to, BytesPerInt);
  pd_conjoint_jints_atomic((const jint*)from, (jint*)to,
  ↪  count);
}

// Copy a span of memory.  If the span is an integral number
↪  of aligned
// longs, words, or ints, copy those units atomically.
// The largest atomic transfer unit is 8 bytes, or the
↪  largest power
// of two which divides all of from, to, and size, whichever
↪  is smaller.
static void conjoint_memory_atomic(const void* from, void*
↪  to, size_t size);

// bytes,                 conjoint array, atomic on each byte
↪  (not that it matters)
static void arrayof_conjoint_jbytes(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  pd_arrayof_conjoint_bytes(from, to, count);
}

// jshorts,               conjoint array, atomic on each
↪  jshort
static void arrayof_conjoint_jshorts(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerShort);
  pd_arrayof_conjoint_jshorts(from, to, count);
}

// jints,                 conjoint array, atomic on each jint
```

```
static void arrayof_conjoint_jints(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerInt);
  pd_arrayof_conjoint_jints(from, to, count);
}

// jlongs,                 conjoint array, atomic on each
↪  jlong
static void arrayof_conjoint_jlongs(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerLong);
  pd_arrayof_conjoint_jlongs(from, to, count);
}

// oops,                   conjoint array, atomic on each oop
static void arrayof_conjoint_oops(const HeapWord* from,
↪  HeapWord* to, size_t count) {
  assert_params_ok(from, to, BytesPerHeapOop);
  pd_arrayof_conjoint_oops(from, to, count);
}

// Known overlap methods

// Copy word-aligned words from higher to lower addresses,
↪  not atomic on each word
inline static void conjoint_words_to_lower(const HeapWord*
↪  from, HeapWord* to, size_t byte_count) {
  // byte_count is in bytes to check its alignment
  assert_params_ok(from, to, HeapWordSize);
  assert_byte_count_ok(byte_count, HeapWordSize);

  size_t count = align_up(byte_count, HeapWordSize) >>
  ↪  LogHeapWordSize;
  assert(to <= from || from + count <= to, "do not overwrite
  ↪  source data");

  while (count-- > 0) {
    *to++ = *from++;
  }
}

// Copy word-aligned words from lower to higher addresses,
↪  not atomic on each word
inline static void conjoint_words_to_higher(const HeapWord*
↪  from, HeapWord* to, size_t byte_count) {
  // byte_count is in bytes to check its alignment
  assert_params_ok(from, to, HeapWordSize);
  assert_byte_count_ok(byte_count, HeapWordSize);
  size_t count = align_up(byte_count, HeapWordSize) >>
  ↪  LogHeapWordSize;
  assert(from <= to || to + count <= from, "do not overwrite
  ↪  source data");

  from += count - 1;
  to   += count - 1;
  while (count-- > 0) {
    *to-- = *from--;
  }
}

/**
 * Copy elements
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
static void conjoint_copy(const void* src, void* dst, size_t
↪  byte_count, size_t elem_size);

/**
 * Copy and *unconditionally* byte swap elements
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
static void conjoint_swap(const void* src, void* dst, size_t
↪  byte_count, size_t elem_size);

/**
 * Copy and byte swap elements from the specified endian to
↪  the native (cpu) endian if needed (if they differ)
 *
 * @param src address of source
 * @param dst address of destination
 * @param byte_count number of bytes to copy
 * @param elem_size size of the elements to copy-swap
 */
template <Endian::Order endian>
static void conjoint_swap_if_needed(const void* src, void*
↪  dst, size_t byte_count, size_t elem_size) {
  if (Endian::NATIVE != endian) {
    conjoint_swap(src, dst, byte_count, elem_size);
  } else {
    conjoint_copy(src, dst, byte_count, elem_size);
    }
  }

  // Fill methods

  // Fill word-aligned words, not atomic on each word
  // set_words
  static void fill_to_words(HeapWord* to, size_t count, juint
   ↪  value = 0) {
    assert_params_ok(to, HeapWordSize);
    pd_fill_to_words(to, count, value);
  }

  static void fill_to_aligned_words(HeapWord* to, size_t count,
   ↪  juint value = 0) {
    assert_params_aligned(to);
    pd_fill_to_aligned_words(to, count, value);
  }

  // Fill bytes
  static void fill_to_bytes(void* to, size_t count, jubyte
   ↪  value = 0) {
    pd_fill_to_bytes(to, count, value);
  }

  // Fill a span of memory.  If the span is an integral number
   ↪  of aligned
  // longs, words, or ints, store to those units atomically.
  // The largest atomic transfer unit is 8 bytes, or the
   ↪  largest power
  // of two which divides both to and size, whichever is
   ↪  smaller.
  static void fill_to_memory_atomic(void* to, size_t size,
   ↪  jubyte value = 0);

  // Zero-fill methods

  // Zero word-aligned words, not atomic on each word
  static void zero_to_words(HeapWord* to, size_t count) {
    assert_params_ok(to, HeapWordSize);
    pd_zero_to_words(to, count);
  }

  // Zero bytes
  static void zero_to_bytes(void* to, size_t count) {
    pd_zero_to_bytes(to, count);
  }

 private:
```

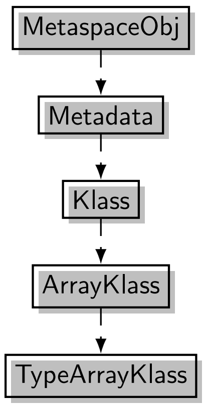
```
static bool params_disjoint(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  if (from < to) {
    return pointer_delta(to, from) >= count;
  }
  return pointer_delta(from, to) >= count;
}

// These methods raise a fatal if they detect a problem.

static void assert_disjoint(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  assert(params_disjoint(from, to, count), "source and dest
   ↪  overlap");
}

static void assert_params_ok(const void* from, void* to,
↪  intptr_t alignment) {
  assert(is_aligned(from, alignment), "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(from));
  assert(is_aligned(to, alignment),   "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(to));
}

static void assert_params_ok(HeapWord* to, intptr_t
↪  alignment) {
  assert(is_aligned(to, alignment), "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(to));
}

static void assert_params_aligned(const HeapWord* from,
↪  HeapWord* to) {
  assert(is_aligned(from, BytesPerLong), "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(from));
  assert(is_aligned(to, BytesPerLong),   "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(to));
}

static void assert_params_aligned(HeapWord* to) {
  assert(is_aligned(to, BytesPerLong), "must be aligned: "
   ↪  INTPTR_FORMAT, p2i(to));
}

static void assert_byte_count_ok(size_t byte_count, size_t
↪  unit_size) {
  assert(is_aligned(byte_count, unit_size), "byte count must
   ↪  be aligned");
}

// Platform dependent implementations of the above methods.
```

```
#include CPU_HEADER(copy)

};
```

- hotspot/cpu/ppc/copy_ppc.hpp

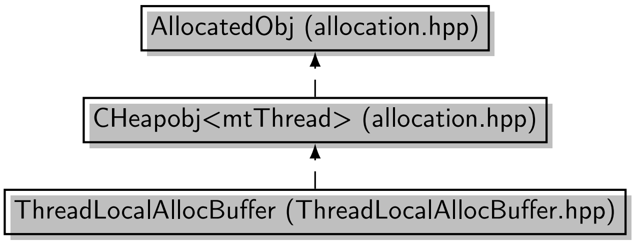
```
// Inline functions for memory copy and fill.

static void pd_conjoint_words(const HeapWord* from, HeapWord*
↪  to, size_t count) {
  (void)memmove(to, from, count * HeapWordSize);
}

// Template for atomic, element-wise copy.
template <class T>
static void copy_conjoint_atomic(const T* from, T* to, size_t
↪  count) {
  if (from > to) {
    while (count-- > 0) {
      // Copy forwards
      *to++ = *from++;
    }
  } else {
    from += count - 1;
    to   += count - 1;
    while (count-- > 0) {
      // Copy backwards
      *to-- = *from--;
    }
  }
}

// ...
```

#### 0.3.2.6 Type Array Inheritance

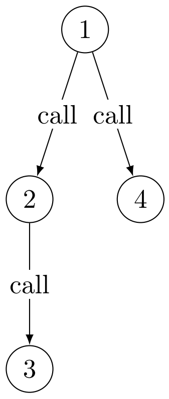


#### 0.3.2.7 TLAB

##### 0.3.2.7.1 Inheritance


AllocatedObj (allocation.hpp)
CHeapobj<mtThread> (allocation.hpp)
ThreadLocalAllocBuffer (ThreadLocalAllocBuffer.hpp)

##### 0.3.2.7.2 Initialization



1. `void OptoRuntime::new_array_C(Klass* array_type, int len, JavaThread *thread)`
2. `typeArrayOop oopFactory::new_typeArray(BasicType type, int length, TRAPS)`
3. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
4. `TypeArrayKlass* TypeArrayKlass::allocate(ClassLoaderData* loader_data, BasicType type, Symbol* name, TRAPS)`
5. `objArrayOop oopFactory::new_objArray(Klass* klass, int length, TRAPS)`

##### 0.3.2.7.3 Notes

The virtual machine must never call one of the implicitly declared global allocation or deletion functions. (Such calls may result in link-time or run-time errors.) For convenience and documentation of intended use, classes in the virtual machine may be derived from one of the following allocation classes, some of which define allocation and deletion functions. Note: `std::malloc` and `std::free` should never called directly.

For objects allocated in the resource area (see resourceArea.hpp). - ResourceObj

For objects allocated in the C-heap (managed by: free & malloc and tracked with NMT) - CHeapObj

For objects allocated on the stack. - StackObj

For classes used as name spaces. - AllStatic

For classes in Metaspace (class data) - MetaspaceObj

The printable subclasses are used for debugging and define virtual member functions for printing. Classes that avoid allocating the vtbl entries in the objects should therefore not be the printable subclasses.

The following macros and function should be used to allocate memory directly in the resource area or in the C-heap, The _OBJ variants of the NEW/FREE_C_HEAP macros are used for alloc/dealloc simple objects which are not inherited from CHeapObj, note constructor and destructor are not called. The preferable way to allocate objects is using the new operator.

WARNING: The array variant must only be used for a homogenous array where all objects are of the exact type specified. If subtypes are stored in the array then must pay attention to calling destructors at needed.

- `NEW_RESOURCE_ARRAY(type, size)`
- `NEW_RESOURCE_OBJ(type)`
- `NEW_C_HEAP_ARRAY(type, size)`
- `NEW_C_HEAP_OBJ(type, memflags)`
- `FREE_C_HEAP_ARRAY(type, old)`
- `FREE_C_HEAP_OBJ(objname, type, memflags)`
- `char* AllocateHeap(size_t size, const char* name);`
- `void FreeHeap(void* p);`

In non product mode we introduce a super class for all allocation classes that supports printing. We avoid the superclass in product mode to save space.

## 0.4 Notes

### 0.4.1 Tuesday, 08 October 2019

#### 0.4.1.1 Kafka Bug Study

- Distributed Concurrency Bugs
  1. [KAFKA-1183] - DefaultEventHandler causes unbalanced distribution of messages across partitions
  2. [KAFKA-1154] - replicas may not have consistent data after becoming follower
  3. [KAFKA-1124] - Sending to a new topic (with auto.create.topics.enable) returns ERROR
  4. [KAFKA-1097] - Race condition while reassigning low throughput partition leads to incorrect ISR information in zookeeper
- Scalability Bugs
  1. [KAFKA-1228] - Socket Leak on ReplicaFetcherThread

#### 0.4.1.2 JVM Deduplication

- Trying to implement the madvise call inside the TLAB allocation (ThreadLocalAllocBuffer::allocate)
  Only saving 44kb for two JVM process and array 139 with its elements 42

### 0.4.2 Friday, 04 October 2019

#### 0.4.2.1 Allocation Classes

1. Allocation, consists of:

1.1. AllocatedObj (Abstract Class)

1.1.1. CHeapObj (Abstract Class)

1.1.1.1. CollectedHeap

1.1.1.1.1. GenCollectedHeap

1.1.1.1.1.1. SerialHeap

1.1.1.1.1.2. CMSHeap

1.1.1.1.2. G1CollectedHeap

1.1.1.1.3. ParallelScavengeHeap

1.1.1.1.4. ZCollectedHeap

1.1.2. StackObj (Abstract Class)

1.1.2.1. MemAllocator

1.1.2.1.1. ObjAllocator

1.1.2.1.2. ObjArrayAllocator

1.1.2.1.3. ClassAllocator

1.1.3. ResourceObj (Abstract Class)

1.2. MetaspaceObj (Abstract Class)

1.3. AllStatic (Abstract Class)

1.3.1. ArrayAllocator

1.3.2. MmapArrayAllocator

1.4. MallocArrayAllocator

#### 0.4.2.2 Notes

- All of the Klass is being instantiated in Metaspace
- EpsilonGC is not creating its own heap

### 0.4.3 Saturday, 19 October 2019

#### 0.4.3.1 JVM Deduplication

- Appending this code into the ‘accessBackend.inline.hpp‘ especially in function ‘arraycopy‘ but for atomic one

```
// @rayandrew
// added this to add execute ksm

if (os::can_execute_ksm() && check_if_tescase_array(length)) {
  // tty->print_cr("Pointer actual size " SIZE_FORMAT "Type
  ↪  %s", sizeof(*dst_raw), demangle(typeid(dst_raw).name()));
  long page_size = sysconf(_SC_PAGE_SIZE);
  tty->print_cr("Is aligned : %d",
  ↪  is_aligned(dst_obj->base(T_BYTE), page_size));
  tty->print_cr("Raw pointer : " PTR_FORMAT ", DST_OBJ "
  ↪  PTR_FORMAT, dst_raw, dst_obj->base(T_BYTE));
  // tty->print_cr("Pointer actual size " SIZE_FORMAT "Type
  ↪  %s", sizeof(*dst_raw), demangle(typeid(dst_raw).name()));
  tty->print_cr("Value : %s", dst_obj->print_value_string());
  tty->print_cr("Raw pointer : " PTR_FORMAT ", DST_OBJ "
  ↪  PTR_FORMAT, dst_raw, dst_obj->base(T_BYTE));
  tty->print_cr("Length : %d", dst_obj->length());
  tty->print_cr("Sizeof arrayOopDesc : %d",
  ↪  sizeof(arrayOopDesc));
  // tty->print_cr("Content %d", &dst_raw[0]);
  tty->print_cr("Element offset %d",
  ↪  (size_t)typeArrayOopDesc::element_offset<jbyte>(2));
  tty->print_cr("New Content 0 ptr " PTR_FORMAT " value %d",
  ↪  static_cast<typeArrayOop>(dst_obj)->byte_at_addr(0),
  ↪  (int)*static_cast<typeArrayOop>(dst_obj)->byte_at_addr(0));
  tty->print_cr("New Content 1 ptr " PTR_FORMAT " value %d",
  ↪  static_cast<typeArrayOop>(dst_obj)->byte_at_addr(1),
  ↪  (int)*static_cast<typeArrayOop>(dst_obj)->byte_at_addr(1));
  tty->print_cr("Content 0 %d", *(int*)((intptr_t)dst_obj));
  tty->print_cr("Content 1 %d", *(int*)((intptr_t)dst_obj +
  ↪  17));
  tty->print_cr("Content %d %d", dst_obj->length(),
  ↪  *(int*)((intptr_t)dst_obj + dst_obj->length()));

  ↪  os::mark_for_mergeable_debug(static_cast<typeArrayOop>(dst_obj)->byte_at_addr(0),
  ↪  length, "RawAccessBarrierArrayCopy::arraycopy [5]");
  // os::mark_for_mergeable_debug(dst_obj->base(T_BYTE),
  ↪  length, "RawAccessBarrierArrayCopy::arraycopy [5]");
}
```

- And the result is as follow :

```
KSM features is enabled
[0.010s][info][gc] Resizeable heap; starting at 2011M, max:
↪  30718M, step: 128M
```

```
[0.010s][info][gc] Using TLAB allocation; max: 4096K
[0.010s][info][gc] Elastic TLABs enabled; elasticity: 1.10x
[0.010s][info][gc] Elastic TLABs decay enabled; decay time:
↪  1000ms
[0.010s][info][gc] Using Epsilon
Phase 1
Phase 2, Copy array
Is aligned : 0
Raw pointer : 0x0000000080697148, DST_OBJ 0x0000000080697148
Value : [B{0x0000000080697138}
Raw pointer : 0x0000000080697148, DST_OBJ 0x0000000080697148
Length : 4096000
Sizeof arrayOopDesc : 16
Element offset 18
New Content 0 ptr 0x0000000080697148 value 1
New Content 1 ptr 0x0000000080697149 value 1
Content 0 1
Content 1 16843009
Content 4096000 16843009
[RawAccessBarrierArrayCopy::arraycopy [5]] Mark page address :
↪  0x0000000080697148 bytes : 4096000 to be mergeable
OpenJDK 64-Bit Server VM warning: Failed to memory pages using
↪  madvise, req_addr: 0x0000000080697148 bytes: 4096000 (errno
↪  = 22).
Phase 3
src == dst : true
^C[5.057s][info][gc] Heap: 30718M reserved, 2011M (6.55%)
↪  committed, 9170K (0.03%) used
```

- The result that can we take  Array is not page aligned : this is actually mandatory in MADV MERGEABLE

## 0.5 Resources

### 0.5.1 Kernel Samepage Merging

1. Documentation
   (a) Madvise
   (b) KSM
   (c) Metaspace Architecture

# Bibliography

[1] S. Rachamalla, D. Mishra, and P. Kulkarni, “Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems,” *20th Annual International Conference on High Performance Computing*, 2013.

[2] A. Arcangeli, I. Eidus, and C. Wright, “Increasing memory density by using ksm,” 01 2009.