Improving Chapel and Array Memory Management

Michael Ferguson, Vassily Litvinov, Brad Chamberlain

June 2, 2017

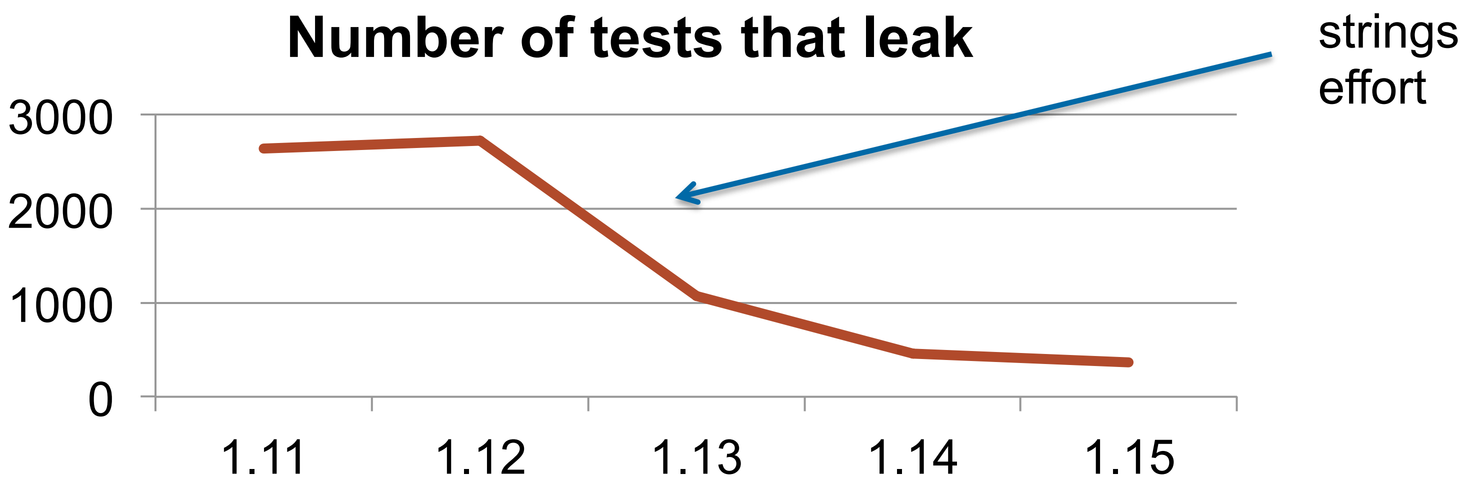
Safe Harbor Statement

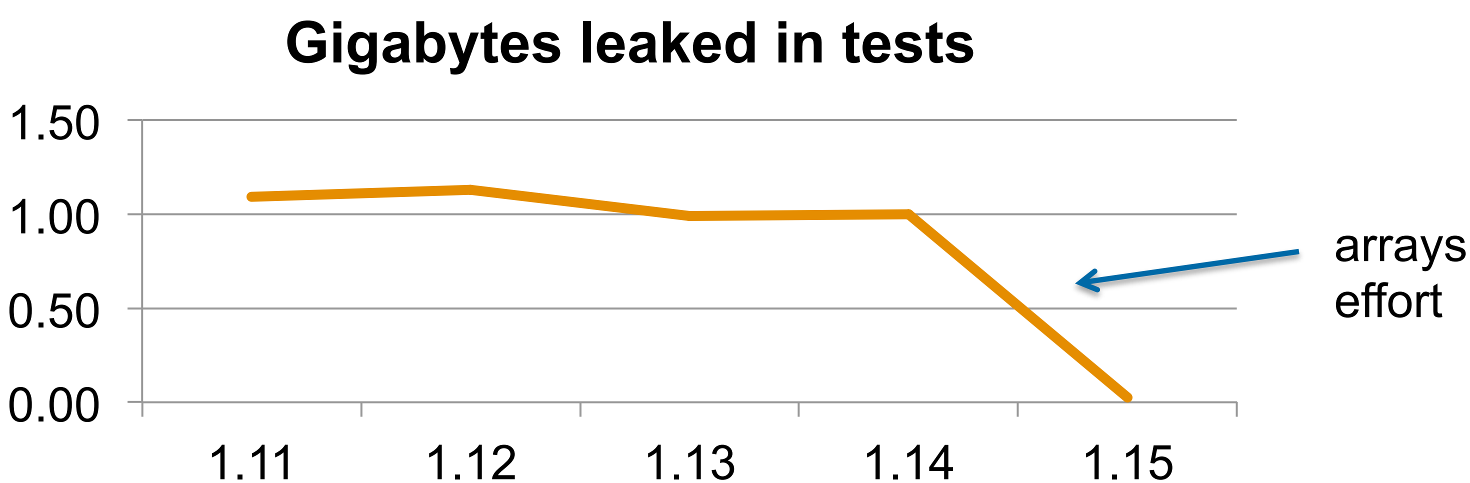
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Big Picture

- **Chapel historically has leaked memory**
- **Made big progress over past several releases**
  - for 1.13, significantly improved strings
  - for 1.15, significantly improved arrays
- **Chapel 1.15 is pretty good!**

Improvements in leaks

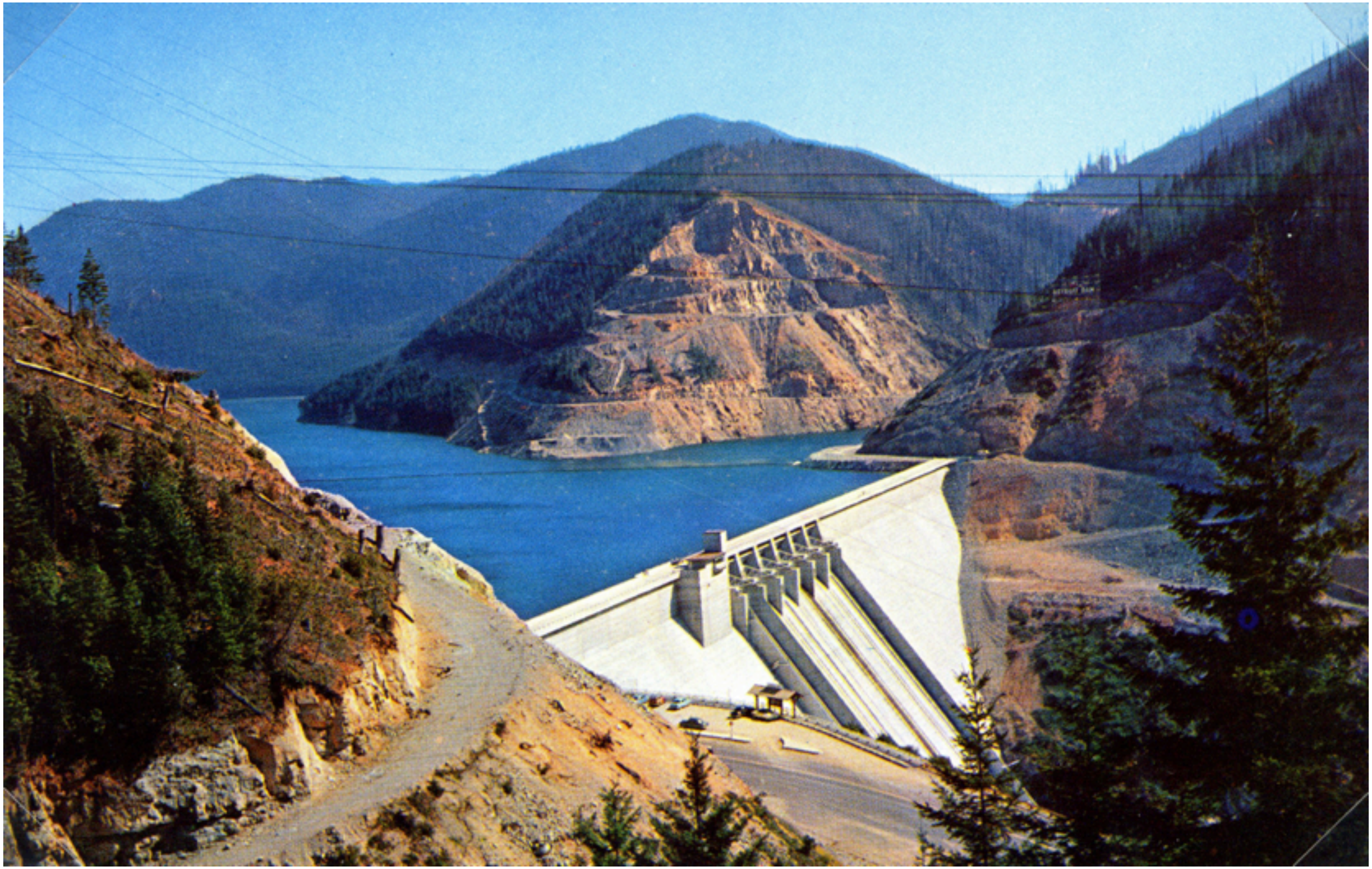




Leaks were always there

Internet Archive Book Images

Fixing leaks required significant effort

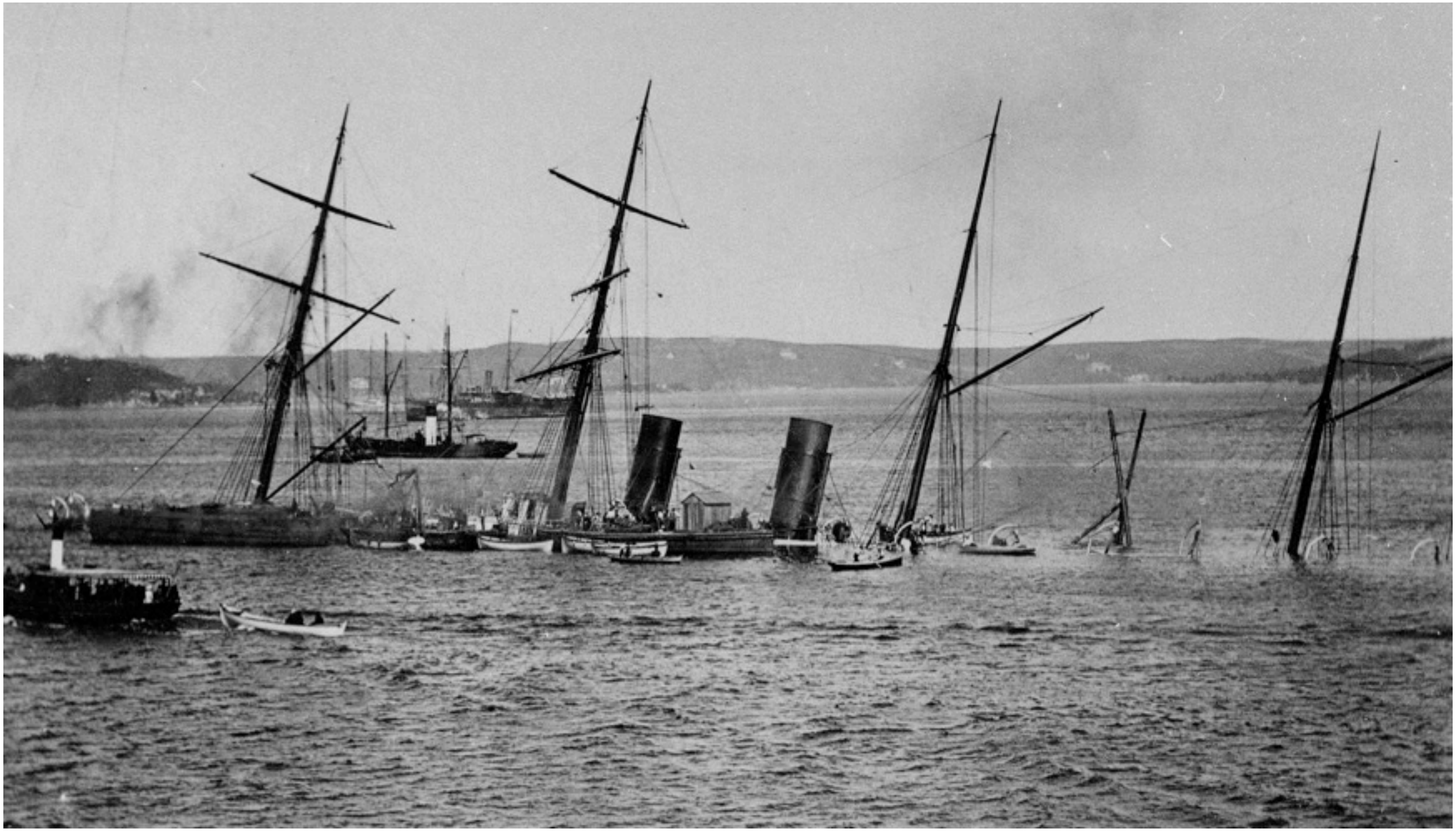


OSU Special Collections & Archives

Rest of the Talk

- What was going wrong with array memory management?
- How did we fix it?
- A challenge we discovered along the way
- Performance Impact

What was going wrong with arrays?



Australian National Maritime Museim

What was going wrong with arrays?

- **Array memory management used reference counting**
  - to enable original language semantics
- **Largest source of memory leaks in Chapel 1.14**
  - distributed arrays accounted for most leaked data
- **Implementation overheads hurt performance**
  - Benchmarks spent significant time handling array reference counting
    - Supported a ‘noRefCount’ setting to measure/reduce impact
    - Sometimes helped dramatically, but guaranteed arrays would be leaked
  - Array memory management overheads could be surprising:

```
var size = A.domain.size; // changed reference counts!
```

Where did the leaks come from?

National Library of Norway

Where did the leaks come from?

- **Array memory management strategy had two goals:**
  1. **Keep arrays alive past lexical scope**
     - when an array slice/view outlives the original array
     - when arrays are used in ‘begin’ statements
  2. **Minimize array copies**

- **But…**
  - Implementation erred on keeping arrays alive to the point of leaking
  - Reference counting approach was expensive and overly conservative
  - Language definition did not clearly specify array return behavior

How did we fix it?

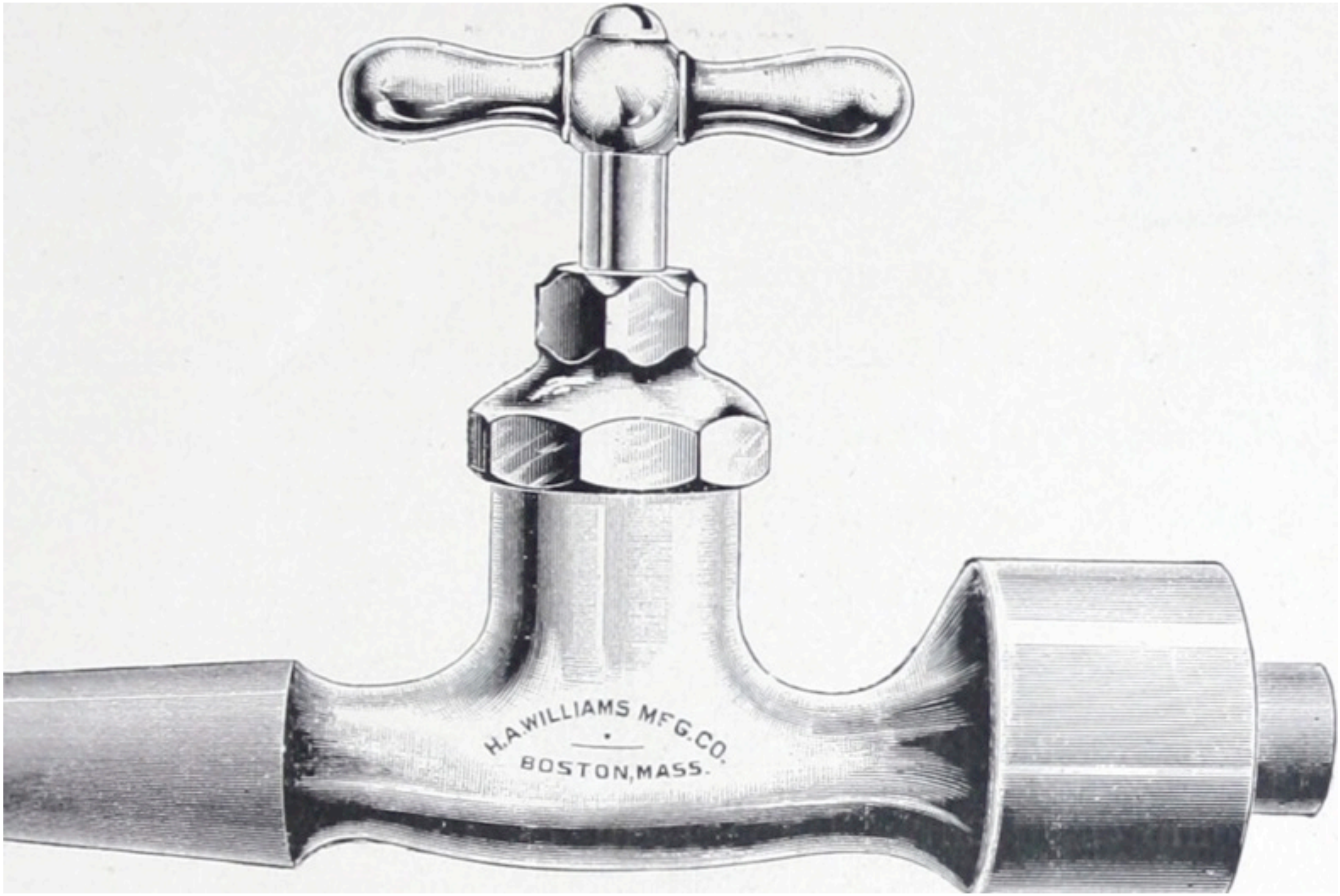
Library of Congress

How did we fix it?

- **Removed array reference counting**
  - no longer necessary
- **Adjusted language to simplify the task**
  - arrays return by value by default
  - arrays no longer outlive lexical scope

… and worked hard on the implementation ...

Returning Arrays



Internet Archive Book Images

Returning Arrays

- **Arrays have historically returned by 'ref' by default**
  - this design interfered with array memory management improvements
- **In 1.15 they return by value by default**
  - to make them more similar to records
  - to simplify the language and its implementation

```
var A: [1..4] int;
proc f() {
  return A; // new in 1.15: return by value
}
ref B = f();
B = 1;
writeln(A);
// printed 1 1 1 1 historically
// prints 0 0 0 0 after this work
```

Returning Arrays

- **When old behavior is desired, use 'ref' return intent**
  - should result in behavior that's backwards-compatible with 1.14

```
var A: [1..4] int;
proc f() ref { // explicit ref return
  return A;
}
ref B = f();
B = 1;
writeln(A);
// prints 1 1 1 1
```

Scoping

Fylkesarkiv - Skjervefossen

Scoping

- **Arrays are now destroyed when they go out of scope**
  - ‘begin’ statements and array slices no longer affect array lifetime

```
proc badBegin() {
  var A: [1..10000] int;
  begin {
    A += 1;
  }
  // User error: A destroyed here at function end, but the begin could still be using it!
}
```

- **Builds upon earlier work**
  - 1.12 no longer extends lifetime for general variables used in 'begin'
  - 1.15 first to rely on this property for arrays

A surprise along the way

State Library of New South Wales

A surprise along the way

We notice a language problem while studying the performance impact of array changes on miniMD

Key background:

1. **Historically, default argument intent for arrays was 'ref'**
   - designed as a convenience for programmers
   - avoids surprising programmers used to modifying array formals
2. **Sparse arrays use return intent overloads**
   - to have different behavior on element read and write
   - writing “zero” values of a sparse array is an error
   - reading the “zero” values of a sparse array is fine

Unintented Consequences

- **Consider this example with a sparse array of int:**

```
var dense = {1..10};
var sps: sparse subdomain(dense); // domain is initially empty

var A: [sps] int; // sparse array of integers, only storing "zero" value

writeln(A[3]); // outputs 0, the "zero" value
```

Unintented Consequences

- **What about a sparse array of arrays?**

```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int; // sparse array of arrays

writeln(A[3]);
```

Unintented Consequences

- **What about a sparse array of arrays?**

```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int;

writeln(A[3]); // surprising: halts
               // attempting to assign a 'zero' value in a sparse array: (3)
```

Unintented Consequences

- What about a sparse array of arrays?

```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int;

writeln(A[3]); // surprising: halts
               // attempting to assign a 'zero' value in a sparse array: (3)
```

- What's happening in this example?
  - writeln() takes its arguments by default intent
  - default intent for an array is 'ref'

  ➔ writeln() appears to the array implementation to set its argument
    - setting a sparse array's “zero” values via indexing is not permitted

Fixing It

Library of Congress

Fixing It

- **Changed the default intent for arrays…**
  …to 'ref' if the formal argument is modified in the function body
  …to 'const ref' if not

```
proc setElementOne(x) {
  // x is modified, so x has 'ref' intent
  x[1] = 1;
}

var A:[1..10] int;
setElementOne(A);
```

```
proc getElementOne(y) {
  // y is not modified,
  // so y has 'const ref' intent
  var tmp = y[1];
}
var B:[1..10] int;
getElementOne(B);
```

Fixing It

- **Motivating cases now work as you'd expect:**

```
var dense = {1..10};
var sps: sparse subdomain(dense);

var A: [sps] [1..5] int;

writeln(A[3]); // prints A[3]
```

- **Why does this now work?**
  - writeln() still takes its arguments by default intent
  - because it only reads its args, the default intent for arrays is 'const ref'

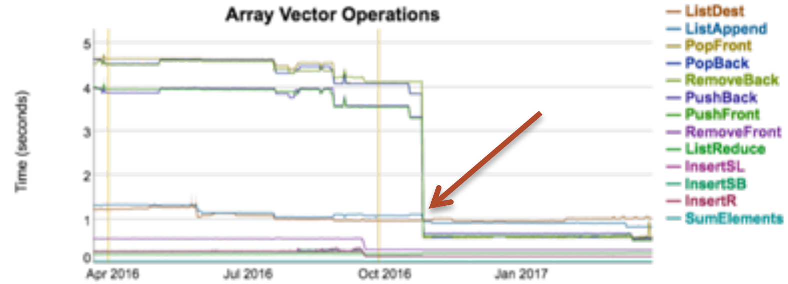
  ➔ writeln now calls the array's read accessor
    - reading a sparse array's "zero" values is fine

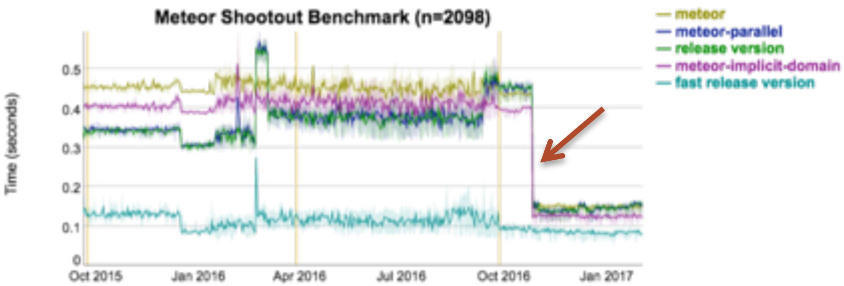
Performance Impact

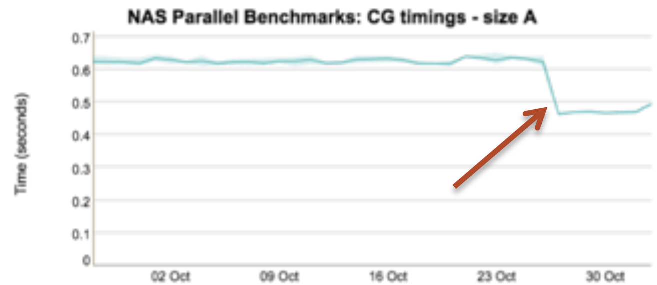
US National Archives

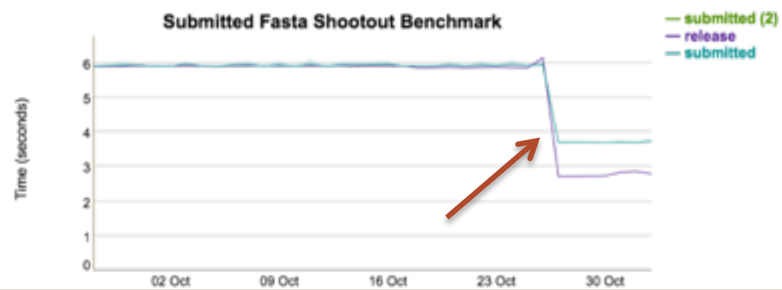
Reworking Array Memory Management

- **Substantial single-locale performance improvements**
  - up to 7x speedup in some cases



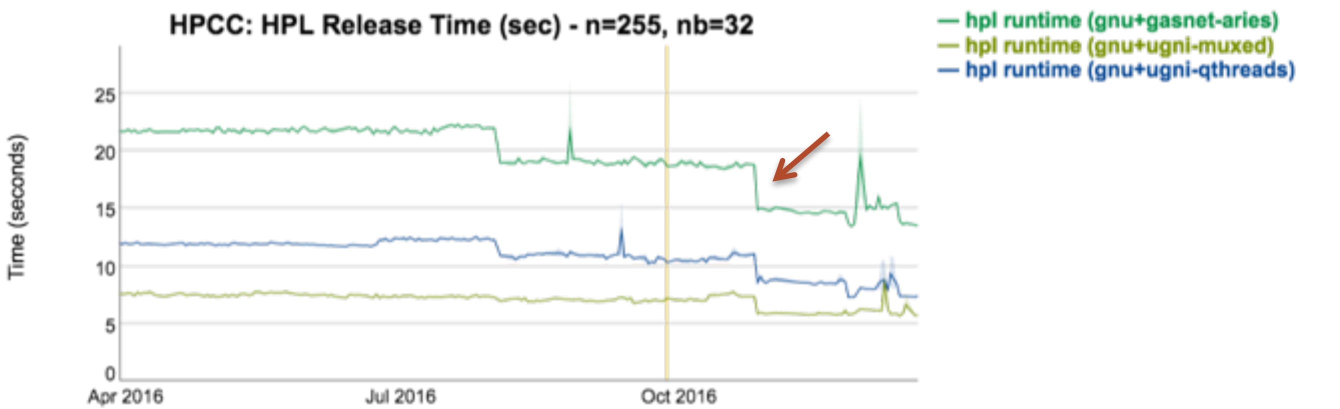


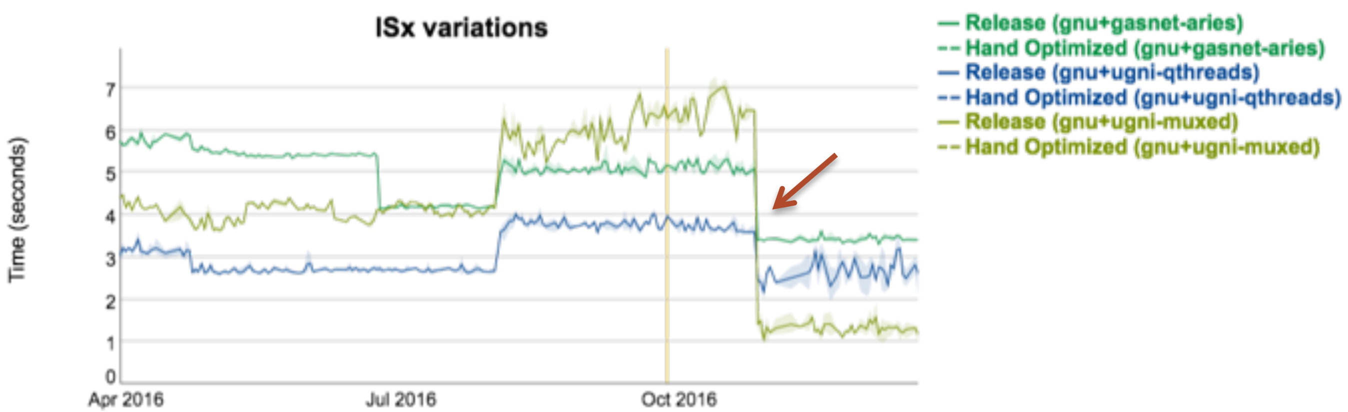




Reworking Array Memory Management

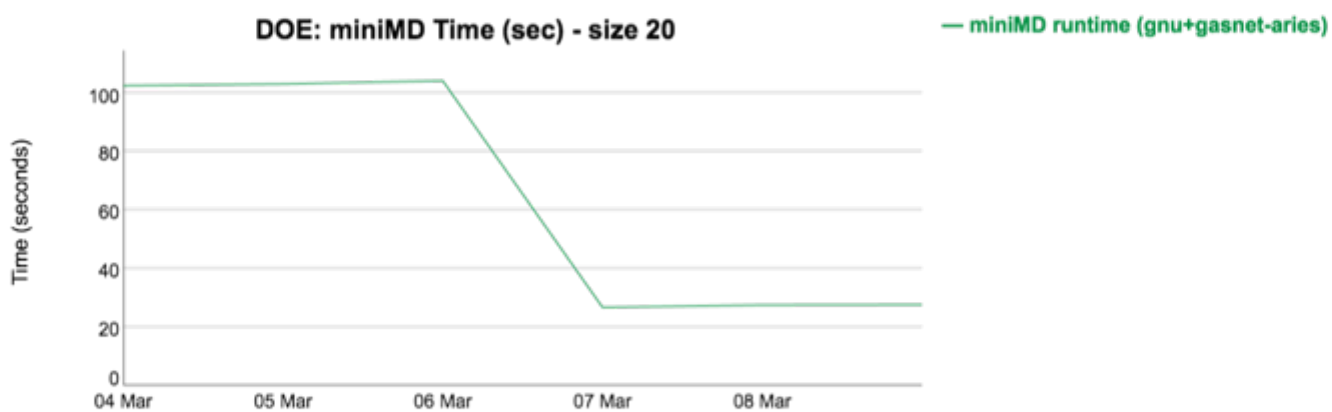
- **Some big multi-locale performance improvements**
  - up to 6x speedup





Adjusting Array Intent

- **Led to about 4x speedup for miniMD on 16 nodes**
  - StencilDist uses return-intent overloads to return from a cache
  - This effort enabled the cache for arrays-of-array stencils, as in MiniMD



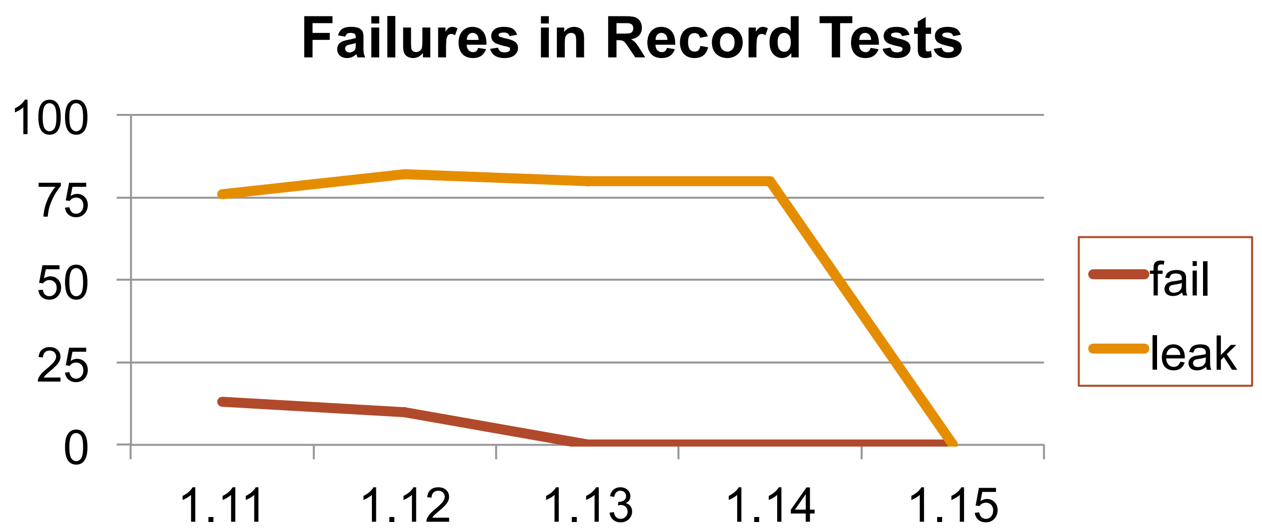
Questions?

- No More Array Reference Counting!
- Arrays Return by Value
- Tuples Capture Arrays by Reference
- Array Intent

Internet Archive Book Images

Building upon Record Memory Management

- Arrays, strings use a record to manage memory
- But records had memory management issues
- Fixing those has enabled progress



Tuples

Lake Clark National Park

Tuples

- **Details of tuple behavior have never been well-defined**
  - things have worked “well enough” for this not to receive more attention

- **Array memory fixes ran afoul of issues with tuples**

- **For example:**

```
proc f( tupleArg ) {
  return tupleArg;
}
var A, B: [1..n] int;
f( (A, B) );
```

  - *are A and B passed by value or by reference into f?*

  - *does returning tupleArg return the contained arrays by value or by ref?*

Tuples

- **Reworked the tuple implementation to support array fixes**
  - guiding principle: 1-element tuples behave similarly to plain elements
  - implementation is now more direct and straightforward

- **Returning to the example:**

```
proc f( tupleArg ) {
  return tupleArg;
}
var A, B: [1..n] int;
f( (A, B) );
```

  - *are A and B passed by value or by reference into f?*
    - by reference, because arrays pass by 'ref' / 'const ref' by default
  - *does returning tupleArg return the contained arrays by value or by ref?*
    - by value, because arrays return by value by default

Language Changes

- **For more information, see**
  - CHIP 13: *When Do Record and Array Copies Occur?*
  - CHIP 6: *Tuple Semantics*

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY