Compiler and Generated Code Improvements

Chapel Team, Cray Inc.

Chapel version 1.10

October 2nd, 2014

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.

Executive Summary

- **Const-ness checking was greatly improved for v1.10**
- **We also undertook two major generated code cleanups:**
  - generation of native C for loops
  - elimination of many common, unnecessary reference temps
- **All executables now support an automatic --about flag**

Outline

Improved Const-ness Checking

Improved Const Checking: Background

Compiler permitted illegal modifications of consts:

- elements of const arrays

```
const A = [ "red" => 5, "yellow" => 6 ];
A["red"] = 7;   // was allowed
```

- loop variables with non-ref iterators

```
// no ref
iter itr() { yield 7; }
for i in itr() do
  i = 5;   // was allowed
```

```
var A: [D] real;
iter itr() ref { yield A[1]; }
for i in itr() do
  i = 5;   // OK - like A[1]=5
```

*cf. a **ref** iterator*

Improved Const Checking: This Effort

Compiler now disallows these modifications

- elements of const arrays

```
const A = [ "red" => 5, "yellow" => 6 ];
A["red"] = 7;
```

compile-time error

- loop variables with non-ref iterators

```
// no ref
iter itr() { yield 7; }
for i in itr() do
  i = 5;
```

compile-time error

cf. a **ref** iterator

```
var A: [D] real;
iter itr() ref { yield A[1]; }
for i in itr() do
  i = 5;   // OK - like A[1]=5
```

Updated some tests

- ones that illegally modified const array elements

Improved Const Checking: Next Steps

- **Check accesses to tuple components**

```
const tup = (1.0, "hi");
tup(1) = 2.0;   // should not be allowed, currently is
```

- **Check accesses to fields of array type**

```
record R { var A: [D] real; }
const r: R;
r.A[1] = 2.0;   // should not be allowed, currently is
```

- **Distinguish initialization from assignment**
  - currently implemented with an imprecise heuristic
  - for variables and fields

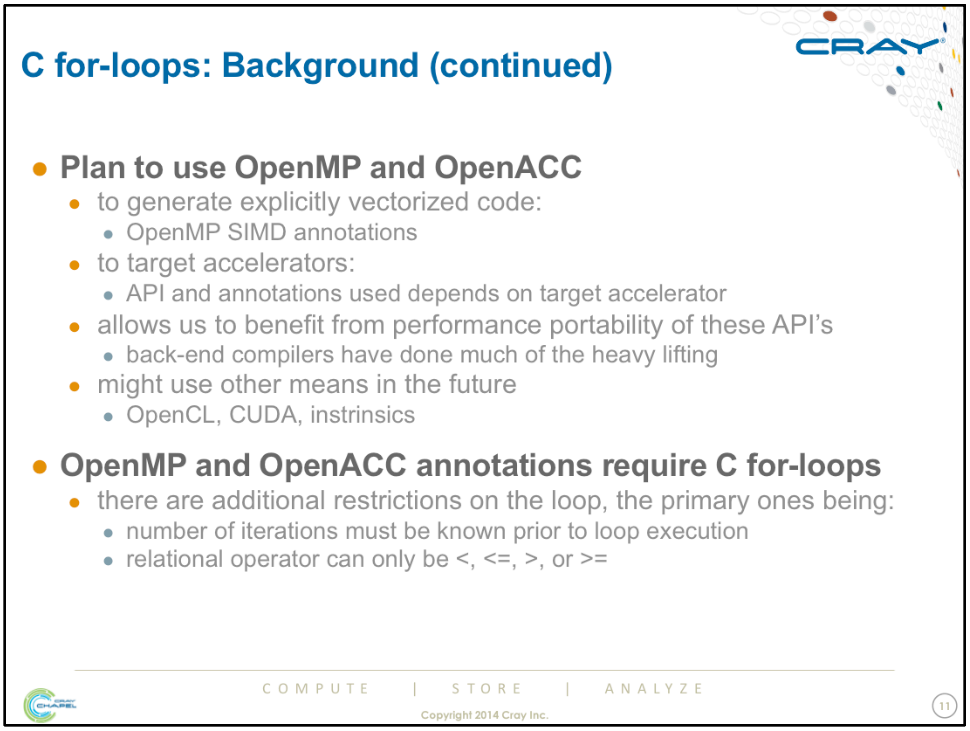
```
class C { const i: int; }
proc C.C() { i = 5; }          initialization – OK
var c = new C(); c.i = 6;      assignment – error
```


The work to distinguish initialization from assignment for class/record members may likely involve language changes, similar to C++ initializers in constructors.

Generation of C for loops

C for-loops: Background

- **Applications must exploit all available parallelism**
  - distributed parallelism (multi-node/multi-locale)
  - thread level parallelism (multi-core)
  - operand level parallelism (vectorization)
  - accelerators (co-processors, GPUs)
- **Chapel compiler responsible for generating parallelism**
  - generates explicit multi-node and multi-core parallelism
  - but relies on backend auto-vectorization for any operand parallelism
  - lacks support for offloading to accelerators



The restrictions on the for-loops that OpenMP and OpenACC can be attached to are similar to the restrictions of Chapel's forall loops

Note that OpenMP is being used for its SIMD/offload capabilities, we do not have any immediate plans to use its multithreading capabilities as our tasking layers are currently responsible for that.

C for-loops: Background (continued)

- **Range iterators drive most loops**
  - most loops iterate over a range
  - or a structure whose iterator forwards to a range iterator
    - e.g. arrays, distributions

```
for a in myArray do                // iterate over an array
  writeln(a);
```

…is implemented in terms of…

```
array.these() {
  for i in myDomain do             // array iterates over its domain
    yield dsiAccess(i);
}
```

…which is implemented in terms of…

```
domain.these() {
  for i in myRange do              // domain iterates over its range(s)
    yield i;
}
```

C for-loops: Background (continued)

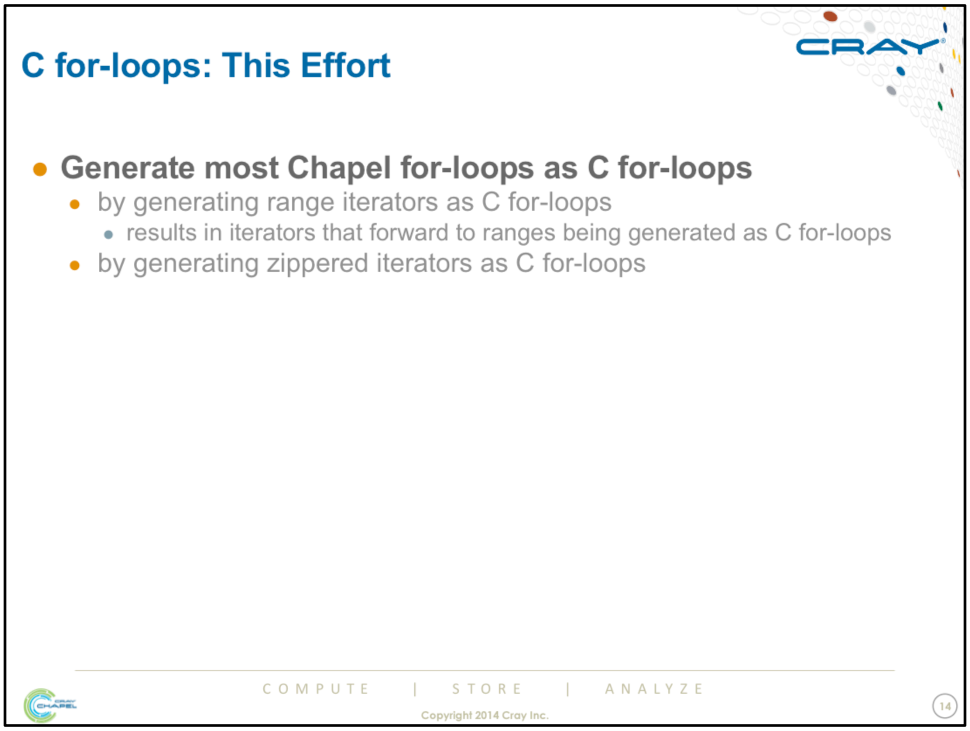
- **Range iterators traditionally generated C while-loops**

```
for i in 1..10                      // range iteration

i = range.first;
end = range.last + 1;
cont = (i != end);
while(cont) {                       // generated while loop
  tmp = (i+1);
  i = tmp;
  cont = (tmp != end);              // != relational operator
}
```

  - not amenable to OpenMP or OpenACC annotations

Range iterators generated while loops because the module code that implements them used a while loop (there was no C-style for loop in chapel to use)



The idea of what this is trying to accomplish is relatively simple: generate range iterators as C for-loops, and you get C for-loops for free in most other cases. We want to generate C for-loops so we can attach OpenMP/OpenACC pragmas to them.

The actual implementation is extremely complicated and unlikely to interest most users.

For those that are interested:

The basic idea is that since Chapel does not have a C-style for-loop in the language, a primitive was added to facilitate representing them.

- instead of while(testExpr) {} (while loops are part of the language) there is a __primitive("C for loop", initExpr, testExpr, incrExpr) {} that is used

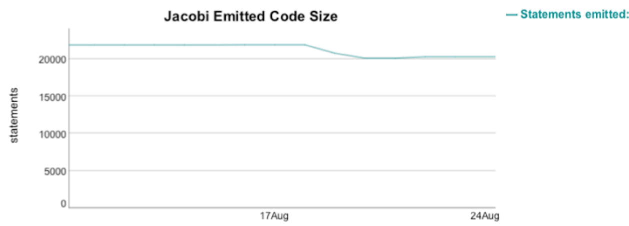
- special codegen routines were added to generate the primitive as a C for-loop

- lower iterators had substantial changes to support lowering and zippering iterators that forward to a C-for loop with iterators that forward to a while-loop as well as with complex iterators and dynamically dispatched iterators

- numerous other changes to the compiler. A few changes include modifications to

C for-loops: Impact

- **Generated code improvements**
  - Decreased generated code size: ~22,000 => ~20,500 for Jacobi



  - Improved readability of generated code

```
for i in 1..10

for (i = start; (i <= end); i += INT64(1))
```

C for-loops: Impact (continued)

- **Generated code for range iteration**

```
for i in 1..10                                 // range iteration
```

was:

```
i = start;                                     // previous
end = end + 1;
cont = (i != end);
while (cont) {
  tmp = (i+1);
  i = tmp;
  conttmp = (tmp != end);
  cont = conttmp;
}
```

is now:

```
for (i = start; (i <= end); i += INT64(1))    // current
```

C for-loops: Impact (continued)

- **Generated code for zippered array iteration**

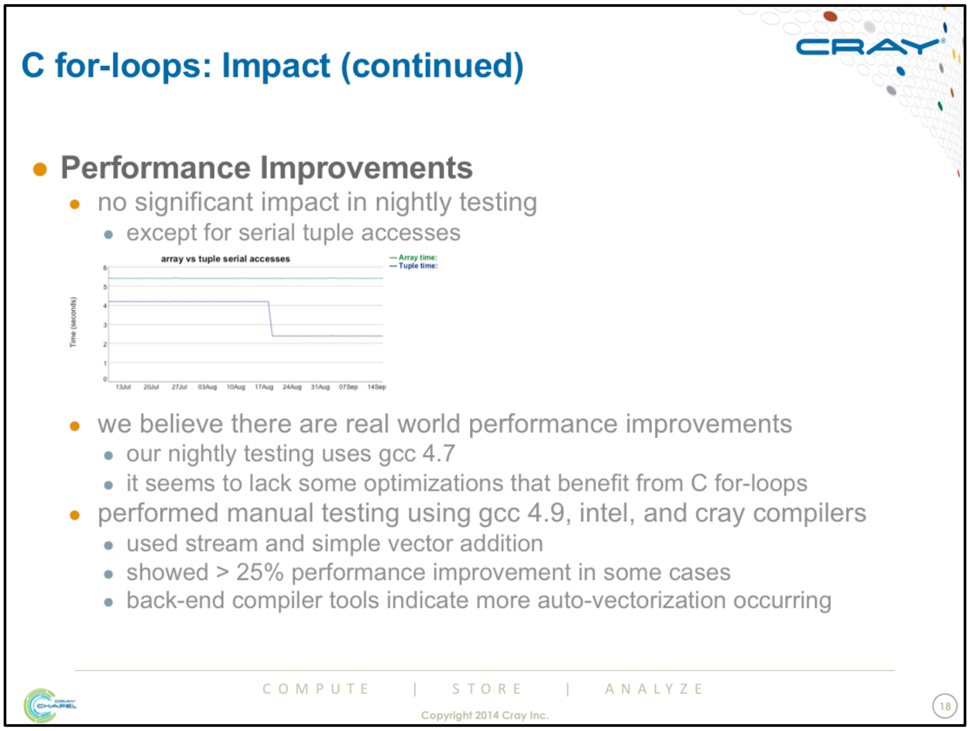
```
for (a, b) in zip(A, B)                          // zippered array iteration
```

was:

```
for (;_cond;) {                                   // previous
   _ref_tmp_5 = &_ic__F6_i;
   *(_ref_tmp_5) += _ic__F4_step;
   tmp31 = (_ic__F6_i != _ic__F5_last);
   if (tmp31)
     _ic__more = INT64(1);
    else
     _ic__more = INT64(0);
   _cond = (_ic__more != INT64(0));
   _ref_tmp_6 = &_ic__F6_i2;
   *(_ref_tmp_6) += _ic__F4_step2;
}
```

is now:

```
for (_ic_i = start1, _ic_i2 = start2;            // current
(_ic_i <= _ic_last); ic_i += _ic_step, _ic_i2 += _ic_step2)
```



We believe there are performance improvements of generating C-for loops (even though that wasn't the motivation for generating them.)

In the past we have observed the Cray compiler perform significantly better by manually replacing the old generated while loops with for loops

- Still need to do more investigation into the performance effects of C for-loops

C for-loops: Impact (continued)

- **Many loops amenable to OpenMP and OpenACC**
  - non-strided, bounded iterators can have annotations attached

    `for i in 1..10, for a in A,` etc.

  - strided, bounded iterators cannot have annotations attached
    - they use !=, which is not allowed by OpenMP or OpenACC
    - future work to make such iterators use <, <=, >, >= when possible

    `for i in 1..10 by 2, etc.`

  - unbounded iterators cannot have annotations attached
    - unbounded iterators will most likely never be amenable to annotations

      e.g., `for i in 1..`
    - number of iterations must be known prior to loop execution

C for-loops: Status and Next Steps

Status:

- Range iterators and iterators that forward to them generate C for-loops
  - most loops in the generated code are C for-loops,
  - many abide the OpenMP and OpenACC loop restrictions

Next Steps:

- Squash range construction for loops over anonymous ranges
  - e.g., '`for i in lo..hi`' does not require us to build the range '`lo..hi`'
    - we can just generate the primitive C for loop directly in the compiler
- Decorate loops with OpenMP and OpenACC annotations
  - initial goal to annotate follower iterators with OpenMP SIMD
  - then move on to more sophisticated annotations
- Detect the sign of a range's stride at compile time when possible
  - strided range iterator uses != since the stride's sign is unknown
  - ranges with strides known at compile time could use <, <=, >, or >=
- Start additional performance testing
  - with newer and varied compilers (gcc 4.9, Intel, Cray, etc.)

Range iteration overflow semantics

Range Overflow: Background

- **In the past, all legal, non-maximal ranges were iterable**

```
for i in max(int)-10 .. max(int)             // fine in 1.9
for i in max(int)-10 .. max(int)-1 by 5      // fine in 1.9
for i in 0:uint .. 10:uint by -1             // fine in 1.9
for i in min(int) .. max(int)                // maximal, iterated 0 times
```

  - However, the generated code could result in undefined behavior

```
for i in 120:int(8)..127:int(8)

end = 127 + 1              // overflows! undefined behavior for C backend
while (i != end) {
  i = i + 1
}
```

    - would be valid in this case for unsigned integers, though

Range Overflow: Background (continued)

- **C for-loops restrict what ranges can be iterated**
  - recall that != is not allowed with OpenMP and OpenACC
  - how to generate loops for ranges whose last index + stride overflows?

```
for i in 245:uint(8) .. 255:uint(8)

// generated as...
for (i = 245; i <= 255;   i+=1)  // all uint(8) <= 255, infinite loop
for (i = 245; i < 255+1;  i+=1)  // no uint(8) < 0 (255+1), 0 iterations
for (i = 245; i != 255+1; i+=1)  // not valid for OpenMP/OpenACC
for (i=0,j=245; i<10; i+=1,j+=1)// works… but adds overhead to every
                                 // iteration, valid for OpenMP/OpenACC?
other options?                   // let us know if you have any!!
```

Range Overflow: This Effort

- **Determine policy for range iterators that will overflow**
  - our current policy: disallow iteration over such ranges by default
    - in the Cray Chapel implementation, NOT in the language specification
    - allows the default range iterator to be highly optimized
    - we believe very few users will be affected by this limitation
    - provide options for users who need to iterate over such ranges
      - e.g., more expensive iterators that can handle such cases
- **Catch attempts to iterate over ranges that will overflow**
  - prior to iterating, range iterators are checked for overflow
    - halts and alerts the user if overflow will occur
    - (if bounds checks are enabled)

Range Overflow: User Options

- **Use a wider type (if possible)**

```
for i in 245:uint(16) .. 255:uint(16)
```

  - not always an option

```
max(int)-10 .. max(int), 0:uint .. 10:uint by -1
```

- **Use general iterator to iterate over any range**
  - including maximal ranges
  - unfortunately, it has a huge performance penalty
    - contains control flow and a break statement
    - prevents iterator optimizations
  - globally controlled with the config param useOptimizedRangeIterators
    - config param replaces ALL range iterators with the general one
  - if iterating directly over a range, the iterator can be called manually

```
for i in (min(int(8)) .. max(int(8))).generalIterator()
```

Range Overflow: Impact and Next Steps

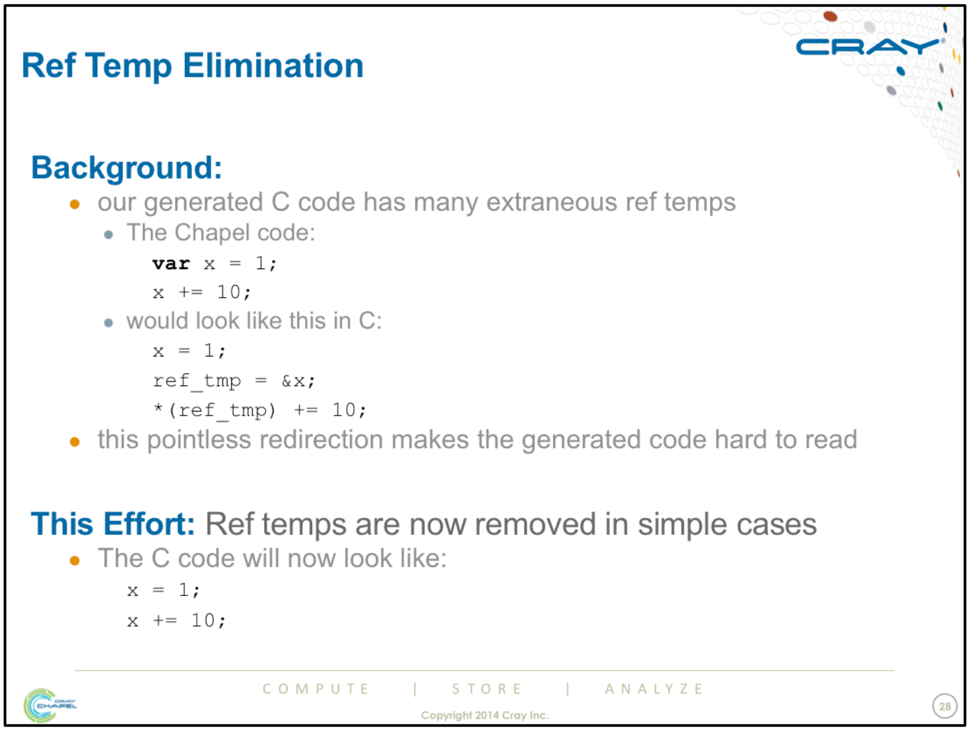
Impact:

- **Some ranges cannot be iterated over by default**
  - there are some options available to user
  - users are warned if they try to iterate over such a range
    - so long as bounds checks are on

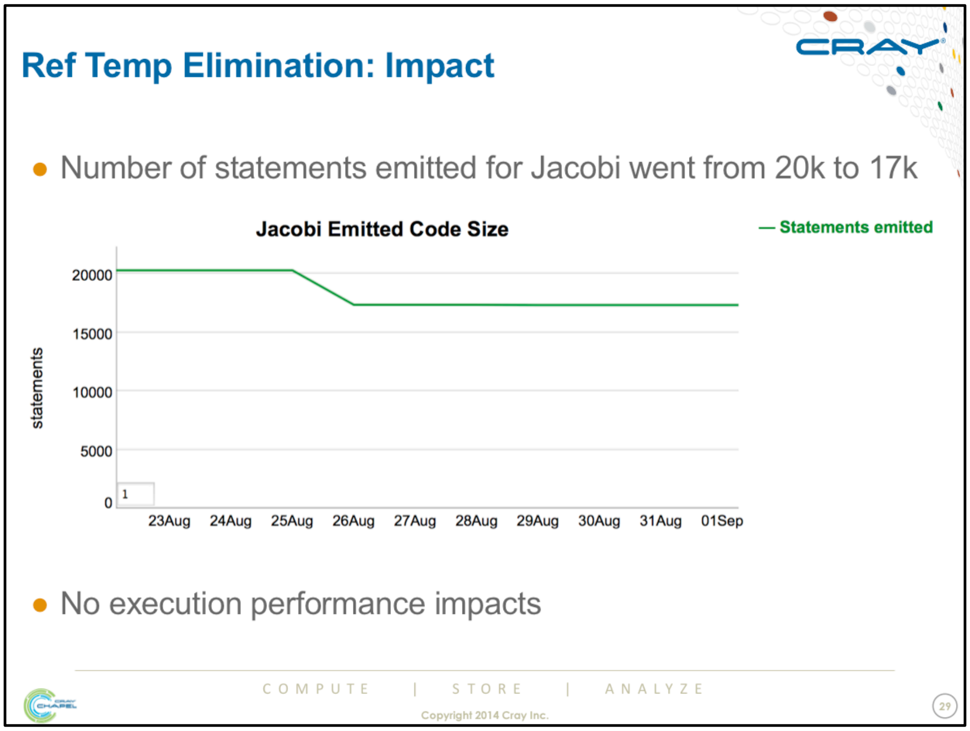
Next Steps:

- **Wait for user feedback**
  - our expectation is that very few users will be affected by this limitation
    - however we encourage anybody who is to contact us
  - design a better solution to the problem if it causes issues for users

Ref Temp Elimination



Simple cases are defined as any call that is a primitive (excluding PRIM_SET_MEMBER) or only contains primitive calls. While not perfect, this does remove ref temps for simple things like =, +=, *=, etc.



There were no execution performance increases because the C compilers were already removing all of these temps.

Ref Temp Elimination: Next Steps

- Remove ref temps in more cases
  - We should be able to detect a few more cases easily
- Avoid inserting unnecessary ref temps from the start
  - Current approach removes them as a peephole optimization

--about flag

--about flag: Background

In addition to compile-time flags, Chapel programs are built using a specified environment

- the environment can be found by running the script `printchplenv`

```
% $CHPL_HOME/util/printchplenv
CHPL_HOME: /path/to/chapel
script location: /path/to/chapel/util
CHPL_HOST_PLATFORM: cray-xc
CHPL_HOST_COMPILER: gnu
CHPL_TARGET_PLATFORM: cray-xc
CHPL_TARGET_COMPILER: cray-prgenv-cray
CHPL_LOCALE_MODEL: flat
CHPL_COMM: gasnet
  CHPL_COMM_SUBSTRATE: aries
  CHPL_GASNET_SEGMENT: fast
…
```

--about flag: This Effort

- **Make the precise compilation environment available using an --about flag built into all Chapel binaries**
- **Generate Chapel compilation environment information**
  - Make all Chapel environment variables available via C const strings
  - Generate a new function called `chpl_program_about()` that prints out the compilation command line, compiler version, and Chapel environment used to compile the program
  - Restructured to make adding Chapel environment variables more maintainable and less error-prone
- **Implement the --about flag in the runtime**
  - Catch the flag before launching the Chapel program (like --help)
  - Call `chpl_program_about()` when the --about flag is used

--about flag: Impact

- **Precise information about how a program was compiled**
  - Useful for performance debugging, user support, etc.

```
% ./a.out -about
Compilation command: chpl hello.chpl -fast
Chapel compiler version: 1.10.0
Chapel environment:
CHPL_HOME: /path/to/chapel
CHPL_HOST_PLATFORM: cray-xc
CHPL_HOST_COMPILER: gnu
CHPL_TARGET_PLATFORM: cray-xc
CHPL_TARGET_COMPILER: cray-prgenv-cray
CHPL_LOCALE_MODEL: flat
CHPL_COMM: gasnet
CHPL_COMM_SUBSTRATE: aries
CHPL_GASNET_SEGMENT: fast
…
```

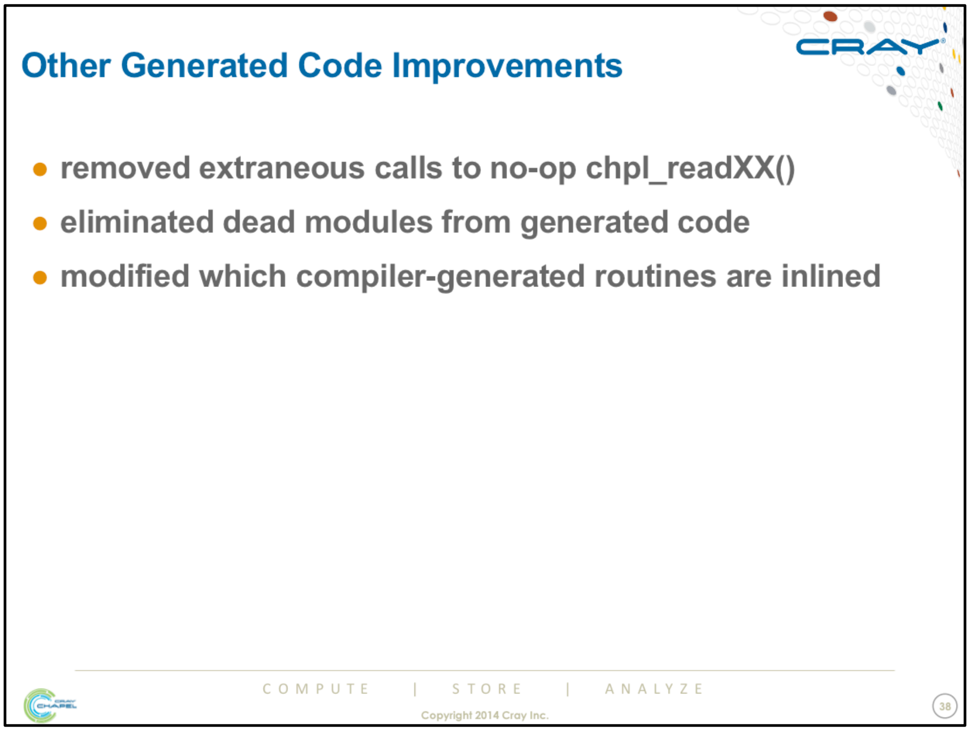
--about flag: Next Steps

- **Add relevant Chapel environment variables that correspond to compile time flags to --about info**
  - Most flags can be set via a similarly named environment variable, e.g., --stack-checks and CHPL_STACK_CHECKS
  - Compilation environment is not captured if the environment variable versions are used
- **Print environment info as seen with printchplenv script**
  - Pros: Non-relevant variables not printed, cons: slows compilation time
- **Auto-generate the CHPL_* environment variable definitions in the compiler**
  - Current the definitions are in module code and must be manually updated when changed
- **Make CHPL_* env vars available as compiler flags**

Other Compiler and Generated Code Improvements

Other Compiler Improvements

- **Converted remaining assignments to use new signature**
  - i.e., `proc =(ref x: t, y: t) { … }`
  - rewrote assignments to use PRIM_ASSIGN rather than PRIM_MOVE
- **Improved error messages for…**
  - function control paths that don't return
  - assignments between unsupported pairs of types
  - op= assignments to a bad l-value
  - applying 'new' to bad expression types
  - trying to use --llvm when it was not enabled
  - distinguishing between 0 and 2+ candidates in function resolution
- **Compilation speed improvements**
  - by rewriting Chapel environment inference scripts in Python
  - by consolidating system calls to query Chapel environment
- **Improvements to --print-passes output**



1st bullet: chpl_readXX() is used in certain circumstances where a variable might be a sync/single but we can't tell for sure. If it turns out it is then we convert that to a real .readXX(); otherwise, we should remove it entirely. We weren't always doing the latter.

Compiler/Generated Code Priorities/Next Steps

Compiler/Generated Code Priorities/Next Steps

- **C for loop improvements**
  - optimized generation of loops over anonymous ranges
  - SIMD-ization/vectorization of generated loops
- **Support for standalone parallel iterators**
- **Squash insertion of ref temps (rather than removing later)**

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, URIKA, and YARCDATA. 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.

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/