Generated Code Improvements

Chapel Team, Cray Inc.

Chapel version 1.14

October 6, 2016

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.

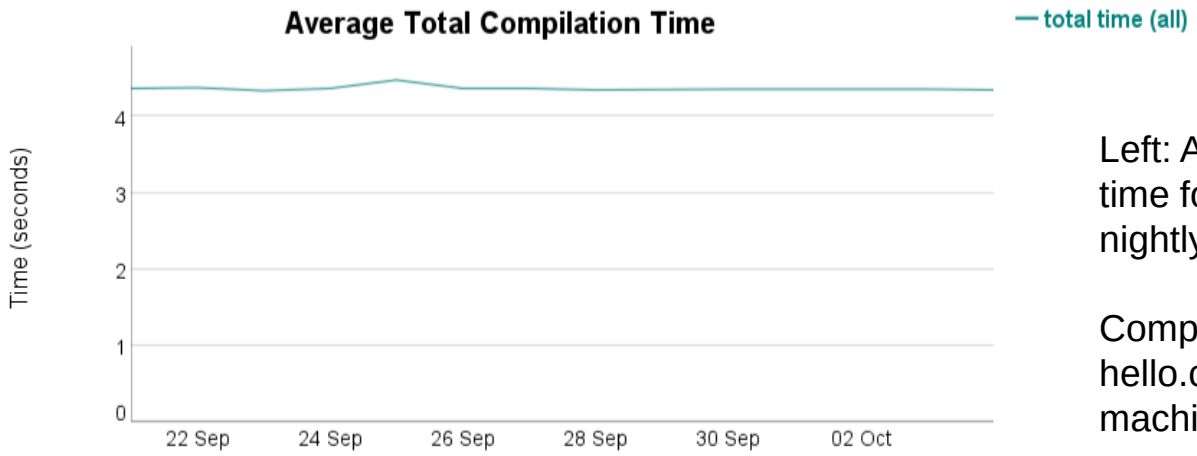
Outline

- Incremental Compilation
- Denormalize Pass
- Vectorization Changes
- LLVM Debug Information
- Other Generated Code Improvements

Incremental Compilation

Incremental Compilation: Background

- **Compiler analyzes whole program at one time**
  - This includes recompilation of standard library code
- **[Re]compilation is relatively slow**
  - Even for minor changes to a small application
- **Goal: provide a quick recompilation mode for users**

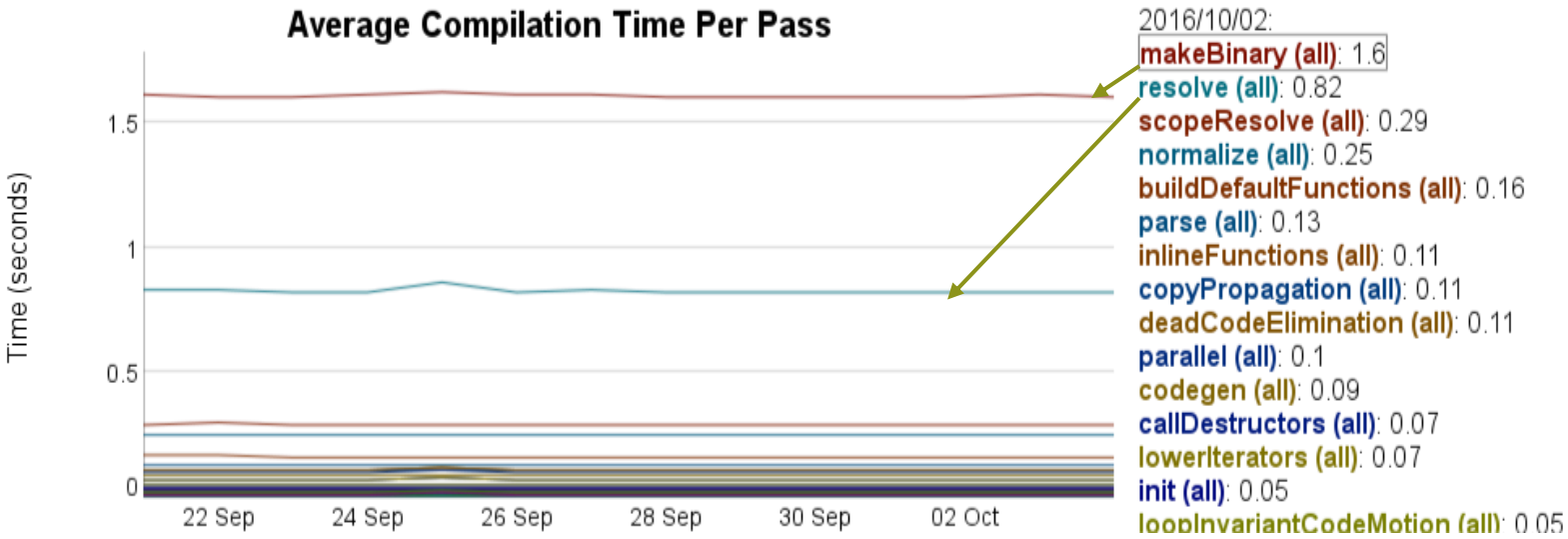


Left: Average compilation time for all tests in our nightly testing suite.

Compilation time for hello.chpl on the same machine is ~3.4 seconds

Incremental Compilation: Background

- **Majority of time spent in 2 passes:**
  - makeBinary – compilation of generated C code
  - functionResolution – type resolution and resolving function calls
- **Improvements here would benefit compilation significantly**
  - makeBinary accounts for ~40% of the compilation time



Incremental Compilation: This Effort

- **Google Summer of Code project**
  - Work from Kushal Singh at IIIT Hyderabad
  - Description of current work and future plans available in CHIP 15
- **Preliminary design for functionResolution pass**
- **Implementation work on codegen/makeBinary passes**

Incremental Compilation: This Effort

- **Preliminary design for functionResolution pass**
  - Including list of cases where a function would need recompilation:
    - Changes to its body
    - Changes to declaration of functions it depends on
    - New potential matches when resolving calls within its body
  - Lower implementation priority than makeBinary

Incremental Compilation: This Effort

- **Implementation work on codegen/makeBinary**
  - makeBinary compiles every generated .c file into one monolithic .o file
    - Any change to one file would trigger a complete rebuild of this .o
  - Moved to generating two .o files and linking them together
    - One for user code, one for library code
    - User changes hopefully would not modify library .o file, saving recomp time
  - Required “purifying” generated header file
    - Moved definitions into another file but left declarations in place
    - In incremental mode, removed “static” keyword from exported symbols
    - Allowed header to be #included by both .o files w/o link time errors

Incremental Compilation: This Effort

- **How to reuse the unmodified parts of the generated code?**
  - Codegen always generated completely new copies of the .c files
    - Even if some of the files would be identical to previous versions
  - Need to maintain persistent state between compiles
    - Compare new generated code against persistent copy…
    - … and move the changed code into the persistent storage location, …
    - … then rebuild only the touched parts of the persistent state
  - Remove the persistent state if changes detected in:
    - Compilation flags
    - Environment variables
    - Changes in library code effectively yield the same result
    - Can be more selective later, but not a high priority

Incremental Compilation: This Effort

- **Add flag to enter this compilation mode, --incremental**
  - Generates two .o's instead of one
  - During first run, saves persistent state and current compilation options
    - Subsequent runs perform check, then overwrite persistent state as needed
- **Challenge: generated .c must be stable, but currently isn't**
  - i.e. An unmodified application should always yield the same .c files
    - Today we see differences in the ordering of functions from run to run
    - This means we must overwrite the persistent state too frequently

Incremental Compilation: Status

- **Known Issues**
  - Generated .c code not completely stable
  - No support for the LLVM back-end
  - Will differ in execution performance from a normal compile
    - Certain gcc optimizations thwarted by multiple .o's, removing static keyword

Incremental Compilation: Next Steps

- **Finish stabilizing generated C code**
- **Discuss next steps before function resolution changes**
- **Make our library .o file reusable across different programs**
  - This would speed up compile time for all programs, not just recompiles
  - Challenge: library code is highly generic and varies with user code
- **Add framework for recompilation to function resolution**
  - Including persistent storage of AST dependencies for functions
- **Other compilation time improvements**

Denormalize

Denormalize: Background

- Generated C code includes a lot of temporary variables
- Consider this Chapel code:

```
var x = 123;
writeln(x*x + x);
```

- It generates the C code:

```
int64_t call_tmp_chpl;
int64_t call_tmp_chpl2;
call_tmp_chpl = (INT64(123) * INT64(123));
call_tmp_chpl2 = (call_tmp_chpl + INT64(123));
writeln_chpl2(call_tmp_chpl2);
```

- The C compiler often optimizes these temps away, but
  - it has to work to do so
  - they increase the complexity for developers looking at the generated C

Denormalize: Background

- Most of the Chapel compiler works with *normalized* AST
- AST is *normalized* so there are no nested call expressions

```
x*x + x
```

becomes

```
call_tmp = x*x
call_tmp + x
```

- The *normalize* pass adds these call_tmp temporaries
- Later passes rely on call expressions not being nested

Denormalize: This Effort

- **Add a pass to remove these call temporaries**
- **Pass runs just before generating C code**
  - only codegen needs to be able to work with a denormalized AST

Denormalize: Impact

- **Remember this Chapel code:**

```
var x = 123;
writeln(x*x + x);
```

- **With --denormalize, it generates C code like**

```
writeln_chpl2( (INT64(123) * INT64(123)) + INT64(123));
```

- **Results in a 25%-50% reduction in lines of C code**

Multi-slide topic: Status and Next Steps

Status:

- --denormalize available in 1.14 release
  - off by default due to insufficient testing prior to release

Next Steps:

- turn --denormalize on by default
- consider further improvements to code generation to clean up
  - useless casts
  - unnecessary parentheses
- improve --denormalize to cover more cases
  - e.g. function calls with a single argument

Changes to --vectorize

Chapel Vectorization: Background

- **Chapel is well-suited for vectorization**
  - limited aliasing
  - support for array programing

```
A = B + C;
```

  - parallelism is a first class citizen

```
forall i in 1..10 do …
```

- **Need to convey Chapel semantics to back-end**
  - do not want to generate explicit vectorization
    - rather, convey when vectorization is legal
    - leverage back-end compilers' sophisticated and refined cost models

Changes to --vectorize

Background:

- Added --vectorize in 1.11.0
- It finds and marks order-independent (data parallel) loops
- And attaches “#pragma ivdep” to the generated code
  - ivdep == ignore vector dependencies
- In 1.14.0, we saw data parallel loops that had vector dependencies
  - in particular with reductions, likely also any loop with “task-private” vars
  - realized order-independence is not sufficient for asserting ivdep

This Effort:

- Stopped enabling --vectorize with --fast
  - is now an opt-in (use at your own risk) flag

Next Steps:

- Determine what additional analysis is needed to safely use ivdep
- Continue exploring other vectorization strategies

LLVM Debug Information

LLVM Debug Information

Background: Chapel includes --llvm code generation option

- generates LLVM IR instead of C code
- initially added in 1.6.0
- supports several research projects
- but, did not generate debug information with -g

This Effort: Include debug information in generated LLVM IR

- contributed to by Matt Baker, Hui Zhang
- finished as part of Google Summer of Code

Impact: --llvm -g works to debug Chapel programs

- debugger can show Chapel source code lines
- debugger can show Chapel global variables

Next Steps: Further improve the debug experience

- make debug variable names match Chapel source code
- generate debug information for local variables

Other Generated Code Improvements

Other Generated Code Improvements

- **Improved accuracy of #line directives for vars, procs**
  - Improves debugging of Chapel code through the generated C
- **Improved quality of 'local' blocks in generated C code**
  - merged adjacent blocks to reduce curly braces
  - added a "/* local block */" comment for developers

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

THE SUPERCOMPUTER COMPANY