Many-Core Compiler Fuzzing

Alastair F. Donaldson

Multicore Programming Group

Joint work with Chris Lidbury,
Andrei Lascu and Nathan Chong

Imperial College London

Importance of compiler reliability

We rely on reliable compilers for:

- Day-to-day programming
- Source code analysis
- IR-level analysis

Are source- or IR-based verification techniques meaningful if the **compiler misbehaves**?

Are compilers reliable?

CPU compilers, by and large: **YES**
(but there are bugs)

OpenCL (GPU) compilers? That's what we wanted to address in this work

Two compiler testing methods

Random differential testing

Cross-check multiple compilers with respect to random programs

Equivalence modulo inputs (EMI) testing

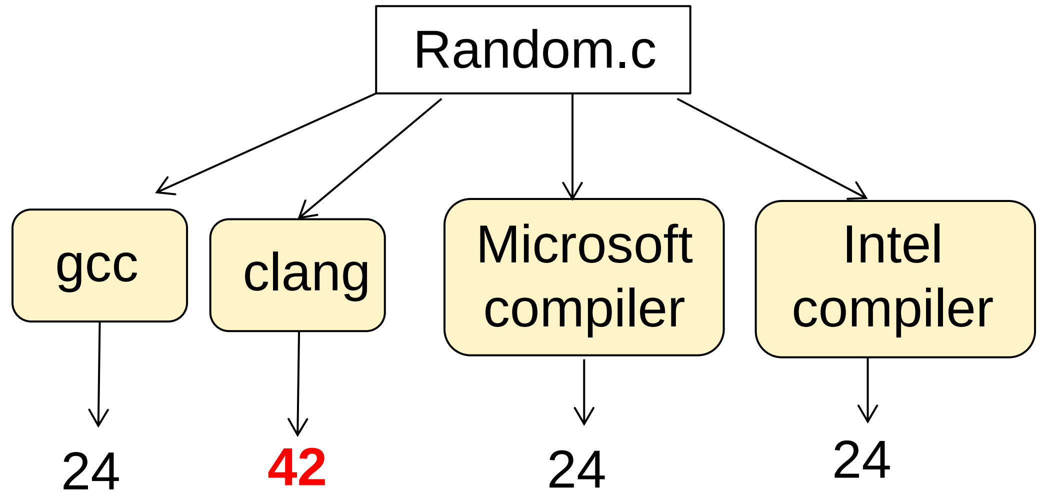
Cross-check single compiler with respect to many programs that should compute identical results

Random differential testing

Generate random programs

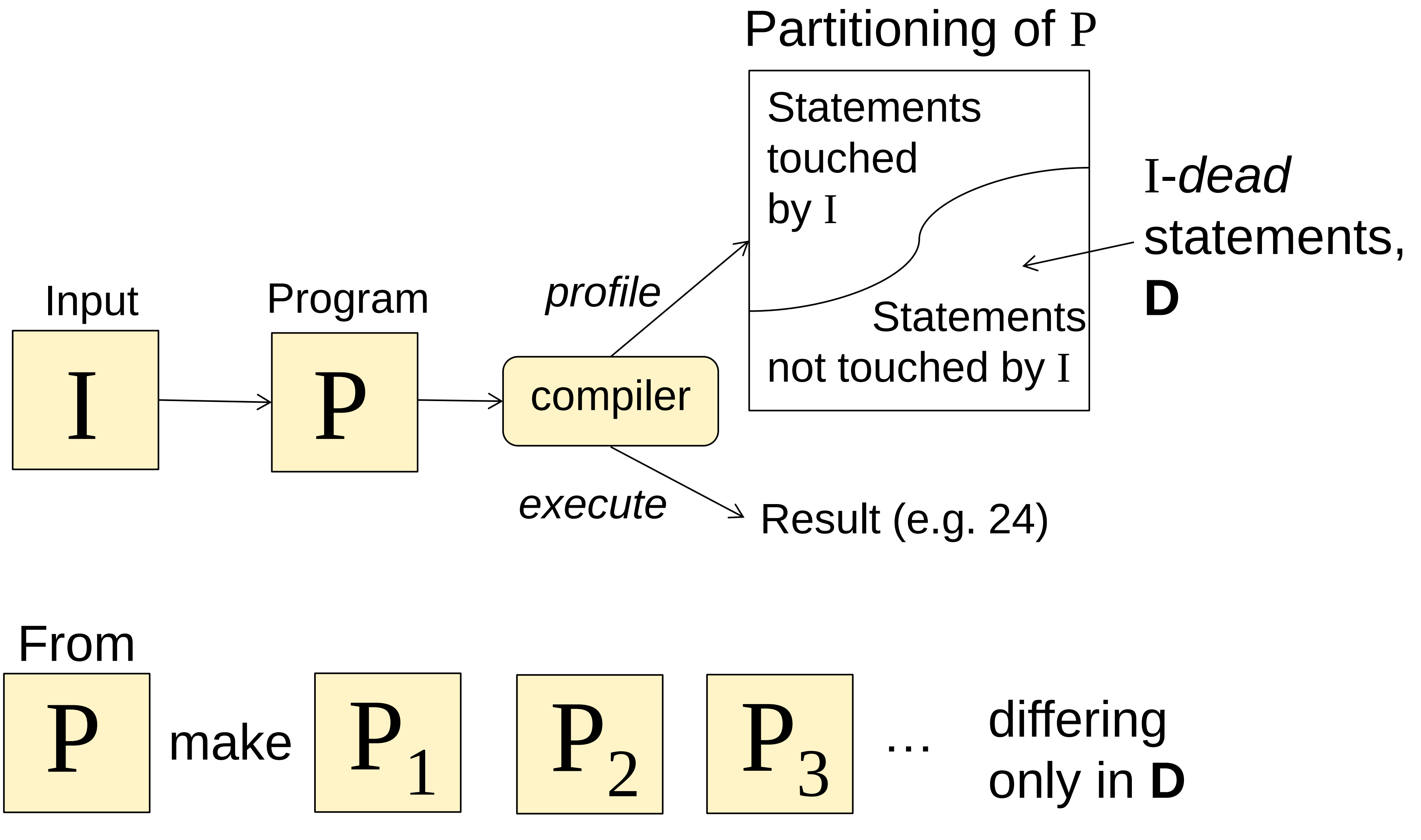
Try them with many compilers

Result mismatches indicate bugs

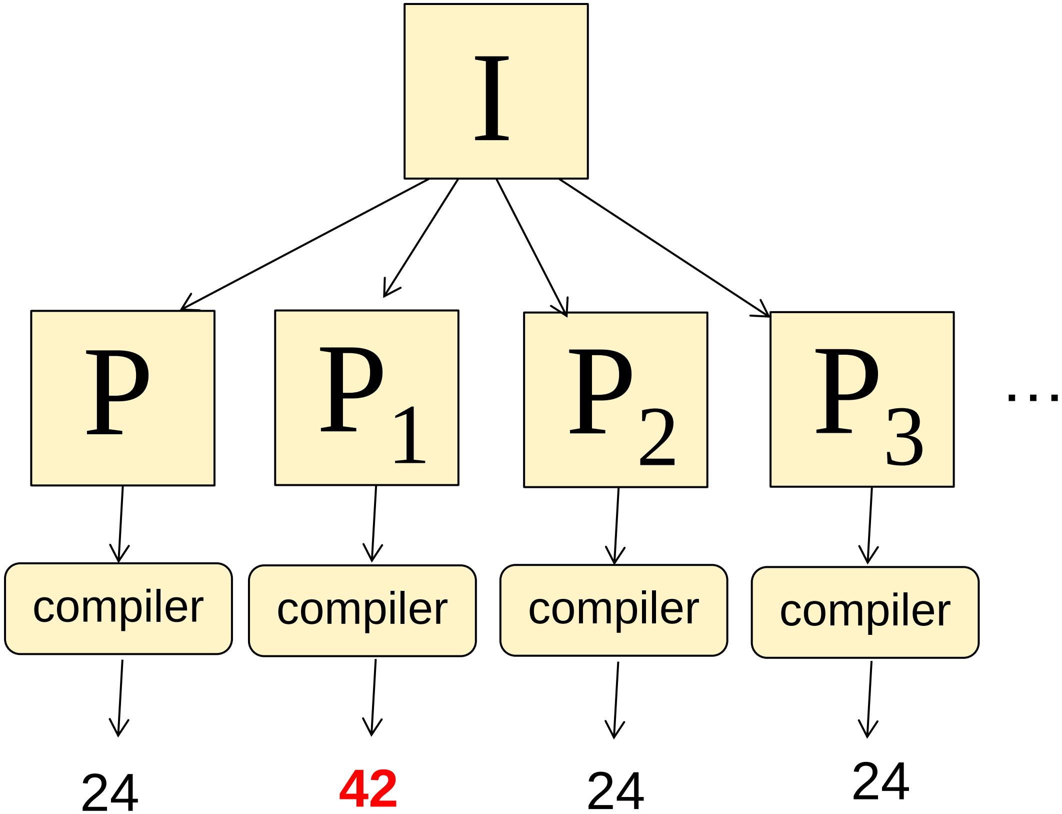


Pioneered by **Csmith**, University of Utah (PLDI'11)

Equivalence modulo inputs testing



Equivalence modulo inputs testing



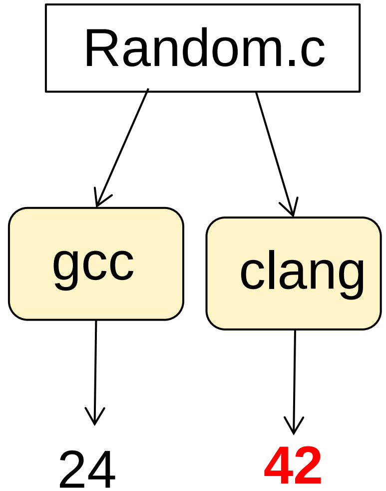
Programs are equivalent modulo I

Mismatches indicate bugs

Pioneered by researchers at UC Davis (PLDI'14)

Does not require multiple compilers

Compiler testing and undefined behaviour



The mismatch is not erroneous if Random.c exercises **undefined behaviour**

If an execution exercises undefined behaviour, the entire execution has no meaning

Any result is acceptable

Let’s see the effects of this in practice

Compiler testing and undefined behaviour

Random differential testing requires programs that are **free from undefined behaviour**

Csmith aims to guarantee this via careful generation, and “safe math” macros

E.g., instead of generating: `e1/e2` (`e1`, `e2` unsigned) generate `safe_div(e1, e2)` defined as follows:

```
#define safe_div(e1, e2) \
     ( (e2) == 0 ? (e1) : (e1) / (e2) )
```

Many core compiler fuzzing

We have lifted random differential testing and EMI testing to **many-core compilers**

Our focus: OpenCL

Discovered defects so far in OpenCL compilers from all main vendors.

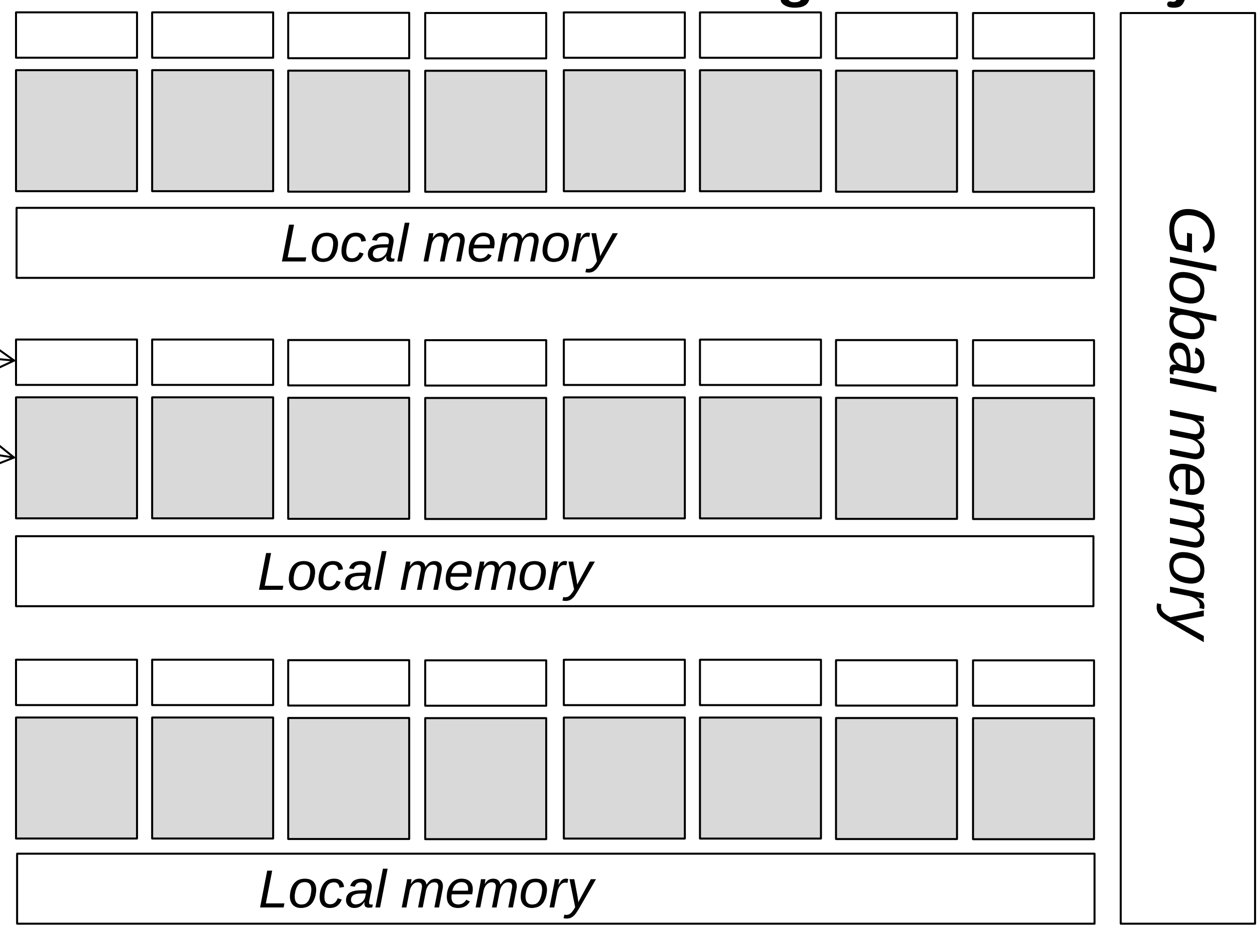
Paper accepted at PLDI'15, CLsmith tool is open source

OpenCL programming model

Many work items

All work items share global memory

Organised into work groups



Work items in same group share local memory

Data races in OpenCL kernels

A **data race** occurs if:

- two **distinct** work items access the **same** memory location
- at least one of the accesses is a **write**
- the work items are in different work groups

  or

  the accesses are **not** separated by a barrier synchronisation operation

OpenCL kernel example

Indicates that function is the kernel's entry point

Indicates that **A** is an array stored in global memory

```
kernel void
add_neighbour(global int* A, int offset) {
  A[id] = A[id] + A[id + offset];
}
```

Read/write data race

Built-in variable which contains work item's id

All work items execute **add_neighbour** –
host specifies how many threads should run

Using barrier to avoid a data race

```
kernel void
add_neighbour(global int* A, int offset) {
  int temp = A[id + offset];
  barrier();
  A[id] = A[id] + temp;
}
```

Accesses cannot be concurrent

Basic lifting of Csmith to OpenCL

- Encapsulate a Csmith program in `func()`
- Make the following OpenCL kernel:

```
kernel void entry(global ulong * result) {
    result[id] = func();
}
```

- Every work item independently computes `func()`
- No communication between work items

Basic lifting of Csmith to OpenCL

Effort required to achieve this:

- OpenCL 1.x does not allow globally scoped variables; these have to be modelled via a struct
- Necessary to disable some non-OpenCL features (e.g. bit-fields)
- Necessary to write a host application to coordinate things

Exercising barriers

Hypothesis: compilation likely to be barrier-sensitive; may be a source of compiler errors

Idea: extend random programs so that threads communicate at barriers, but **maintain determinism**

Generate 2D array, permutations, for each *i*, `permutations[`*i*`]` is a permutation of work item ids.

Equip work group with array A, initially uniformly constant (e.g. `{ 3, 3, 3, …, 3 }`)

Equip work item with private variable `tid`, initialised to `permutations[0][id]`

Exercising barriers

During execution, a work item **owns** `A[tid]` – it can read from and write to this location without conflicts

At random execution points, change ownership:

```
barrier();
tid = permutations[C][id];
```

where `C` is an index into permutations, chosen randomly at generation time

Because `permutations[C]` is permutation, `tid` is different for every work item

EMI testing for OpenCL

Real-world OpenCL kernels do not typically contain code that is dead for some inputs

Synthesised OpenCL kernels does contain such code, but doing code coverage on OpenCL is non-trivial

Our idea: inject *dead-by-construction* code

Dead-by-construction code injection

```
kernel void entry(<params>) {

        Statements

        Statements

}
```

Dead-by-construction code injection

```
kernel void entry(<params>, global uint * emi) {
  Statements

  // guaranteed to be false at runtime
  if(emi[12] > emi[24])
  {
    Any syntactically valid code
  }
  Statements
}
```

At kernel launch time, fill `emi` with: `{ 0, 1, 2, … }`

Behaviour of kernel should not be affected by injection

Experimental evaluation

Evaluated 21 (device, driver) configurations

- Intel CPUs and GPU
- Nvidia GPUs
- AMD GPUs
- Altera emulator and FPGA
- Oclgrind emulator
- Three further anonymous configurations

Full discussion of findings in our paper

Let's see some bugs!

Ongoing and future work

Random differential testing for OpenGL

- Can we diff sets of pixels in a meaningful way?

Finding floating point compiler bugs

Compiler fuzzing on nondeterministic programs that use OpenCL 2.0 atomics

Automated reduction and ranking of bugs

Check out our PLDI'15 paper