UW PL Security Research
(And Research Ideas)

Dan Grossman
(then Sorin Lerner)
(includes work by Michael Ringenburg)

A scatter-plot…

Goal: Tell you about
- Neat stuff we're doing
- Neat stuff we want to do ASAP

Grossman:
- *Cyclone* for memory-safe systems software
- *Clamp* for modular & portable systems software
- Error-message search for more reliable compilers
- Atomic for more secure concurrent programming

Lerner: Rhodium for provably correct compiler optimizations

Cyclone: what

A safe C-like language
- Implemented like C (pointers are addresses)
- Memory management: stack pointers, arenas, unique pointers, garbage collection
- Array bounds: static analysis, known bounds, bounds in variables, bounds in struct fields, Java-style arrays
- Parametric polymorphism for generics: `` `a `` instead of `void*`, no code duplication
- Interoperability with C (same calling convention and data rep, can give C code Cyclone types)

Cyclone: how

Enforce known idioms with known approaches:
- Intra-procedural static analysis:

```
char buf[10];
for(int i=0; i<10; ++i) buf[i] = f();
```

- Types for inter-procedural invariants:

```
void g(tag_t n, char buf[n]);
```

- Explicit dynamic checks if appropriate:

```
void h(char *@fat buf) { buf[e]; }
```

- Synergy: **h** can do **`g(numelts(buf),buf)`** and analysis of **g** knows **n** is a constant

Cyclone: where

- The Cyclone compiler and libraries (100K lines)
- STP (extensible transport protocols)
- MediaNet (multimedia overlay network)
- OKE (extensible kernel)
- RBClick (extensible router with resource bounds)
- Windows device driver (can still crash kernel though)

And fairly portable because compile to gcc:
- Linux, OS/X, Cygwin, Lego Mindstorm, …
- Current implementation assumes 32-bit words

Just a start

Last week I argued memory safety is necessary but insufficient for secure systems

So Cyclone brings something necessary to the C-level, but we still need:
- Strong and correct interfaces
- A reliable compiler

Clamp

Clamp is a C-like Language for Abstraction, Modularity, and Portability (and it holds things together)

In part, go beyond Cyclone by *using a module system to encapsulate low-level assumptions*, e.g.,:

- Module X assumes big-endian 32-bit words
- Module Y uses module X
- Do I need to change Y when I port?

(Similar ideas in Modula-3 and Knit, but no direct support for the data-rep levels of C code.)

Clamp does not exist (help! Or share your woes!)

Error Messages

Here's what happens:

1. A researcher implements an elegant new analysis in a compiler that is great for correct programs.
2. But the error messages are inscrutable, so the compiler gets hacked up:
   - Pass around more state
   - Sprinkle special cases and strings everywhere
   - Slow down compilation
   - Introduce bugs

Yesterday, I fixed a dangerous bug in Cyclone resulting from not type-checking **`e->f`** as **`(*e).f`**

A new approach

- One solution: write 2 checkers, trust the elegant one, use the slow one for messages
  - – Hard to keep in sync; slow one no easier to write
- My plan: use fast one as a subroutine for *search:*
  - – Human speed can be really slow (1-2 secs)
  - – Find a similar term (with holes) that type-checks!
    - Easier to read than types anyway
    - Can offer different ones and rank them
- Example: "**`f(e1,e2,e3)`** doesn't type-check, but **`f(e1,_,e3)`** does and **`f(e1,e2->foo,e3)`** does"
- Help! (PL, compilers, AI, HCI, …)

Atomic – what

An easier-to-use and harder-to-implement synchronization primitive:

```
void deposit(int x){
synchronized(this){
  balance += x;
}}
```

```
void deposit(int x){
atomic {
  balance += x;
}}
```

semantics:
lock acquire/release

semantics:
(behave as if)
no interleaved execution

No fancy hardware, disabling interrupts, or code restrictions (there is a catch…)

Atomic – how

Elegant, efficient solution for this special case:

No threads sharing memory run at truly the same time (on separate processors). E.g.,:

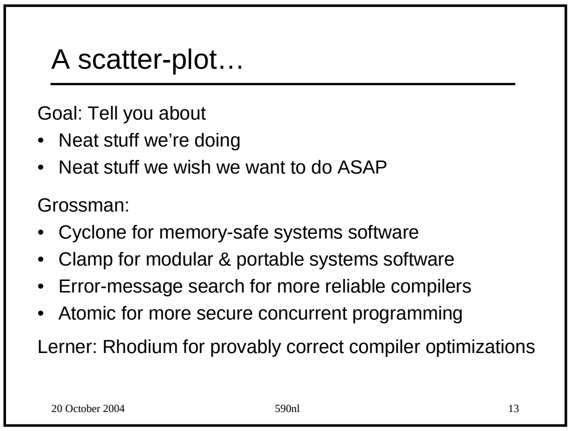
- – Every uniprocessor
- – MSR's Singularity, OCaml, DrScheme, Cecil, …
- – Concurrency for I/O masking and GUIs
- Bag of tricks (see Mike for details)
  1. Atomic code *logs writes* and *buffers sends*
  2. Two versions of functions (atomic & non-atomic)
  3. Scheduler *rolls-back* unfinished atomic blocks
  4. No *blocking for receives*

Atomic – why

- Often what you want conceptually
- Can implement locks & co-exist with locking code
- Efficient: − Non-atomic code unchanged
  − Reads in atomic code unchanged
- Atomic code never starved or corrupted by bad code!

```
void deposit(int x){
atomic {
 int tmp = balance;
 tmp += x;
 balance = tmp;
}}
```

```
void bad1(){
 int tmp = balance;
 balance = tmp*1.01;
}
void bad2(){
 atomic { bad2(); }
}
```



A scatter-plot…

Goal: Tell you about

- Neat stuff we're doing
- Neat stuff we wish we want to do ASAP

Grossman:

- Cyclone for memory-safe systems software
- Clamp for modular & portable systems software
- Error-message search for more reliable compilers
- Atomic for more secure concurrent programming

Lerner: Rhodium for provably correct compiler optimizations