A Type-Safe Dialect of C

Greg Morrisett

Harvard University

Collaborators: D.Grossman, T.Jim, M.Hicks

C is a *terrible* language:

- Must bypass the type system to do simple things (e.g., allocate and initialize an object).
- Libraries put the onus on the client to do the “right thing” (e.g., check return codes, allocate data of right size, pass in array sizes, etc.).
- Manual memory management leads to leaks, data corruption.
- No information at runtime to do needed checks.(e.g., printf is passed arguments of the right type).
- "Portability" is in the #ifdef's, #defines, and Makefiles.

But C Is Also Very Useful:

Almost every critical system is coded in C:

- ported to lots of architectures.
- low-level control over data structures, memory management, instructions, etc.
- features useful for building device derivers, operating systems, protocol stacks, language runtimes, etc.
- the portability of the world is encoded in .h files

Questions:

- How do we achieve type safety for legacy C code?
- What should a next-generation C look like?

A Number of Recent Projects:

- LCLint, Splint [Evans]
- ESC M3/Java [Leino et al.]
- Prefix, Prefast [MS]
- SLAM [Ball, Rajamani]
- ESP [Das, Adams, Jagannathan]
- Vault, Fugue [Fahndrich, DeLine]
- Metal [Engler]
- CCured [Necula]

General Flavor

- Find bugs & inconsistencies in *real* source code.
  - e.g., Windows, Linux, Office, GCC, etc.
  - buffer overruns, tainted input, protocol violations, etc.
- A variety of analysis techniques.
  - ast analysis, dataflow analysis, type inference, constraint solving, model checking, theorem proving, spell checking,...
- Key needs:
  - minimize "false positives"
    - tool won't be used if it's not finding real bugs.
    - skip soundness, add annotations, add run-time checks, etc.
  - attention to scale
    - modular analysis, avoiding state explosion, etc.
  - good user interface
    - e.g., minimal error traces, integration with build system, etc.

The Cyclone Project

Cyclone is a type-safe dialect of C:

- primary goal: guarantee *fail-stop* behavior.
  - if we can't verify statically, we verify it dynamically.
  - whether or not we issue a warning is heuristic.
- second goal: retain virtues of C
  - syntax and semantics in the spirit of the language.
  - avoid hidden state (i.e., type tags, array bounds).
  - make it easy to interoperate with C (e.g., <kernel.h>).
  - ultimately: attractive for writing systems code.
- final goal: keep verification modular and scalable.
  - want this to be used as part of *every* build.
  - local analysis and inference only.
  - defaults, porting tool to minimize annotation burden.

Cyclone Users

- In-kernel Network Monitoring [Penn]
- MediaNet [Maryland & Cornell]
- Open Kernel Environment [Leiden]
- RBClick Router [Utah]
- xTCP [Utah & Washington]
- Lego Mindstorm on BrickOS [Utah]
- Cyclone on Nintendo DS
- Cyclone compiler, tools, & libraries
  - Over 100 KLOC
  - Plus many sample apps, benchmarks, etc.
  - Good to eat your own dog food…

This Talk

- A little bit about the Cyclone design:
  - Refining C types
  - Flow analysis
  - Type-safe Manual Memory management
- Lessons learned:
  - Theory vs. Practice
  - Why you shouldn’t trust tools
- Where we’re heading:
  - Open, trustworthy analysis framework

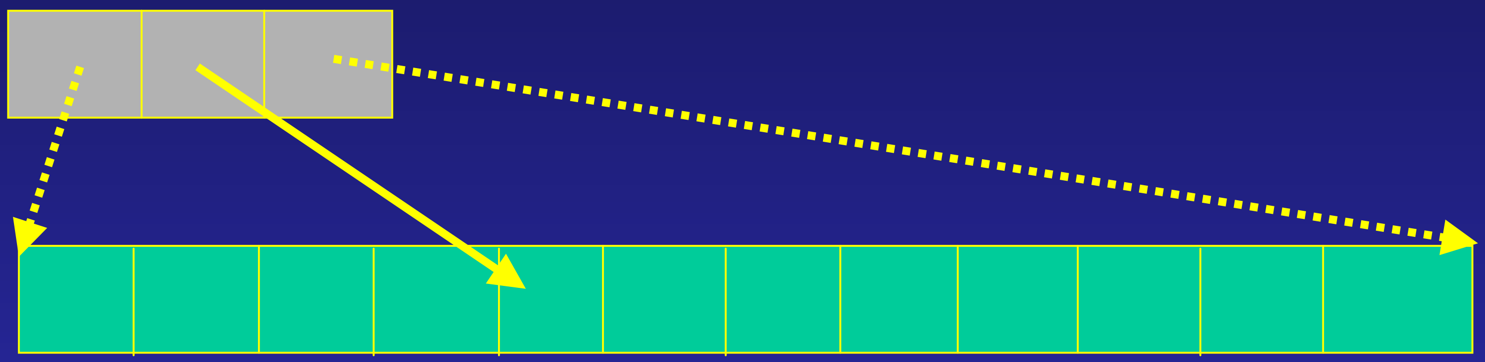
Hello World in Cyclone

```
#include <stdio.h>

int main(int argc, char*@zeroterm *@fat argv)
{
  if (argc != 2) {
    fprintf(stderr,"usage: %s <name>\n",argv[0]);
    exit(-1);
  }
  printf("Hello, %s.\n",*(++argv));
  return 0;
}
```

Fat Pointers:

To support dynamic checks, we must insert extra information (e.g., bounds for an array):



This is similar to what's done in Java, but we need more information to support pointer arithmetic.

Avoiding Overheads:

Dynamic checks make porting from C easy and our static analysis eliminates much of the overhead.

But often programmers want to *ensure* there will be no overhead and no potential failure.

To achieve this, programmers can leverage Cyclone's *refined types* and *static assertions.*

Pointer Qualifiers Clarify

Thin pointers: same representation as C, but restrictions on pointer arithmetic.

`char *`: a (possibly NULL) pointer to a character.

`char *@notnull`: a (definitely not NULL) pointer to a character.

`char *@numelts{c}`: pointer to a sequence of *c* characters.

`char *@zeroterm` : pointer to a zero-terminated sequence.

Fat pointers: arbitrary arithmetic but the representation is different (3 words):

`char *@fat` : a "fat" pointer to a sequence of characters.

`numelts(s)` : returns number of elements in sequence s

Subtyping Is Crucial:

Some Subtyping:

```
@numelts{42} <= @numelts{3}
@notnull <= @nullable
(mutable) <= @const
```

Some *No-check* Coercions:

```
@thin @numelts{42} <:= @fat
@thin @zeroterm <:= @fat @zeroterm
@fat @zeroterm <:= const @fat @nozeroterm
```

Some *Checked* Coercions:

```
@fat <#= @numelts{42}
@nullable <#= @notnull
```

Determining Qualifiers

Programmers:

- provide qualifiers for procedure interfaces

Compiler:

- infers qualifiers for local variables using a constraint-based inference algorithm.
- inserts coercions to adjust where necessary and possible.
- emits warnings for (most) checked coercions.

Porting Tool:

- global analysis tries to infer qualifiers, using only equality constraints (linear time).
- may be unsound(!) but compiler will flag problems

Checked Coercions & Warnings

In Cyclone stdio library:

```
FILE* fopen(const char *,const char *);
int getc(FILE *@notnull);
```

A client of the library:

```
FILE *f = fopen("foo.txt", "r");
c = getc(f);
```

Warning: argument might be NULL – inserting runtime check

Should Be Able to Avoid Warnings:

1. `cyclone -nowarn`

2.
```
FILE @f = (FILE @)fopen("foo.txt", "r");
c = getc(f)
```

3.
```
FILE *f = fopen("foo.txt", "r");
if (f == NULL) {
  perror("cannot find foo.txt\n");
  exit(-1);
}
c = getc(f)
```

Flow Analysis

Simple intraprocedural flow-sensitive, path-insensitive analysis used to determine:

- whether pointer variables are NULL.
  - used to avoid NULL checks, warnings.
- whether variables and fields within data structures are initialized.
  - warning on "bits-only" types, error otherwise.
- unsigned integer inequalities on variables.
  - used to avoid bounds checks, warnings.
- aliasing (essentially k-level with k = 2).
- "noreturn" attribute (e.g., calls exit).

An Example:

```
int strcmp(const char *@fat s1,
           const char *@fat s2) {
  unsigned n1 = numelts(s1);
  unsigned n2 = numelts(s2);
  unsigned n = (n1 <= n2) ? n1 : n2;
  for (int i = 0; i < n; i++) {
    ... s1[i] ... s2[i] ...
  }
    ...
}
```

The analysis is not able to prove that `i` is in bounds, so it inserts run-time tests...

Using Static Asserts

```
int strcmp(const char *@fat s1,
           const char *@fat s2) {
  unsigned n1 = numelts(s1);
  unsigned n2 = numelts(s2);
  unsigned n = (n1 <= n2) ? n1 : n2;
  @assert(n <= n1 && n <= n2);
  for (int i = 0; i < n; i++) {
    ... s1[i] ... s2[i] ...
  }
    ...
}
```

Here, we have
`n1 == numelts(s1) &`
`n <= n1 &  i < n`

In Practice:

- Initial code has lots of dynamic checks.
  - Choice of warning levels reveals *likely* points of failure.
- Two options:
  - Turn up knob on analyses
    - e.g., explore up to K paths
  - Refine types, add assertions
    - Programmer intensive

In either case, programmer views task as *optimizing* code when in fact, they're providing the important bits of a proof of safety.

One Big Wrinkle: Order

Order of evaluation is not specified for many compound expressions.

Consider: $e(e_1,e_2,\ldots,e_n)$

- Worst case: compiler could evaluate each expression in parallel.
- Even if you assume compiler does some permutation, you still have (n+1)! orderings.
- Could calculate all flows and then join, but that's too expensive in practice.

Solutions:

Originally, we had a sophisticated, iterative analysis to deal with the ordering issue.

- Complicated, difficult to maintain.

Now we force the order of evaluation.

- Greatly simplifies the analysis.
- Very little perceived loss in performance.
- But confuses GCC in some instances (e.g., self-tail calls.)

Moral: shouldn't be afraid to change the language to suit verification task.

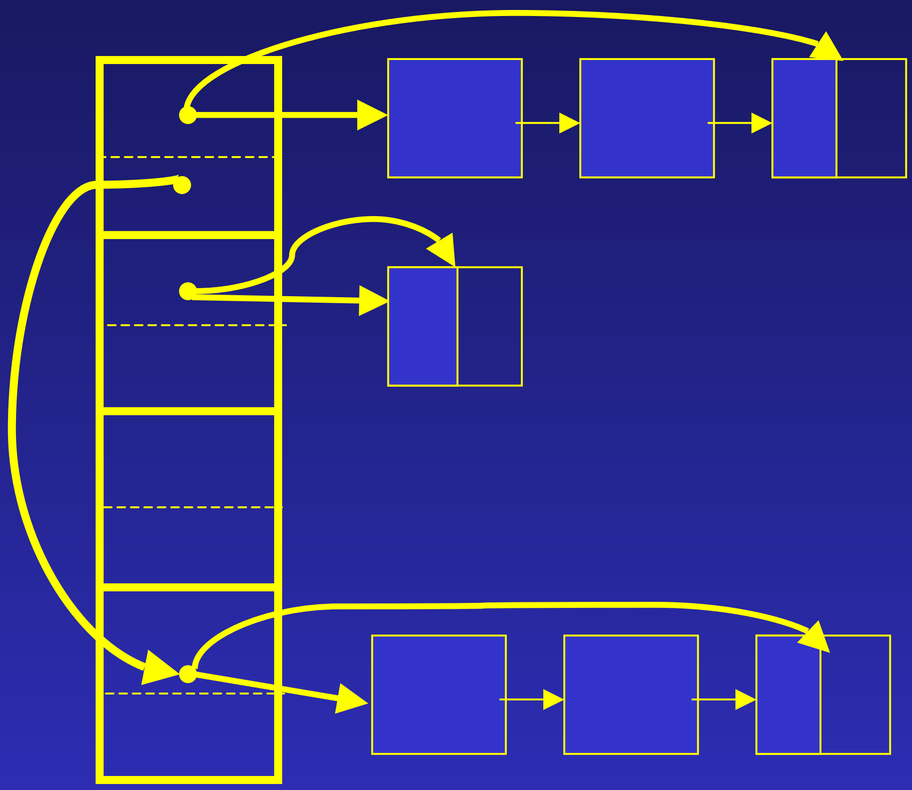
Other Cyclone Features:

- Unions
  - `union Foo {int x; float y;};`
    - can read or write either element
  - `union Bar {int *x; float y;};`
    - can write either element, but only read float
  - `@tagged union Baz {int *x; float y;};`
    - can read/write, but extra tag is inserted
- Parametric Polymorphism, Pattern-Matching, Existential Types, and Exceptions
- Limited dependent types over integer expressions (*a la* Dependent ML)
- Region-based memory management.

Region Goals

- Provide some mechanism to avoid GC.
  - no hidden tags.
  - no hidden pauses.
  - small run-time.
  - but ensure safe pointer dereferences.
  - scalable and modular analysis.
- Regions (a la Tofte & Talpin) fit the bill.
  - group objects with similar lifetimes into regions.
  - put region names on pointer types (**`int *`r`**).
  - track whether or not a region is live (effects).
  - allow dereferencing a pointer only if region is live.

Runtime Organization



runtime stack

Regions are linked lists of pages.

Arbitrary inter-region references.

Similar to arena-style allocators.

The Good News

Stack allocation happens a lot in C code.

- Thread local
- Cheap

Lexical region allocation works well for:

- "callee" allocates idioms (e.g., rgets)
- temporary data (e.g., environments)

Automatic deallocation.

All checks are done statically.

Real-time memory management.

The Bad News:

LIFO region lifetimes are too strict.

- No “tail-call” for regions.
- Lifetimes must be statically determined.
- Consider a server that creates some object upon a request, and only deallocates that object upon a subsequent request…

Creating/destroying a region is relatively expensive compared to malloc/free.

- Must install exception handler.
- Makes sense only when you can amortize costs over many objects.

To Address Shortcomings

- Unique pointers
  - Lightweight when compared to a region.
  - Can deallocate (free) at will.
  - But you can’t make a copy of the pointer.
- Dynamic regions
  - Can allocate or deallocate the arena at will.
  - Use a unique pointer as a “key” for access.

The combination actually subsumes lexical regions and provides the flexibility needed to optimize memory management for clients.

The Flexibility Pays: MediaNET

TTCP benchmark (packet forwarding):

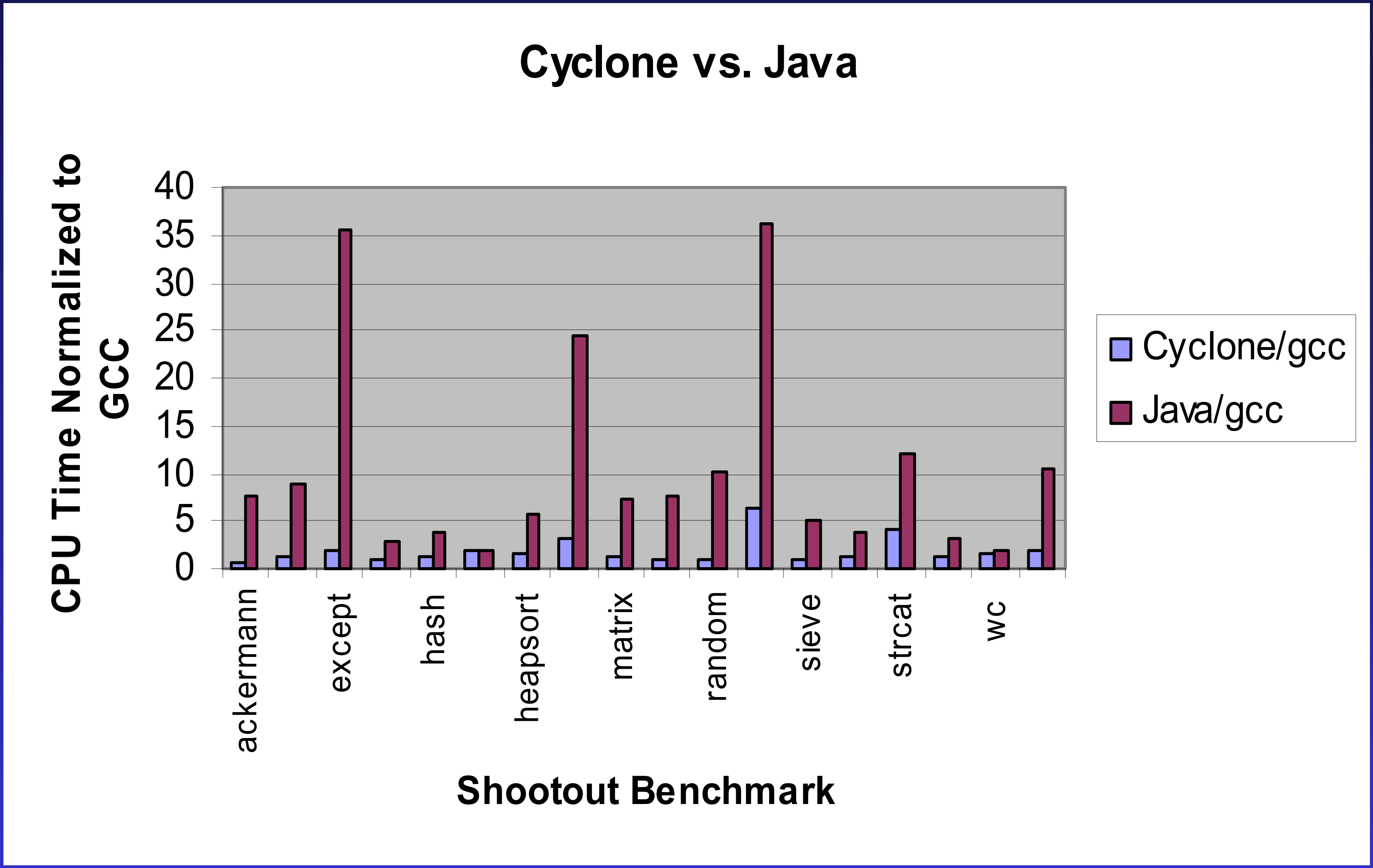
Cyclone v.0.1 (lexical regions & BDW GC)

- High water mark: 840 KB
- 130 collections
- Basic throughput: 50 MB/s

Cyclone v.0.5 (unique ptrs + dynamic regions)

- High water mark: 8 KB
- 0 collections
- Basic throughput: 74MB/s

Cyclone vs. Java



Comparing to Java

Program	Cyclone/gcc	Java/gcc
Ackermann	0.75	7.57
Ary3	1.21	8.85
Except	2.02	35.45
Fibo	1.00	2.86
Hash	1.35	3.83
Hash2	1.80	1.82
Heapsort	1.58	5.84
Lists	3.04	24.33
Matrix	1.24	7.30
Nestedloop	0.99	7.72
Random	0.99	10.11
Reversefile	6.45	36.28
Sieve	0.99	5.17
Spellcheck	1.15	3.67
Strcat	4.22	12.00
Sumcol	1.20	3.21
Wc	1.73	2.02

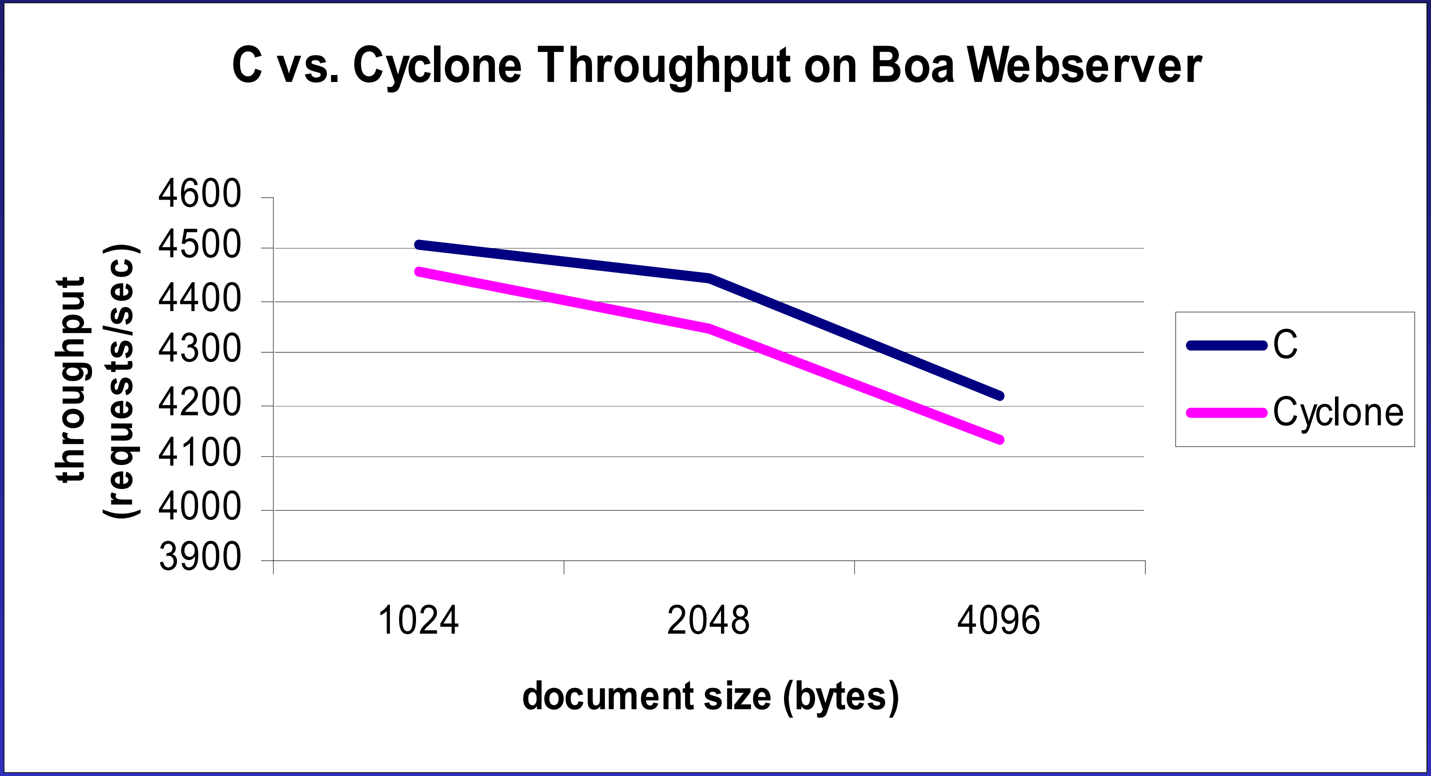
Bagley's Language Shootout comparing Sun's Java 2 RTE v1.4.1_03-b02.

CPU time normalized to gcc's.

On average:
Cyclone: 1.87
Java : 10.47

Macro-benchmarks:

We have also ported a variety of security-critical applications where we see little overhead (e.g., 2% for the Boa Webserver.)



Some Lessons Learned

- Don’t try to “fix” C:
  - Example: auto-break in switch cases
  - Instead, explicit “fallthru” annotation.
- There is no ANSI C:
- People matter, performance doesn’t
  - Porting code is still too painful.
  - Error messages are crucial.
- Interoperability is crucial.

Very Important Lessons

The compiler at this point is huge:

- ~ 50KLOC
- We kept finding subtle bugs in the analyses (c.f., order of evaluation.)
- Is it trustworthy?

Furthermore, there's no end to the refinements needed.

- Can we simplify the approach?

Current Thrust:

We're currently working on a more trustworthy, extensible infrastructure:

- As in ESC and SPLint:
  - Compiler computes verification conditions (using strongest-post-conditions.)
  - Infers some minimal loop invariants, but programmers can supply better invariants.
  - Uses an internal theorem prover to discharge most of the VCs.
- Unlike ESC/SPLint:
  - The prover is *not* trusted: must give witness.
  - If we can't prove it, then we do the run-time check.

Longer Term:

No need to stick with our prover:

- Should be able to discharge VCs using any plug-in prover, as long as it can produce a witness that we can check.
- In fact, should be able to discharge some proofs by hand!

Problem:

- Very few sound, witness-producing provers with useful decision procedures.
- For instance, few of them deal with machine arithmetic, and those that do don’t scale well.

The Program Logic

Another issue is fixing the logic to deal with issues such as memory mgmt.

The usual encoding of memory as a big array is insufficient for many reasons.

Hoping to leverage the emerging spatial logics (e.g., Reynolds & Ohearn's BI).

Open question: decision procedures.

Summary:

Cyclone is a type-safe dialect of C

- *Much* better performance than previous type-safe languages.
- In large part because programmers can tune performance (erm, safety) by adding additional information.
- More suited to writing new systems code than porting legacy code.
- Our ultimate goal is to make it possible (but not necessary) to eliminate all run-time checks.

More info...

www.eecs.harvard.edu/~greg/Cyclone