

# Precise Garbage Collection for C

Jon Rafkind\*

Adam Wick<sup>+</sup>

John Regehr\*

Matthew Flatt\*

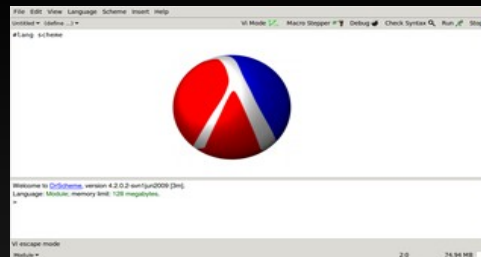
\* University of Utah

<sup>+</sup> Galois, Inc.

# Motivation

## C

- Used to implement important programs
  - Web servers
  - Operating systems
  - Virtual machines



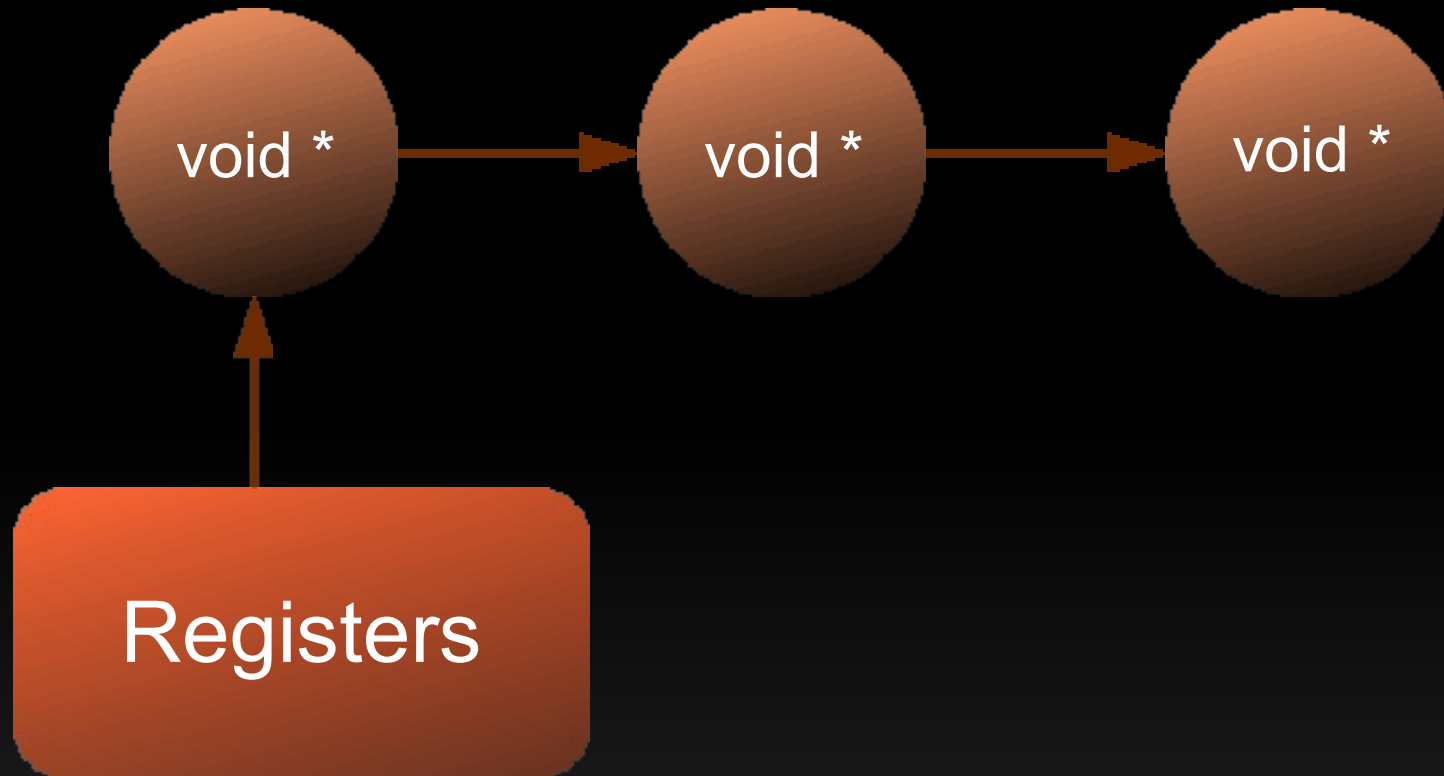
- Memory management is difficult to get right

# Conservative GC

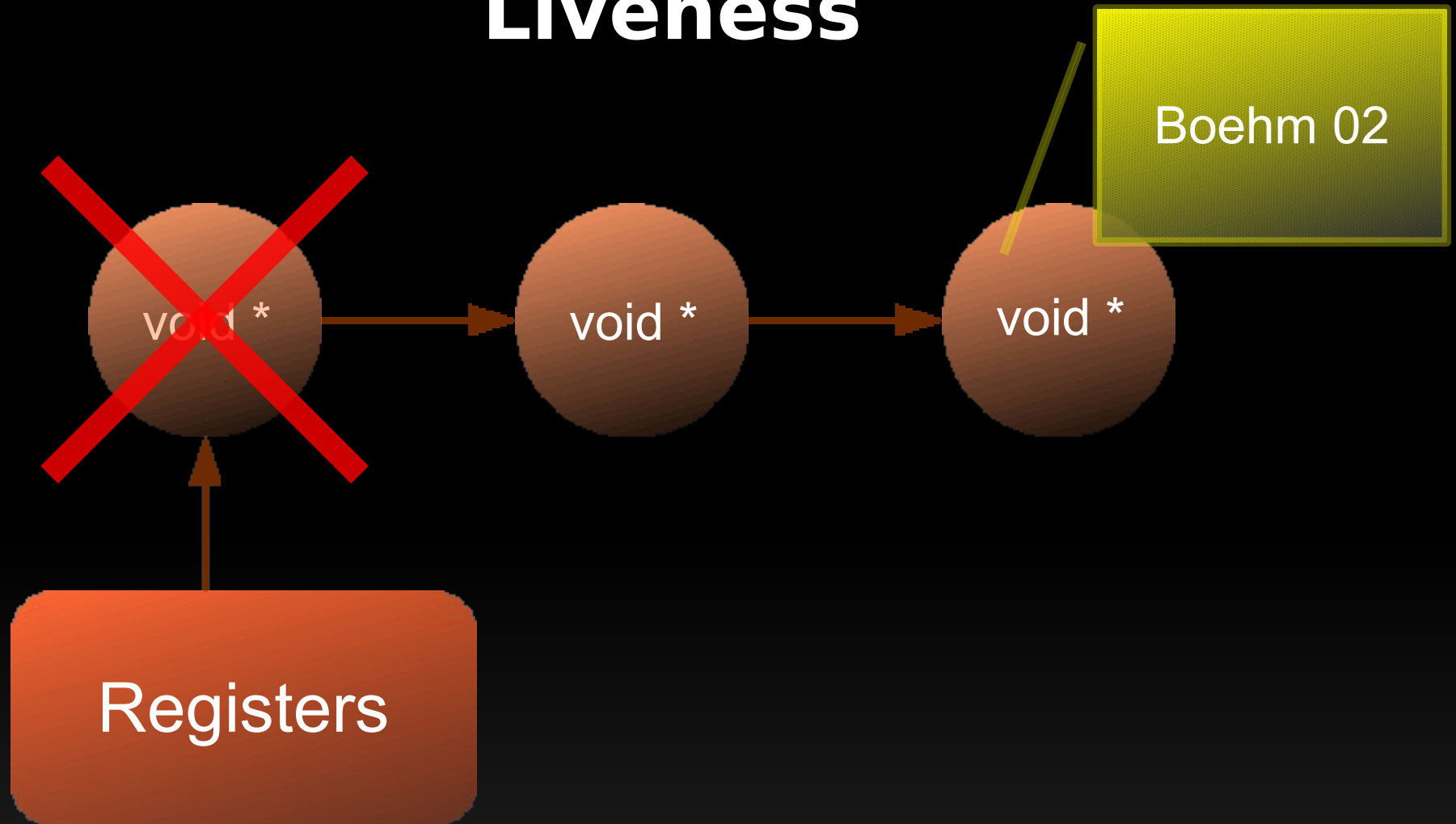
## Scans for pointer values

- Works well for weakly typed languages
  - Objects are stored in memory as sequences of bytes
- Unobtrusive to the original program
- Works for many C programs

# Liveness



# Liveness



Long running programs have possibility of liveness errors

# Liveness inaccuracy

Machine state outside the realm of C

Registers

Non-pointer values as pointers

void \* x

0xba324100

long y

0xbd100000

# Mitigating liveness

Atomic blocks (memory regions with no pointers)

Custom tracers

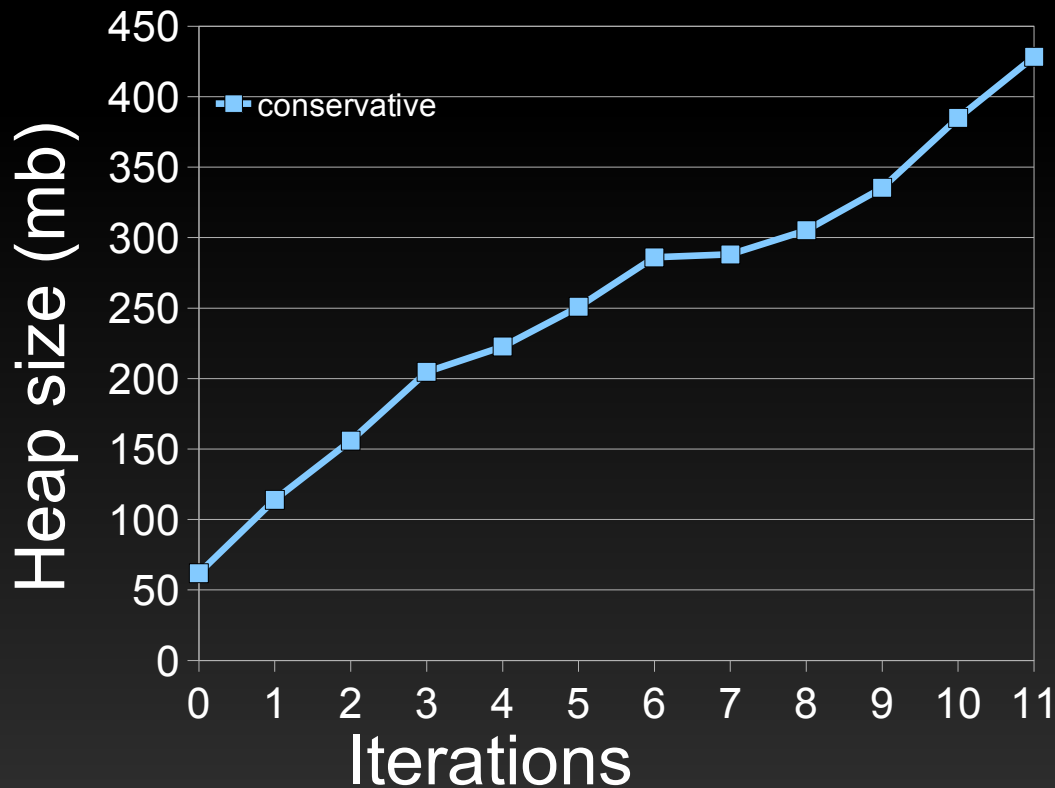
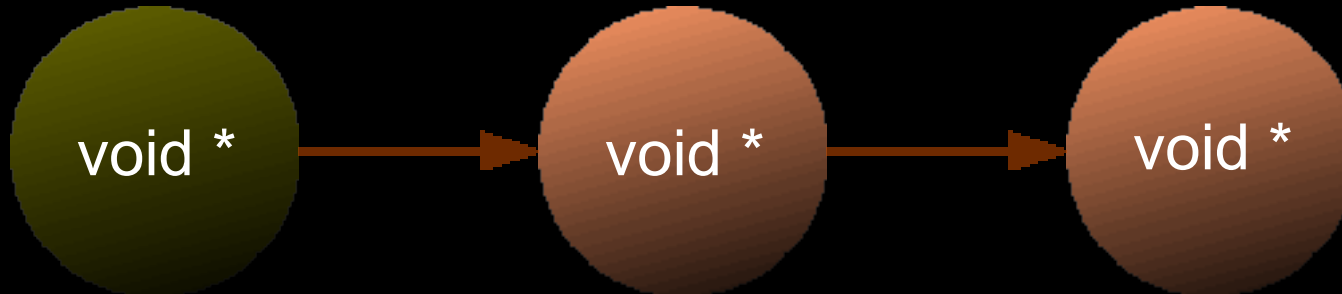
Liveness errors have non-zero  
chance of occurring

# Memory leaks

thread 1

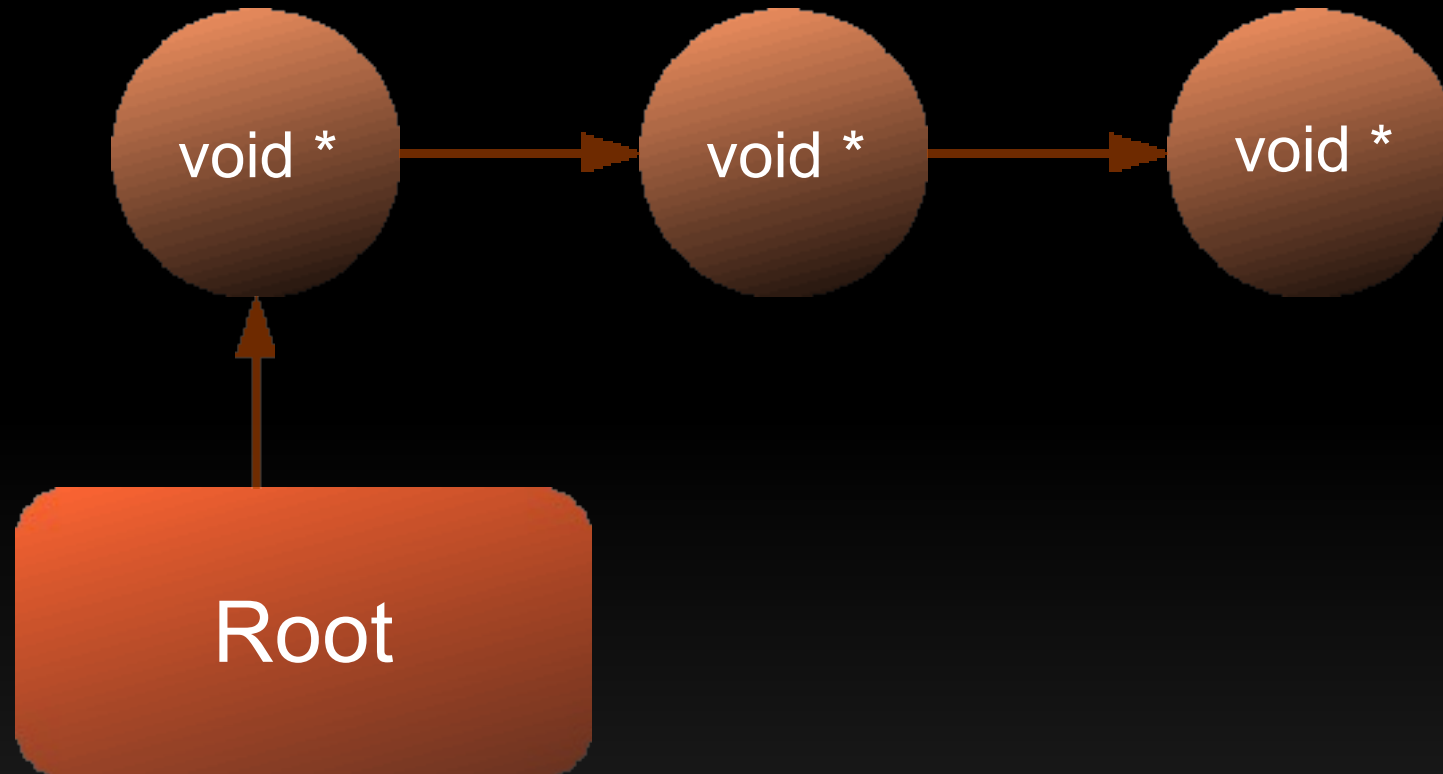
thread 2

thread 3





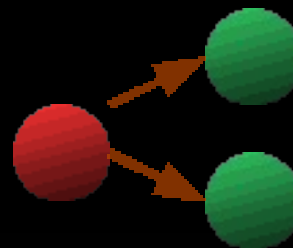
# Precise GC



# Precise GC

Liveness is apparent at the source level

```
struct painter * p = ...;  
p->brush = ...;  
p->color = ...;
```



```
p->brush = 0;
```



# Precise GC

Successfully added a precise garbage collector to *long running* C programs

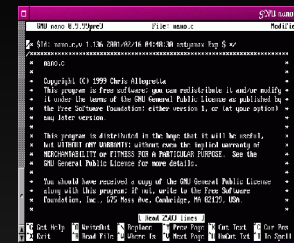
PLT Scheme



ZSnes Emulator



Nano text editor



Linux



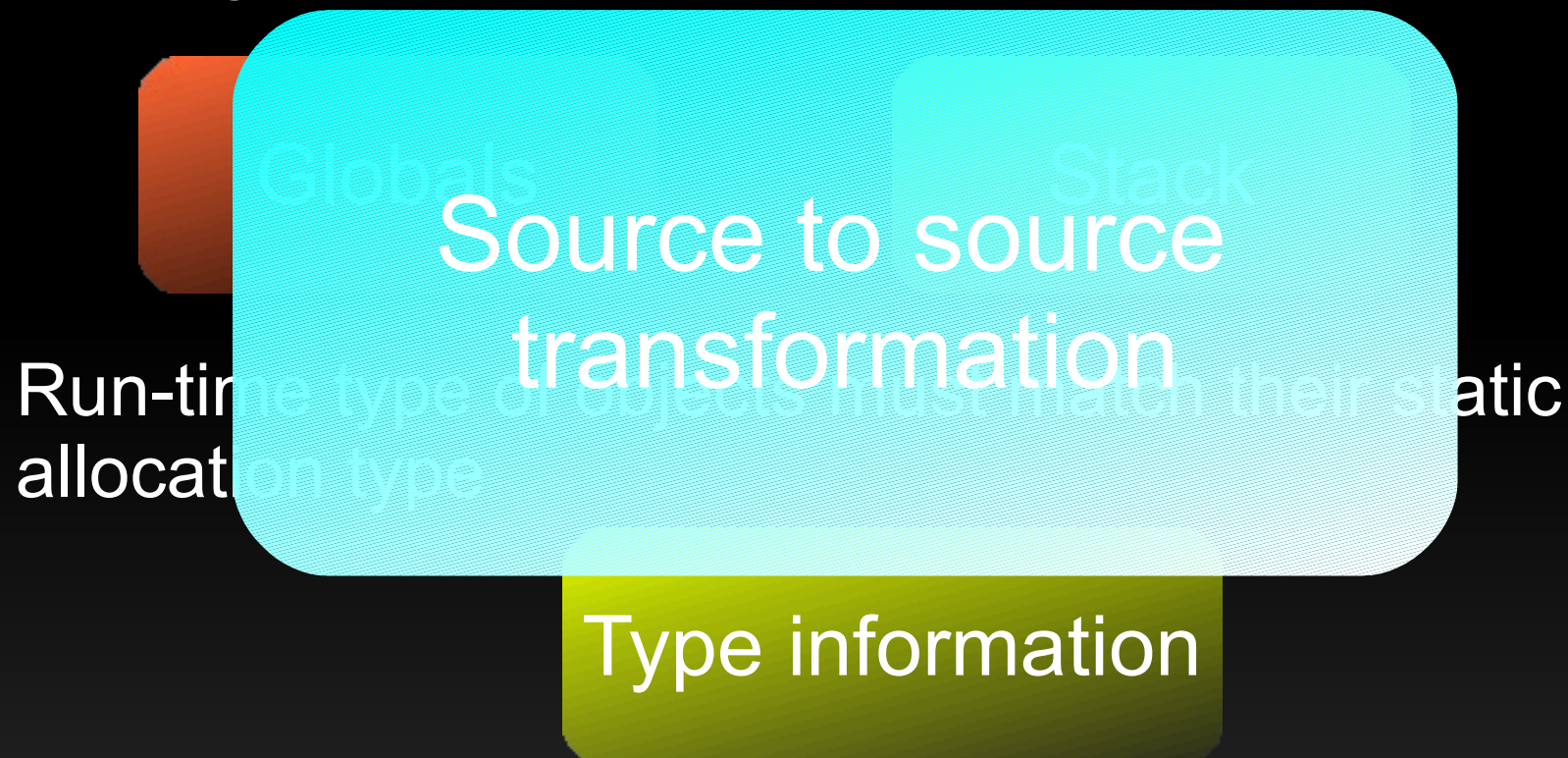
# Precise

All live objects must be reachable through pointers starting from roots

Run-time type of objects must match their static allocation type

# Precise for C

All live objects must be reachable through pointers starting from roots



# Insight

C programs provide enough type information to precisely distinguish pointers from non-pointers

# Transformation

- Register global variables as roots
- Dynamically register stack variables
- Rewrite allocations with run-time information
- Add traversal routines for structures

# Precise for C

All live objects must be reachable through pointers starting from roots

Globals

Stack

Run-time type of objects must match their static allocation type

Type information



# Roots

```
int * counter;  
int * maker(int * value){  
    int * x = malloc(10);  
    *x = *value;  
    *counter += 1;  
    return x;  
}
```

# Shadow stack

Henderson 02

```
int * counter;  
int * maker(int * value){  
  
    int * x = malloc(10);  
    *x = *value;  
    *counter += 1;  
  
    return x;  
}
```

```
void * frame[3];  
frame[0] = POINTER;  
frame[1] = &x;  
frame[2] = &value;  
GC_set_stack(frame);
```

# Shadow stack

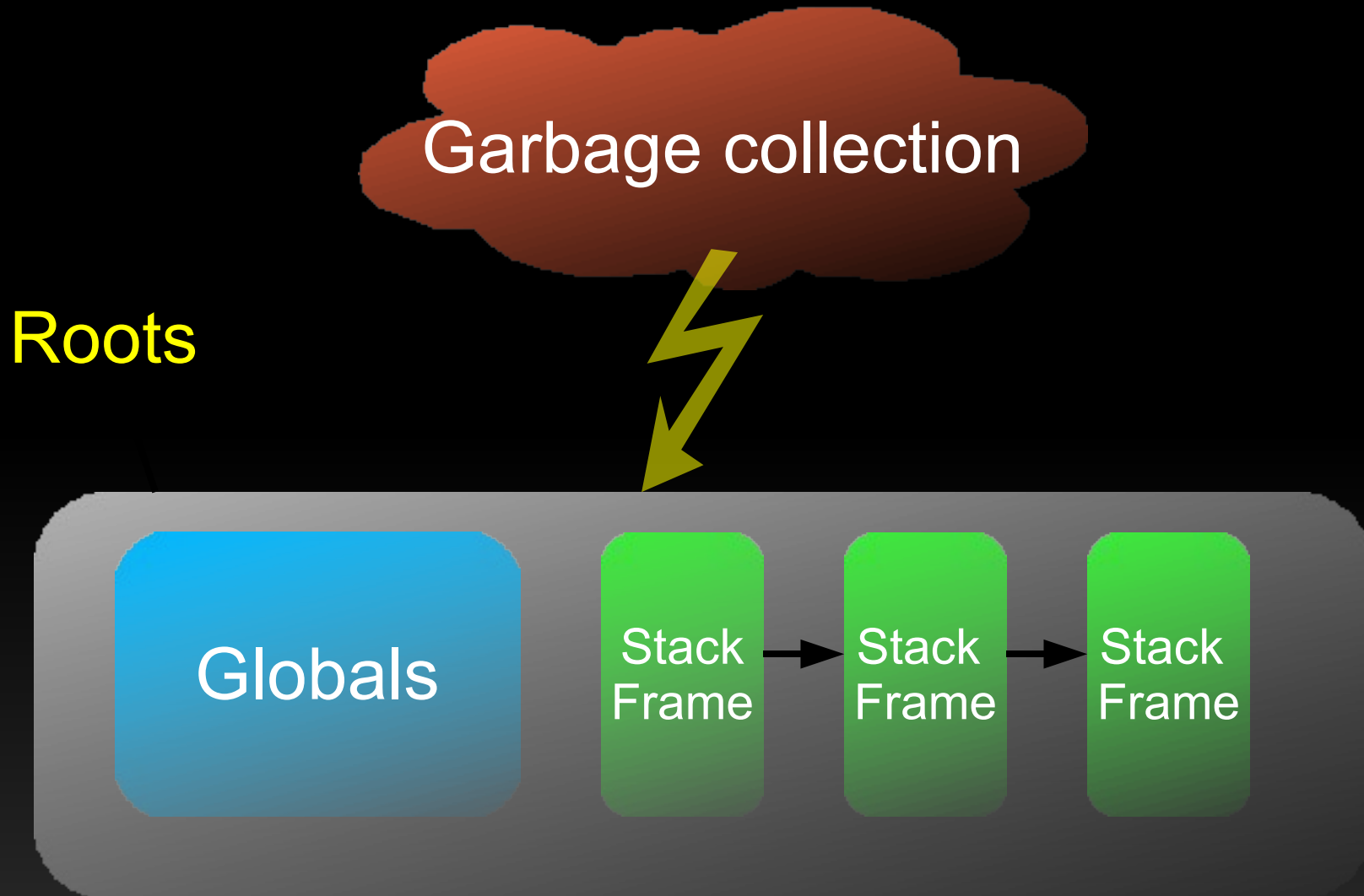
Henderson 02

```
int * counter;  
int * maker(int * value){  
    int * x = GC_malloc(10);  
    *x = *value;  
    *counter += 1;  
    return x;  
}
```

```
void * frame[3];  
frame[0] = POINTER;  
frame[1] = &x;  
frame[2] = &value;  
GC_set_stack(frame);
```

```
GC_set_stack(last_stack);
```

# Shadow stack

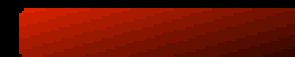


# Stack optimizations

Call graph analysis  
detects functions that do  
not need shadow stacks



Potentially  
allocates



Does not  
allocate

Need shadow  
stacks

Don't need  
shadow stacks

Gzip: 80% non-allocating  
H264: 40% non-allocating

# Precise for C

All live objects must be reachable through pointers starting from roots

Globals

Stack

Run-time type of objects must match their static allocation type

Type information

# Allocation rules

Heuristic used to statically detect run-time type

```
var = malloc(<expr>);
```

*sizeof(t)* – allocate single t structure

*sizeof(t) \* e* – allocate array of t structures

*sizeof(t\*) \* e* – allocate pointer array

*e* – allocate atomic block

# Transforming allocations

```
var = malloc(sizeof(struct cat));
```

*sizeof(t)* – allocate single t structure

*sizeof(t) \* e* – allocate array of t structures

*sizeof(t\*) \* e* – allocate pointer array

*e* – allocate atomic block



```
var = GC_malloc(gc_tag_struct_cat, sizeof(struct cat));
```



# Traversal functions

```
struct cat {  
    int * paws;  
    char * nose;  
};
```

Used during mark phase



```
void gc_struct_cat_mark(struct cat * c){  
    GC_mark(c->paws);  
    GC_mark(c->nose);  
}
```

Generate traversal functions



```
void gc_struct_cat_repair(struct cat * c){  
    GC_repair(&c->paws);  
    GC_repair(&c->nose);  
}
```

Updates references to moved objects



# Result of transformation

The transformed source program will operate semantically equivalent to what it did before and is capable of supporting a moving collector

# C Support

## Internal pointers

```
Int * x = obj->x;
```

## Pointer arithmetic

```
char * p = ...;  
while (*p){  
    p++;  
}
```

# Unions

Active variant is tracked

```
union object {  
    int value;  
    void * info;  
};
```

```
union object o;  
o.value = 1;  
GC_autotag(&o, 0);  
o.info = get();  
GC_autotag(&o, 1);
```

# External libraries

Libraries that store pointers

- Annotate objects that should not be moved by the collector

Memory allocated by libraries

- Safe to use because the GC will ignore it

# Unsupported C

Pointer obfuscation

```
int * p = malloc(20);
```

GC cannot traverse  
the pointer

Constructing pointers

```
int * x = (int*) 0x323429;
```

Liveness issues

Ruby

Pointers as integers

```
long make(){  
    return malloc();
```

GC cannot find pointer

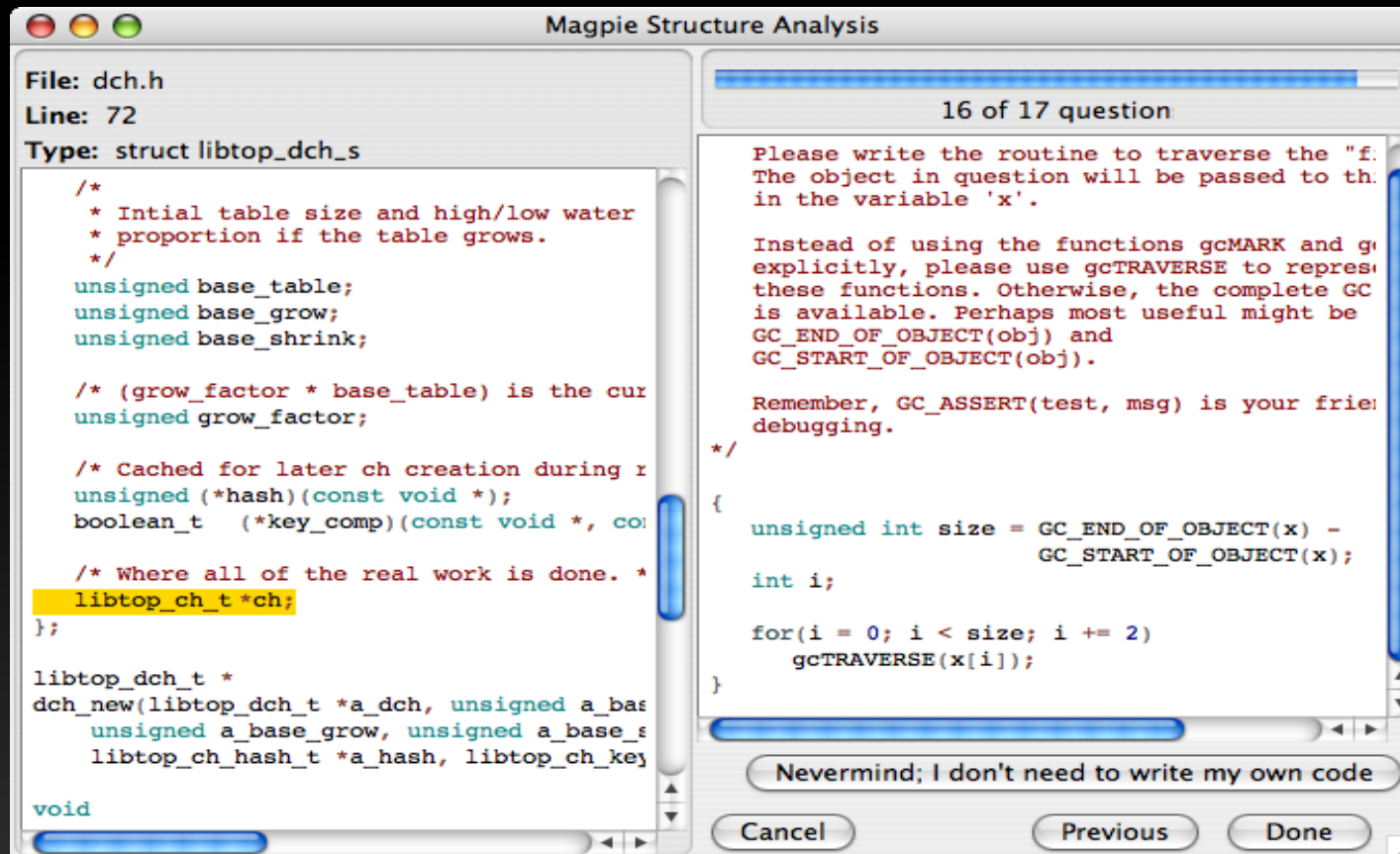
Perl

Arrays of open sized  
arrays

```
struct foo{  
    GC doesn't know how large  
    individual elements are
```

# Tool support

GUI that verifies allocation heuristic and traversal functions



# Tool support

Two tools that can be used on C programs

- ~2500 line Ocaml program based on CIL (Necula 2002)
- Capable of parsing the Linux kernel source
- Fully automatic



# Benchmarks

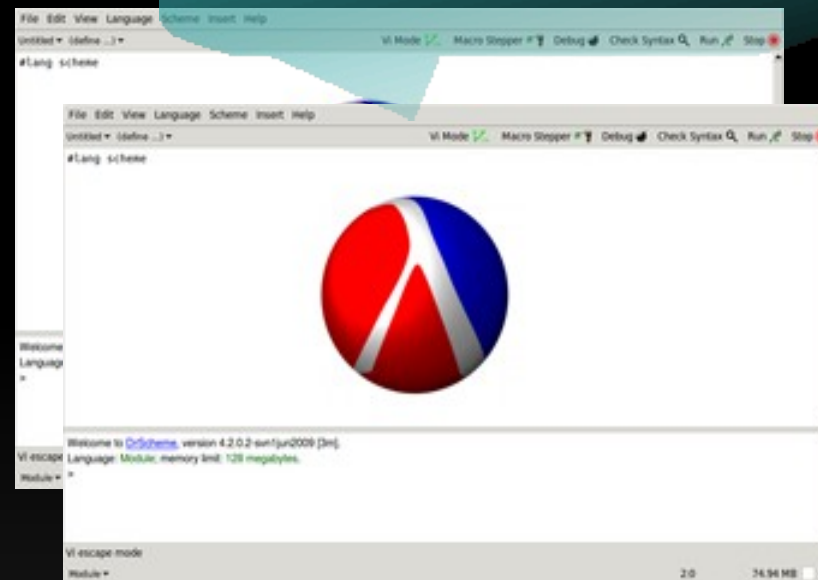
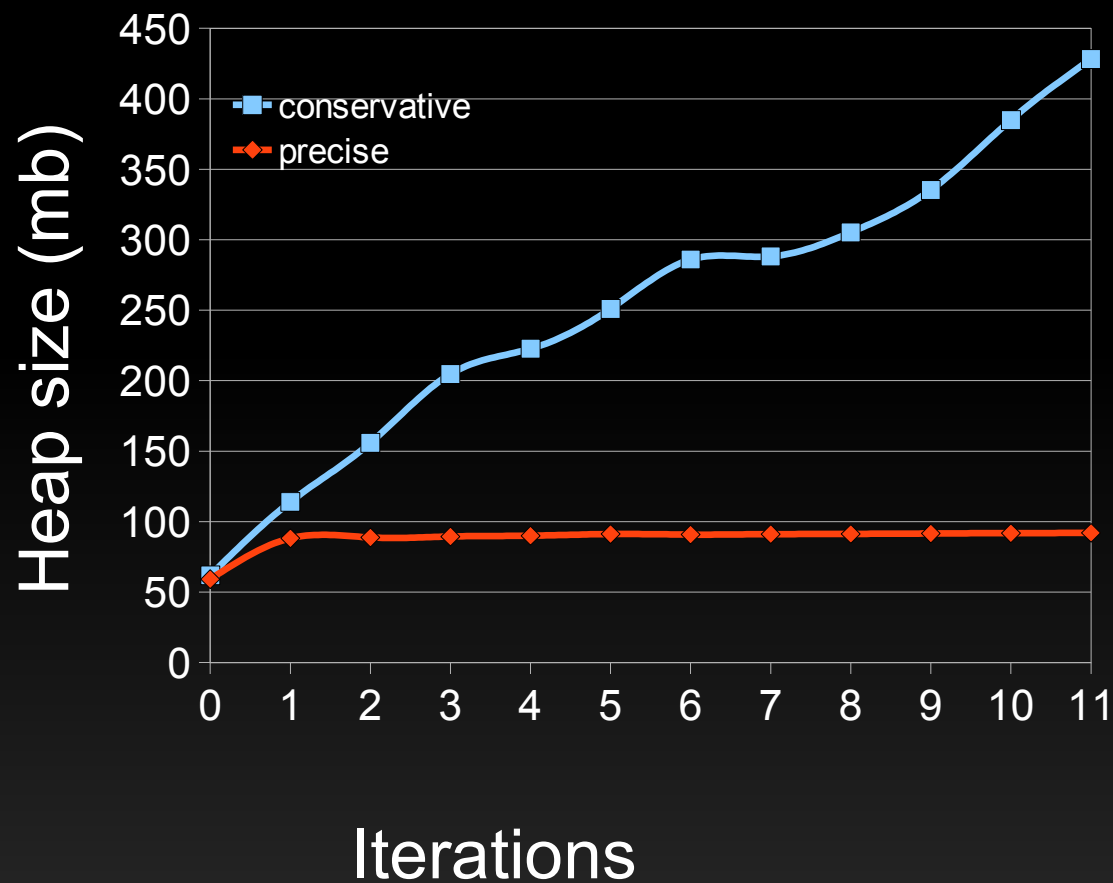
Precise vs Conservative

- memory
- runtime

Source transformation overhead

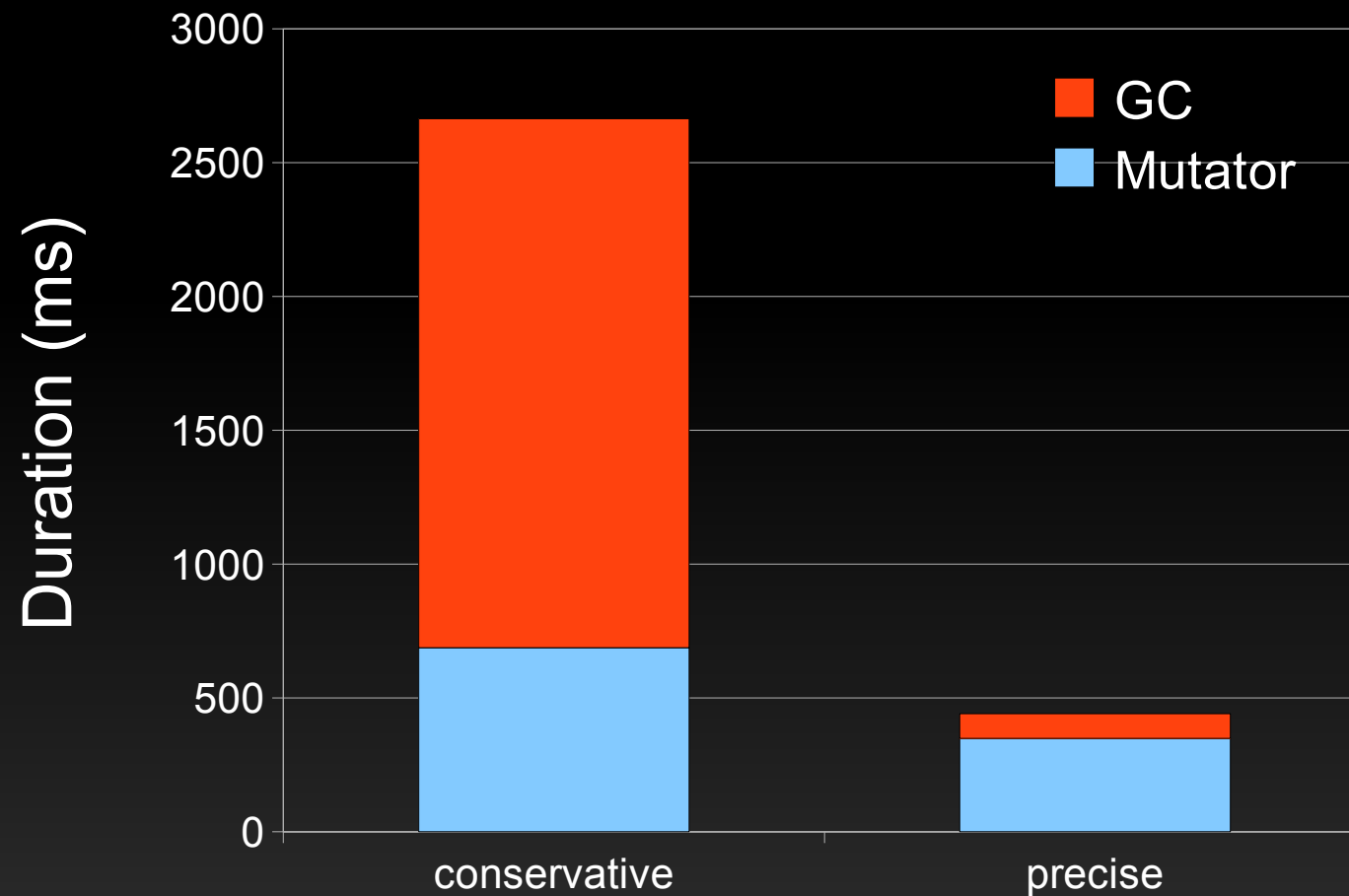
# Precise is bounded

Repeated executions  
of drscheme



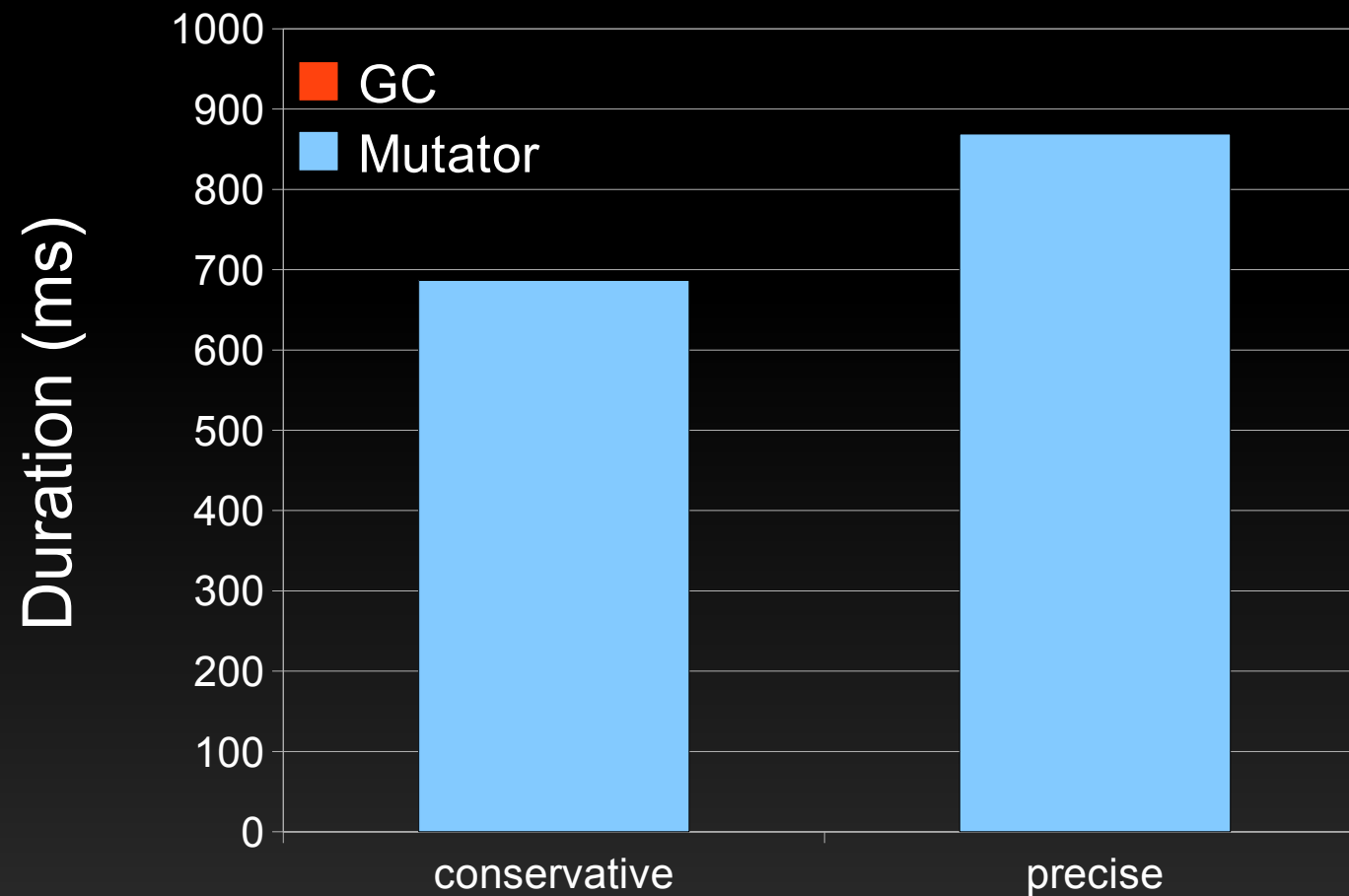
# PLT Scheme Performance

Benchmark: CPStack



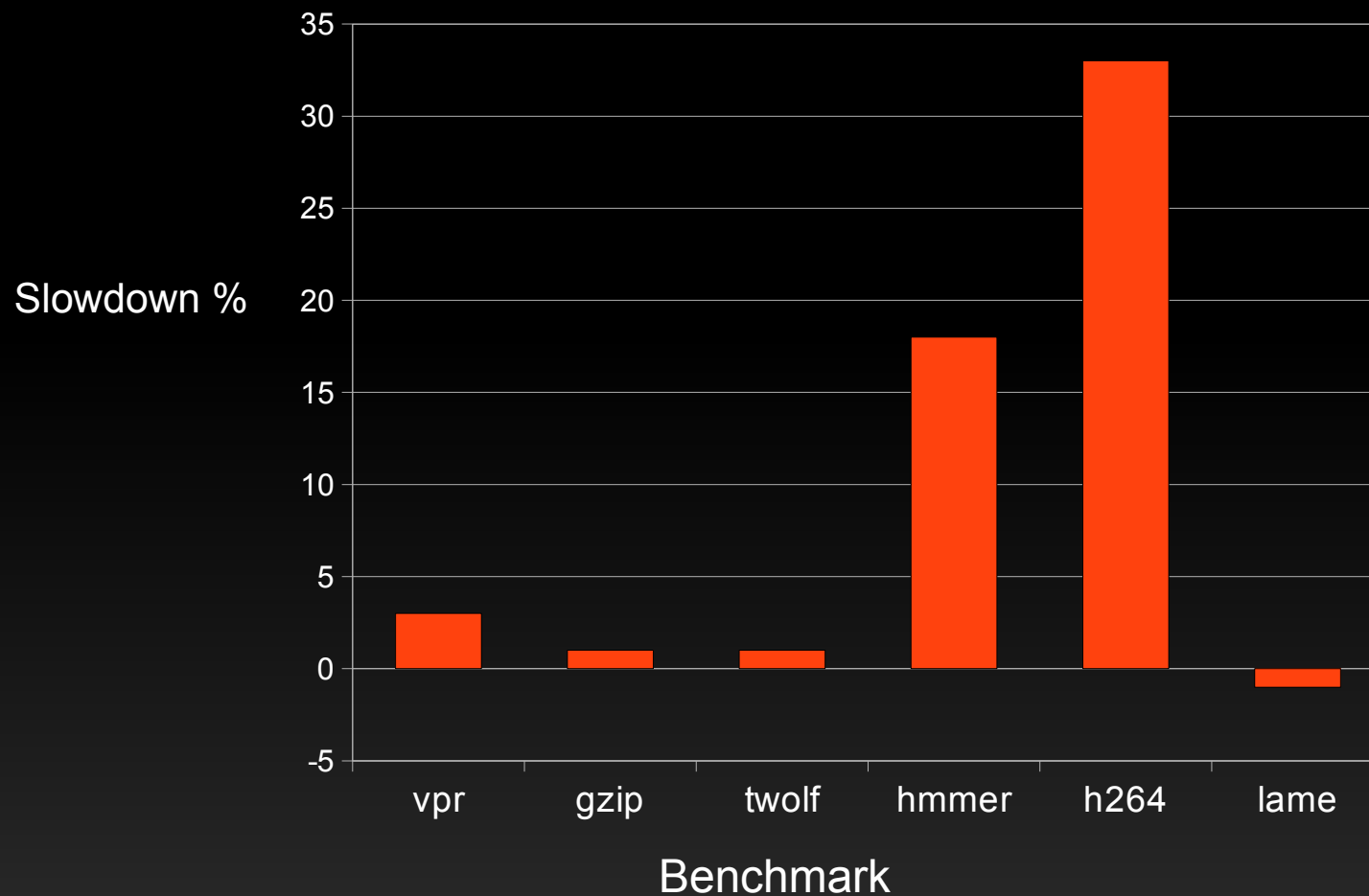
# PLT Scheme Performance

Benchmark: Takl



# GC overhead

## Spec 2k benchmarks



# Linux kernel

- Test the limits of precise for C
- Longest running program in a system
- Harsh C environment

# Linux kernel: Implementation

Transformed 74,975 lines of C code

Ext3 and ipv4

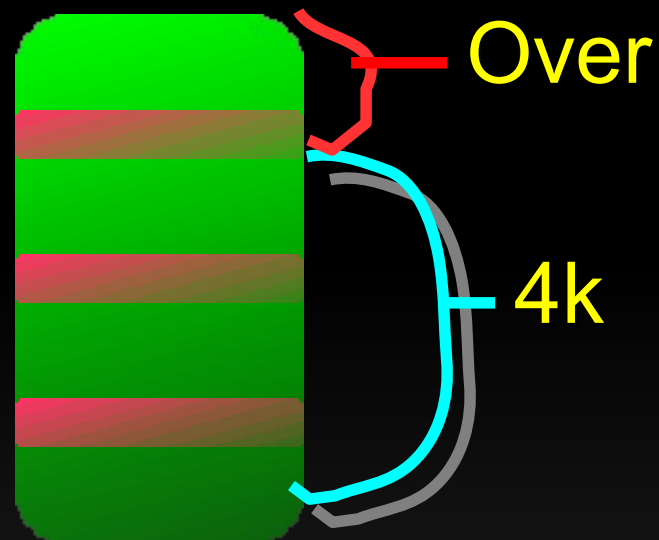
85 manual changes

GC is non-moving / non-generational

Interface to non-transformed  
modules

# Linux kernel: Stack

Stack limit reached with additional shadow stack information





# Linux kernel: Memory regions

## Separate memory regions

kmalloc

Physically contiguous

vmalloc

Virtually contiguous

GC only returns physically contiguous memory

# Linux kernel: heuristics

Allocation heuristic broken

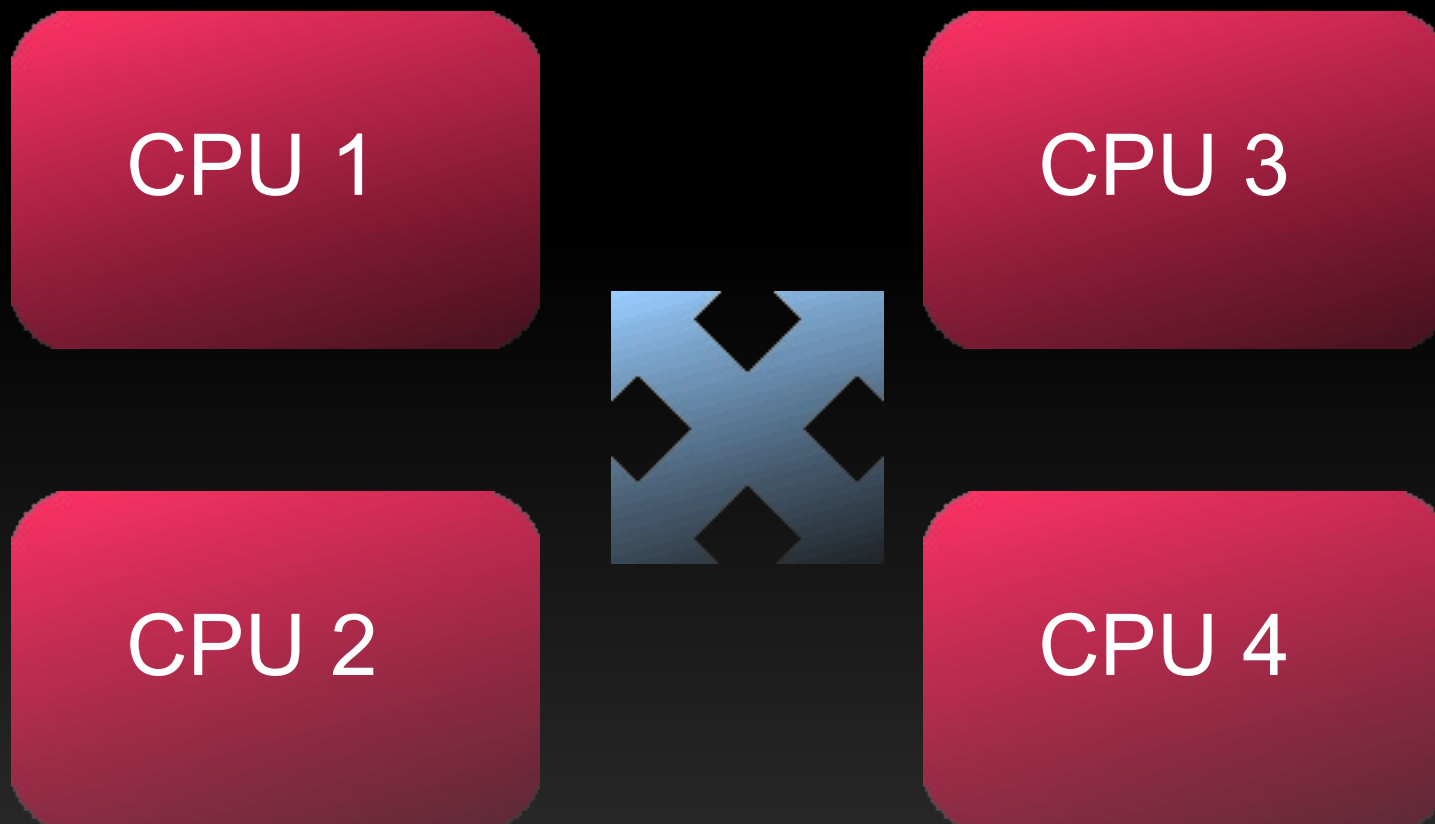
$X = \text{kmalloc}(\text{sizeof}(\dots))$

$Y = \text{kmem\_cache\_alloc}(\text{cache})$

Type of  $Y$  is cannot be discerned

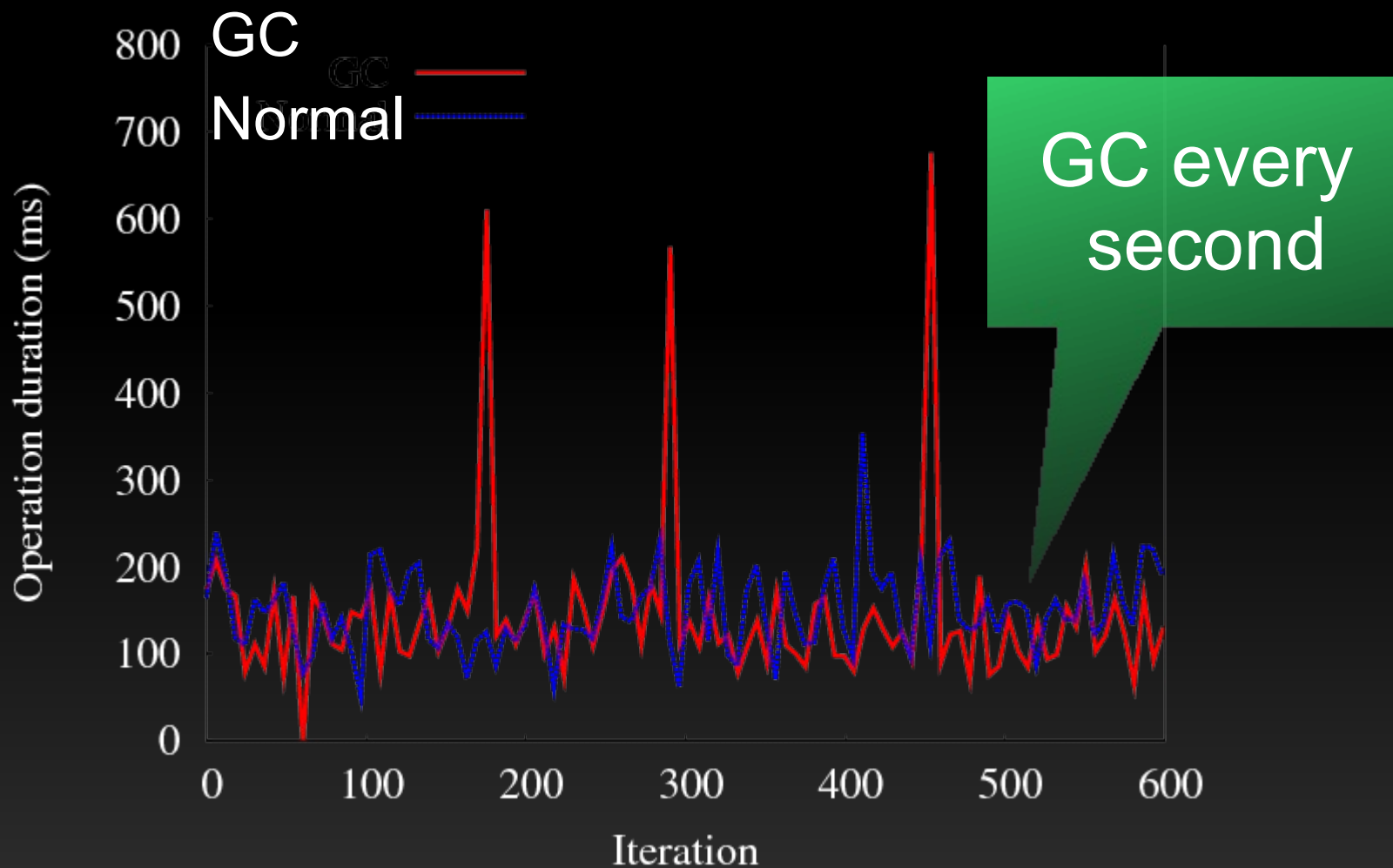
# Linux kernel: finalizers

RCU (Read-Copy-Update) is a finalizer mechanism in Linux



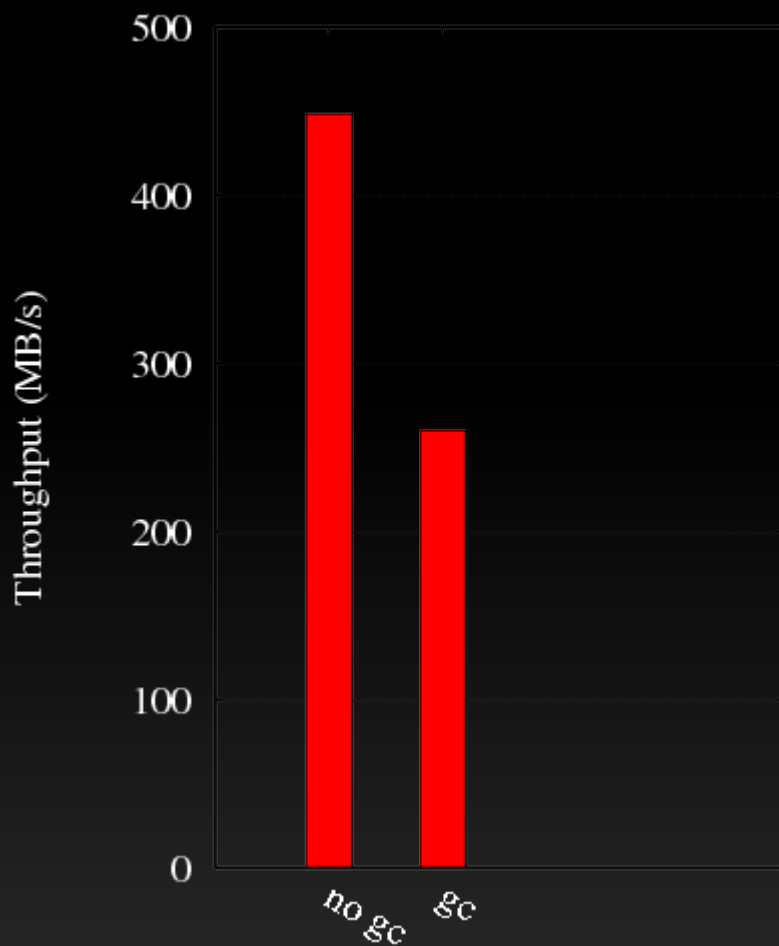
# Performance

## dbench filesystem benchmark

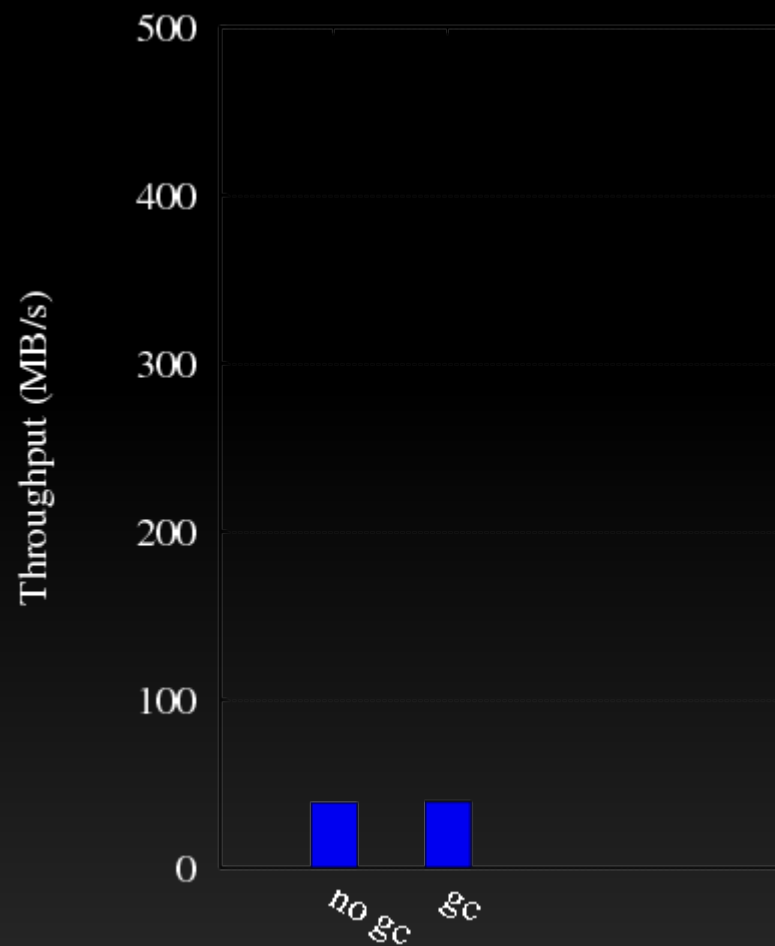


# Performance

dd on memory  
filesystem



dd on disk  
filesystem



# Conclusions

- Precise garbage collection is a practical for C because most C programs embed enough information required to extract the required properties
- Long running programs benefit from precise garbage collection

Thank you  
rafkind@cs.utah.edu

# Precise Garbage Collection for C

Jon Rafkind



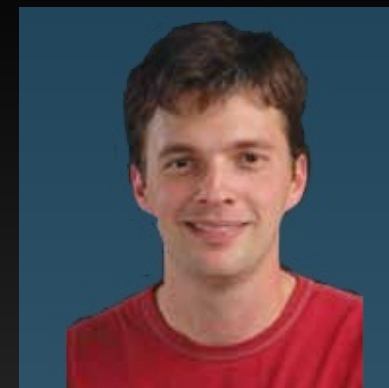
Adam Wick



John Regehr



Matthew Flatt





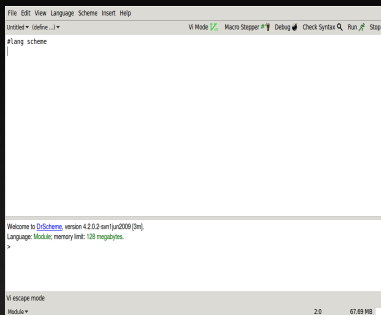


# C is a weakly typed language

Magpie

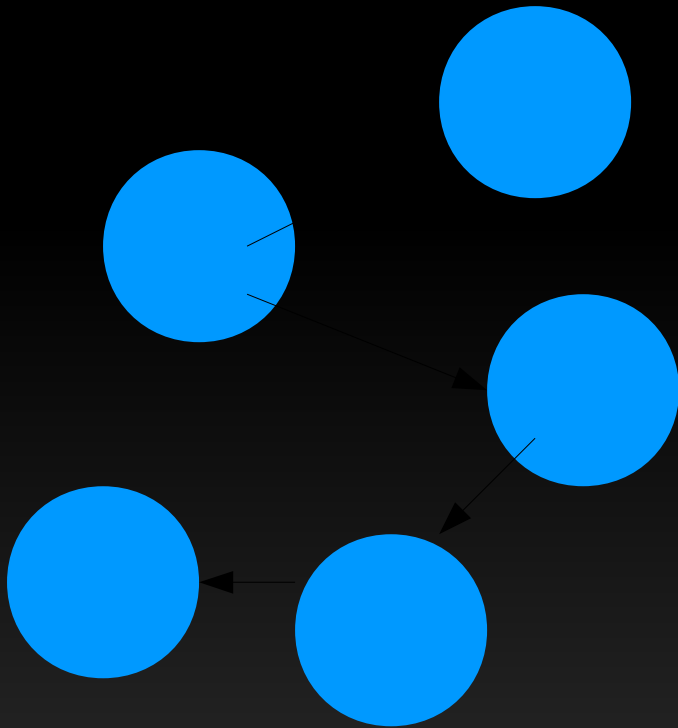
Written a tool that makes C programs use precise garbage collection

Applied to PLT Scheme, Linux kernel, benchmarks

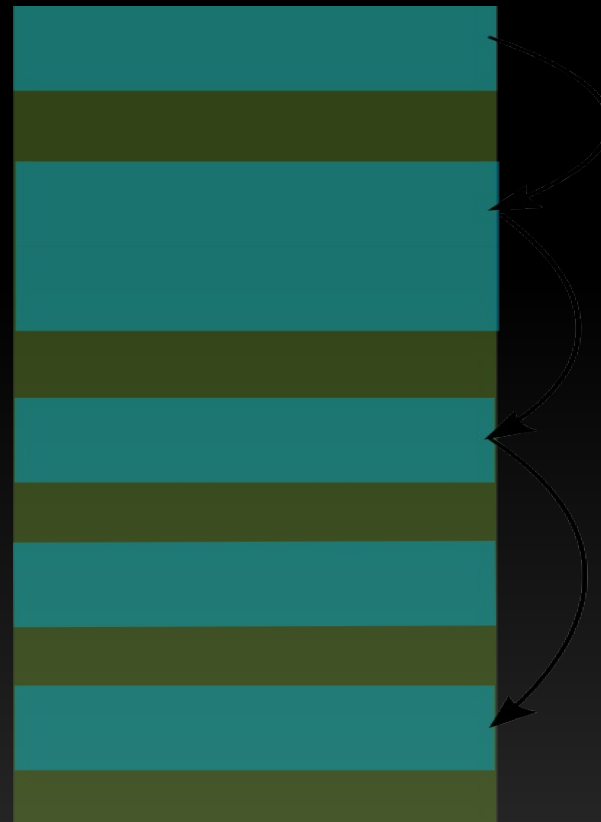


spec

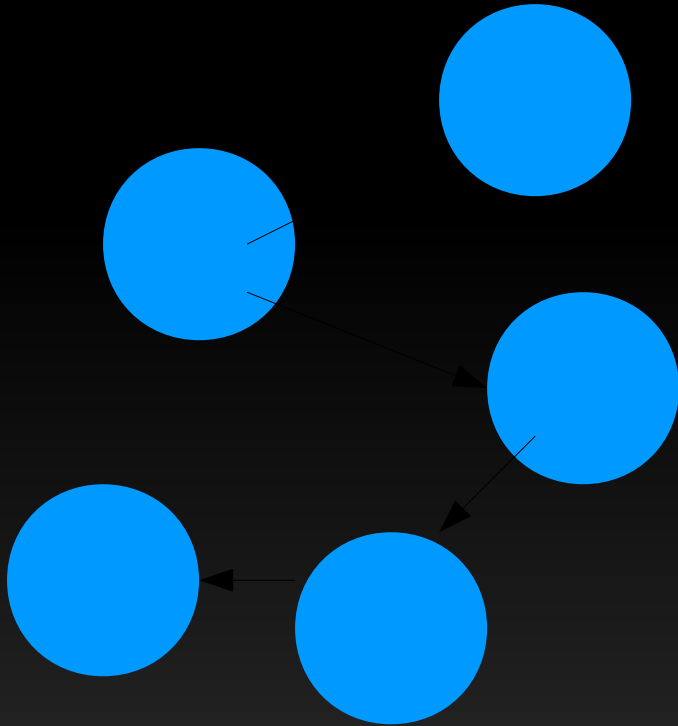
Precise collection



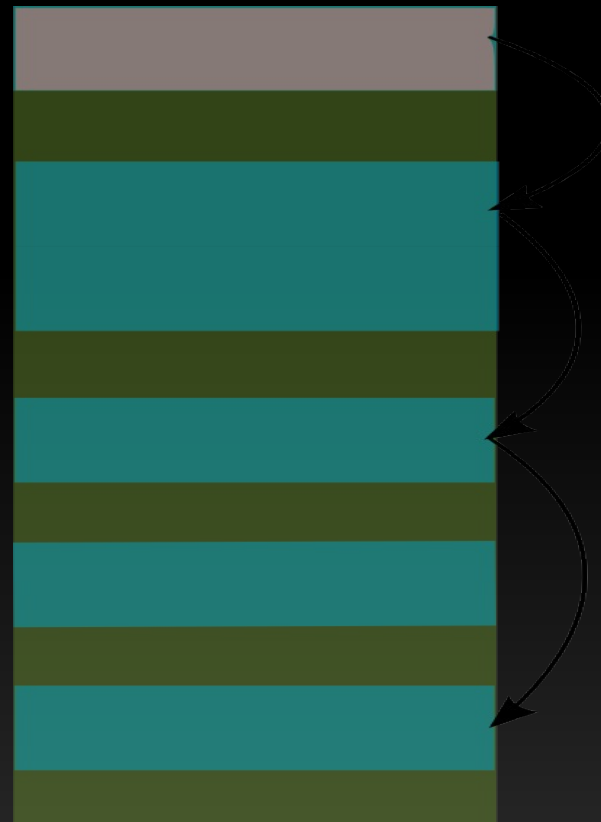
Conservative collection



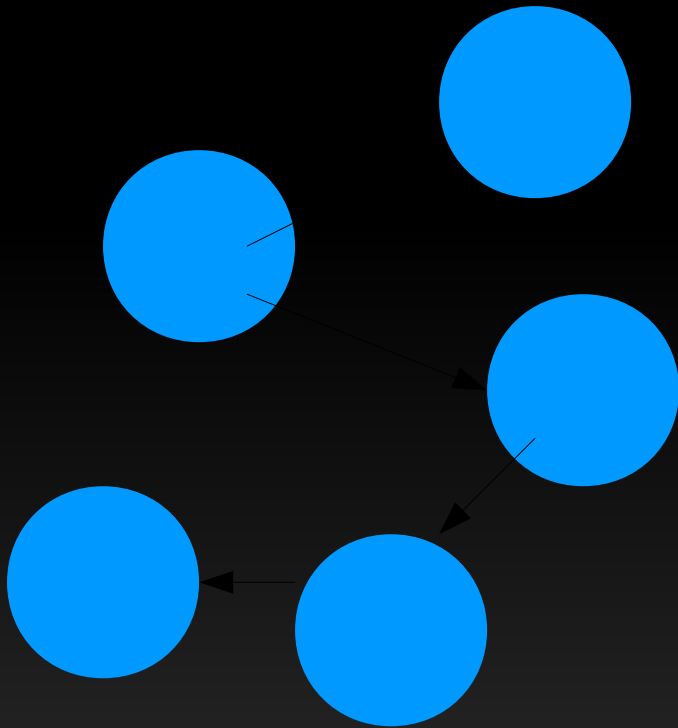
Precise collection



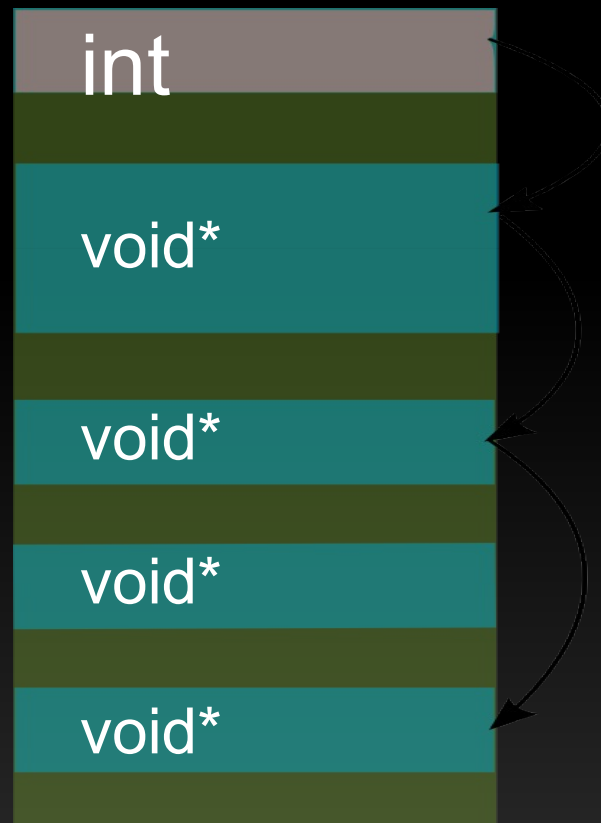
Conservative collection

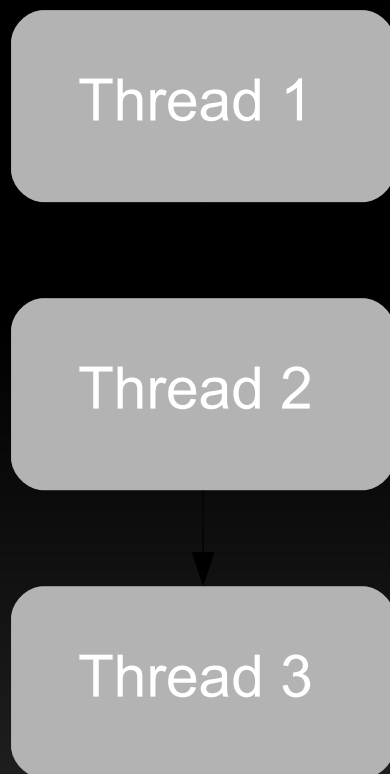


Precise collection



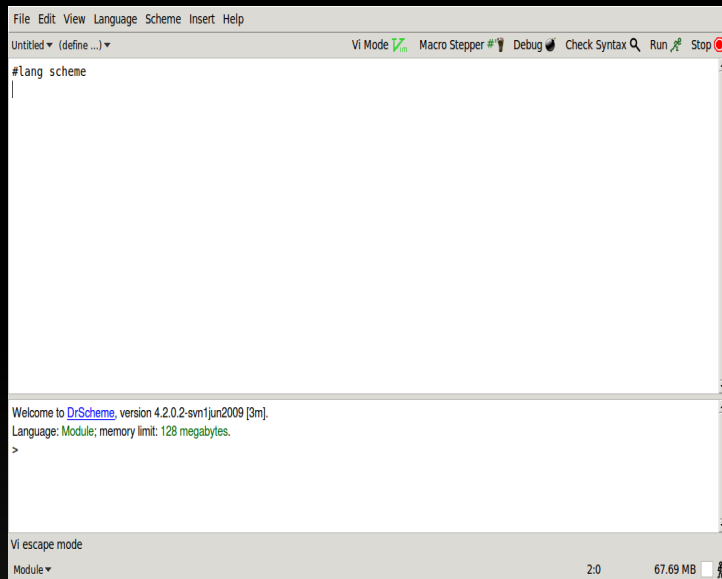
Conservative collection





Lists of threads and stacks  
create unreclaimable chains

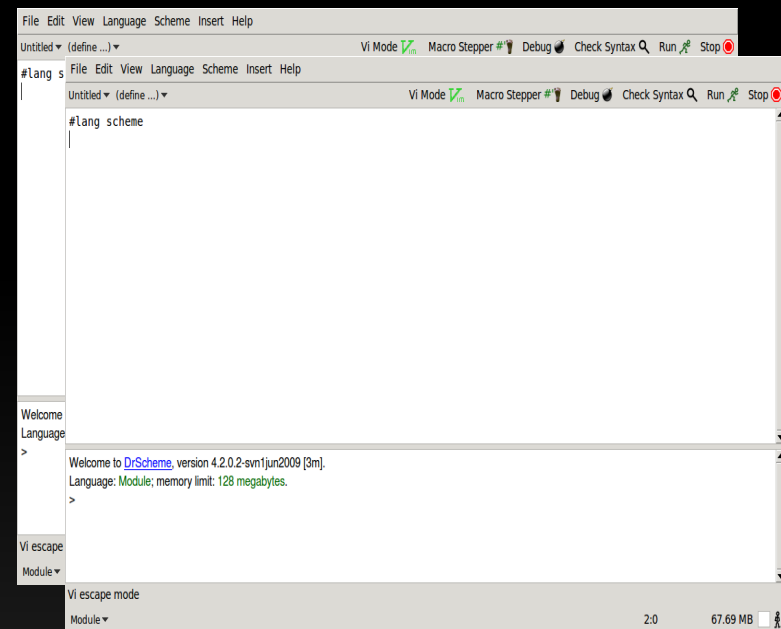
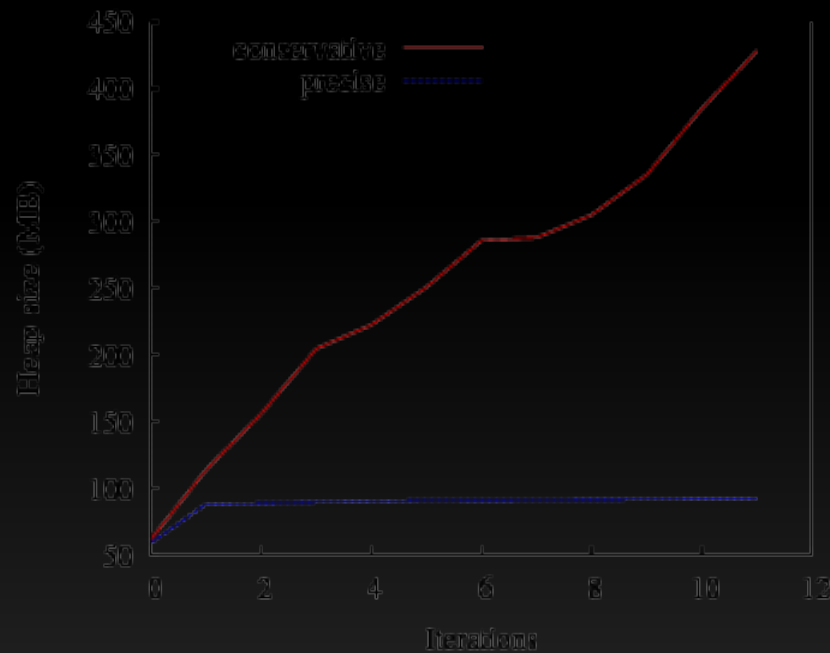
Web servers, OS's, etc can experience unstable memory usage with conservative collectors



```
File Edit View Language Scheme Insert Help
Untitled - (define ...) Vi Mode Macro Stepper Debug Check Syntax Run Stop
#Lang scheme
|
Welcome to DrScheme, version 4.2.0.2-svn1jun2009 [3m].
Language: Module; memory limit: 128 megabytes.
>
Vi escape mode
Module 2:0 67.69 MB
```

- Core of Drscheme written in C
- 10+ years of managing Boehm collector

# Certain classes of programs are susceptible to memory leaks







# ZSNes emulator – Super Nintendo emulator

## Nano – text editor

```

GNU nano 2.7.7@mac3  /tmp/nano.c  [root@fedora]
$ cat nano.c 1.5K 2001/03/16 04:08:38 editpage by 5.42
-----
*  nano.c
*
*  Copyright (C) 1999 Charles Allegretto
*  This program is free software; you can redistribute it and/or modify
*  it under the terms of the GNU General Public License as published by
*  the Free Software Foundation; either version 3, or (at your option)
*  any later version.
*
*  This program is distributed in the hope that it will be useful,
*  but WITHOUT ANY WARRANTY; without even the implied warranty of
*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*  GNU General Public License for more details.
*
*  You should have received a copy of the GNU General Public License
*  along with this program.  If not, see the file COPYING.
*  Foundation, Inc., 529 Mass Ave, Cambridge, MA 02139, USA
*
-----
[Show GNU GPL text]
[Get Help] [New Window] [Open] [New Page] [Get Text] [Get Pos]
[Exit] [Load File] [Save File] [Work Space] [Insert Text] [Spell]

```

Unbounded memory with conservative, stable with precise

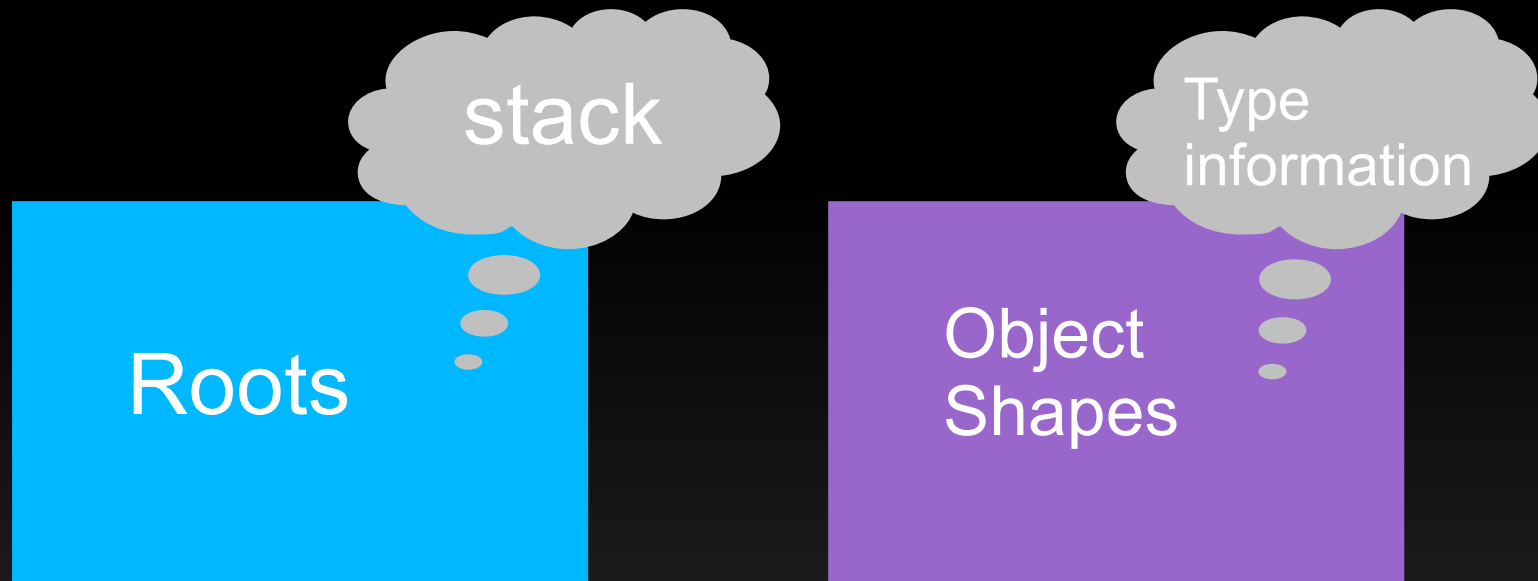
Source to source transformation of C code can embed information to let precise garbage collection work:

*Invariants:*

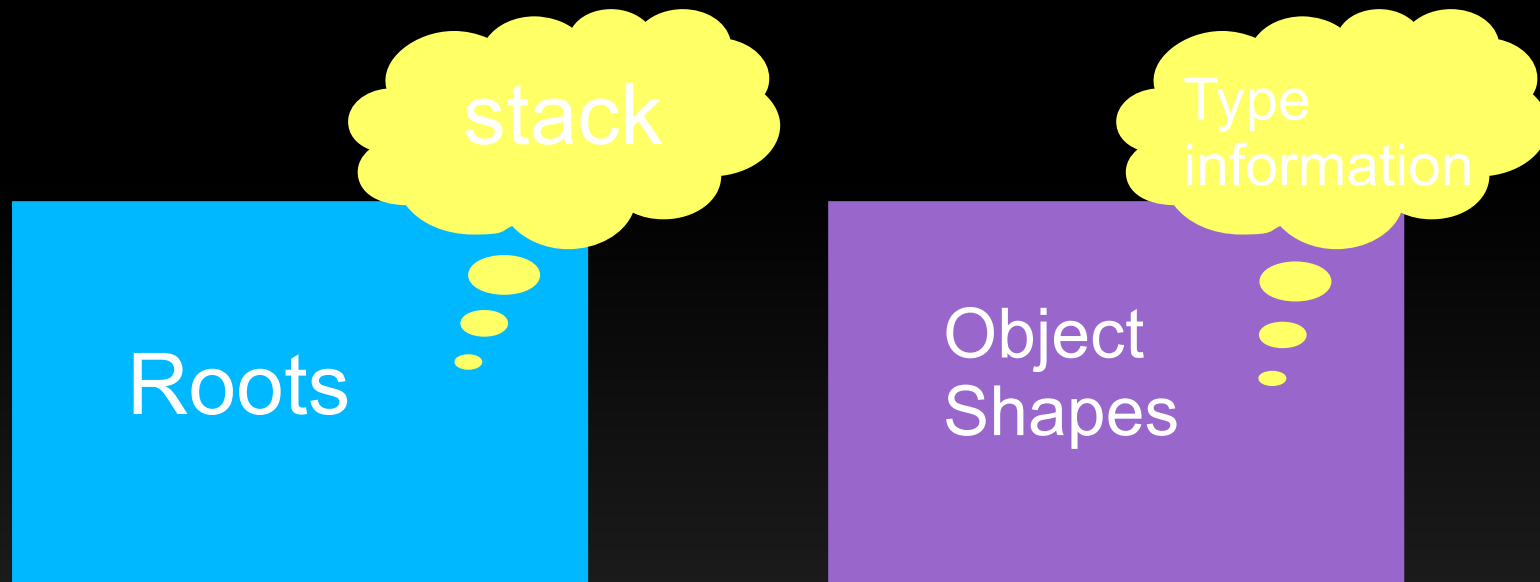
All pointers are known

Run-time type of objects is the same as their static allocation type

- Registers are not scanned for pointers
- All live pointers in the system are known



# Source transformation to retain this information



```
var = malloc(<expr>);
```

*sizeof(t)* – allocate single t structure

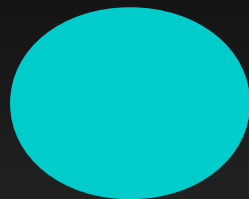
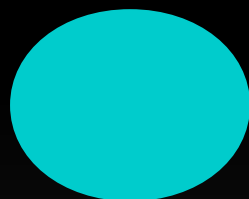
*sizeof(t) \* e* – allocate array of t structures

*sizeof(t\*) \* e* – allocate pointer array

*e* – allocate atomic block

Traverse objects:

- Marking
- Updating pointers



```
struct cat{  
    int * paws;  
    char * nose;  
};
```

```
void gc_struct_cat_mark(struct cat * c){  
    GC_mark(c->paws);  
    GC_mark(c->nose);  
}
```

```
void gc_struct_cat_repair(struct cat * c){  
    GC_repair(&c->paws);  
    GC_repair(&c->nose);  
}
```



Generate traversal functions

## Active variant is tracked

```
union object{  
    int value;  
    void * info;  
};
```

```
union object o;  
o.value = 1;  
GC_autotag(&o, 0);  
o.info = get();  
GC_autotag(&o, 1);
```



## Pointer obfuscation

```
int * p = malloc(20);  
p -= 10;  
p[10] = 2;
```

## Pointers as integers

```
int make(){  
    return malloc();  
}
```

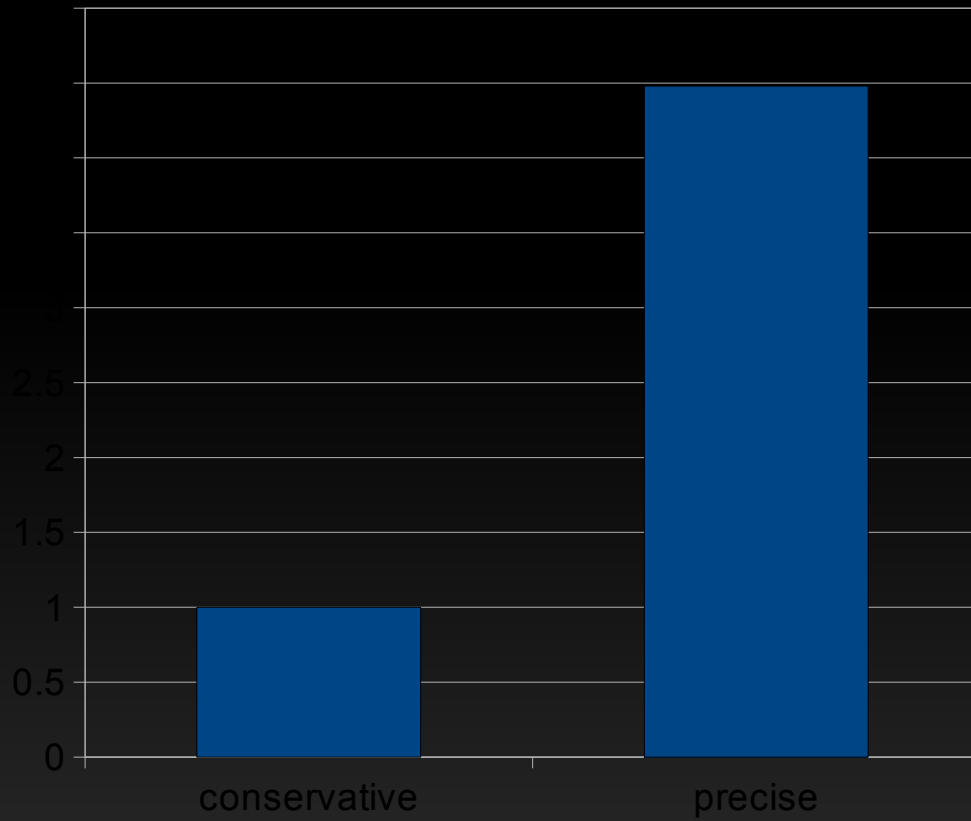
## Constructing pointers

```
int * x = (int*) 0x323429;
```

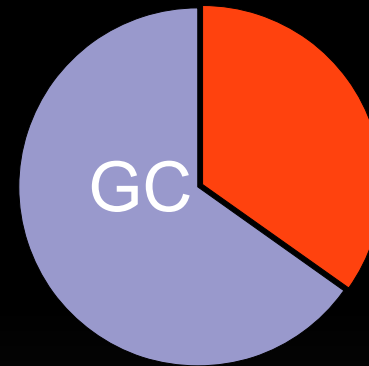
## Arrays of open sized arrays

```
struct foo{  
    ...  
    char rest[0];  
};
```

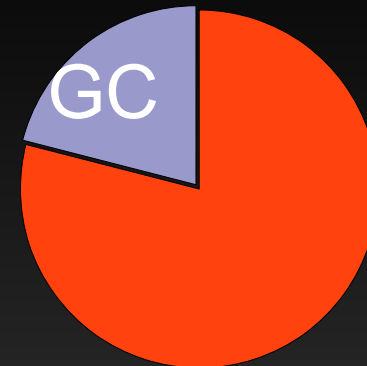
# Benchmark: CPStack



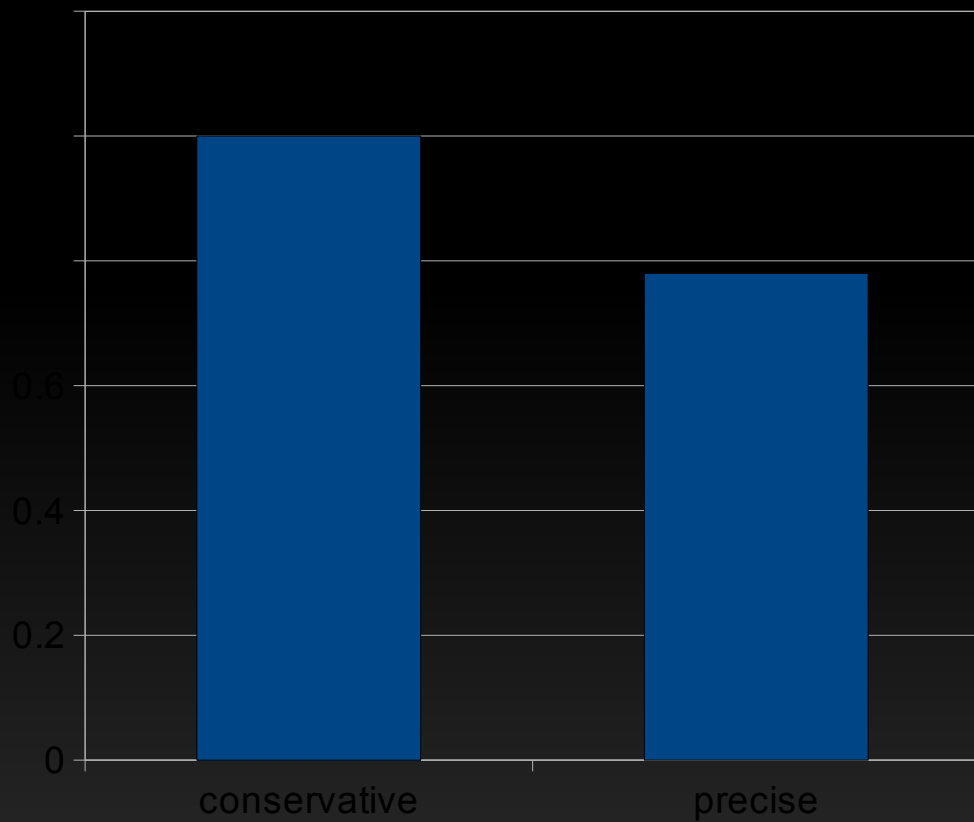
## Conservative



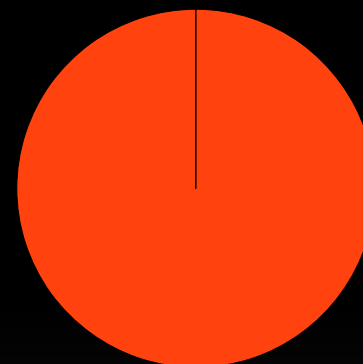
## Precise



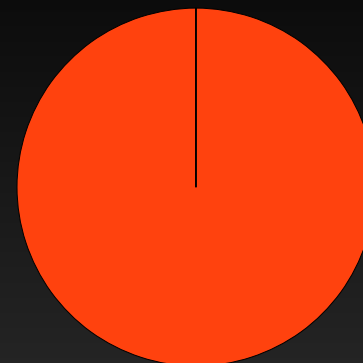
# Benchmark: Takl

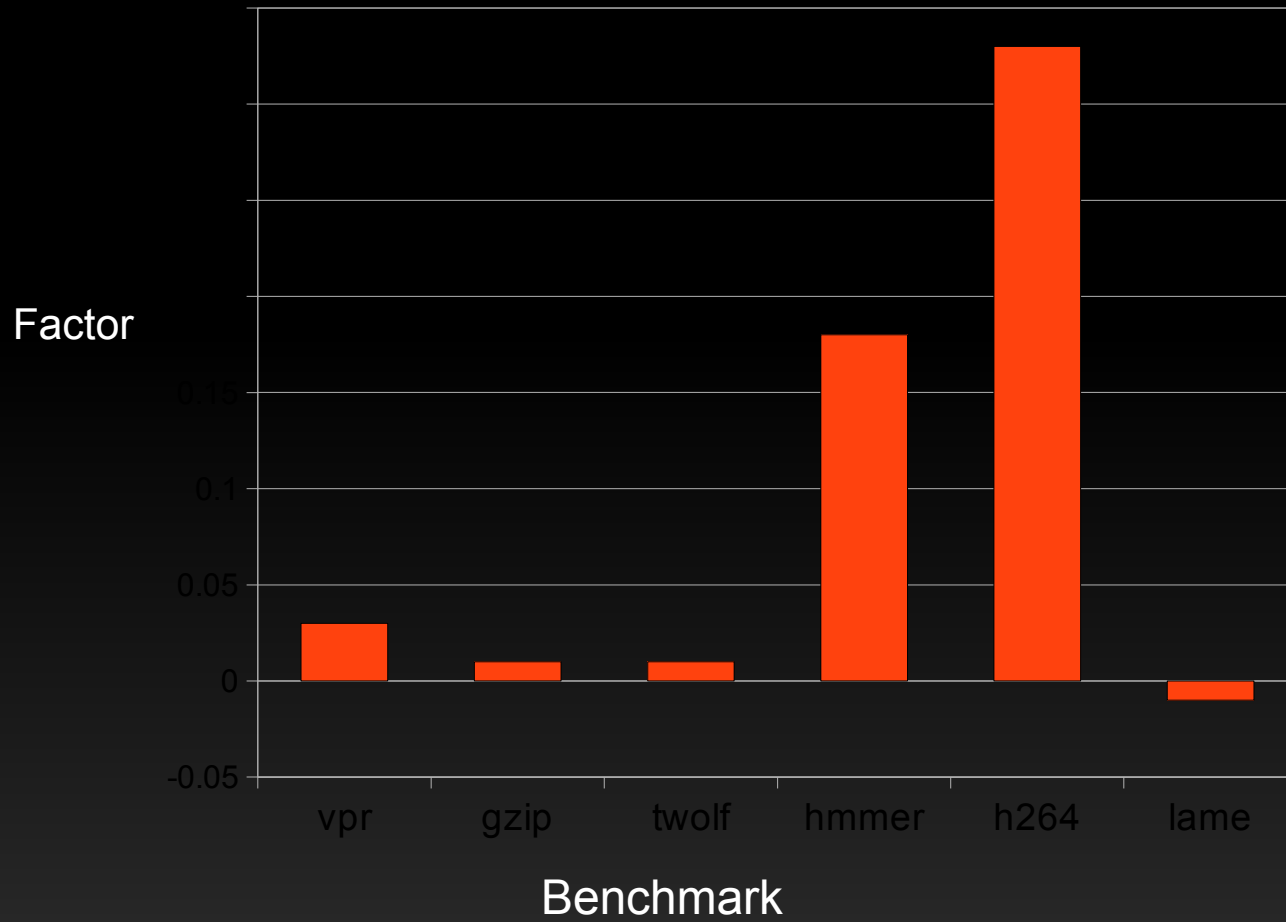


Conservative



precise





# Two tools that can be used on C programs

The screenshot shows the Magpie Structure Analysis tool interface. On the left, a code editor displays the definition of the `libtop_dch_t` struct from `dch.h`. The line number is 72. The struct definition includes fields for `base_table`, `base_grow`, `base_shrink`, `grow_factor`, `hash`, `key_comp`, and `ch`. The `ch` field is highlighted in yellow. On the right, a dialog box titled "16 of 17 question" contains a text-based question asking for a routine to traverse the "f" object. The question text is: "Please write the routine to traverse the 'f'. The object in question will be passed to the function in the variable 'x'." Below the question, it provides instructions: "Instead of using the functions gcMARK and gcFREE explicitly, please use gcTRAVERSE to represent these functions. Otherwise, the complete GC is available. Perhaps most useful might be GC\_END\_OF\_OBJECT(obj) and GC\_START\_OF\_OBJECT(obj)." It also includes a reminder: "Remember, GC\_ASSERT(test, msg) is your friend for debugging." The dialog box shows a partial C code snippet for the answer:
 

```

  /*
  {
    unsigned int size = GC_END_OF_OBJECT(x) -
                      GC_START_OF_OBJECT(x);
    int i;

    for(i = 0; i < size; i += 2)
      gcTRAVERSE(x[i]);
  }
  
```

 At the bottom of the dialog, there are buttons for "Nevermind; I don't need to write my own code", "Cancel", "Previous", and "Done".

File: dch.h  
Line: 72  
Type: struct libtop\_dch\_s

```

/*
 * Initial table size and high/low water
 * proportion if the table grows.
 */
unsigned base_table;
unsigned base_grow;
unsigned base_shrink;

/* (grow_factor * base_table) is the current
unsigned grow_factor;

/* Cached for later ch creation during r
unsigned (*hash)(const void *);
boolean_t (*key_comp)(const void *, const void *);

/* Where all of the real work is done. */
libtop_ch_t *ch;
};

libtop_dch_t *
dch_new(libtop_dch_t *a_dch, unsigned a_base_table,
        unsigned a_base_grow, unsigned a_base_shrink,
        libtop_ch_hash_t *a_hash, libtop_ch_key_t *a_key,
        void
  
```

16 of 17 question

Please write the routine to traverse the "f". The object in question will be passed to the function in the variable 'x'.

Instead of using the functions gcMARK and gcFREE explicitly, please use gcTRAVERSE to represent these functions. Otherwise, the complete GC is available. Perhaps most useful might be GC\_END\_OF\_OBJECT(obj) and GC\_START\_OF\_OBJECT(obj).

Remember, GC\_ASSERT(test, msg) is your friend for debugging.

```

/*
{
  unsigned int size = GC_END_OF_OBJECT(x) -
                    GC_START_OF_OBJECT(x);
  int i;

  for(i = 0; i < size; i += 2)
    gcTRAVERSE(x[i]);
}
  
```

Nevermind; I don't need to write my own code

Cancel Previous Done

## Two tools that can be used on C programs

- Based on Cil (Necula 2002)
- Parses C and GCC extensions
- Fully automatic

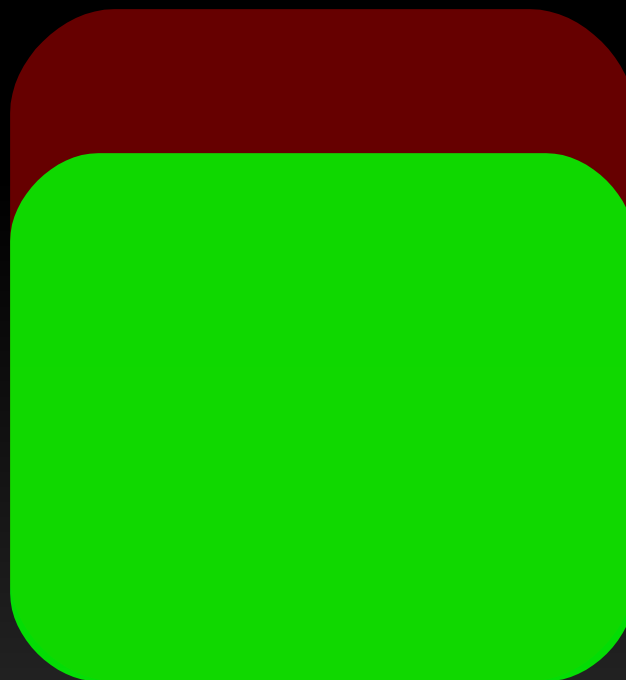
Too large to fully transform

Subsystems:

Ext3

IPV4

User Mode Linux stack is limited to 4k – GC  
shadow stack went over this limit



Shadow stack



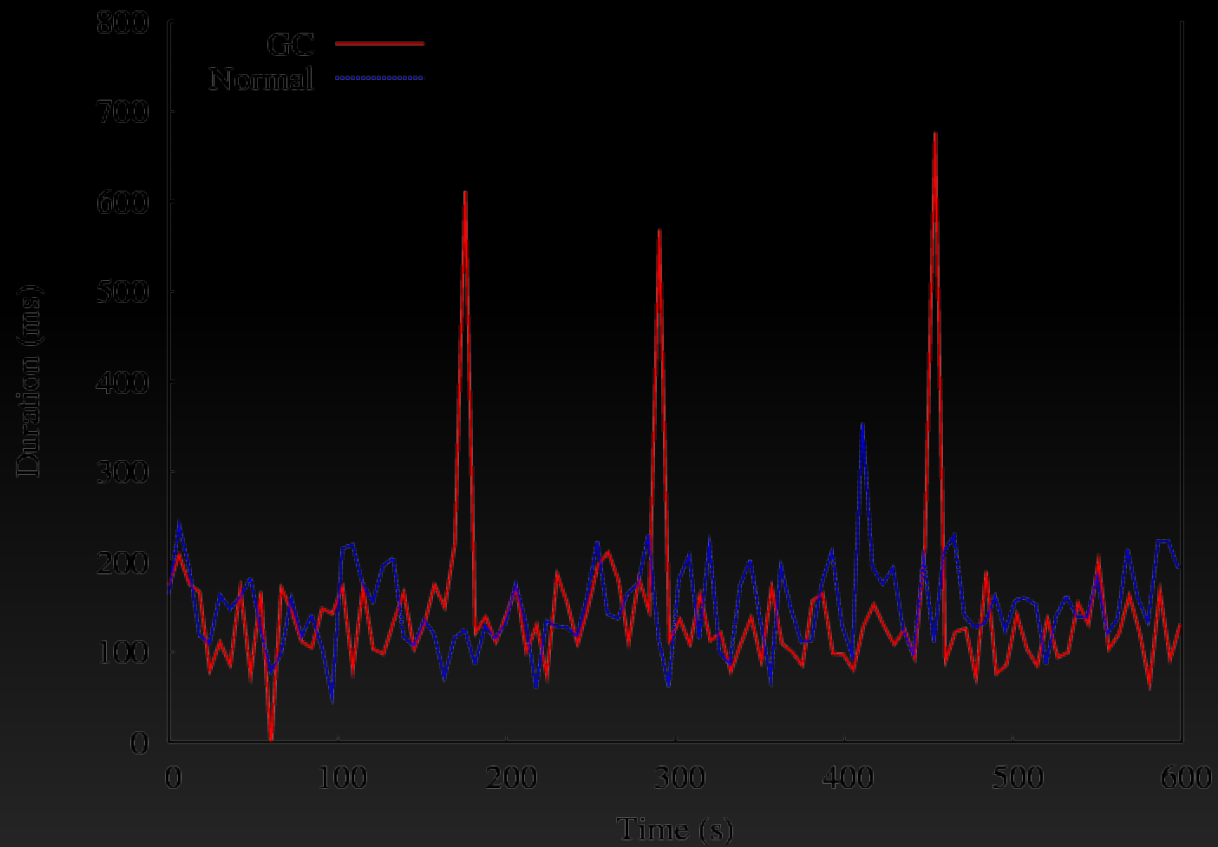
## Manual changes

RCU (Read-Copy-Update) finalizers

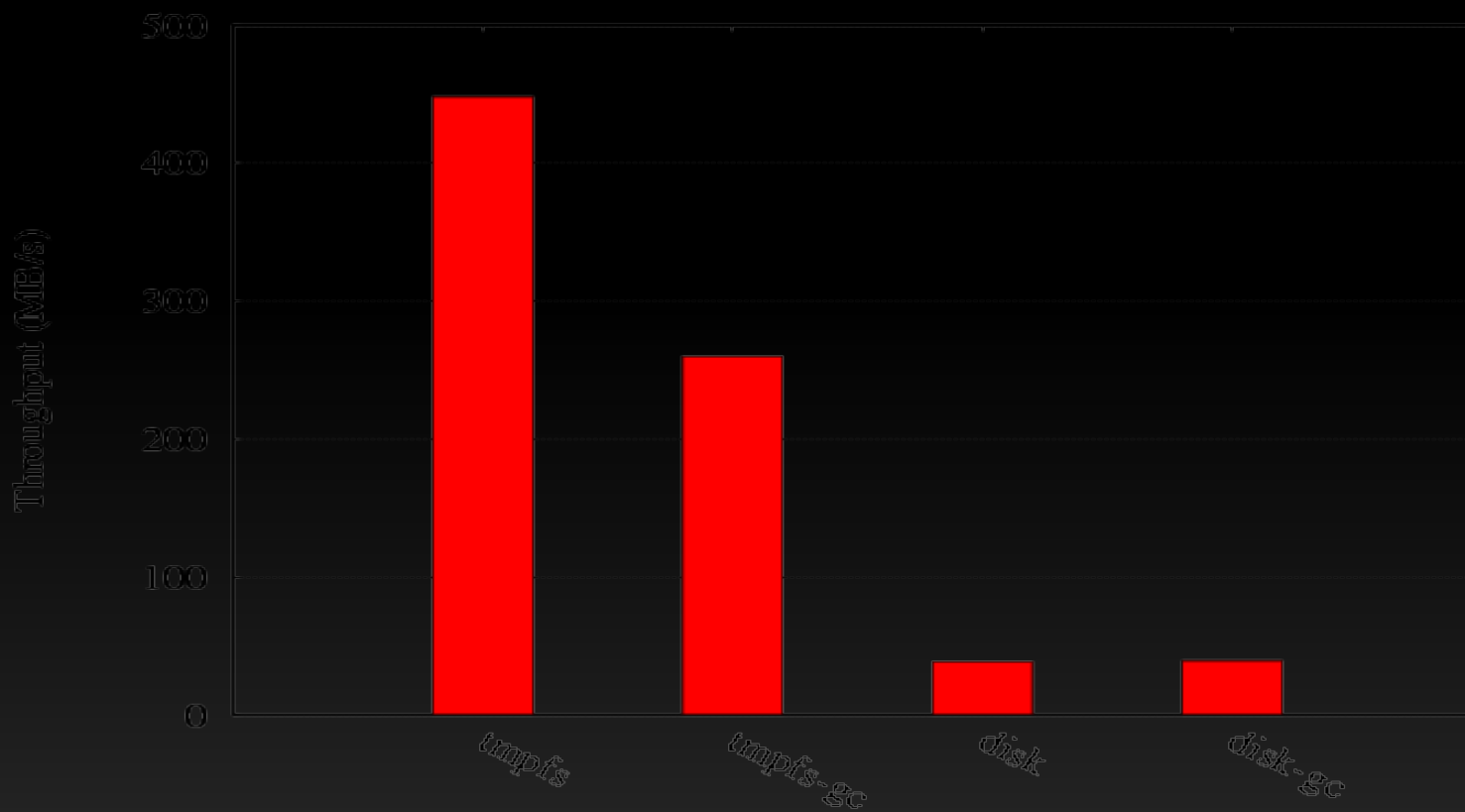
Important structures allocated with `vmalloc`

Custom allocators (`kmem_cache_alloc`)

# GC can cause high latencies



# GC overhead



- Precise garbage collection is a practical alternative for C
- Long running programs benefit from precise garbage collection

CPStack

Takl

