Software Performance Optimisation Group

Imperial College London

GCC in software performance research: just plug in

Paul Kelly
Software Performance Optimisation Group
Imperial College London
http://www.doc.ic.ac.uk/~phjk

Joint work with
David J. Pearce, Alex Lamaison, James Huggett, Olav Beckmann, Michael Mellor and others

Brasov, September 2007

Mission statement

- Software Performance Optimisation group

- Extensible optimising compiler technology
- Extensible performance profiling tools
- To tackle challenging contexts:
  - Dynamically-configured software
  - Diverse target hardware
  - Cross-component optimisation
  - Domain-specific optimisations
  - Distributed systems

What have we done with GCC?

- Field-sensitive pointer analysis
- Bounds checking for pointers and arrays
- A graph query language for Gimple
- Runtime code generation: using GCC as a library

What have we done with GCC?

- Field-sensitive pointer analysis
- Bounds checking for pointers and arrays
- A graph query language for Gimple
- Runtime code generation: using GCC as a library
- **Each demonstrates a different form of engagement with the GCC community**
- **This talk explores:**
  - **How GCC has helped our research**
  - **How our research aims to improve GCC**

Software Performance Optimisation Group
Imperial College London

Compiler research is really scary

- Most compiler research results never end up in compiler products
- Good compiler research ideas interact with other issues/phases/optimisations
- GCC is >1M lines of code
- Students get intimidated!

- So research strategy has to be designed with some care…

Field-sensitive pointer analysis

- Thesis work of David Pearce, now at Victoria University, New Zealand

```
int *f(int *p) {
  return p;
}
int g() {
  int x,y,*p,*q,**r,**s;
  s=&p;
  if(...) p=&x;
  else p=&y;
  r=s;
  q=f(*r);
}
```

(1) $f_* \supseteq f_p$

(2) $g_s \supseteq \{g_p\}$

(3) $g_p \supseteq \{g_x\}$

(4) $g_p \supseteq \{g_y\}$

(5) $g_r \supseteq g_s$

(6) $f_p \supseteq *g_r$

(7) $g_q \supseteq f_*$

Variable s of function g might point to variable p of function g

R might point to anything s might point to

f's p might point to anything r might point to

q might point to anything f returns

- Goal: for each pointer variable (p,q,r,s), find the set of objects it might point to at runtime

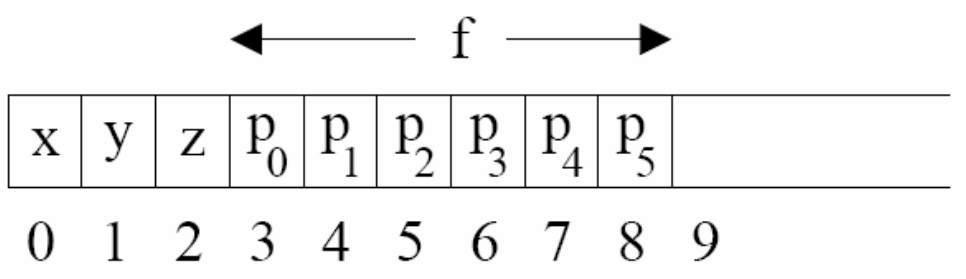
Set-constraint formulation of pointer analysis

- For each pointer variable, we have a points-to set
- Elements are variables whose address has been taken, together with abstract objects for each malloc call

$$[trans]\ \frac{p \supseteq \{q\} \quad r \supseteq p}{r \supseteq \{q\}} \qquad [deref_1]\ \frac{p \supseteq *q \quad q \supseteq \{r\}}{p \supseteq r} \qquad [deref_2]\ \frac{*p \supseteq q \quad p \supseteq \{r\}}{r \supseteq q}$$

- First, walk the code and extract the inclusion relationships between the points-to sets
- Then, iteratively sweep the graph to find a fixed point where all the inclusions are satisfied
- Subtlety: pointer dereferencing results in new inclusion relationships being added

Field sensitivity

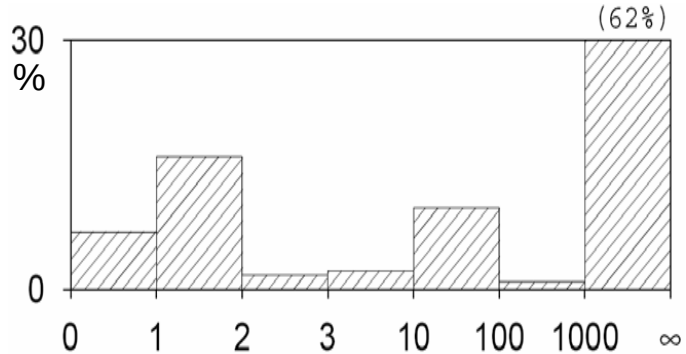


- Key idea: each field of each aggregate is represented by a points-to set indexed by an integer
- Access fields by $*(p_0+k)$
- Do the same for each parameter of each function
- Now our constraint graph has *weighted* arcs
- We can even have cycles:
  - $p_0 = (p_0+k)$
  - (because in C we can take address of a field)
  - These have a very negative effect on analysis time and quality
  - But occur rather rarely, often due to malloc wrappers

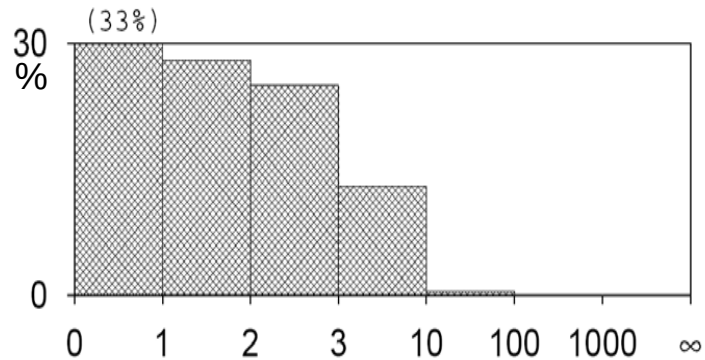
Solving the constraints (fast)

- We have quite a large constraint graph
  - Eg for 126.gcc from SPEC95:
    - 194KLOC (132K without comments etc)
    - 51K constraint variables (22K of them heap)
    - 7.4K “trivial” constraints
    - 39K “simple” constraints
    - 25K “complex” constraints (due to dereferencing)
    - No positive-weighted cycles
- Need to bring together several tricky techniques to get sensible solution times
  - Difference-sets: propagate only changes so you can track what has changed
  - Topological sort: visit nodes in order that maximises solution propagation
  - Cycle detection: zero-weighted cycles can be collapsed
  - Dynamically: dereferencing pointers adds new edges
  - 0.61s (900MHz AMD Athlon)

- Histogram of points-to set size at dereference sites for 126.gcc:



- Field *insensitive*



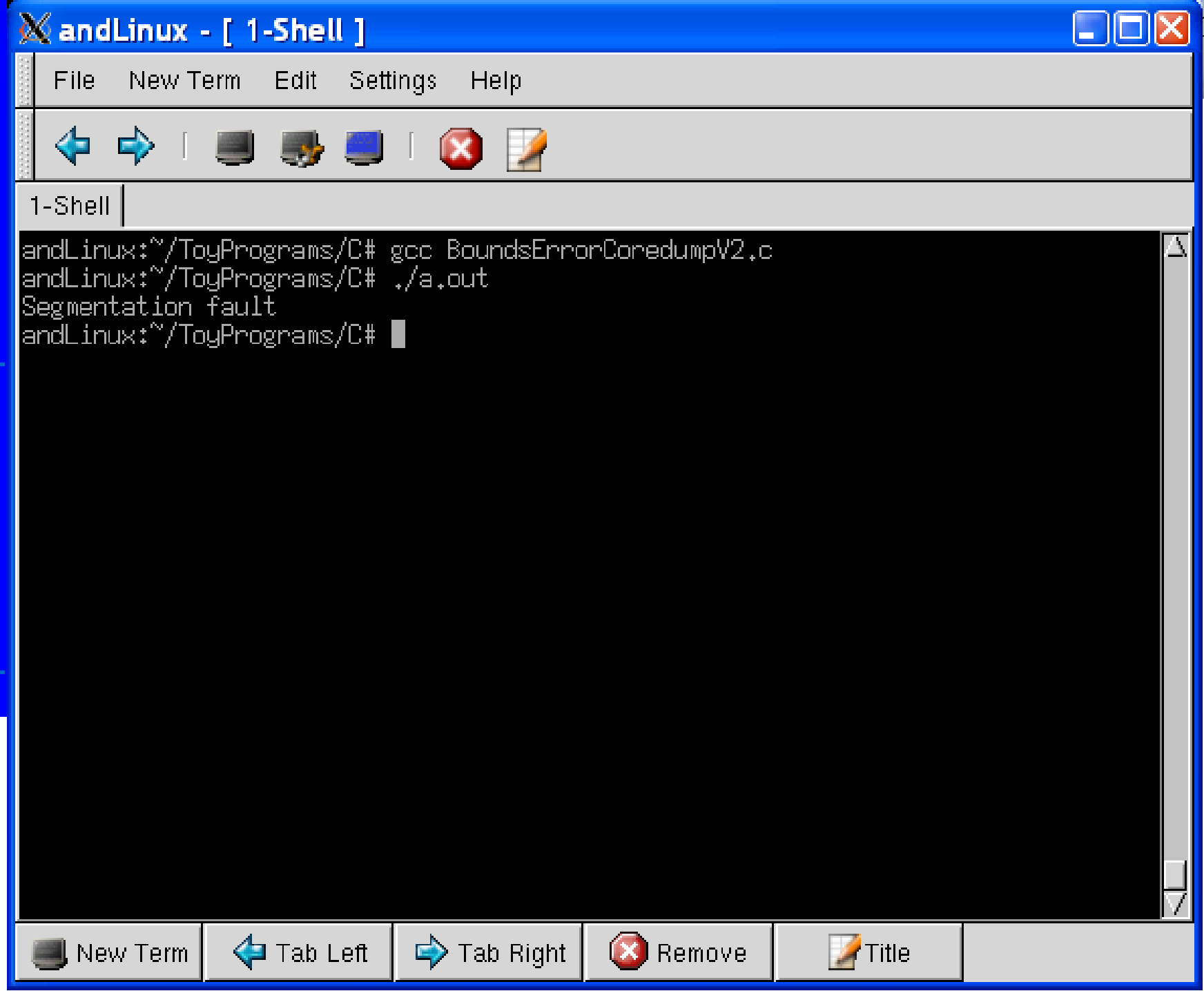
- Field *sensitive*

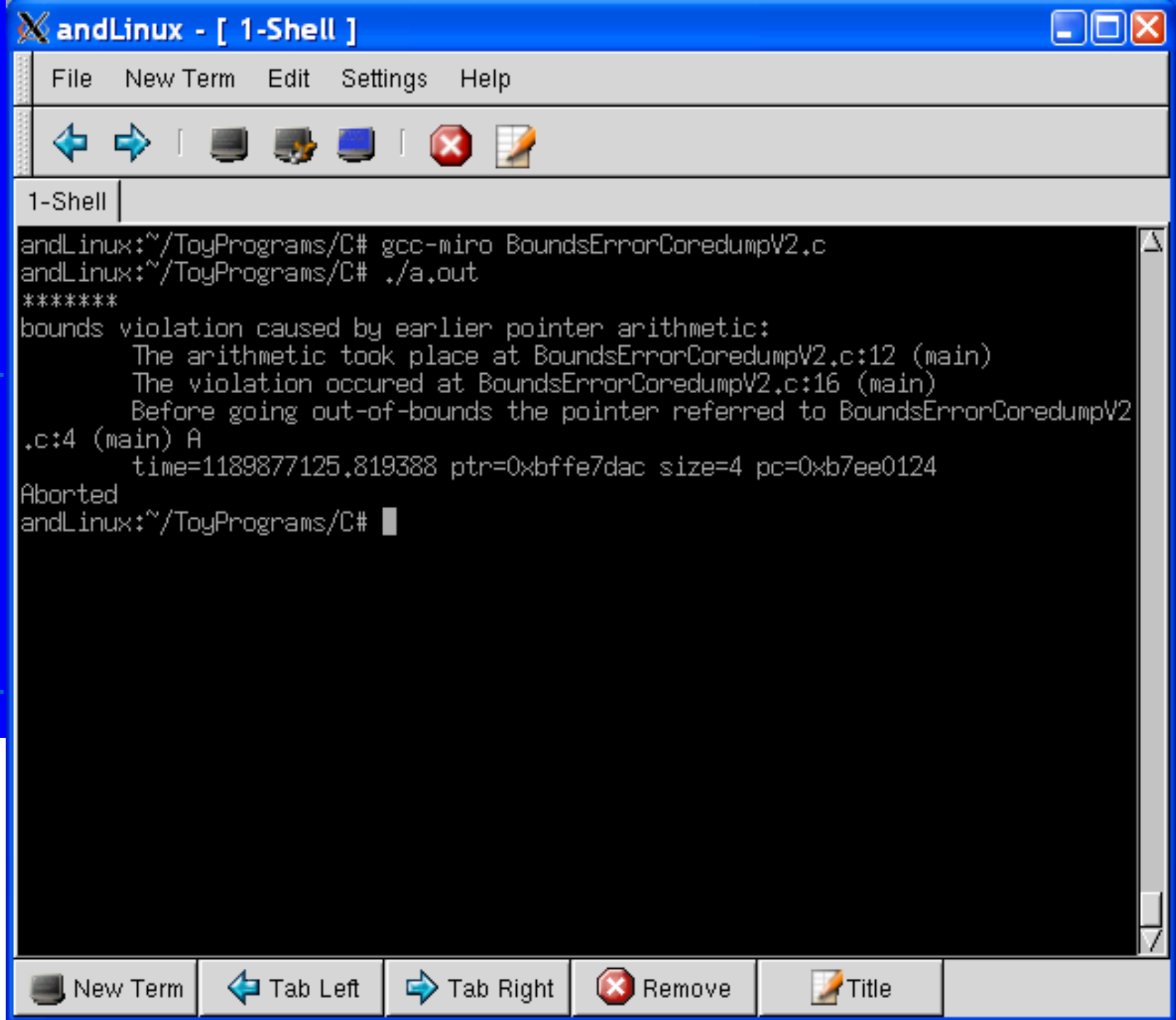
But…

- We didn’t use GCC for this
- Instead, we used SUIF

But…

- We didn't use GCC for this
- Instead, we used SUIF
- But our work was re-implemented within GCC
  - By Dan Berlin
  - Fully acknowledged
  - With some interesting refinements
  - Shipped in GCC4.1 snapshot
  - The week of David's PhD defence




andLinux - [1-Shell]
File
New Term
Edit
Settings
Help
1-Shell
andLinux:~/ToyPrograms/C# gcc-miro BoundsErrorCoredumpV2.c
andLinux:~/ToyPrograms/C# ./a.out
****
bounds violation caused by earlier pointer arithmetic:
The arithmetic took place at BoundsErrorCoredumpV2.c:12 (main)
The violation occured at BoundsErrorCoredumpV2.c:16 (main)
Before going out-of-bounds the pointer referred to BoundsErrorCoredumpV2
.c:4 (main) A
time=1189877125.819388 ptr=0xbffe7dac size=4 pc=0xb7ee0124
Aborted
andLinux:~/ToyPrograms/C#
New Term
Tab Left
Tab Right
Remove
Title

Bounds checking

- In civilised languages, when you calculate an address, you have a handle on the object you're indexing:

  "A[i]"

- In C the user of a calculated pointer has no access to the object:

  "*p"

- Bounds checking in C is harder!

- Classical solution: change representation of "p" so it includes "A" ("fat pointers")

- Breaks binary compatibility…

Strengthening the invariant: Referent Objects

- How can we figure out what object p is *supposed* to point into?
- Suppose you calculate a pointer:

  “p = q + 1024”
- p might now point to a different array…
- But that would be wrong.
- Idea: **invariant**
  - Suppose we enforce that all stored pointers (eg p and q) point into the object they’re supposed to?

Referent objects

- **Referent object** of a pointer:
  - *The unit of storage allocation into which the pointer can correctly be used to access data*
- So we add code to check all pointer arithmetic (as well as dereferencing)
- Maintain table of allocated storage objects
- Using splay tree to find a pointer's referent object (range query)
- Make sure that pointer arithmetic based on p always yields a pointer with the same Referent Object

Glitch: out of bounds objects

- The Referent Objects scheme alone is too strict
- Programmers calculate invalid pointers and expect them to work right
  - (the ANSI C standard says they usually shouldn't but they do)
- When an invalid pointer is calculated, we represent it as a pointer to an "out-of-bounds object" (OOB)
- Intercept pointer arithmetic and comparisons so OOBs caught and the right values are used

Does it work?

- MIRO: Mudflap improved with Referent Objects

Program	Test	GCC		Mudflap		MIRO	
		unpatched	patched	unpatched	patched	unpatched	patched
sendmail	s1	FAIL seg	PASS nf	PASS v	PASS n	PASS v	PASS nf
	s2	FAIL seg	PASS nf	PASS v	FAIL fp	PASS v	PASS nf
	s3	FAIL x	PASS nf	FAIL x	PASS nf	PASS v	PASS nf
	s4	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	s5	FAIL x	PASS nf	FAIL x [a]	PASS nf	PASS v	PASS nf
	s6	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	s7	FAIL seg	PASS nf	FAIL fp [b]	FAIL fp	PASS v	PASS nf
bind	b1	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b2	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b3	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b4	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
wu-ftpd	f1	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	f2	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	f3	FAIL x	PASS nf	PASS v	FAIL fp	PASS v	PASS nf

- Using suite of known defects in large open-source applications (Zhivich, Leek and Lippmann, 2005)

Bounds checking and the GCC community

- BCC (Jones and Kelly)
  - Patches for GCC released ca.1996
  - Operates in C front-end
- McGary
  - Internal GCC fat pointers project
- Mudflap (Eigler)
  - Checks accesses are valid but does not track referent objects
- CRED (Ruwase and Lam)
  - Introduces OOB objects to reduce false positives
  - http://sourceforge.net/projects/boundschecking/
- MIRO (Lamaison and Kelly)
  - Reimplementation of referent objects+OOB scheme using tree-SSA
  - Independent of front-end – works with C++

Plugging into GCC – querying the IR

- Drvar is a yacc-like language for specifying queries over linked structures in C. It generates more-or-less readable C.
- Fundamentals:

```
Valid(x): p1(x) & p2(x)
```

  - Node x is valid if p1(x) and p2(x) are satisfied

```
Valid(x): -> p(x)
```

  - Node x is valid if it has a child for which p(x) holds
  - where “child” is defined by a iterator map definition for the type “node”

Plugging into GCC – querying the IR

- More Drvar examples:

```
valid(operand_tree stmt):
    'TREE_CODE($stmt) != SSA_NAME'
```

  - Node stmt is valid it's tagged as an SSA name (quoted C is embedded in generated code)

```
zeroComp(operand_tree stmt):
    'TREE_CODE ($stmt) == EQ_EXPR'
    -> 'TREE_CODE ($stmt) == SSA_NAME'
    -> 'integer_zerop ($stmt)'
```

  - Stmt is a comparison with zero

Plugging into GCC – querying the IR

- More Drvar examples:

```
start() : for_loops ((struct loops *)'current_loops');
```

  - Iterate through GCC's current_loops, using the iterator map for "struct loops *".

```
for_loops(loop_tree loop):
    ! has_yield_bb (loop, loop) & { warning ("No call to yield
    , !->! (END | for_loops(loop));
```

  - If this loop has no call to the "yield" method (details omitted), then print warning
  - Then (","), repeat for all ("!->!") children unless/until END found.

Drvar - implementation

- Drvar compiler generates C code to traverse C data structures
- Iterator map file specifies how Drvar iterates over each of the C types, and provides implementation for “->”, “END” etc
- Generated code is fairly straightforward – optimisations avoid code duplication and turn tail recursion into loops
- Special GCC pass dynamically links the generated code into GCC
- So Drvar/GCC compile-edit cycle is very fast

Drvar – applications, performance

- Static debugging
- Validity conditions for optimisations
  - Loop-invariant code motion
- Instrumentation

- Related work:
  - Our own DeepWeaver tool for Java
  - Condate for GCC
  - Datalog-based tools (Semmle, bddbddb)
  - Path queries (JunGL, etc)
  - AST-based tools (Stratego, ROSE, etc)

Using GCC as a library

- Suppose you could call GCC just like a normal function?
- Suppose you could build and run code programmatically?
- Suppose you could control the compilation process programmatically?

- The TaskGraph library is a portable C++ package for building and optimising code on-the-fly
- Compare:
  - `C (tcc) (Dawson Engler)
  - MetaOCaml (Walid Taha et al)
  - Jak (Batory, Lofaso, Smaragdakis)
- But there's more…

```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main() {
  int c = 1;
  TaskGraph < Par < int, int >, Ret < int > > T;
  taskgraph( T, tuple2(x, y) ) {
    tReturn( x + y + c );
  }
  T.compile( tg::GCC );
  int a = 2;
  int b = 3;
  printf( "a+b+c = %d\n", T.execute( a, b ) );
}
```

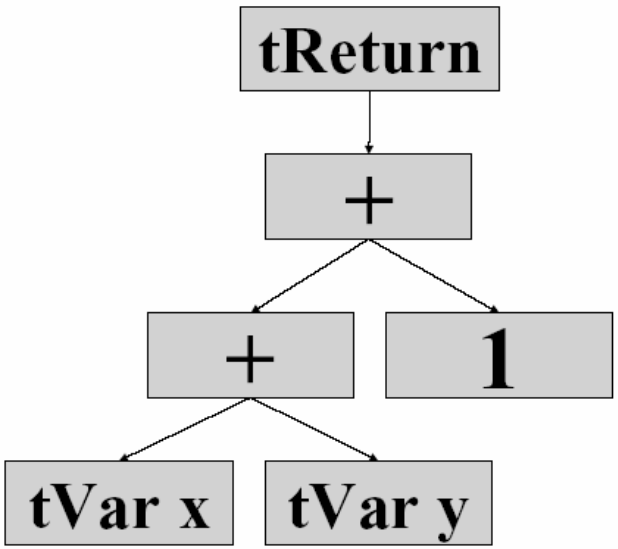
- A taskgraph is an abstract syntax tree for a piece of executable code
- Syntactic sugar makes it easy to construct
- Defines a simplified sub-language
  - With first-class multidimensional arrays, no alliasing

```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main() {
  int c = 1;
  TaskGraph < Par < int, int >, Ret < int > > T;
  taskgraph( T, tuple2(x, y) ) {
    tReturn( x + y + c );
  }
  T.compile( tg::GCC );
  int a = 2;
  int b = 3;
  printf( "a+b+c = %d\n", T.execute( a, b ) );
}
```

- Binding time is determined by type
- In this example
  - c is static
  - x and y dynamic



- built using value of c at construction time

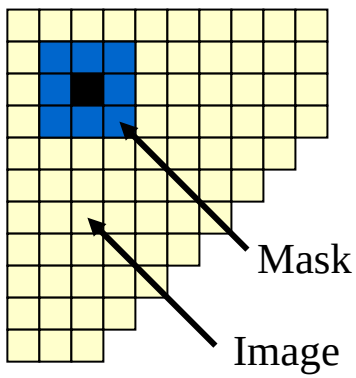
```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main() {
  int c = 1;
  TaskGraph < Par < int, int >, Ret < int > > T;
  taskgraph( T, tuple2(x, y) ) {
    tReturn( x + y + c );
  }
  T.compile( tg::GCC );
  int a = 2;
  int b = 3;
  printf( "a+b+c = %d\n", T.execute( a, b ) );
}
```

Better example:

- Applying a convolution filter to a 2D image
- Each pixel is averaged with neighbouring pixels weighted by a stencil matrix




```
void filter (float *mask, unsigned n, unsigned m,
             const float *input, float *output,
             unsigned p, unsigned q)
{
  unsigned i, j;
  int      k, l;
  float    sum;
  int half_n = (n/2);
  int half_m = (m/2);

  for (i = half_n; i < p - half_n; i++) {
    for (j = half_m; j < q - half_m; j++) {
      sum = 0;

          // Loop bounds unknown at compile-time
          // Trip count 3, does not fill vector registers

      for (k = -half_n; k <= half_n; k++)
        for (l = -half_m; l <= half_m; l++)
          sum += input[(i + k) * q + (j + l)]
                 * mask[k * n + l];

      output[i * q + j] = sum;
    }
  }
}
```

- TaskGraph representation of this loop nest
- Inner loops are static – executed at construction time
- Outer loops are dynamic
- Uses of mask array are entirely static

- This is deduced from the types of mask, k, m and l.

emacs@SECONDSELF

Buffers Files Tools Edit Search Mule C++ Help

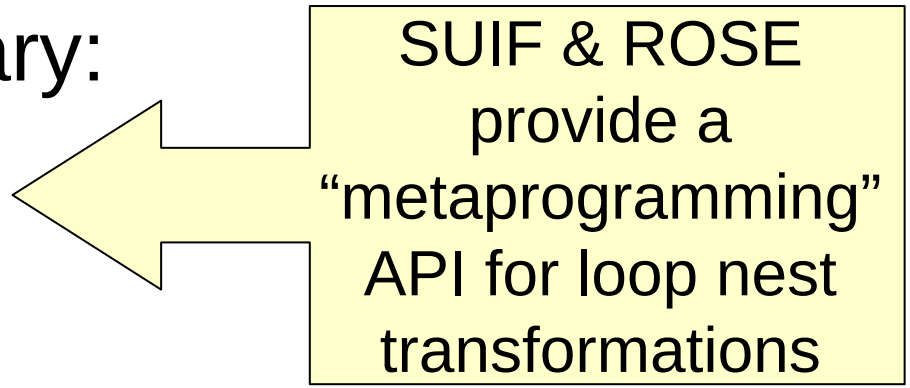
```
void specialize_convolution(
  TaskGraph < Par <float[IMG_SIZE][IMG_SIZE],
                   float[IMG_SIZE][IMG_SIZE]>,
              Ret < void > > &T,
  const int IMGSZ, const int CSZ, const float *mask )
{
  int ci, cj;
  assert( CSZ % 2 == 1 );
  const int c_half = ( CSZ / 2 );
  taskgraph( T, tuple2(tgimg, new_tgimg) ) {
    tVar ( int, i );
    tVar ( int, j );
    // Loop iterating over image
    tFor( i, c_half, IMGSZ - (c_half + 1) ) {
      tFor( j, c_half, IMGSZ - (c_half + 1) ) {
        new_tgimg[i][j] = 0.0;
        // Loop to apply convolution mask
        for( ci = -c_half; ci <= c_half; ++ci ) {
          for( cj = -c_half; cj <= c_half; ++cj) {
            new_tgimg[i][j] +=
              tgimg[i+ci][j+cj] * mask[c_half+ci][c_half+cj];
        } } } }
  }
}
```

// Inner loops fully unrolled
// j loop is now vectorisable

--\-- Convolution.cc (C++)--L4--All----------

Taskgraph – how it works

- We have built several back-ends for the Taskgraph library:
  - SUIF
  - ROSE
  - Hand-rolled



- The library builds the SUIF/ROSE IR at run-time, then dumps it as C to a file
- Calls the C compiler on the file
- Links the binary back into the host app
- About 100ms minimum

Taskgraph – GCC backend

- Build GCC IR directly

- Call GCC directly
  - Build GCC IR
  - Get GCC to generate assembler
  - Link binary back into host app

  - About half the overhead
  - Opportunity: use GCC to do metaprogramming of loop nest transformations?

What does it take to use GCC?

- All the work I have talked about was done by my students

- Drvar:
  - 3yr BEng CS final-year project (James Huggett)
  - November-June mostly overlapped with other studies
- Bounds checking - first 1995-6 version
  - 3yr BSc Maths+CS final-year project (Richard WM Jones)
  - Started November, released patches March
- Bounds checking – MIRO 2006-7 version
  - 4yr MEng CS final-year project (Alex Lamaison)
  - November-June mostly overlapped with other studies
- TaskGraph/GCC
  - PhD student's pet project, a few weeks (Michael Mellor)
- Field-sensitive pointer analysis
  - PhD project, 3.5 years (David J Pearce)
  - Reimplementation by Daniel Berlin

Working with GCC -

- Model of engagement – **tech transfer**
  - We publish, others implement
  - (as with field-sensitive pointer analysis)
    - Of course this happens all the time - hopefully
    - In closed-source projects – we never find out
    - What's exciting with GCC is that the code is there for all to read, and openly acknowledges its sources
    - We get to claim that every person you meet owns a device running code generated using the result of our research
    - We don't have to do any maintenance, bug-fixing etc!

Working with GCC -

- Model of engagement - **patches**
  - We release patches
  - As with bounds checking
    - 100-200 downloads/month from sourceforge (the CRED version including Ruwase&Lam improvements)
    - We have a measure of the level of interest
    - Many people who should use it don't
    - We were very lucky: other people volunteered to pick up the maintenance and updating tasks
    - Good students really can create robust patches

Working with GCC -

- Model of engagement – **GCC as a toolbox**
  - We use GCC as a component
  - As with our runtime code-generation work
  - GCC is exceptionally rich in tools for systems research
  - GCC is not designed as a research infrastructure
  - Much work to be done

Working with GCC -

- Model of engagement - **pluggability**
  - We design a generic plug-in architecture for optimisations
  - Support easy plug-and-play
  - “Marketplace” of optimisation/analysis components
    - Download, load and go, “gcc –fpauls-latest-hack”
  - Some progress on analysis side:
    - Drvar and Condate (“-ftree-check”)
  - Lots of research issues here…

Conclusions

- GCC is a vast resource
- GCC is not a research project
- GCC can deliver your research
- GCC can enable your research
- GCC is not a research infrastructure
  - But lots of exciting work is going on
  - With concerted effort it will become the pre-eminent research tool

- Find out more:
  - HiPEAC GCC Tutorial
    - http://www.hipeac.net/node/746