SHARE
Technology • Connections • Results

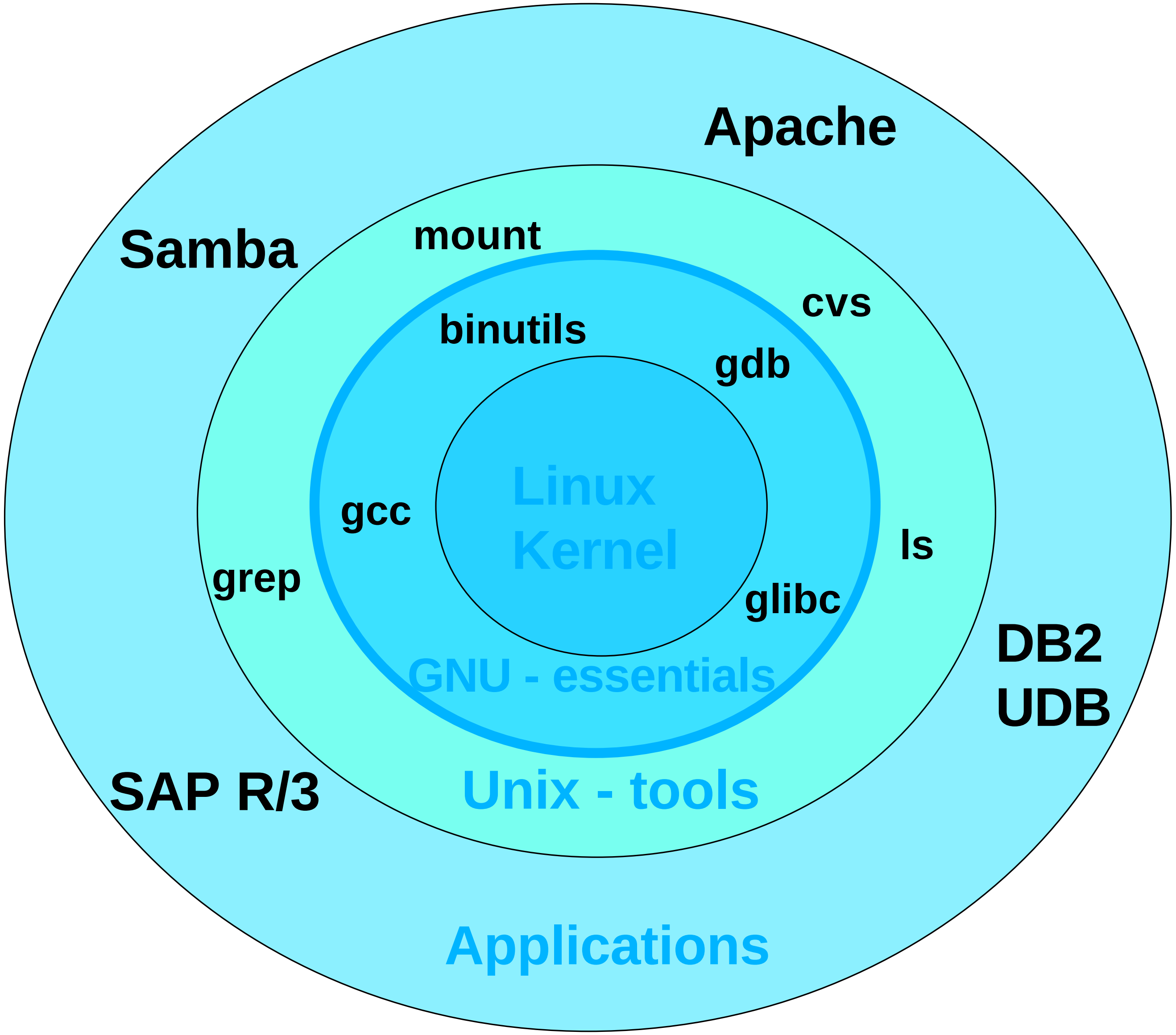
The GNU Compiler Collection on zSeries

Dr. Ulrich Weigand
Linux for zSeries Development, IBM Lab Böblingen
Ulrich.Weigand@de.ibm.com

Agenda

- GNU Compiler Collection
  - History and features
  - Architecture overview
- GCC on zSeries
  - History and current status
  - zSeries specific features and challenges
- Using GCC
  - GCC optimization settings
  - GCC inline assembly
- Future of GCC

GCC and Linux



GCC History

- Timeline
  - January 1984: Start of the GNU project
  - May 1987: Release of GCC 1.0
  - February 1992: Release of GCC 2.0
  - August 1997: EGCS project announced
  - November 1997: Release of EGCS 1.0
  - April 1999: EGCS / GCC merge
  - July 1999: Release of GCC 2.95
  - June 2001: Release of GCC 3.0
  - May/August 2002: Release of GCC 3.1/3.2
  - March 2003: Release of GCC 3.3 (estimated)

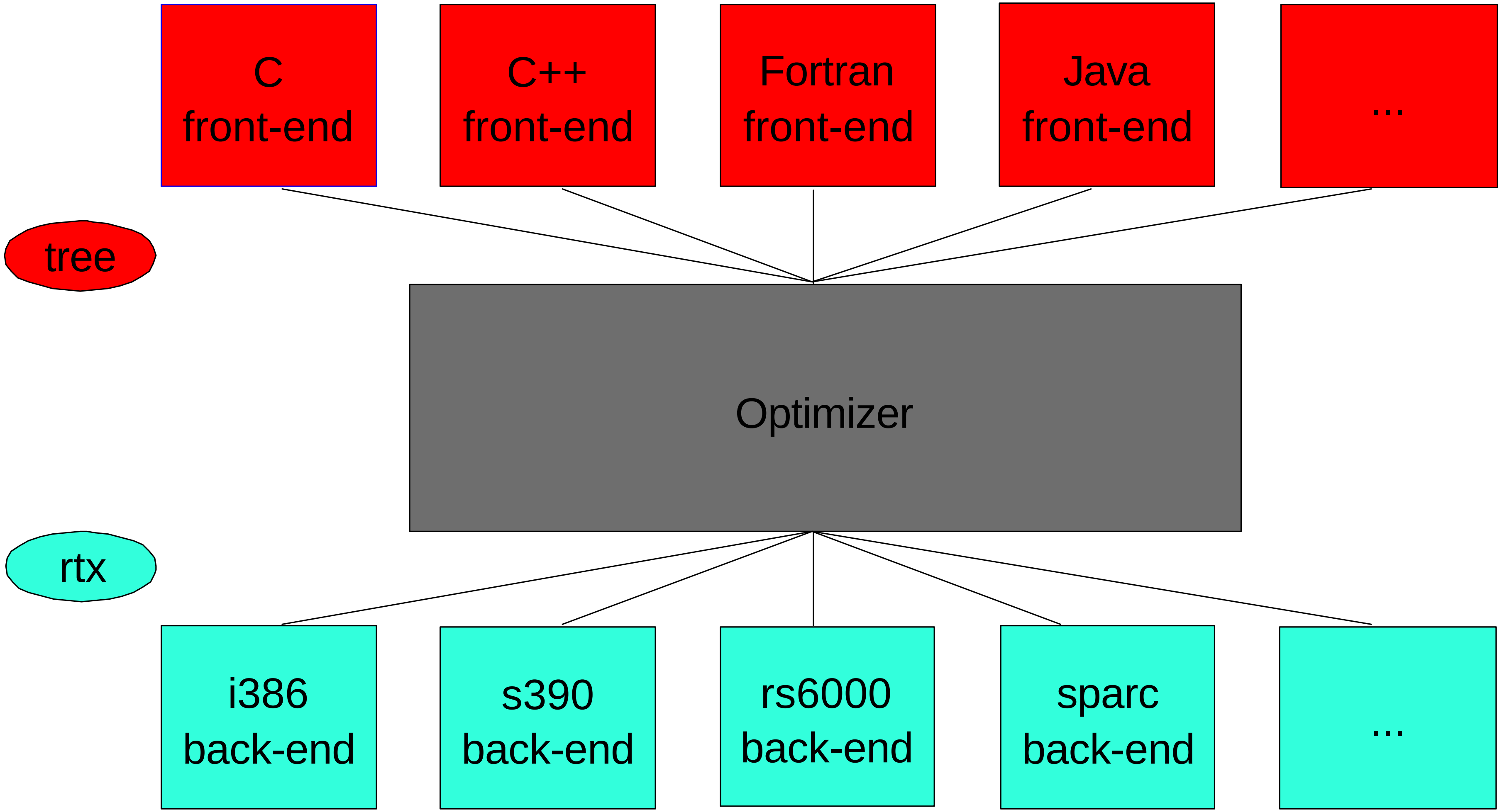
GCC Features

- Supported Languages
  - part of GCC distribution:
    - C, C++, Objective C
    - Fortran 77
    - Java
    - Ada
  - distributed separately:
    - Pascal
    - Modula-3
  - under development:
    - Fortran 95
    - Cobol

GCC Features (cont.)

- Supported CPU targets
  - i386, ia64, rs6000, s390
  - sparc, alpha, mips, arm, pa-risc, m68k, m88k
  - many embedded targets
- Supported OS bindings
  - Unix: Linux, *BSD, AIX, Solaris, HP/UX, Tru64, Irix, SCO
  - DOS/Windows, Darwin (MacOS X)
  - embedded targets and others
- Supported modes of operation
  - native compiler
  - cross-compiler
  - 'Canadian cross' builds

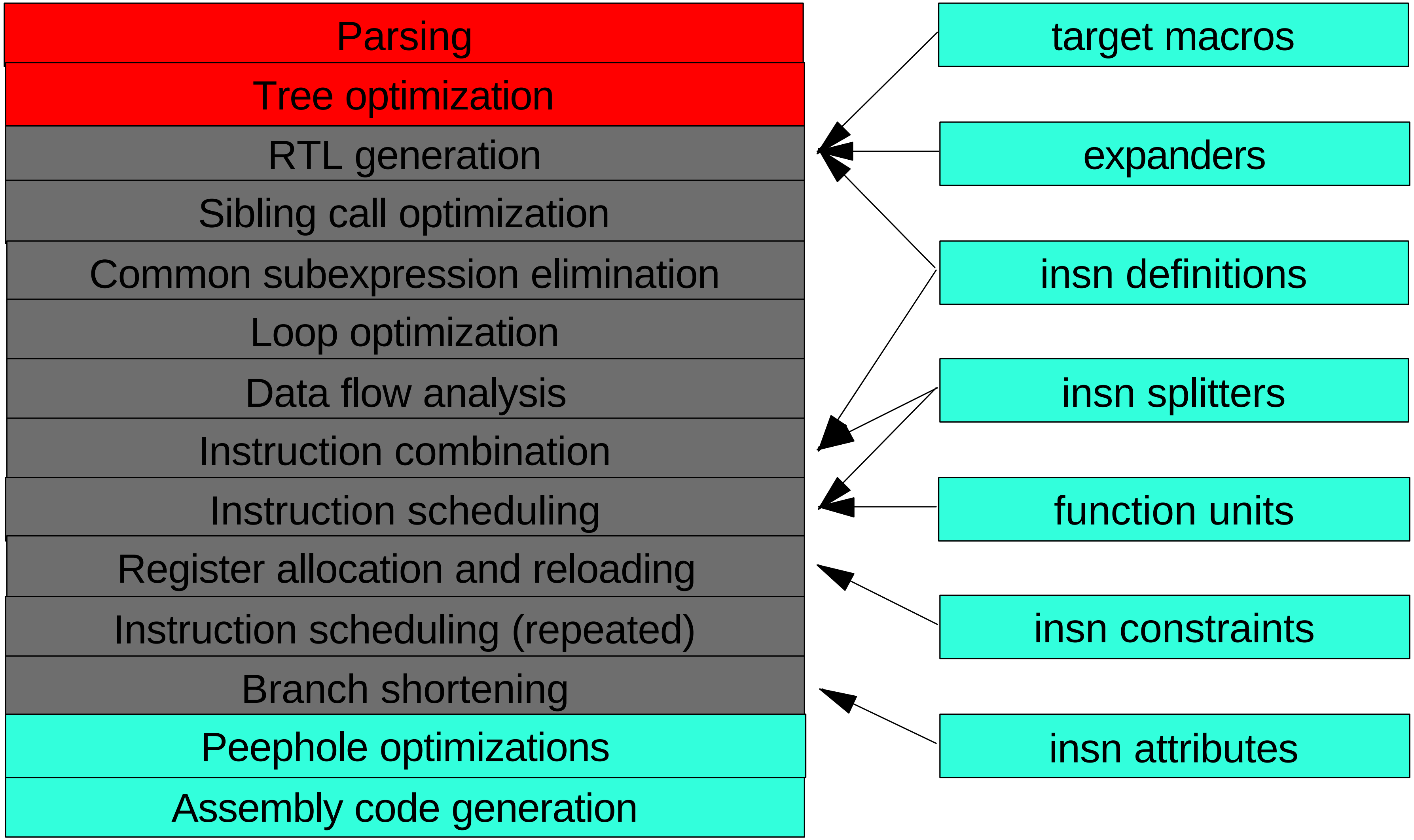
GCC Architecture: Overview



GCC Architecture: Passes

Parsing
Tree optimization
RTL generation
Sibling call optimization
Common subexpression elimination
Loop optimization
Data flow analysis
Instruction combination
Instruction scheduling
Register allocation and reloading
Instruction scheduling (repeated)
Branch shortening
Peephole optimizations
Assembly code generation

GCC Architecture: Passes



Parsing
Tree optimization
RTL generation
Sibling call optimization
Common subexpression elimination
Loop optimization
Data flow analysis
Instruction combination
Instruction scheduling
Register allocation and reloading
Instruction scheduling (repeated)
Branch shortening
Peephole optimizations
Assembly code generation

target macros
expanders
insn definitions
insn splitters
function units
insn constraints
insn attributes

GCC for zSeries: History

- Timeline
  - in 1998: Work on the S/390 backend started
  - in 1999: Linux for S/390 project started
  - December 1999: Code drop to developerWorks (gcc 2.95.1)
  - October 2000: Linux for S/390 GA distribution (gcc 2.95.2)
  - December 2000: Experimental 64-bit support (gcc 2.95.2)
  - April 2001: Merged 31-bit and 64-bit back-ends
  - June 2001: Improved back-end dropped (gcc 2.95.3)
  - July/August 2001: Integration into FSF CVS repository
  - August 2001: gcc 3.0.1 released
  - November 2002: gcc 3.2 based GA distribution

GCC for zSeries: Status

- Supported environments
  - 31-bit platform: ESA/390 + optional features
    - relative and immediate instructions (S/390 G2+)
    - IEEE floating point instructions (S/390 G5+)
  - 64-bit platform: z/Architecture
  - Linux ELF Application Binary Interface
- Performance
  - Competitive with other compilers on the platform
  - Many (but not all) GCC / platform features exploited
  - Still room for improvement

GCC for zSeries: Status

- Versions
  - gcc 2.95.2: superseded 31-bit only compiler
  - gcc 2.95.3: stable 31-bit and 64-bit compiler
    - Largest installed base (SuSE, Red Hat, Millennux, Debian)
    - Used to build most middleware and ISV software
  - gcc 3.0.x: Never in wide-spread use
  - gcc 3.1.x: Superseded by gcc 3.2 (ABI issues)
  - gcc 3.2.x: current recommended compiler
    - Used with recent/upcoming distributions
    - C++ compatibility/transition issues

GCC for zSeries: Status

- New features in gcc 3.2 vs. 2.95.3
  - Improved support for ISO C99 features
  - Improved ISO C++ standard conformance
  - Stable C++ ABI
  - Integrated C/C++ preprocessor
  - New optimization passes
  - Improved support for function inlining
  - Profile-directed optimizations
  - Internal infrastructure enhancements

GCC for zSeries: Challenges

- 'Unusual' architecture features
  - 31-bit addressing mode
  - Instruction-dependent address formats
  - Limited address displacements and immediate literals
  - Condition code handling

GCC for zSeries: Optimization example

- Source code

  -
    ```
    void f (long a)
    {
      if ((a & 32) && !(a & 4))
        g ();
    }
    ```
- Optimal translation into zSeries assembler
  - TEST UNDER MASK instruction: TMLL %reg,36
  - Check for condition code 2: Selected bits mixed zeros and ones, and leftmost is one

GCC for zSeries: Optimization example

- Non-optimized code

```
f:      stmg    %r11,%r15,88(%r15)
        larl    %r13,.L3
        aghi    %r15,-168
        lgr     %r11,%r15
        stg     %r2,160(%r11)
        lg      %r1,160(%r11)
        ng      %r1,.LC0-.L3(%r13)   # .quad 32
        ltgr    %r1,%r1
        je      .L1
        lg      %r1,160(%r11)
        ng      %r1,.LC1-.L3(%r13)   # .quad 4
        ltgr    %r1,%r1
        jne     .L1
        brasl   %r14,g
.L1:    lg      %r4,280(%r11)
        lmg     %r11,%r15,256(%r11)
        br      %r4
```

GCC for zSeries: Optimization example

- Optimized code (gcc 3.3 with -O1):
  - `f:`

```
        stmg    %r14,%r15,112(%r15)
        aghi    %r15,-160
        tmll    %r2,36
        jnh     .L1
        brasl   %r14,g
.L1:
        lg      %r4,272(%r15)
        lmg     %r14,%r15,272(%r15)
        br      %r4
```

Using GCC: Optimization

- -O0 (default): no optimization
  - shortest compilation time, best results when debugging
- -O1 (-O): default optimization
  - moderately increased compilation time
- -O2: heavy optimization
  - significantly increased compilation time
  - no optimizations with potentially adverse effects
- -O3: optimal execution time
  - may increase code size, may make debugging difficult
- -Os: optimal code size
  - may imply slower execution time than -O3

Using GCC: Function inlining

- What is function inlining?
  - Incorporate the called function's body into the caller
  - Replace formal parameters with arguments
- Benefits
  - Avoid function call overhead
  - Optimize combined function as a whole
- Disadvantages
  - Increased code size
  - Increased compilation time

Using GCC: Function inlining (cont.)

- Functions explicitly declared for inlining
  - Use **`inline`** keyword in function declaration
  - Define C++ member functions inside class body
- Functions automatically chosen for inlining
  - Heuristics based on function size and 'complexity'
  - Activated via **`-finline-functions`** (part of -O3)
- Inlining limits and overrides
  - Maximum size of inlined functions: **`-finline-limit=`***n*
  - Warn if non-inlined: **`-Winline`**
  - Force inlined: **`__attribute__((always_inline))`**
  - Force non-inlined: **`__attribute__((noinline))`**

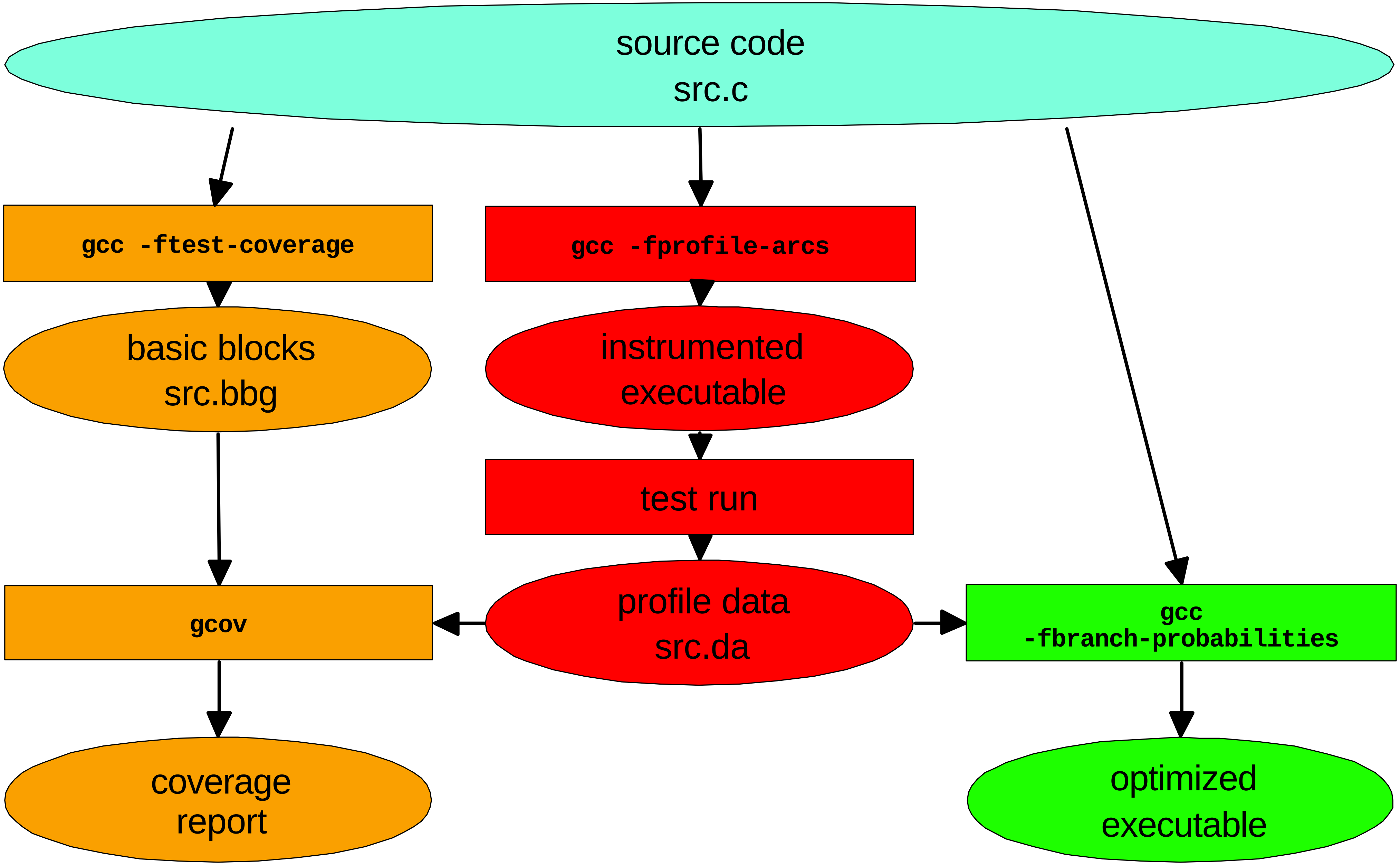
Using GCC: Profile-directed optimizations

- Basic blocks
  - Block of code that is always executed sequentially
  - Bounded by branches or branch target labels
- Program flow arcs
  - Potential transfers of control between basic blocks
  - Fall-through, branches, function call/return, exceptions
- Branch probabilities
  - How often is any given branch taken vs. non-taken?
  - How often is any given basic block executed?

Using GCC: Profile-directed optimizations

- Utilizing branch probability data
  - Profiling
  - Test coverage analysis
  - Profile-directed optimizations
- Generating branch probability data
  - Build instrumented executable: **`-fprofile-arcs`**
  - Generate basic block graph: **`-ftest-coverage`**
  - Profile-directed optimizations: **`-fbranch-probabilities`**
  - GNU test coverage tool: **`gcov`**

Using GCC: Profile-directed optimizations



source code
src.c

`gcc -ftest-coverage`

basic blocks
src.bbg

`gcc -fprofile-arcs`

instrumented
executable

test run

profile data
src.da

`gcov`

coverage
report

`gcc -fbranch-probabilities`

optimized
executable

Using GCC: Static branch probabilities

- Sources of branch probability data
  - Guessed by the compiler
  - Profile-directed feedback (**`-fbranch-probabilities`**)
  - Specified by the programmer (**`__builtin_expect`**)
- Using __builtin_expect
  - Specification:

    ```
    long __builtin_expect (long expression,
                           long expected)
    ```
  - Example:

    ```
    if (__builtin_expect (ptr == NULL, 0))
      error ();
    ```

Using GCC: Inline assembly

- Why inline assembly?
  - Use low-level architecture features (CS, STCK, ...)
  - Optimize hot spots
- GCC inline assembly features
  - Generate arbitrary assembler code
  - Access high-level data operands
  - Expose detailed semantics to the compiler
  - Fully participate in compiler optimizations

Using GCC: Inline assembly

- Syntax of "asm" construct

```
asm (   assembler template
      : output operands             [optional]
      : input operands              [optional]
      : clobber statments           [optional]);
```

- Assembler template
  - String passed to assembler
  - May contain operand placeholders **`%0`**, **`%1`**, ...
  - Registers specified as **`%%r0`**, **`%%r1`**, ...
- Clobber statements
  - Specify registers changed by template: **`"0"`**, **`"1"`**, ...
  - Special clobbers: **`"cc"`** (condition code), **`"memory"`**

Using GCC: Inline assembly

- Operand specification
  - Format: List of `"`***`constraint`***`" (`***`expression`***`)`
  - Constraint letters
    - "d" / "f" - general purpose / floating point register
    - "a" - address register (i.e. general purpose register except **`%r0`**)
    - "m" - general memory operand (base + index + displacement)
    - "Q" - S-operand (base + displacement) - gcc 3.3 only
    - "i" - immediate constant
  - Constraint modifier characters
    - "=" / "+" - write-only / read-write output operand
    - "&" - operand modified before all inputs are processed
  - Matching constraints
    - "0", "1", ... - operand must match specified operand number

Using GCC: Inline assembly examples

- Simple register constraint
  - `asm ("ear %0,%%a0" : "=d" (ar0_value));`
- Simple memory constraint
  - `asm ("cvb %0,%1" : "=d" (bin) : "m" (dec));`
- Handling S-operands
  - `asm ("stck %0" : "=Q" (time) : : "cc");`

  -
    ```
    asm ("stck 0(%0)" : : "a" (&time)
                        : "memory", "cc");
    ```

  -
    ```
    asm ("stck 0(%1)" : "=m" (time)
                      : "a" (&time) : "cc");
    ```

Using GCC: Inline assembly examples

- Compare and swap

  -
    ```
    asm ("cs %0,%3,0(%2)"
         : "=d" (old_val), "+m" (*loc)
         : "a" (loc), "d" (new_val),
           "0" (expected_val) : "cc");
    ```
- Atomic add (using compare and swap)

  -
    ```
    asm ("0: lr  %1,%0\n\t"
         "   ar  %1,%4\n\t"
         "   cs  %0,%1,0(%3)\n\t"
         "   jl  0b"
         : "=&d" (old_val), "=&d" (new_val),
           "=m" (*counter)
         : "a" (counter), "d" (increment),
           "0" (*counter)
         : "cc");
    ```

Using GCC: Inline assembly examples

- System call (using register asm variables)

  -
    ```
    int read(int fd, char *buf, off_t count) {
        register int __arg1 asm("2") = fd;
        register char *__arg2 asm("3") = buf;
        register off_t __arg3 asm("4") = count;
        register int __res asm("2");
        __asm__ __volatile__ (
                    "svc %1"
                    : "=d" (__res)
                    : "i" (__NR_read),
                      "0" (__arg1),
                      "d" (__arg2),
                      "d" (__arg3)
                    : "cc", "memory" );
        return __res;
    }
    ```

Future of GCC

- gcc 3.3 (scheduled for March 2003)
  - Improved profile-directed optimizations
  - Improved instruction scheduling
  - Type-based alias analysis for C++ aggregate types
  - Thread-local storage support
  - Enable full Java support on zSeries
  - Bi-arch compile support for zSeries
  - Miscellaneous zSeries back-end performance optimizations

Future of GCC (cont.)

- gcc 3.4 (estimated Year End 2003)
  - Precompiled header support for C/C++/Objective-C
  - New C++ parser for full ISO C++ conformance
  - Improved loop optimizer (?)
  - Improved register allocator (?)
  - Tree-based optimization passes (?)
  - Compile-time speed enhancements (?)
  - More zSeries back-end improvements

Resources

- GNU Compiler Collection home page

  http://gcc.gnu.org

- Linux for zSeries developerWorks page

  http://www.software.ibm.com/
  developerworks/opensource/linux390/index.html

- Linux for zSeries technical contact address

  linux390@de.ibm.com