



redhat.

GCC

Yesterday, Today and Tomorrow

Diego Novillo

dnovillo@redhat.com

Red Hat Canada

2nd HiPEAC GCC Tutorial

Ghent, Belgium

January 2007

Yesterday

Brief History

- GCC 1 (1987)
 - Inspired on Pastel compiler (Lawrence Livermore Labs)
 - Only C
 - Translation done one statement at a time
- GCC 2 (1992)
 - Added C++
 - Added RISC architecture support
 - Closed development model challenged
 - New features difficult to add

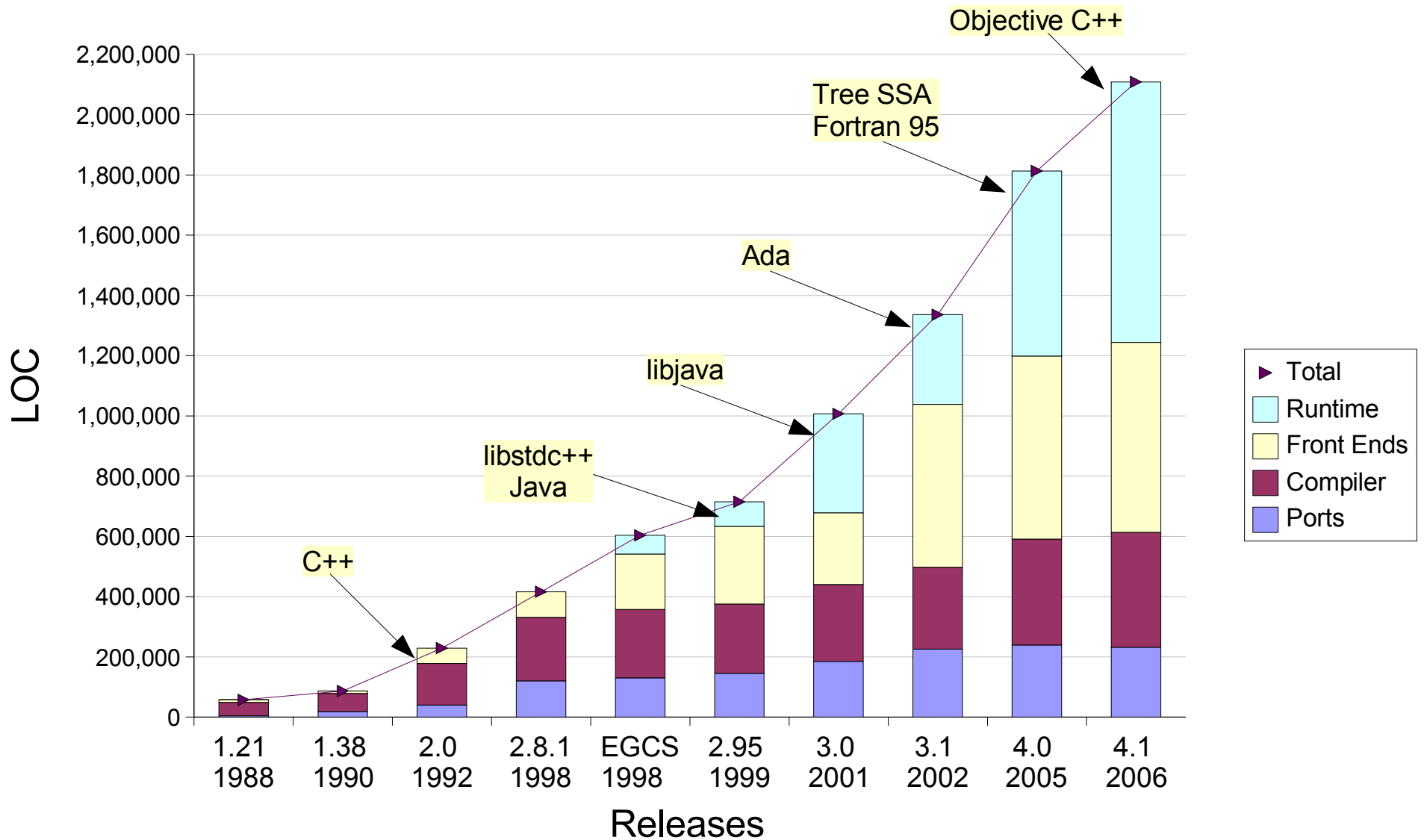
Brief History

- EGCS (1997)
 - Fork from GCC 2.x
 - Many new features: Java, Chill, numerous embedded ports, new scheduler, new optimizations, integrated libstdc++
- GCC 2.95 (1999)
 - EGCS and GCC2 merge into GCC
 - Type based alias analysis
 - Chill front end
 - ISO C99 support

Brief History

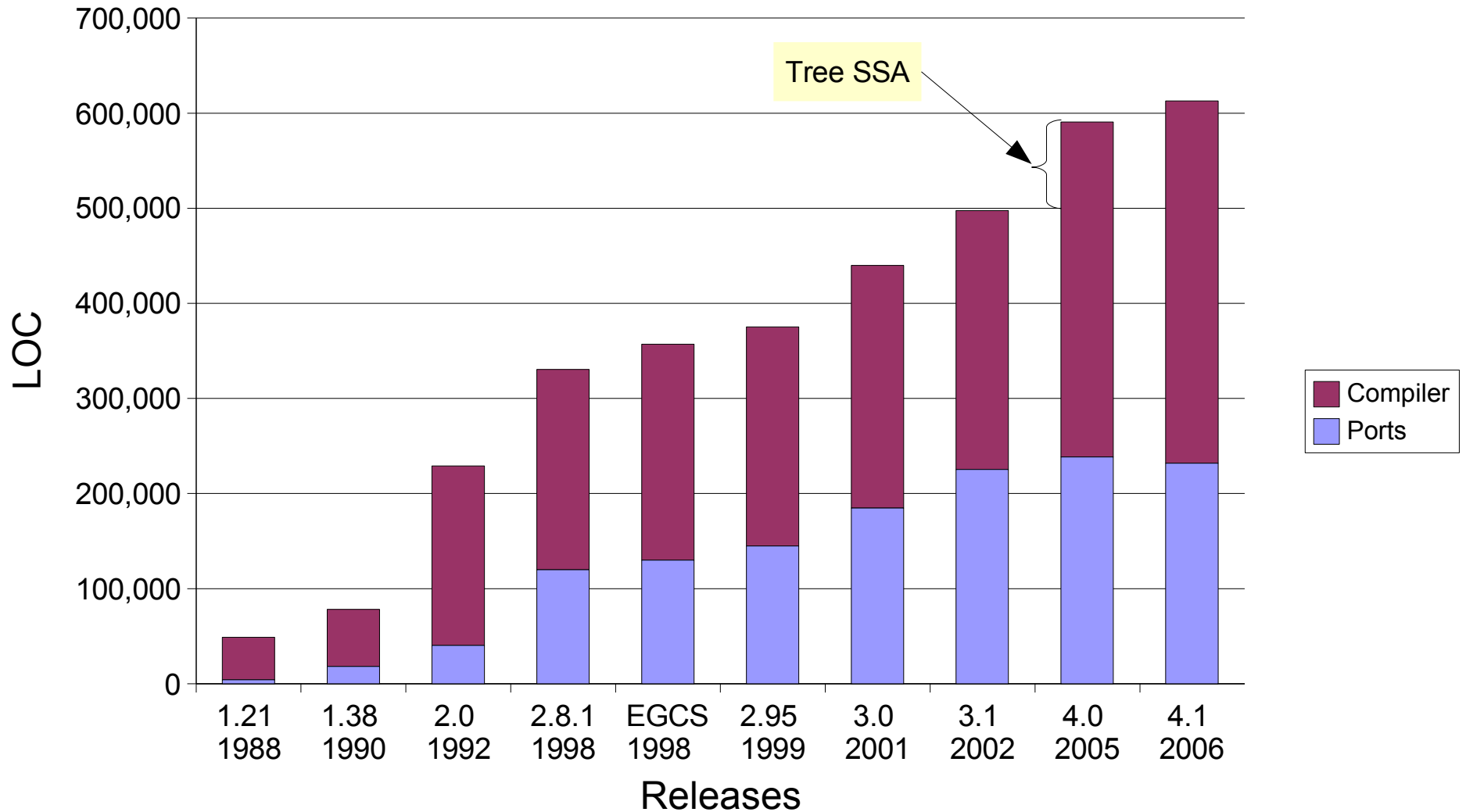
- GCC 3 (2001)
 - Integrated libjava
 - Experimental SSA form on RTL
 - Functions as trees (crucial feature)
- GCC 4 (2005)
 - Internal architecture overhaul (Tree SSA)
 - Fortran 95
 - Automatic vectorization

GCC Growth¹



¹generated using David A. Wheeler's 'SLOCCount'.

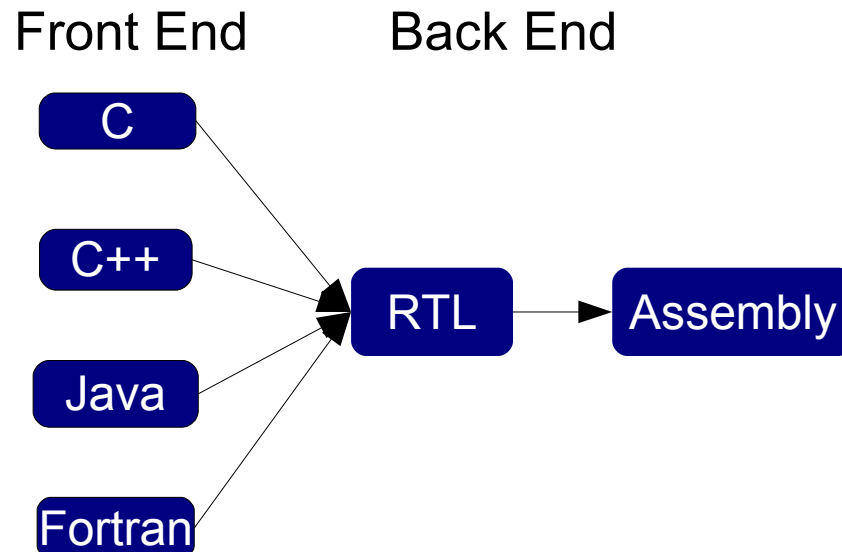
Core Compiler Growth¹



¹generated using David A. Wheeler's 'SLOCCount'.

Growing pains

- Monolithic architecture
- Front ends too close to back ends
- Little or no internal interfaces
- RTL inadequate for high level optimizations



Tree SSA Design

- Goals
 - Separate FE from BE
 - Evolution vs Revolution
- Needed IL layers
 - Two ILs to choose from: Tree and RTL
 - Moving down the abstraction ladder seemed simpler
- Started with FUD chains over C Trees
 - Every FE had its own variant of Trees
 - Complex grammar, side-effects, language dependencies

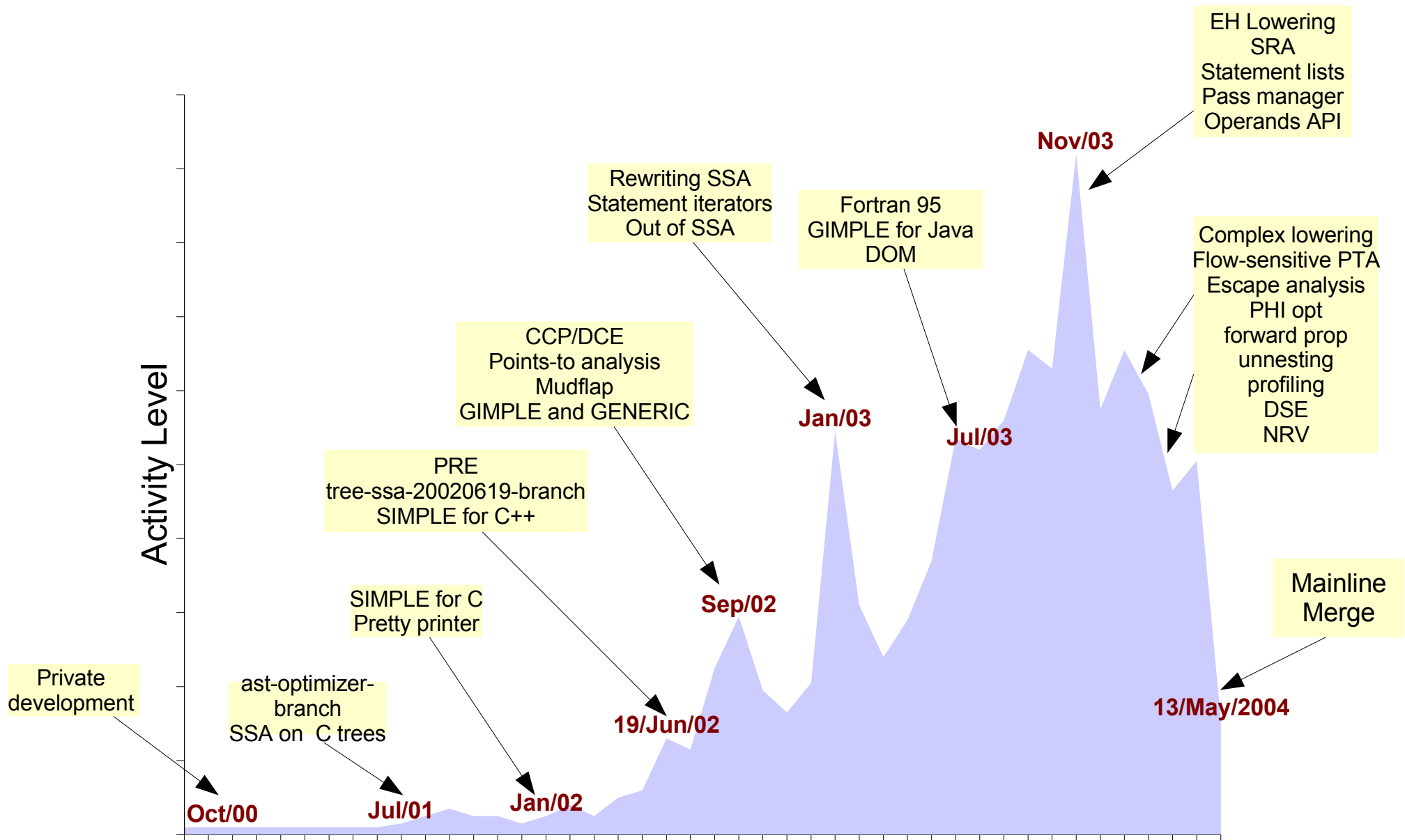
Tree SSA Design

- SIMPLE (McGill University)
 - Used same `tree` data structure
 - Simplified and restrictive grammar
 - Started with C and followed with C++
 - Later renamed to GIMPLE
 - Still not enough
- GENERIC
 - Target IL for every FE
 - No grammar restrictions
 - Only required to remove language dependencies

Tree SSA Design

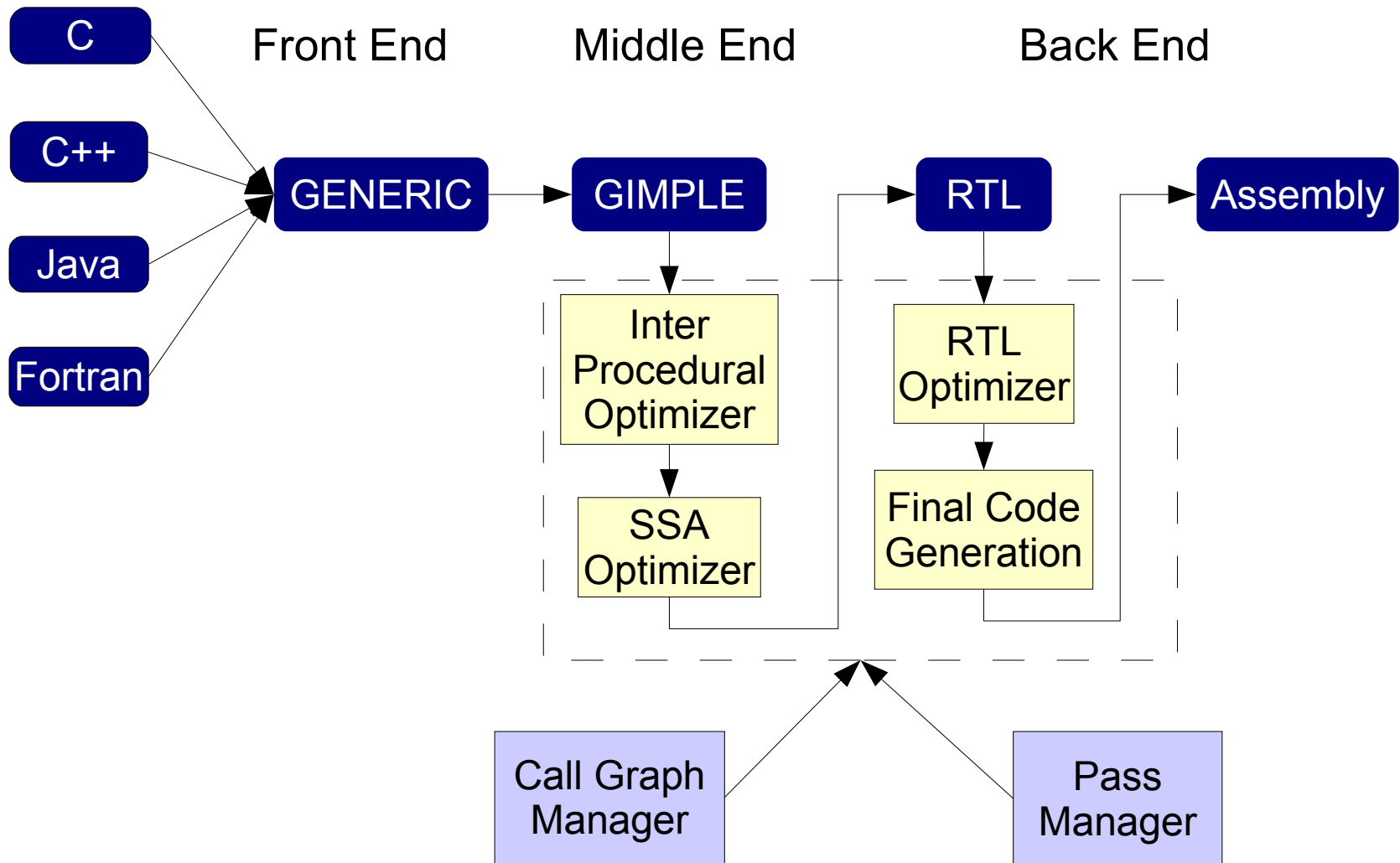
- FUD-chains unpopular
 - No overlapping live ranges (OLR)
 - Limits some transformations (e.g., copy propagation)
- Replaced FUD-chains with rewriting form
 - Kept FUD-chains for memory expressions (Virtual SSA)
 - Needed out-of-SSA pass to cope with OLR
- Several APIs
 - CFG, statement, operand manipulation
 - Pass manager
 - Call graph manager

Tree SSA Timeline



Today

Compiler pipeline



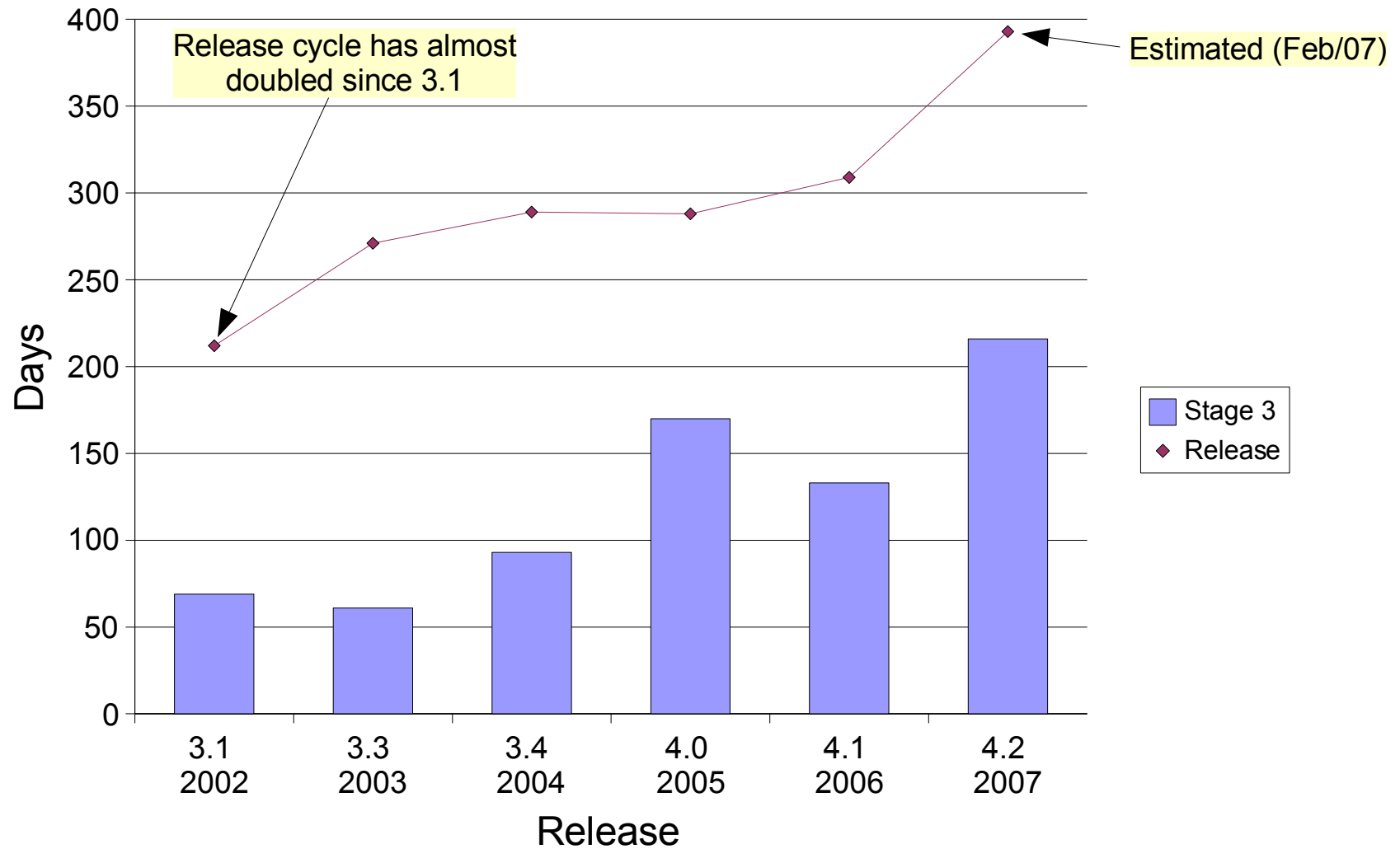
Major Features

- De-facto system compiler for Linux
- No central planning (controlled chaos)
- Supports
 - All major languages
 - An insane number of platforms
- SSA-based high-level global optimizer
- Automatic vectorization
- OpenMP support
- Pointer checking instrumentation for C/C++

Major Issues

- Popularity
 - Caters to a wide variety of user communities
 - Moving in different directions at once
- No central planning (controlled chaos)
- Not enough engineering cycles
- Risk of “featuritis”
 - Too many flags
 - Too many passes
- Warnings depending on optimization levels

Longer Release Cycles



A Few Current Projects

- RTL cleanups
- OpenMP
- Interprocedural analysis framework
- GIMPLE tuples
- Link-time optimization
- Memory SSA
- Sharing alias information across ILs
- Register allocation
- Scheduling

RTL Cleanups

- Removal of duplicate functionality
 - Mostly done
 - Goal is for RTL to only handle low-level issues
- Dataflow analysis
 - Substitute ad-hoc flow analysis with a generic DF solving framework
 - Increased accuracy. Allows more aggressive transformations
 - Support for incremental dataflow information
 - Backends need taming

OpenMP

- Pragma-based annotations to specify parallelism
- New hand-written recursive-descent parser eased pragma recognition
- GIMPLE extended to support concurrency
- Supports most platforms with thread support
- Available now in Fedora Core 6's compiler
- Official release will be in GCC 4.2

OpenMP

```
#include <omp.h>

main()
{
    #pragma omp parallel
    printf ("[%d] Hello\n", omp_get_thread_num());
}
```

```
$ gcc -fopenmp -o hello hello.c
$ ./hello
[2] Hello
[3] Hello
[0] Hello ← Master thread
[1] Hello
```

```
$ gcc -o hello hello.c
$ ./hello
[0] Hello
```

Interprocedural Analysis

- Keep the whole call-graph in SSA form
- Increased precision for inter-procedural analyses and transformations
- Unify internal APIs to support inter/intra procedural analysis
 - No special casing in pass manager
 - Improve callgraph facilities
- Challenges
 - Memory consumption
 - Privatization of global attributes

GIMPLE Tuples

- GIMPLE shares the `tree` data structure with FE
- Too much redundant information
- A separate tuple-like data structure provides
 - Increased separation with FE
 - Memory savings
 - Potential compile time improvements
- Challenges
 - Shared infrastructure (e.g. `fold()`)
 - Compile time increase due to conversion

Link Time Optimization

- Ability to stream IL to support IPO across
 - Multiple compilation units
 - Multiple languages
- Streamed IL representation treated like any other language
- Challenges
 - Original language must be represented explicitly
 - Complete compiler state must be preserved
 - Conflicting flags used in different modules

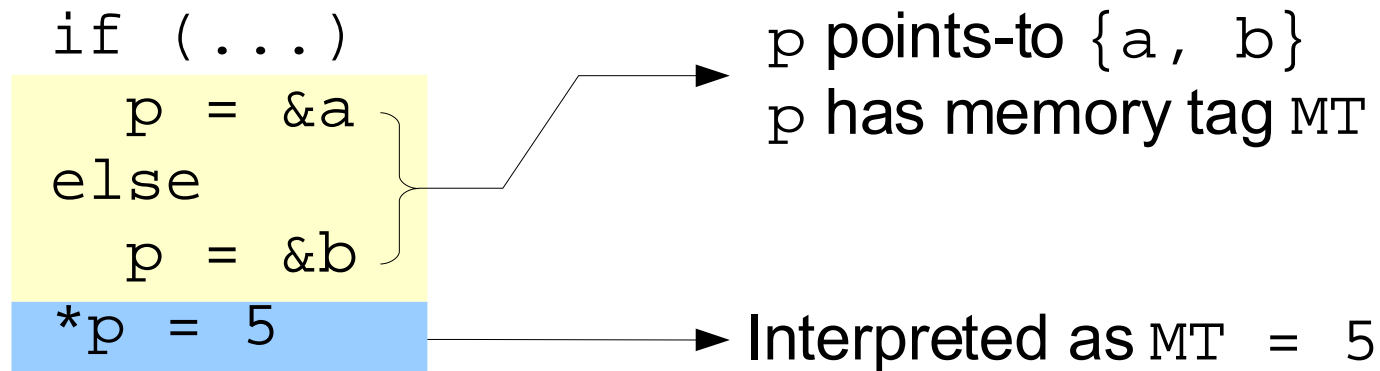
Alias Analysis

Overview

- GIMPLE represents alias information explicitly
- Alias analysis is just another pass
 - Artificial symbols represent memory expressions (virtual operands)
 - FUD-chains computed on virtual operands → Virtual SSA
- Transformations may prove a symbol non-addressable
 - Promoted to GIMPLE register
 - Requires another aliasing pass

Symbolic Representation of Memory

- Pointer P is associated with memory tag MT
 - MT represents the set of variables pointed-to by P
- So $*P$ is a reference to MT



Associating Memory with Symbols

- Alias analysis
 - Builds points-to sets and memory tags
- Structural analysis
 - Builds field tags (sub-variables)
- Operand scanner
 - Scans memory expressions to extract tags
 - Prunes alias sets based on expression structure

Alias Analysis

- Points-to alias analysis (PTAA)
 - Based on constraint graphs
 - Field and flow sensitive, context insensitive
 - Intra-procedural (inter-procedural in 4.2)
 - Fairly precise
- Type-based analysis (TBAA)
 - Based on input language rules
 - Field sensitive, flow insensitive
 - Very imprecise

Alias Analysis

- Two kinds of pointers are considered
 - Symbols: Points-to is flow-insensitive
 - Associated to Symbol Memory Tags (SMT)
 - SSA names: Points-to is flow-sensitive
 - Associated to Name Memory Tags (NMT)
- Given pointer dereference $*ptr_{42}$
 - If ptr_{42} has NMT, use it
 - If not, fall back to SMT associated with ptr

Structural Analysis

- Separate structure fields are assigned distinct symbols

```
struct A
{
  int x;
  int y;
  int z;
};
```

```
struct A a;
```

- Variable a will have 3 sub-variables
{ SFT.1, SFT.2, SFT.3 }
- References to each field are mapped to the corresponding sub-variable

IL Representation

```
foo (i, a, b, *p)
{
  p =(i > 10) ? &a : &b
  *p = 3
  return a + b
}
```

```
foo (i, a, b, *p)
{
  p = (i > 10) ? &a : &b
```

```
# a = VDEF <a>
# b = VDEF <b>
*p = 3
```

```
# VUSE <a>
t1 = a
```

```
# VUSE <b>
t2 = b
```

```
t3 = t1 + t2
return t3
}
```

Virtual SSA Form

- VDEF operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

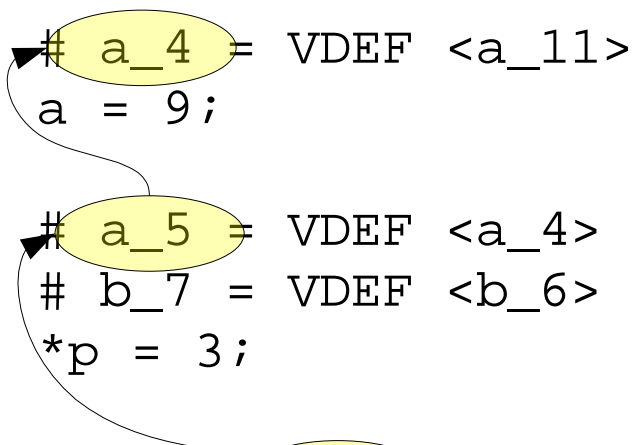
```
foo (i, a, b, *p)
{
    p_2 = (i_1 > 10) ? &a : &b

    # a_4 = VDEF <a_11>
    a = 9;

    # a_5 = VDEF <a_4>
    # b_7 = VDEF <b_6>
    *p = 3;

    # VUSE <a_5>
    t1_8 = a;

    t3_10 = t1_8 + 5;
    return t3_10;
}
```



Virtual SSA – Problems

- Big alias sets → Many virtual operators
 - Unnecessarily detailed tracking
 - Memory
 - Compile time
 - SSA name explosion

Memory SSA

- New representation of aliasing information
- Big alias sets → Many virtual operators
 - Unnecessarily detailed tracking
 - Memory, compile time, SSA name explosion
- Main idea
 - Stores to many locations create a single name
 - Factored name becomes reaching definition for all symbols involved in store

Memory SSA

```
# .MEM_10 = VDEF <.MEM_0>  
*p_3 = ...
```

p_3 points-to { a, b, c }

```
# .MEM_11 = VDEF <.MEM_0>  
*q_4 = ...
```

q_4 points-to { n, o, p }

```
# b_12 = VDEF <.MEM_10>  
b = ...
```

At most one VDEF and one VUSE per statement

```
# .MEM_13 = VDEF <.MEM_10, b_12>  
*p_3 = ...
```

Virtual operators may refer to more than one operand

```
# VUSE <.MEM_13>  
t_14 = b
```

```
# VUSE <.MEM_11>  
t_15 = o
```

Factored stores create “sinks” that group multiple incoming names

Memory SSA

- Challenges
 - Overlapping live ranges create a problem for some passes
 - Requires more detailed tracking in SSA rewriting
 - Dynamic association between Φ nodes and symbols
- Memory partitions
 - Have more than one .MEM object
 - Create static associations between symbols
 - Association is independent from alias relations
 - Heuristics control partitioning

Tomorrow

Future Work

- Wide variety of projects
- A small sample
 - Plug-in support
 - Scheduling
 - Register pressure reduction
 - Register allocation
 - Incremental compilation
 - Dynamic compilation
 - Dynamic optimization pipeline

Plug-in Support

- Extensibility mechanism to allow 3rd party tools
- Wrap some internal APIs for external use
- Allow loading of external shared modules
 - Loaded module becomes another pass
 - Compiler flag determines location
- Versioning scheme prevents mismatching
- Useful for
 - Static analysis
 - Experimenting with new transformations

Scheduling

- Several concurrent efforts targetting 4.3 and 4.4
 - Schedule over larger regions for increased parallelism
 - Most target IA64, but benefit all architectures
- Enhanced selective scheduling
- Treeregion scheduling
- Superblock scheduling
- Improvements to swing modulo scheduling

Register Allocation

- Several efforts over the years
- Complex problem
 - Many different targets to handle
 - Interactions with reload and scheduling
- YARA (Yet Another Register Allocator)
 - Experimented with several algorithms
- IRA (Integrated Register Allocator)
 - Priority coloring, Chaitin-Briggs and region based
 - Expected in 4.4
 - Currently works on x86, x86-64, ppc, IA64, sparc, s390

Register pressure reduction

- SSA may cause excessive register pressure
 - Pathological cases → ~800 live registers
 - RA battle lost before it begins
- Short term project to cope with RA deficiencies
- Implement register pressure reduction in GIMPLE before going to RTL
 - Pre-spilling combined with live range splitting
 - Load rematerialization
 - Tie RTL generation into out-of-ssa to allow better instruction selection for spills and rematerialization

Dynamic compilation

- Delay compilation until runtime (JIT)
 - Emit bytecodes
 - Implement virtual machine with optimizing transformations
- Leverage on existing infrastructure (LLVM, LTO)
- Not appropriate for every case
- Challenges
 - Still active research
 - Different models/costs for static and dynamic compilers

Incremental Compilation

- Speed up edit-compile-debug cycle
- Speeds up ordinary compiles by compiling a given header file “once”
- Incremental changes fed to compiler daemon
- Incremental linking as well
- Side effects
 - Refactoring
 - Cross-referencing
 - Compile-while-you-type (e.g., Eclipse)

Dynamic Optimization Pipeline

- Phase ordering not optimal for every case
- Current static ordering difficult to change
- Allow external re-ordering
 - Ultimate control
 - Allow experimenting with different orderings
 - Define `-On` based on common orderings
- Problems
 - Probability of finding bugs increases
 - Enormous search space