CSC 408F/CSC2105F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC 408H/CSC 2105H in the Fall term 2004/2005 at the University of Toronto.

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

Software Configuration Management

- Manage the *structure* of a software system over its lifetime
- Software Configuration Management includes
  - **System Modeling** at the module interconnection level
  - **Composition** - Managing system building and integration
  - **Version control** - managing and controlling source code evolution
  - **Change control** - managing and controlling changes to the system
  - **Release Management** - managing the software Release process
- Software configuration management attempts to bring order to the chaos of a continually evolving software system
- Chaos is increased by current trends toward distributed, concurrent, multinational development of software.

System Modeling

- Assume that a software system is composed of an arbitrary collection of modules.
  - Each module has an *implementation part (body)* that implements one or more *interfaces*
  - An *interface* is the link between module(s) that provide a service and the client(s) that use the service
  - Each interface is implemented by one or more modules. i.e. several modules can cooperatively implement an interface
  - The connection between clients and servers (through interfaces) can be arbitrary
  - Module bodies and interfaces evolve over time as a system is developed and maintained
- A *System Model* is a *complete and detailed* description of the client/server relationships in a software system at a given point in time

Identification and Interconnection

- An item in a software system is either a *base item* or a *composite (or derived) item*

- As a software system evolves over time, there will be many *versions* of each base item

- A composite item is specified by rule(s) for deriving it from specific instances of base items

System Model Example

Module	Files	Version	Timestamp	Directory
main	main.c	3.10.2	2000.10.21/10:35:03	/project/projectA/master/driver/
	utillib.a	4.25	2000.11.06:14:30:25	/utils/util/current/lib
utillib	utility.h	4.25.17	2000.11.04/19:42:37	/utils/utilib/current/src/
	utility.c	4.25.27	2000.11.05/10:03:05	/utils/utilib/current/src/
	utildebug.c	4.25.01	2000.09.25/11:35:42	/utils/utilib/current/src

- System model requires several coordinate systems
  - File version number.
  - File time stamps
  - Host (not shown above) and directory
- These coordinate systems must be maintained over time This is difficult in a distributed development environment.

System Modeling

- **Consistent Composition** - A software system is *consistently composed* if for every client/server relationship, the client and the server(s) agree *exactly* on the interface between them.

  Lack of agreement constitutes an error in the software system

- For every client/server relationship in a software system the System Model must specify

  - The version of the interface used to create the relationship
  - The version of the module(s) used to implement this relationship

  This information is required to correctly compose the system

- Clients, servers and interfaces and therefore system models evolve over time as a system is developed and maintained

- The System Model is used to select *versions* of software components when the system is being composed

The System Composition Process

- Input: System Model and software source/binaries

  Output: a correctly composed system

- Assume: modules are separately compiled to produce binary modules

  A formal *interface* is the only compile time link between client and server modules

- For each client/server relationship specified in the System Model, check that the client and server were compiled with the *same* version of the interface between them

- Use the linkage editor to bind the binary modules together into a complete executable system, using the binding rules expressed in the System Model

The Consistent Recompilation Problem

- Assume that some interface in the system has been changed as a result of development or maintenance activity

- **Question:** *What is the smallest set of modules that must be recompiled so as to make consistent composition possible ?*

- **Answer:**

  1. All of the modules that implement the interface
  2. All modules that use the modified interface directly
  3. All modules in the transitive completion of the client relationships originating from the server modules compiled in step 2

- Note the possibility of a *phase error*, an interface and the corresponding implementation are changed after a client has been compiled using the previous version of the interface

A tool to reduce recompilations

- Basic concepts
  - Here an interface is a header, “.h” file
  - Here a module is a compilation unit, “.c” file
  - An implementing module is a module implementing the interface
  - A dependent module is a module using the interface
- Remove redundancies
  - Here is a module “hello.c”:

    ```
    #include <stdio.h>
    int main(int argc, char * argv) {
       printf("Hello, world!");
       return 0;
    }
    ```

  - Preprocessor `gcc -E -P hello.c` expands the module into 745 lines of code! That’s what happened in compilation.

- The following program is equivalent:

```
extern int printf (__const char *__restrict __format, ...);
int main(int argc, char ** argv)
{
        printf("Hello, world!");
        return 0;
}
```

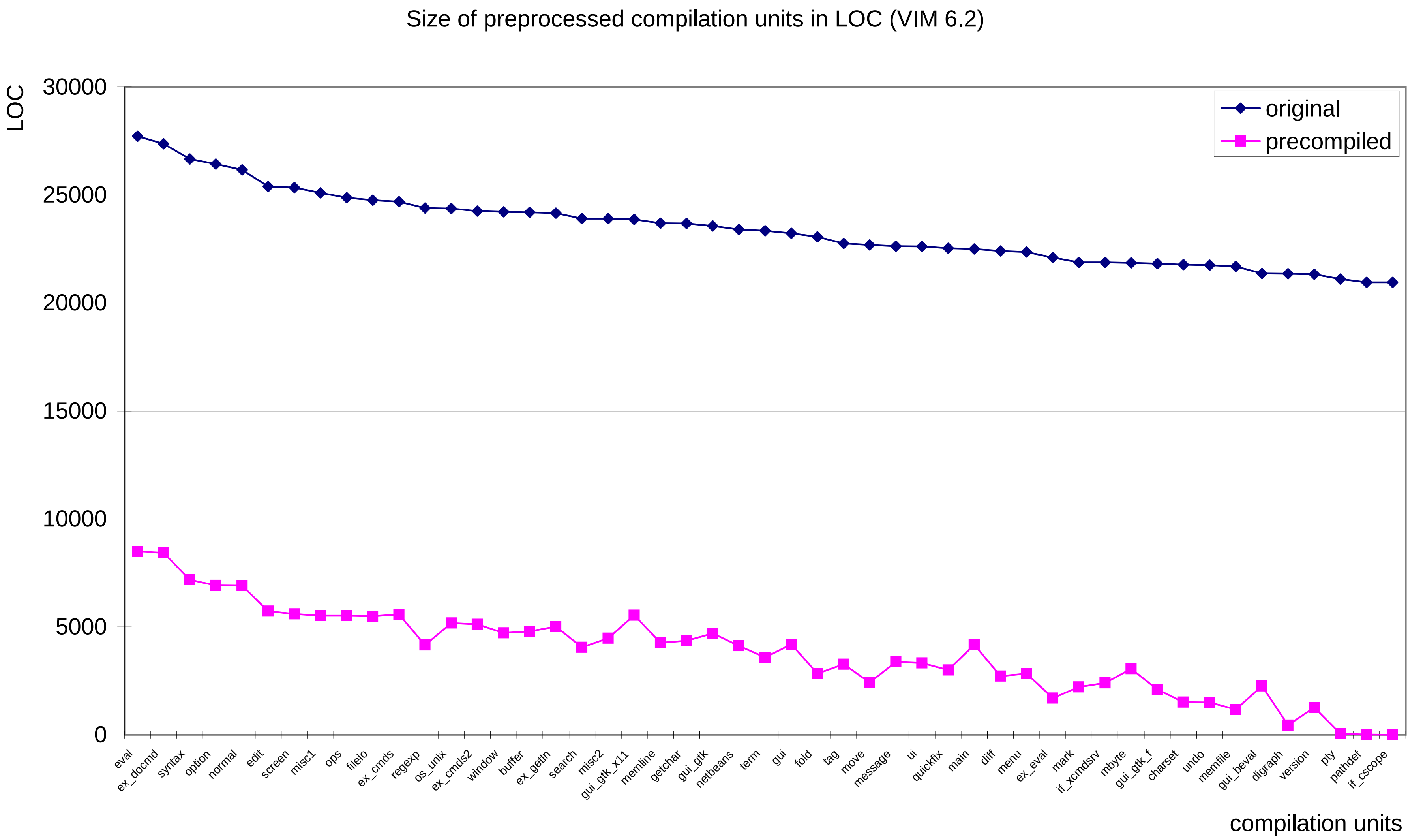
- What are the benefits of removing redundancies?
    - ∗ Smaller precompiled code size.
    - ∗ Faster compilation of the module.
    - ∗ Smaller module interface.
    - ∗ Compiler independent code.
- What are the drawbacks?
    - ∗ Whenever there is a change to the interface, the recompilation is not reduced since you need to re-embed them into the modules.
    - ∗ How to mitigate the drawbacks? (1) Automatically remove redundancies; (2) Header restructuring.

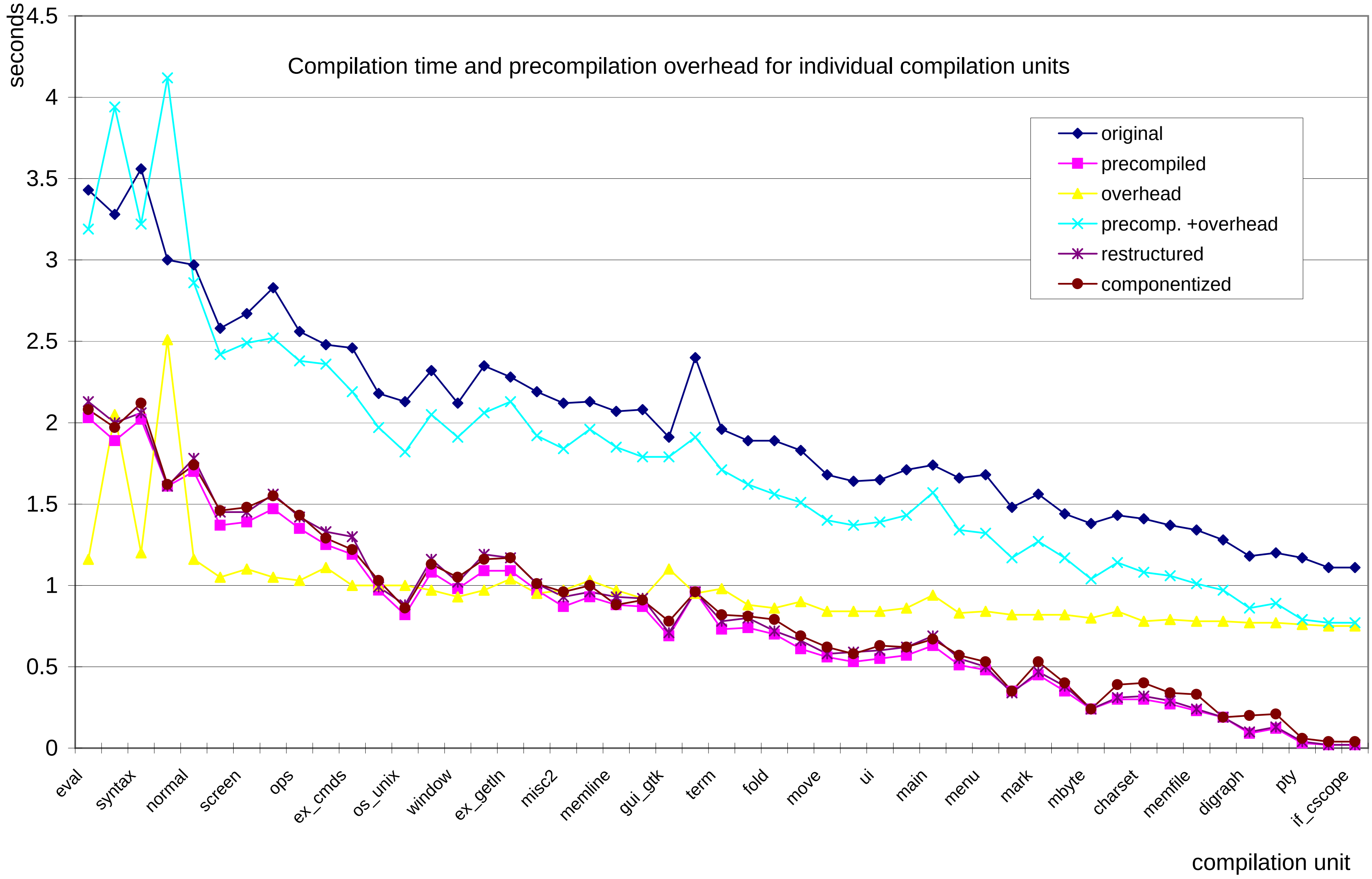
- Remove false dependencies
  - What is a false dependency? A module depends on an interface that is larger than necessary.

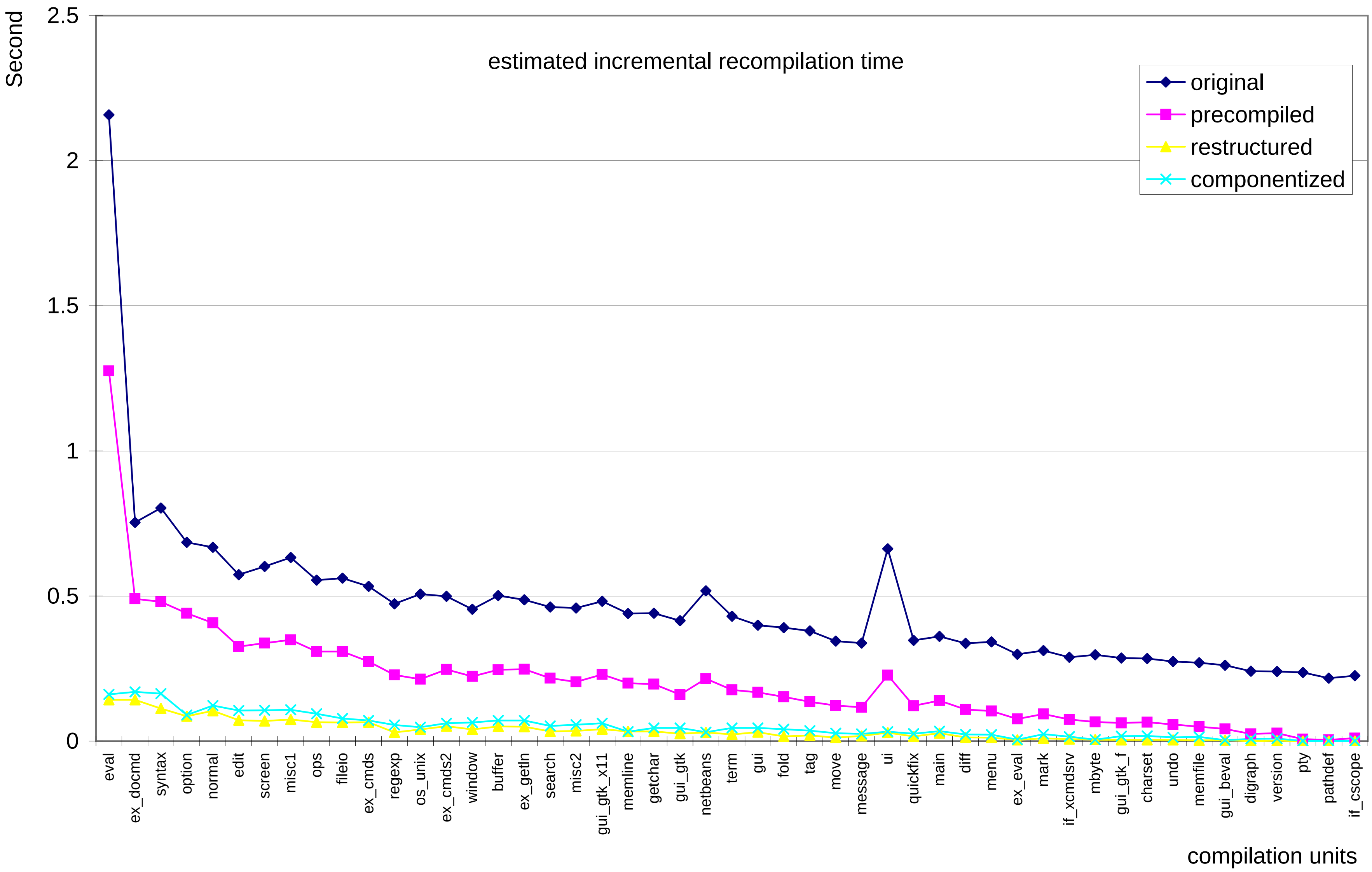
```
/* header.h */
void foo();
void bar();
/* module.c */
#include "header.h"
void moo() {
  foo();
}
---------------------
/* header’.h */
void foo();
/* module’.c */
#include "header’.h"
void moo() {
  foo();
}
```

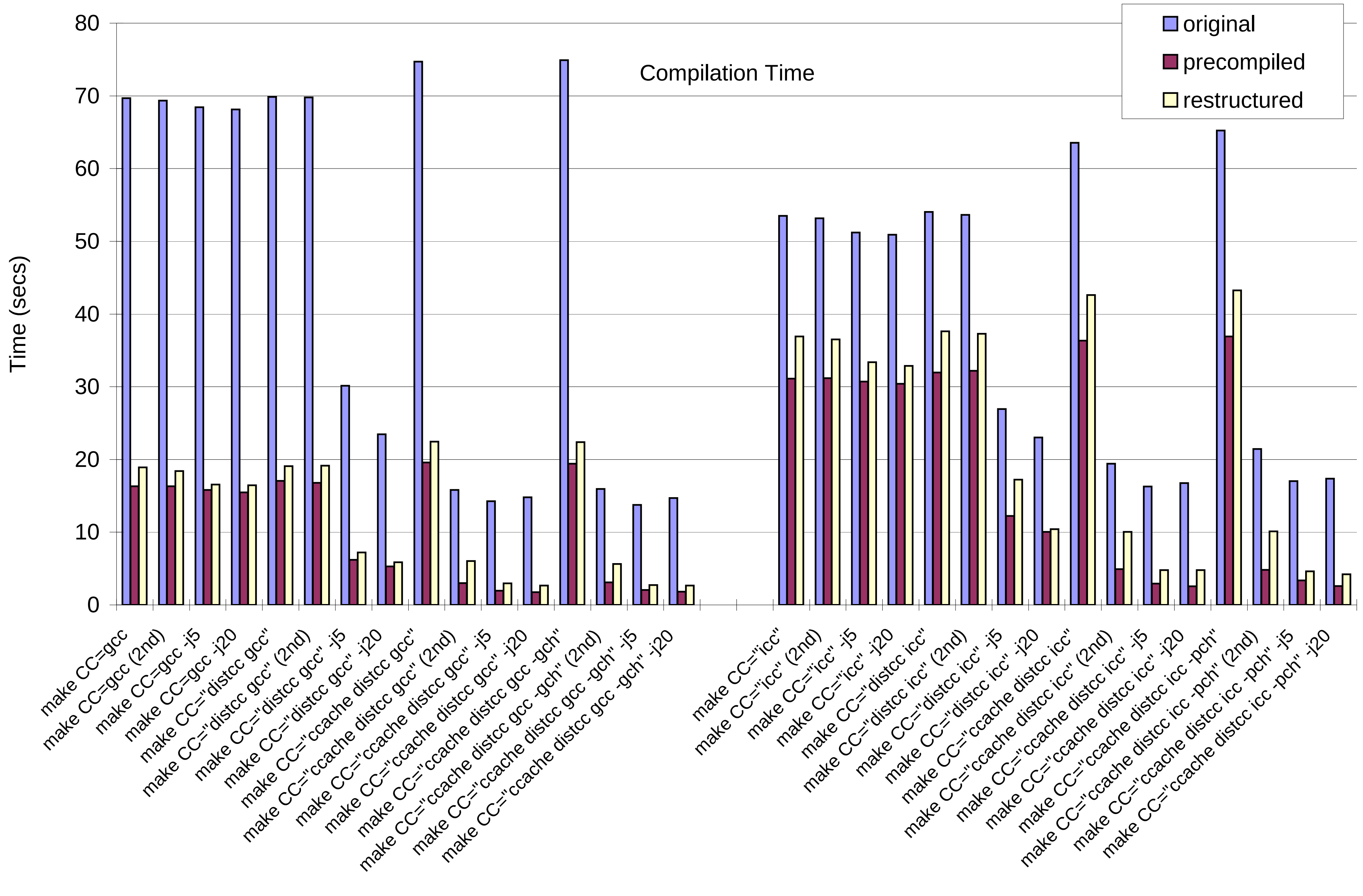
- What are the benefits?
  - Keeping the interfaces, but removing all false dependencies
  - Reducing recompilations of the dependent modules whenever a change to the refactored interface
- What is the draback?
  - Too many interfaces generated
- Further improvements:
  - Componentization: group modules into components, for example, Model-View-Controller
  - Group sub-module interfaces into component interfaces
  - Minimal number of interfaces generated.

- Experiments: VIM 6.2


Size of preprocessed compilation units in LOC (VIM 6.2)
LOC
30000
25000
20000
15000
10000
5000
0
original
precompiled
eval
ex_docmd
syntax
option
normal
edit
screen
misc1
ops
fileio
ex_cmds
regexp
os_unix
ex_cmds2
window
buffer
ex_getln
search
misc2
gui_gtk_x11
memline
getchar
gui_gtk
netbeans
term
gui
fold
tag
move
message
ui
quickfix
main
diff
menu
ex_eval
mark
if_xcmdsrv
mbyte
gui_gtk_f
charset
undo
memfile
gui_beval
digraph
version
pty
pathdef
if_cscope
compilation units


Compilation time and precompilation overhead for individual compilation units
seconds
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
original
precompiled
overhead
precomp. +overhead
restructured
componentized
eval
syntax
normal
screen
ops
ex_cmds
os_unix
window
ex_getln
misc2
memline
gui_gtk
term
fold
move
ui
main
menu
mark
mbyte
charset
memfile
digraph
pty
if_cscope
compilation unit


estimated incremental recompilation time
Second
2.5
2
1.5
1
0.5
0
original
precompiled
restructured
componentized
eval
ex_docmd
syntax
option
normal
edit
screen
misc1
ops
fileio
ex_cmds
regexp
os_unix
ex_cmds2
window
buffer
ex_getln
search
misc2
gui_gtk_x11
memline
getchar
gui_gtk
netbeans
term
gui
fold
tag
move
message
ui
quickfix
main
diff
menu
ex_eval
mark
if_xcmdsrv
mbyte
gui_gtk_f
charset
undo
memfile
gui_beval
digraph
version
pty
pathdef
if_cscope
compilation units


Compilation Time
Time (secs)
0
10
20
30
40
50
60
70
80
original
precompiled
restructured
make CC=gcc
make CC=gcc (2nd)
make CC=gcc -j5
make CC=gcc -j20
make CC="distcc gcc"
make CC="distcc gcc" (2nd)
make CC="distcc gcc" -j5
make CC="distcc gcc" -j20
make CC="ccache distcc gcc"
make CC="ccache distcc gcc" (2nd)
make CC="ccache distcc gcc" -j5
make CC="ccache distcc gcc" -j20
make CC="ccache distcc gcc -gch"
make CC="ccache distcc gcc -gch" (2nd)
make CC="ccache distcc gcc -gch" -j5
make CC="ccache distcc gcc -gch" -j20
make CC="icc"
make CC="icc" (2nd)
make CC="icc" -j5
make CC="icc" -j20
make CC="distcc icc"
make CC="distcc icc" (2nd)
make CC="distcc icc" -j5
make CC="distcc icc" -j20
make CC="ccache distcc icc"
make CC="ccache distcc icc" (2nd)
make CC="ccache distcc icc" -j5
make CC="ccache distcc icc" -j20
make CC="ccache distcc icc -pch"
make CC="ccache distcc icc -pch" (2nd)
make CC="ccache distcc icc -pch" -j5
make CC="ccache distcc icc -pch" -j20

Industrial Strength SCM[a]

Workspaces Where developers build, test and debug.

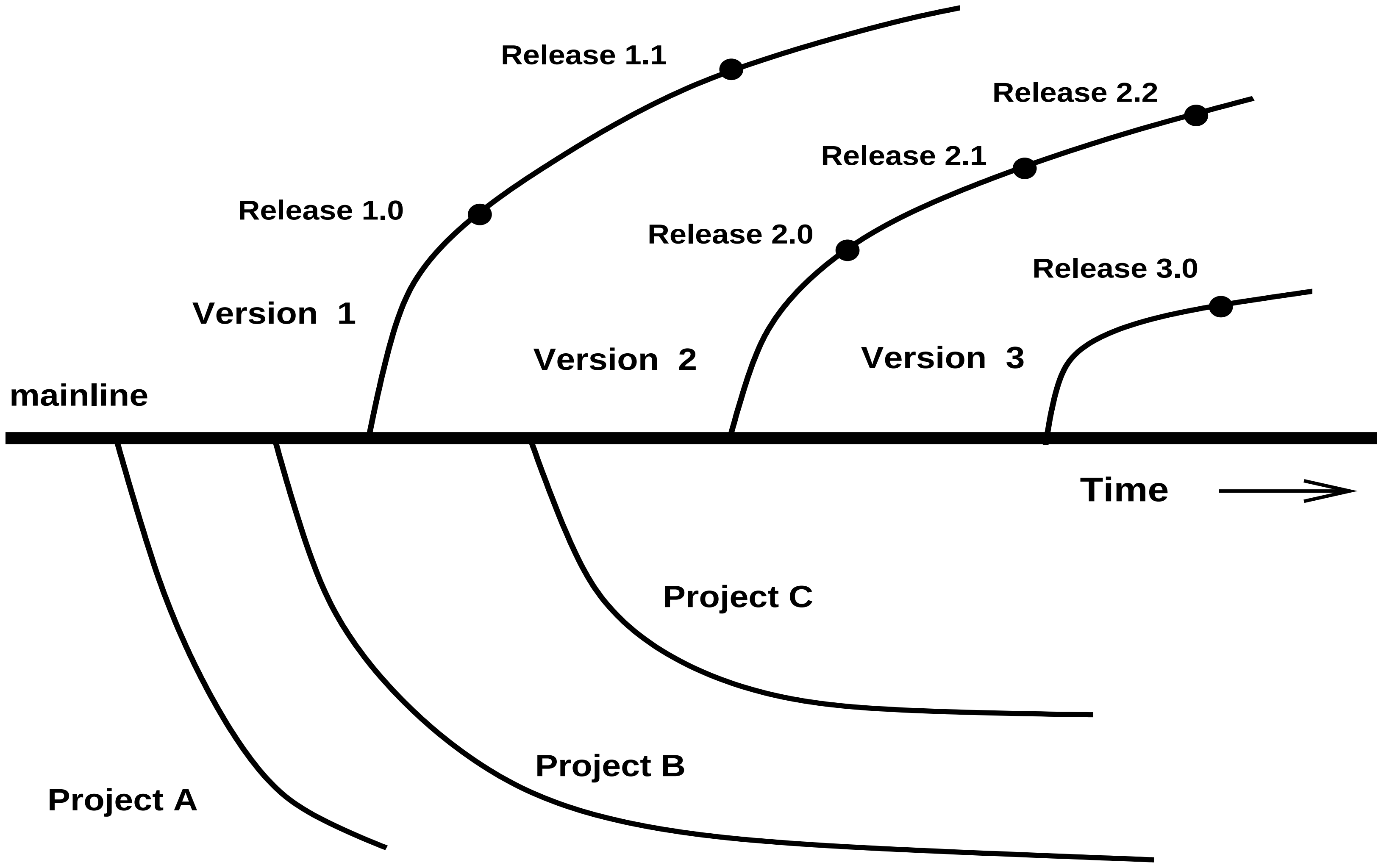
- Don't share workspaces.
- Don't work outside of managed SCM workspaces.
- Don't use jello views (e.g. links outside workspace)
- Stay in sync with the codeline (e.g. cvs update)
- Check in often (e.g. cvs checkin)

These rules are intended to give each developer complete control over changes made within their workspace.

All of the developers work should be visible to and managed by the SCM system.

[a] Inspired by: L. Wingerd & C. Siewald, *High-Level Best Practices in Software Configuration Management*, ECOOP 98, SCM-8, Springer-Verlag LNCS 1439, 1998

Codelines and Branches



Codelines the *canonical* sets of source files.

- Give each codeline a policy.

  The policies specify the fair use (e.g. development, release) and permissible checkins for the codeline.

  Checkin permissions might include timing (e.g not near deadline), type (e.g. only bug fixes), and amount of testing required before checkin (e.g full regression test).

- Give each code line an owner.

  Person for setting and interpreting policy for the codeline.

  The owner can also grant exceptions to the general policy.

- Have a mainline.

  The mainline is the codeline that evolves forever.

  It is the base from which new codelines (e.g. releases) are started.

Examples of Codelines and Policies

Development Codeline

- interim code changes may be checked in
- checked in components must be buildable

Release Codeline

- components must build *and* pass regression tests *before* checkin.
- checkin limited to bug fixes.
- no new features or functionality may be checked in.
- after checking branch is frozen until complete QA cycle has been completed.

Mainline

- all components must compile, link and pass regression tests.
- completed, tested new features may be checked in.

Branches variants of the codeline

- Branch only when necessary. Branches cause extra work.
- Don’t copy when you mean to branch.

  Copies can get outside the SCM system, cause extra work.
- Branch on incompatible policies.
- Branch late. i.e. stay with the existing codeline as long as possible.
- Branch instead of freezing a codeline.

  Don’t make everyone wait for a frozen codeline, (e.g frozen for a full regression test).

Change propagation getting changes from one codeline to another.

- Make original changes in the branch that has evolved *least* since branching.
- Propagate changes early and often
- Get the right (e.g. most knowledgeable) person to do the merge.

Builds turning source code files into products

- product is source *plus* tools.
- Checkin all source before build.
- Segregate built objects from source, i.e. build to a separate directory tree.
- Use common build tools for all parts of a project.
- Build often
- Keep build logs and build output.

Process rules for all of the above

- Track change *packages* (i.e. files changed together)
- Track change package propagations.
- Distinguish change requests from change packages.
- Give everything an owner. A thing without an owner is dead.
- Use living documents. Process documentation should be available and updatable.

Version Control Issues

- The development of a software system involves the creation and evolution of thousands of source objects.

  These objects may be source code, documentation, designs, diagrams, etc.

- Every change to a source object logically creates a new version of the object

  There may be hundreds of versions of an object over the lifetime of a system

- Any given instance of a software system is composed of specific versions of every object in the system.

  For example: `main.c 4.35.7 14 Mar 1994 11:35:03`

- Requirements for a version control system
  - Need systematic rule for enumerating versions of an object
  - Need to be able to retrieve *any* version
  - Want to encourage documentation of versions
  - Need to store versions in a space efficient fashion
  - Provide version differencing as a debugging tool

Version Control

- A *version control system* is used to manage the multiple versions of base items that exist over time during the evolution of a software system
- Basic version control facilities
  - Reliable archival storage of all versions.
    Store version differences to save disk space
  - The ability to create new versions and variants of versions
    Typical naming scheme: *major.minor.variant*
  - Automatic logging and management of version identification information
  - Retrieval of any version or variant
  - Mutual exclusion and locking to allow access to multiple programmers
  - Differential comparison of versions for change tracking and debugging
- SCCS, RCS and CVS are examples of version control systems

How to Use Version Control

- Establish a *base version* (major#) at significant points in the development of the software. Major versions typically correspond to releases of the software.
- **Checkpoint frequently** by checking in new minor versions after any significant change (e.g. an hour's or day's work)
- Use logging facilities to document changes between versions for future reference
- Link source code versions to corresponding binary objects occurring in the System Model
- Use differential version compare to identify the exact differences between different versions of the software
- **Use version control religiously. It will help you out of problems**

Change Control

- In a large software system *change must be managed*
- Typically a *change committee* comprised of senior developers and managers has control over all changes to the software
- A formal process is followed for all proposed changes to the system
- Change committee
  - Determines desirability of a change
  - Acts as a first level filter to detect conflicting or overlapping changes
  - Estimates the cost and impact of changes

    May do cost/benefit analysis to decide whether the change should be done
  - Schedules changes relative to Releases of the software
  - Tracks the change process to make sure changes are actually applied