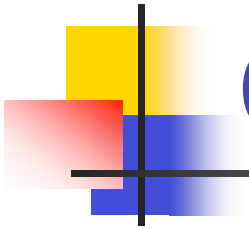
Evolution in Open Source Software: A Case Study

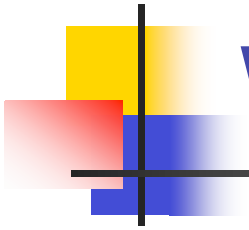
Michael W. Godfrey
Qiang Tu

Software Architecture Group
University of Waterloo



Overview

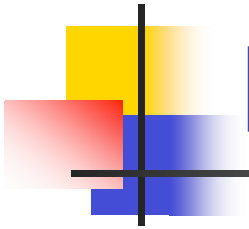
- What is software evolution?
  - Why should we care?
- Previous research
- A case study: The Linux OS kernel
- Observations, hypotheses, and future research



What is software evolution?

"Evolution is what happens
while you're busy
making other plans."

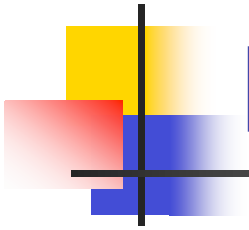
- Usually, we consider evolution to begin once the first version has been delivered:
  - *Maintenance* is the planned set of tasks to effect changes.
  - *Evolution* is what actually happens to the software.



Previous research

- Lehman's laws
- Parnas on software geriatrics
- Eick *et al.* on code decay (10 MLOC telecom)
- Gall *et al.* (10 MLOC telecom)
- Munro, Burd *et al.* (2 MLOC `gcc`)

Lehman's Laws of Software Evolution

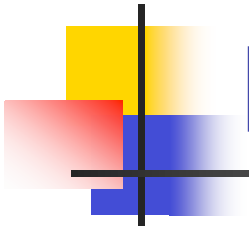


1. *Continuing change* — An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
2. *Increasing complexity* — As a program is evolved, its complexity increases unless work is done to maintain or reduce it.
3. *Self regulation* — The program evolution process is self-regulating with close to normal distribution of measures of product and process attributes.

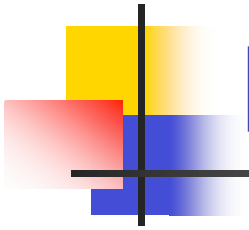
Lehman's Laws of Software Evolution

4. *Invariant work rate* — The average effective global activity rate on an evolving system is invariant over the product lifetime.
5. *Conservation of familiarity* — During the active life of an evolving program, the content of successive releases is statistically invariant.
6. *Continuing growth* — Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

Lehman's Laws of Software Evolution



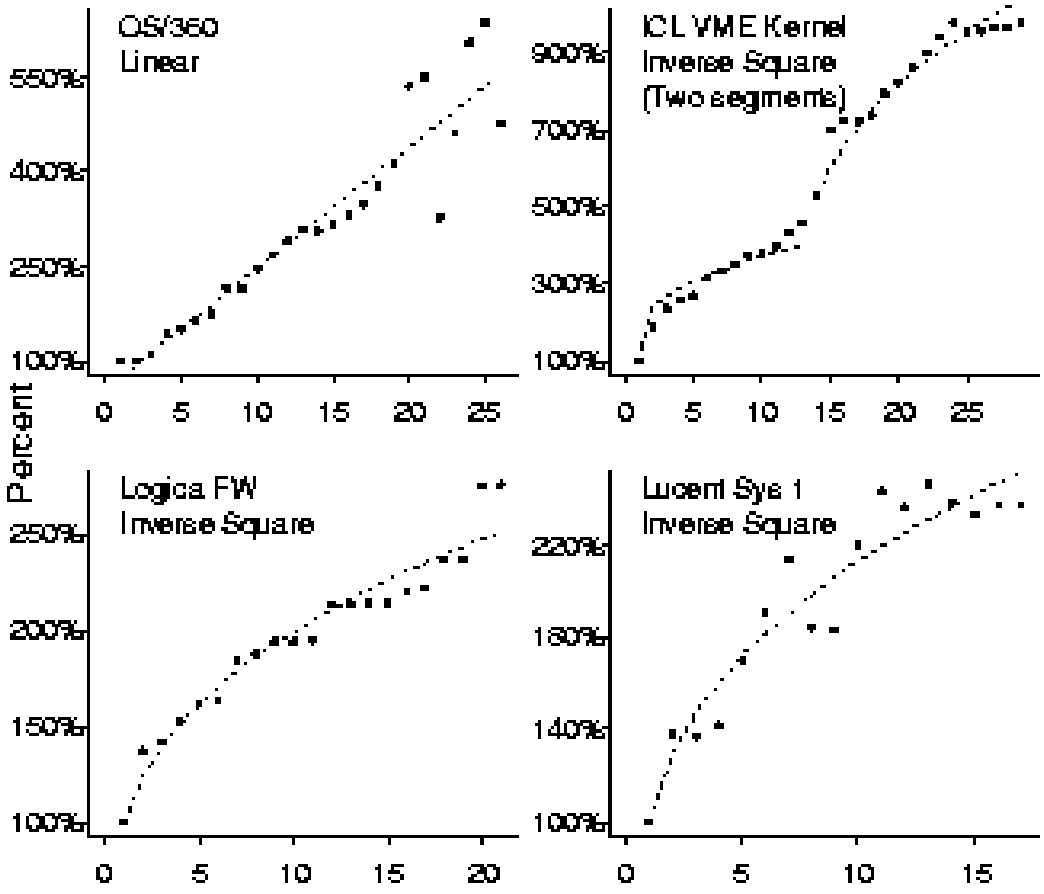
7. *Declining quality* — E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operation environment.
8. *Feedback system* — E-type programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.



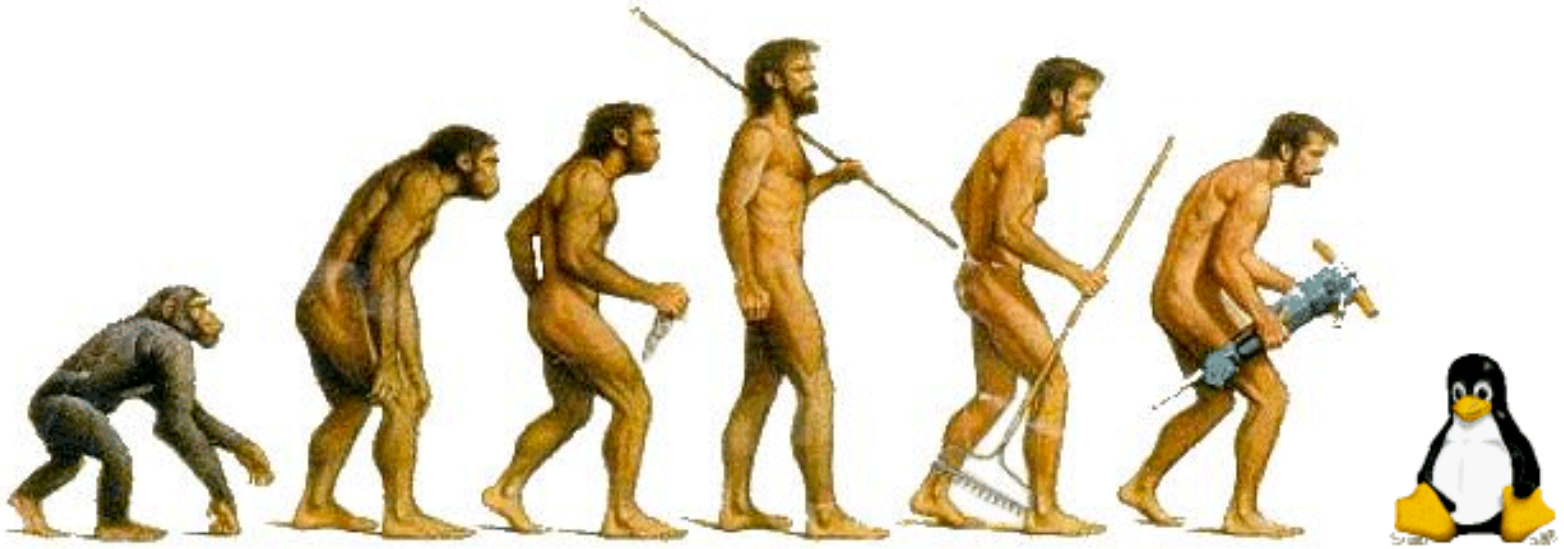
Lehman's Laws in a nutshell

- Observations:
  - (Most) useful software must evolve or die.
  - As a software system gets bigger, its resulting complexity tends to limit its ability to grow.
  - Development progress/effort is (more or less) constant; growth is *at best* constant.
- Advice:
  - Need to manage complexity.
  - Do periodic redesigns.
  - Treat software and its development process as a feedback system (and not as a passive theorem).

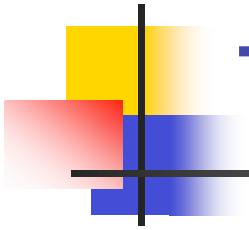
Lehman's examples



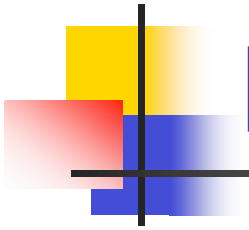
A case study in evolution: The Linux OS kernel



A case study in evolution: The Linux OS kernel

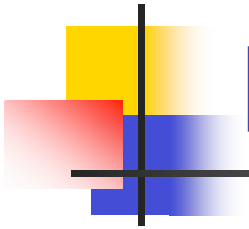


- It's Linux!
  - Large system, very stable, many releases over several years, many developers
  - Growing mainstream adoption
- Open source development model
  - Interesting phenomenon in itself
  - Easy to track, can publish results, many experts
  - Not much previous study



Linux background

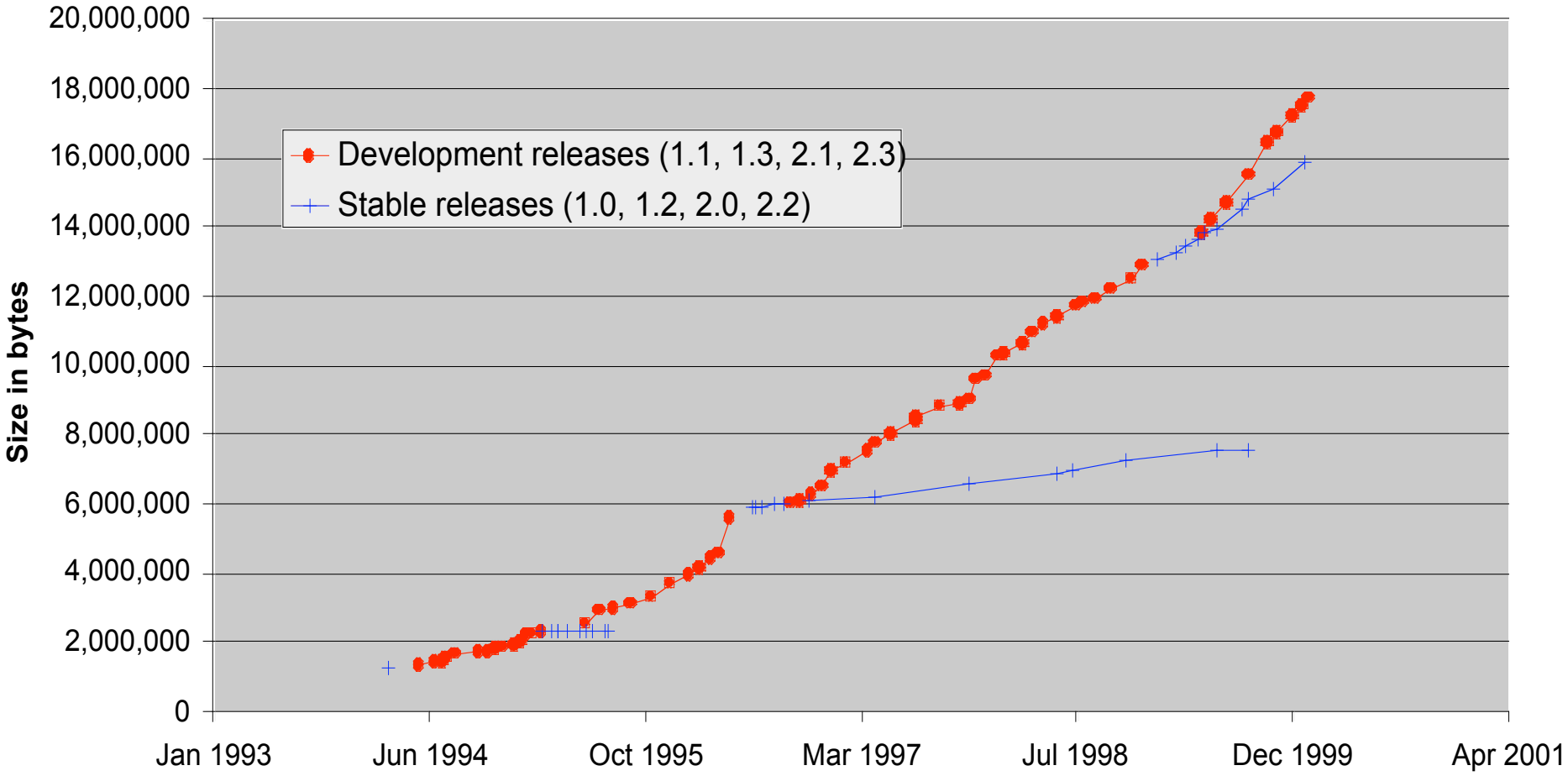
- Linux kernel v1.0 released March 1994
  - 487 source files, 165 KLOC, i386 only
- Linux kernel v2.3.39 released January 2000
  - 4854 source files, 2.2 MLOC, 10 hardware architectures supported, over 300 developers credited
- Maintained along two parallel paths:
  - *development* and *stable*



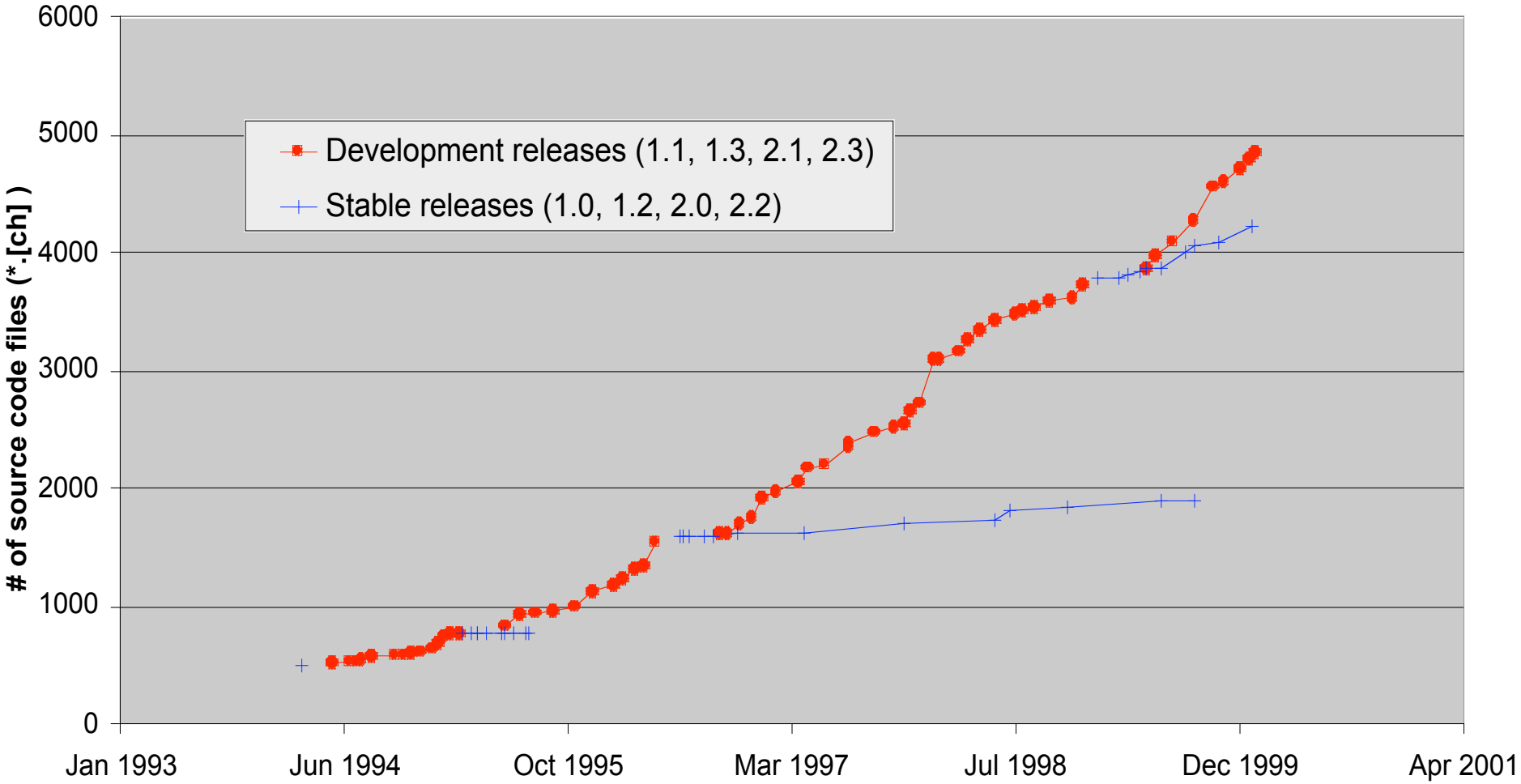
Methodology

- Examined 96 versions of Linux kernel
  - 34 of the 67 stable releases
  - 62 of the 369 development releases
- All measures considered only `.c/.h` files contained in the tarball
  - Counted LOC using "`wc -l`" and an `awk` script that ignored comments and blank lines
  - Counted # of fcns/vars/macros using `ctags`
  - Architectural model (SSs hierarchy) based on default directory structure
- We plotted growth against calendar time
  - Lehman suggests plotting growth against release number

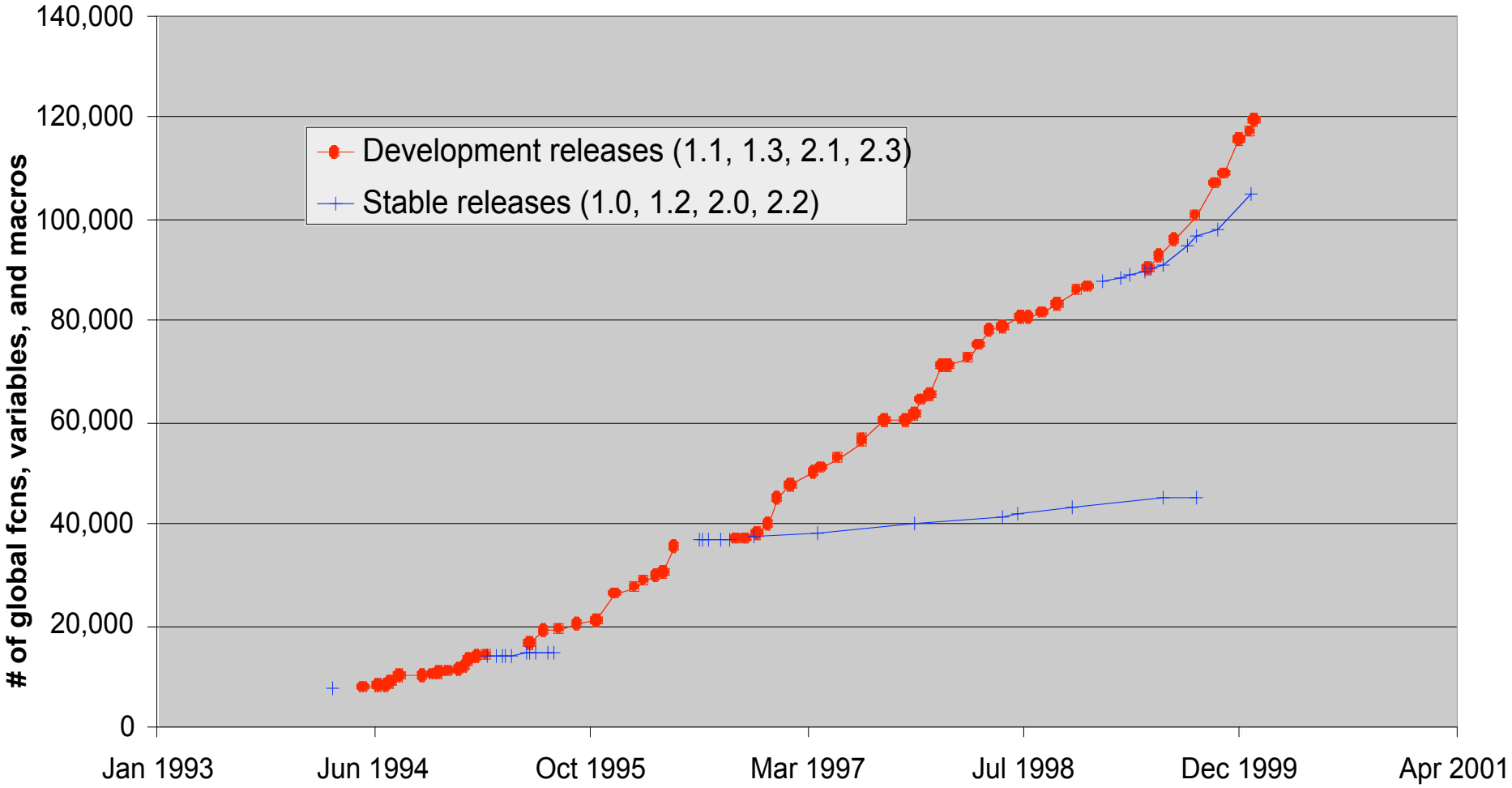
Growth of compressed tar file



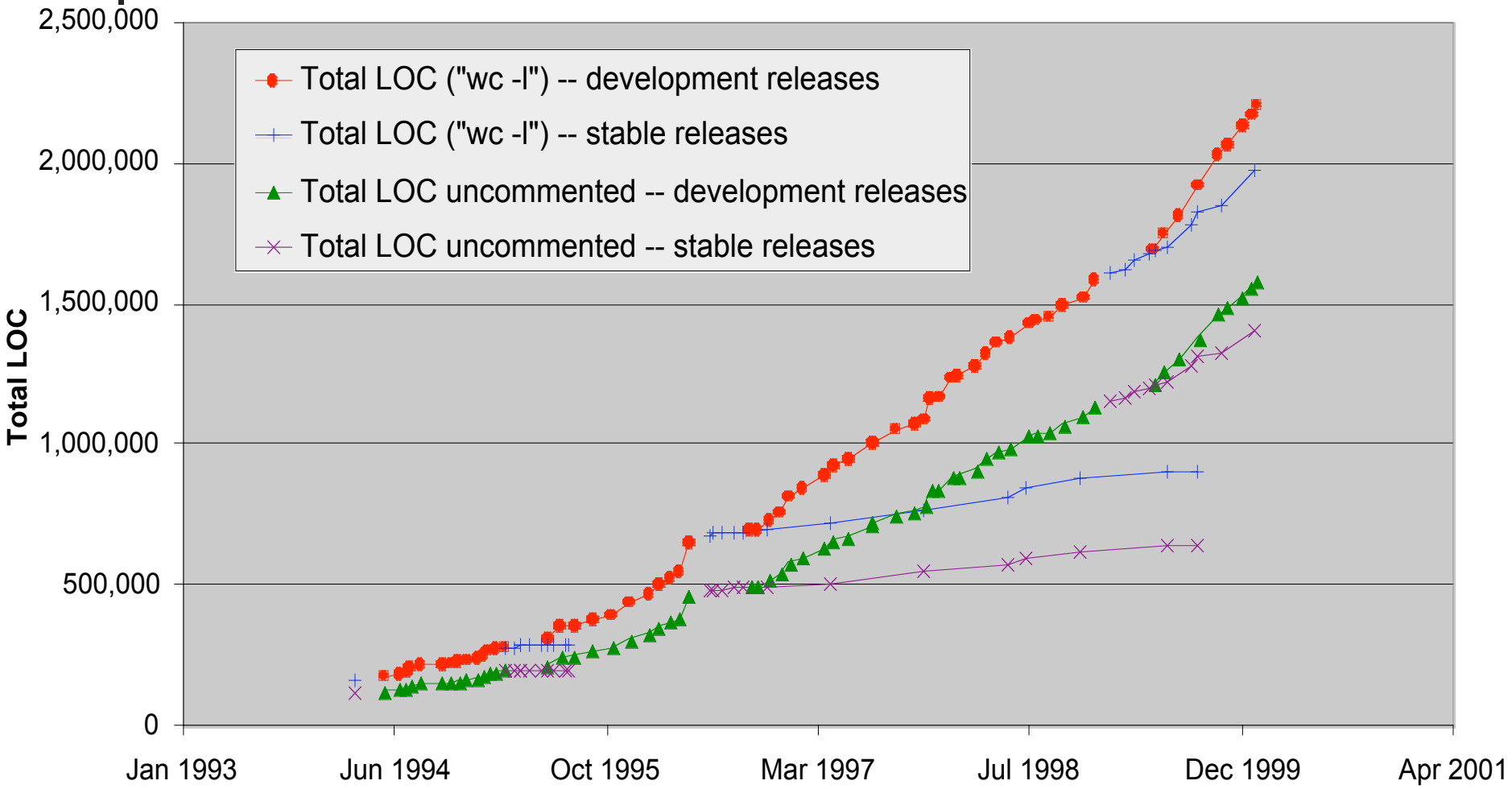
Growth of # of source files



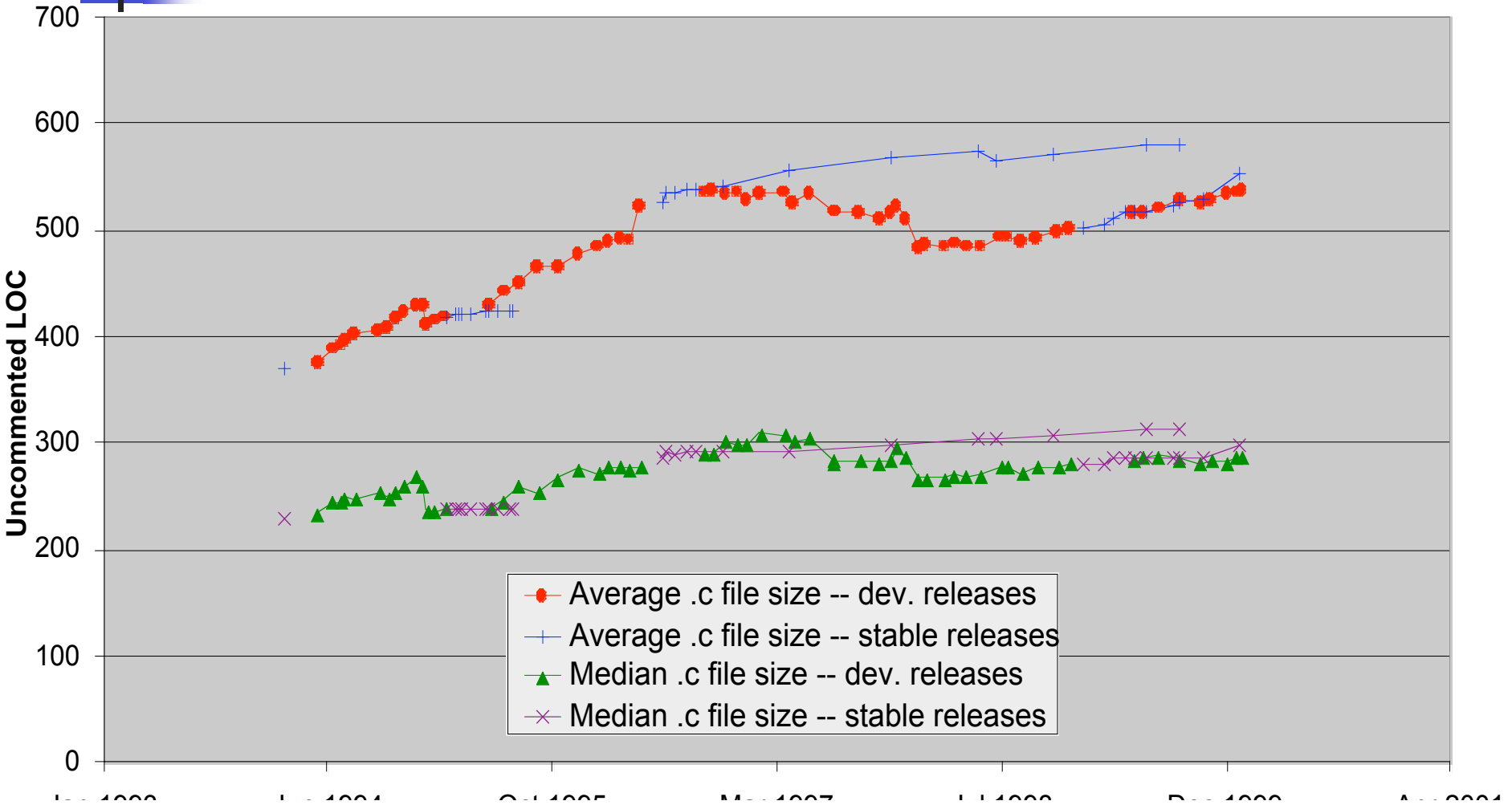
Growth of # of global fcns, variables, and macros



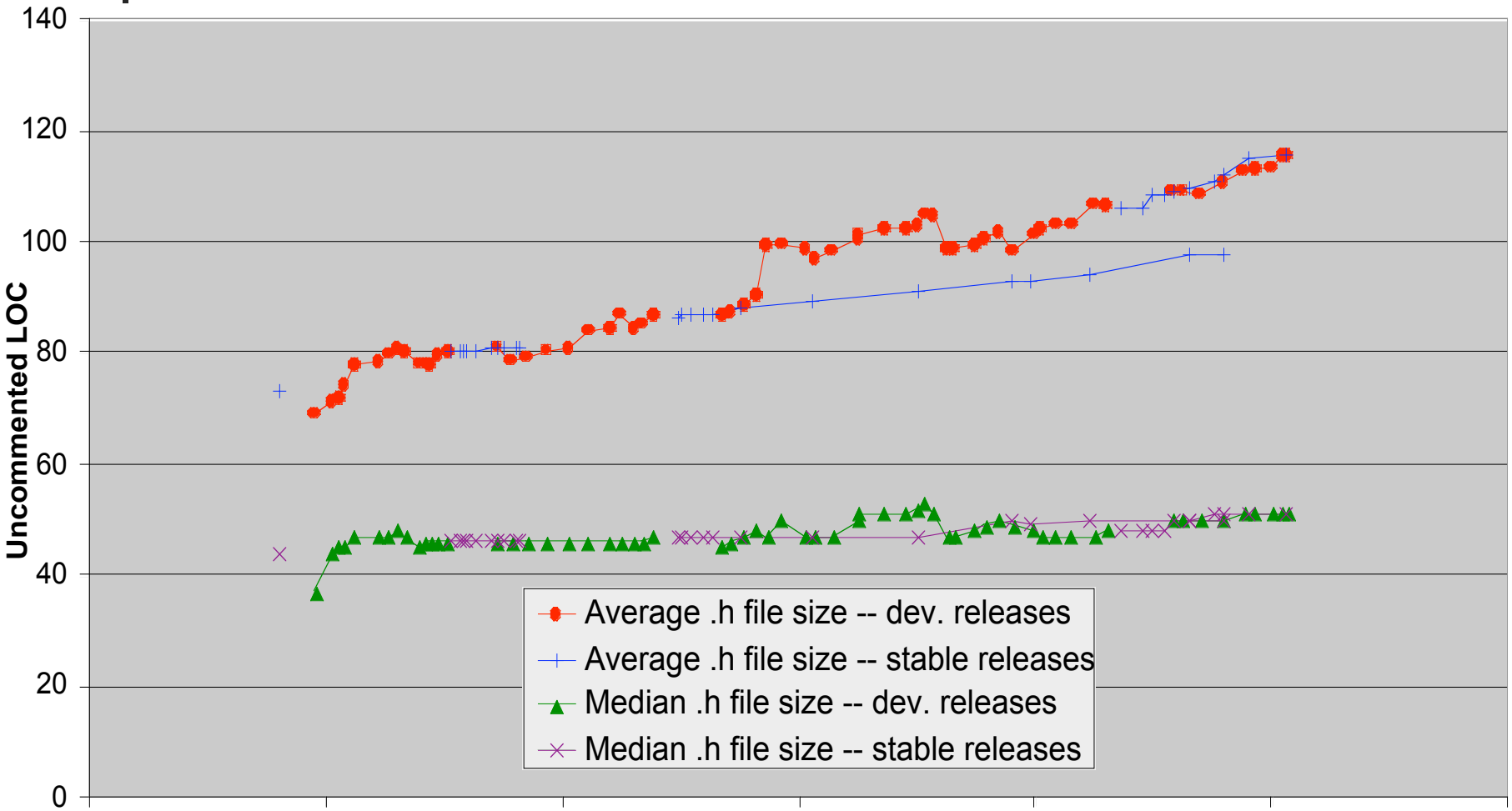
Growth of Lines of Code (LOC)



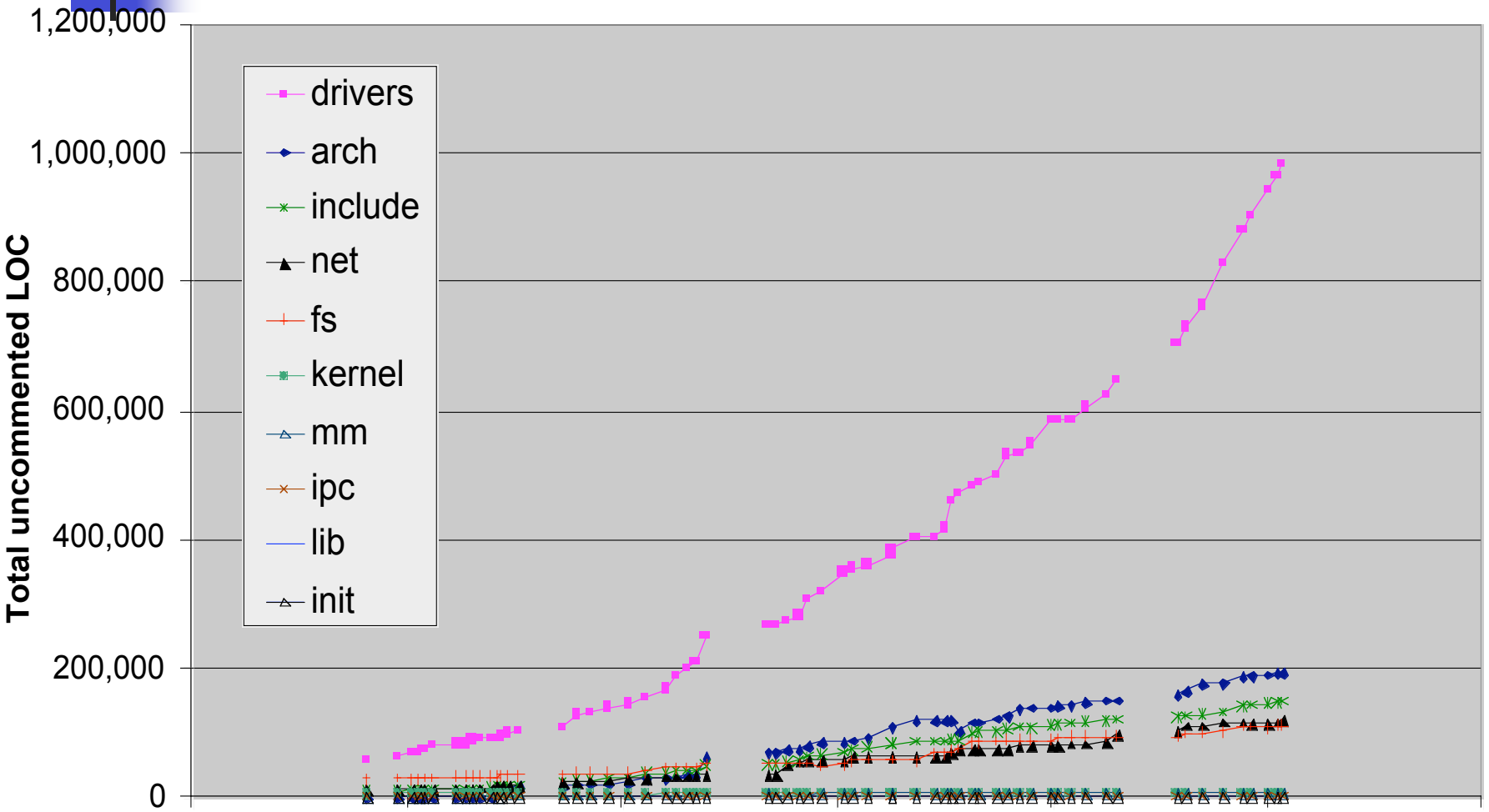
Average/median .c file size



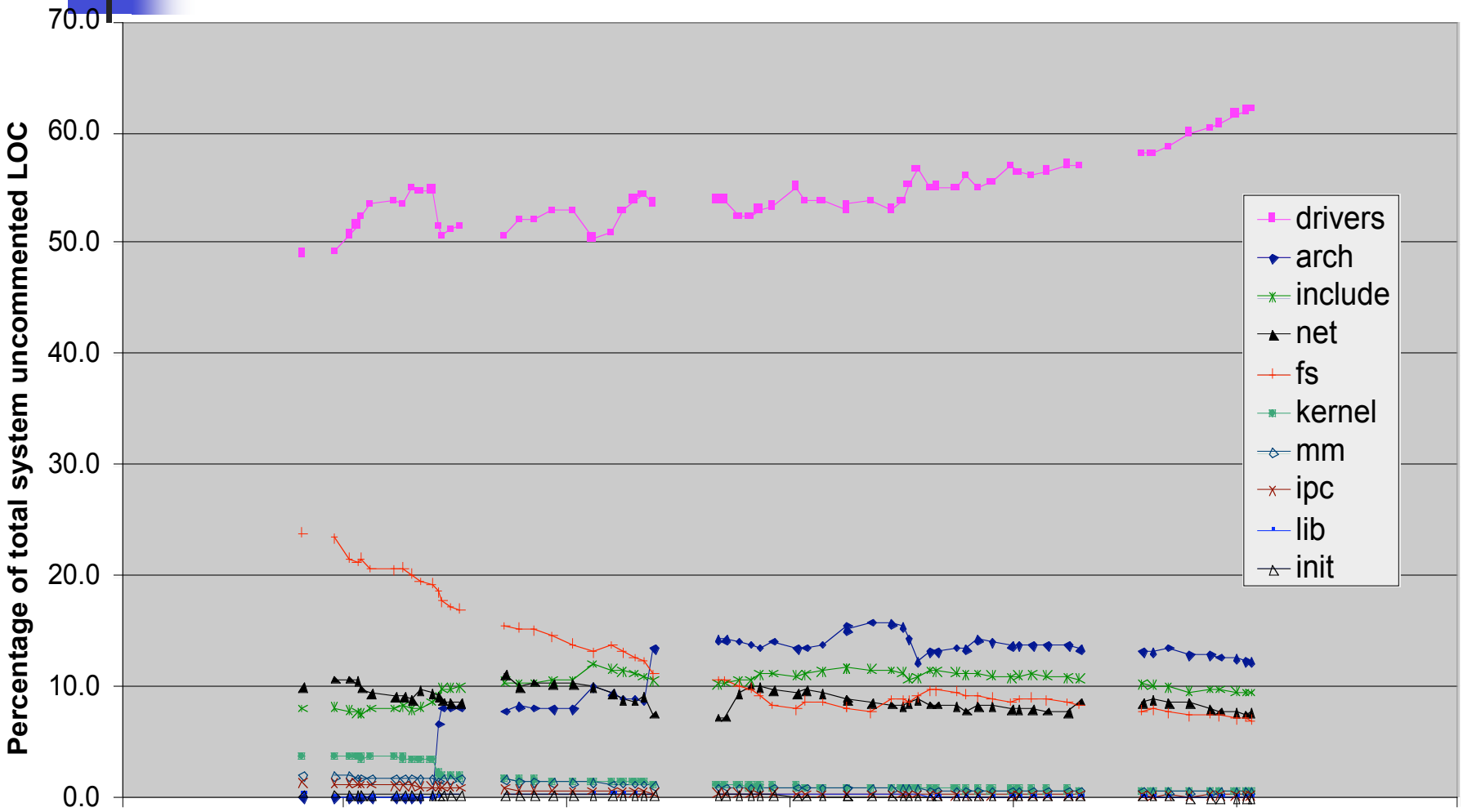
Average/median `.h` file size



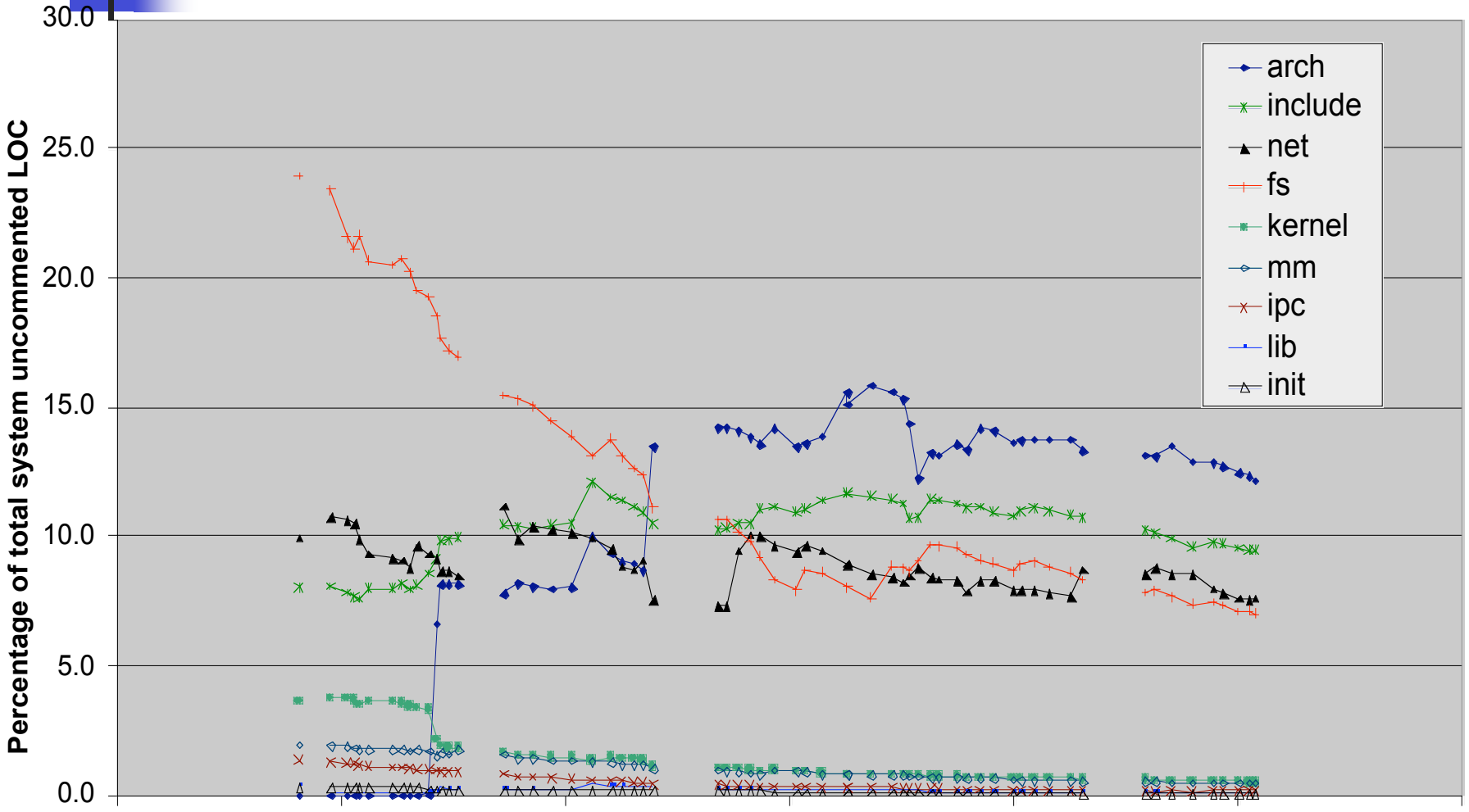
Growth of major SSs (dev. releases)



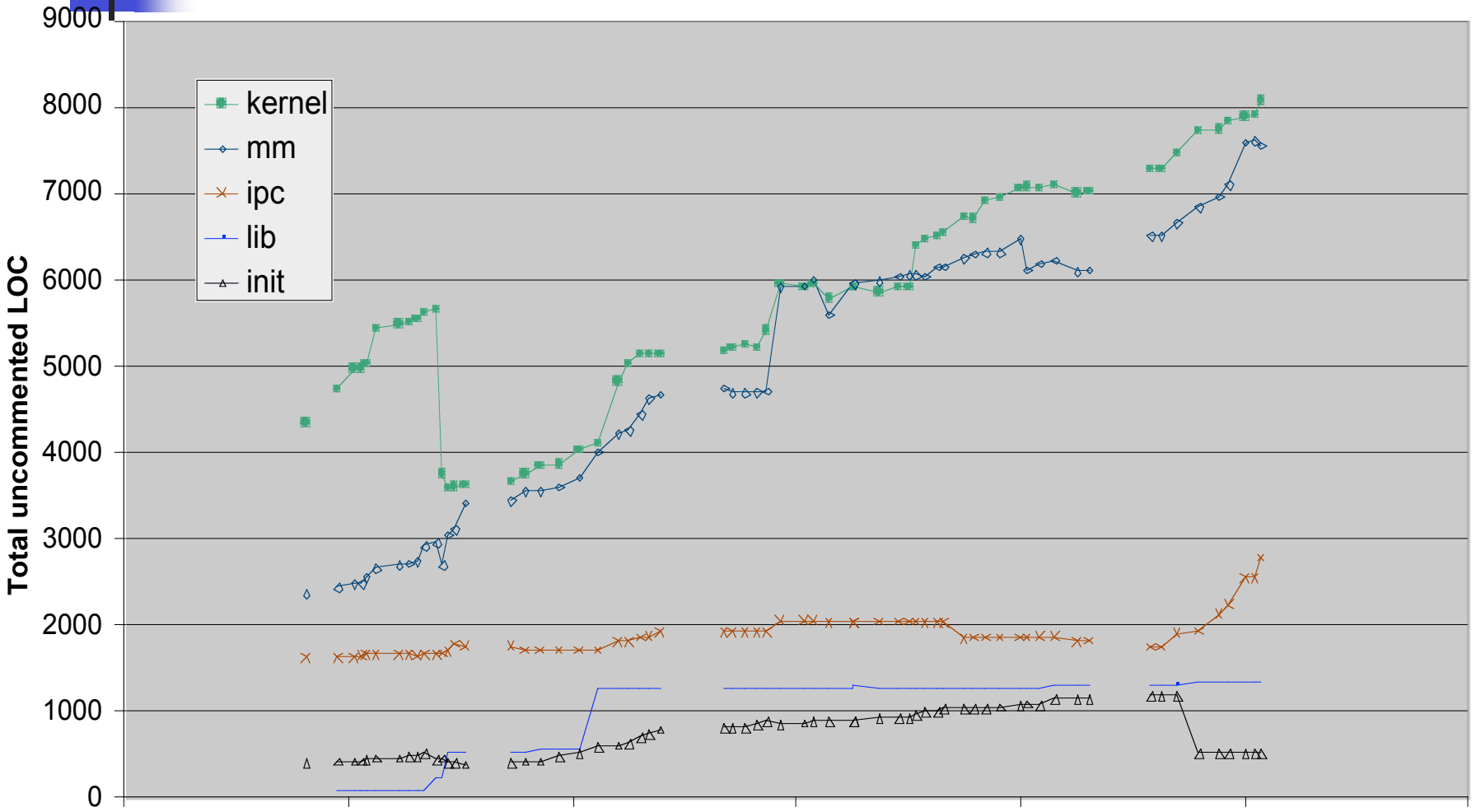
SS LOC as percentage of total system



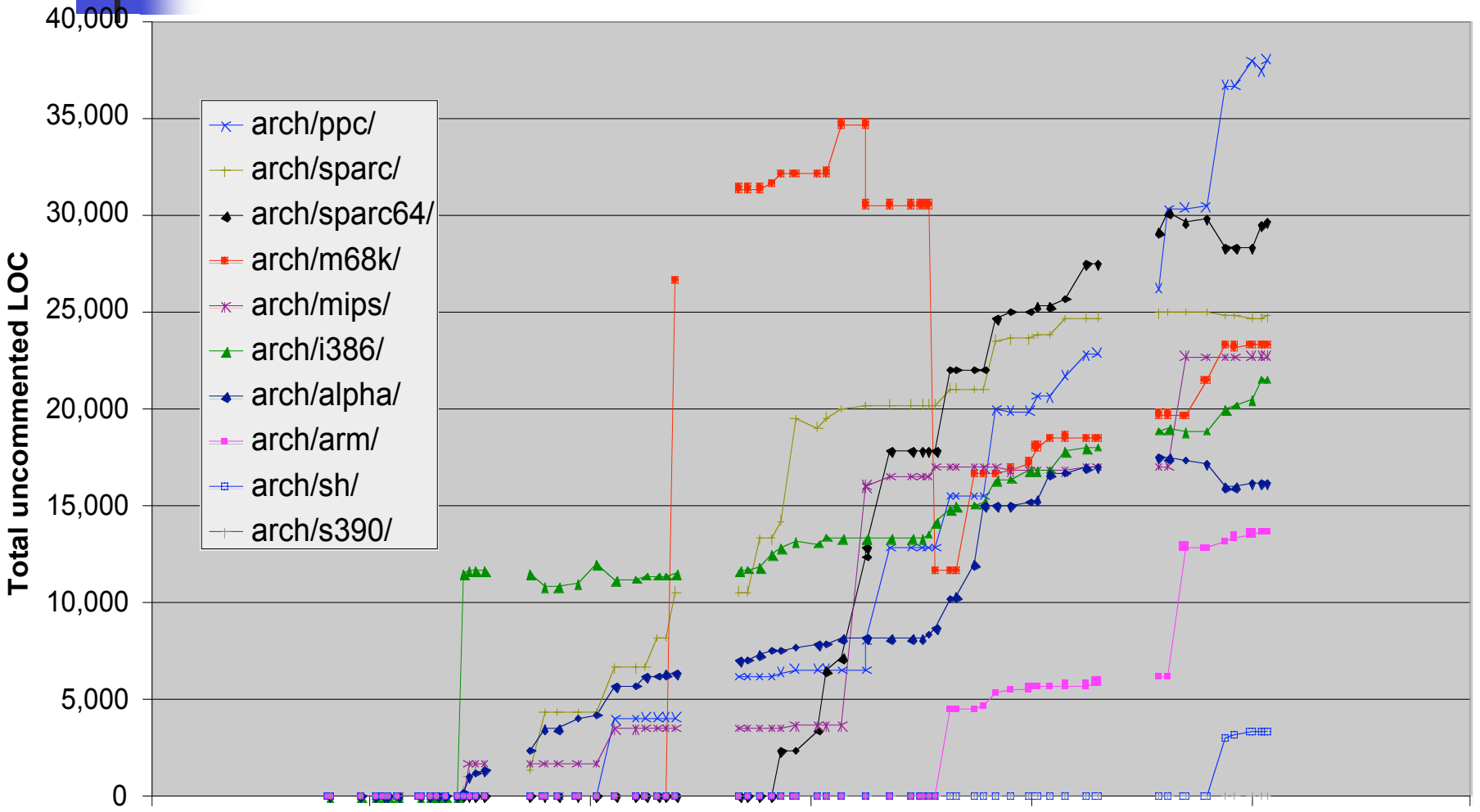
SS LOC as percentage of total system (ignoring drivers)



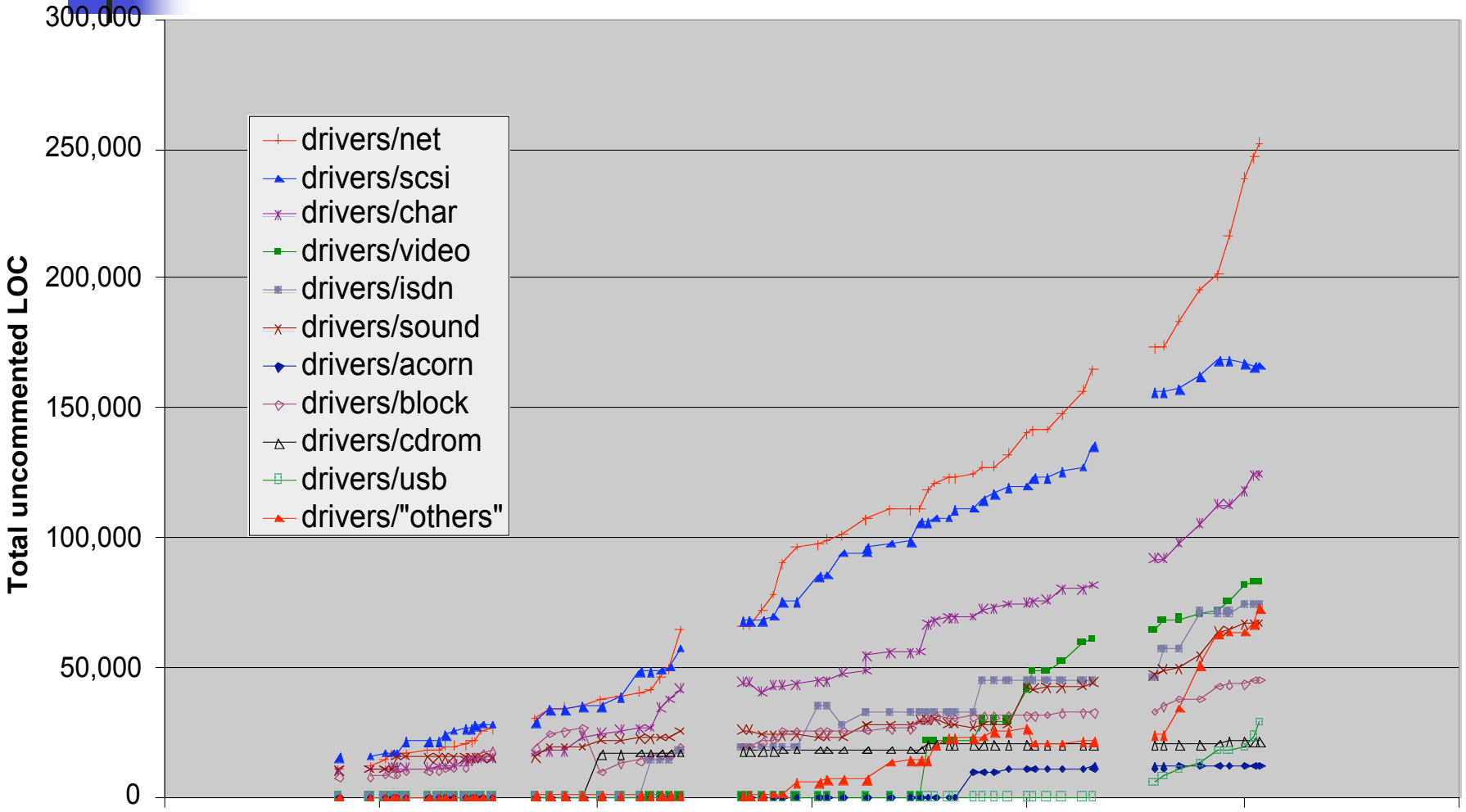
Growth of small core SSs

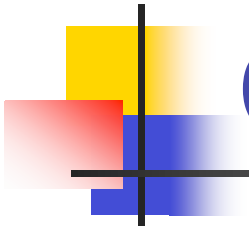


Growth of arch SSs



Growth of drivers SSs





Observations and hypotheses

- Growth along devel. path is super-linear

  y = .21*x^2 + 252*x + 90,055 r2=.997

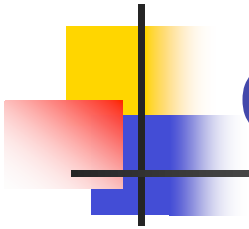
  y = size in LOC

  x = days since v1.0

  r2 is "coefficient of determination" using least squares

  [Lehman/Turski's model: y' = y + E/y^2 Ξ (3Ex)^(1/3)]

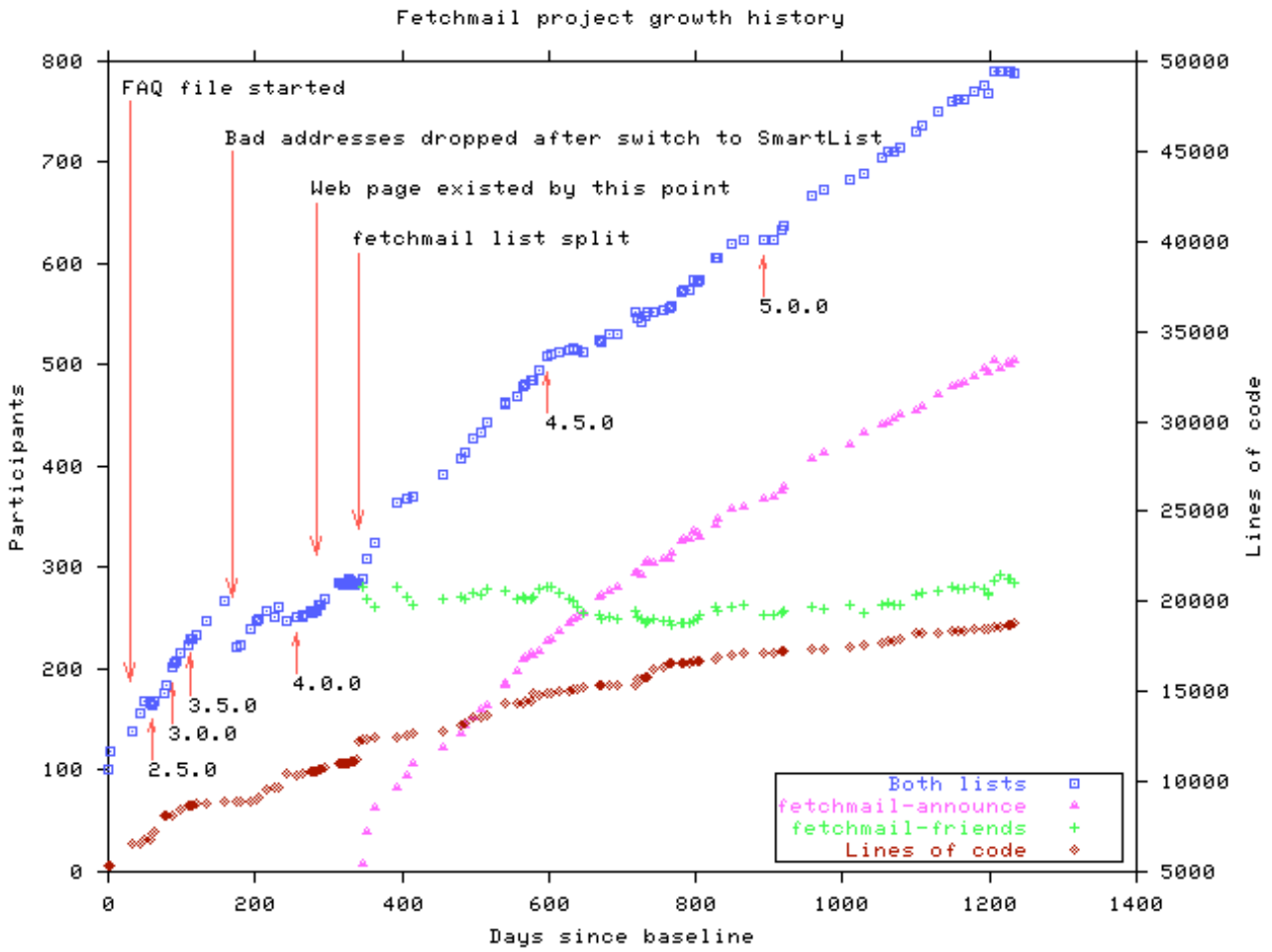
  - Linux's strong growth is continuing.
  - This is stronger growth at MLOC level than observed by others (Lehman, Gall), even for other OSs.



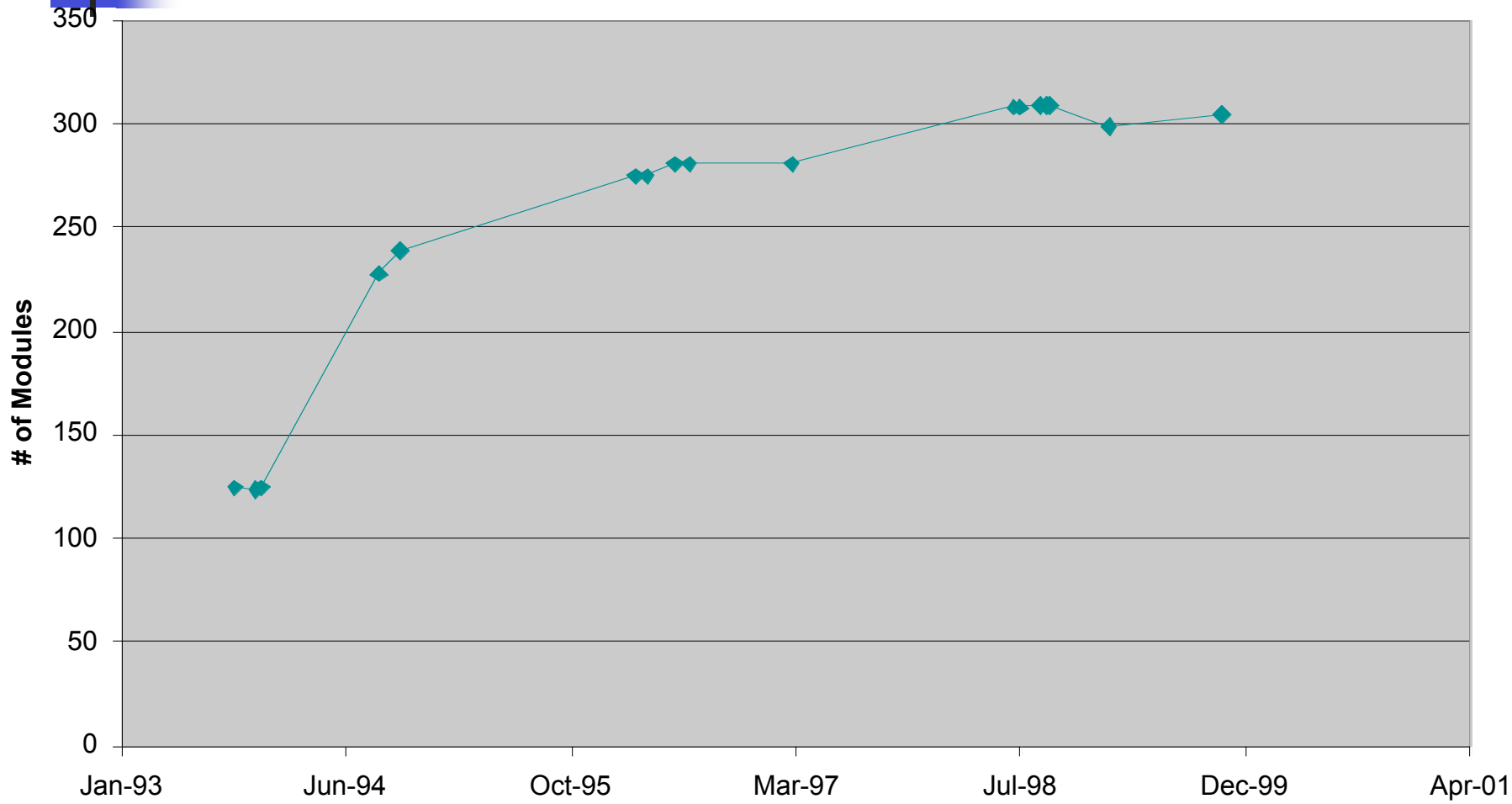
Why has Linux been able to continue its geometric growth?

- Core code quality is carefully maintained
- Architecture/problem domain
  - It's largely drivers
  - Much of the code is "parallel"
  - It's not as big as you might think
    - Vanilla configuration used only 15% of files
- Development model (OSD) and its sociology
  - Popularity and visibility has encouraged outsiders (both hackers and industry) to contribute

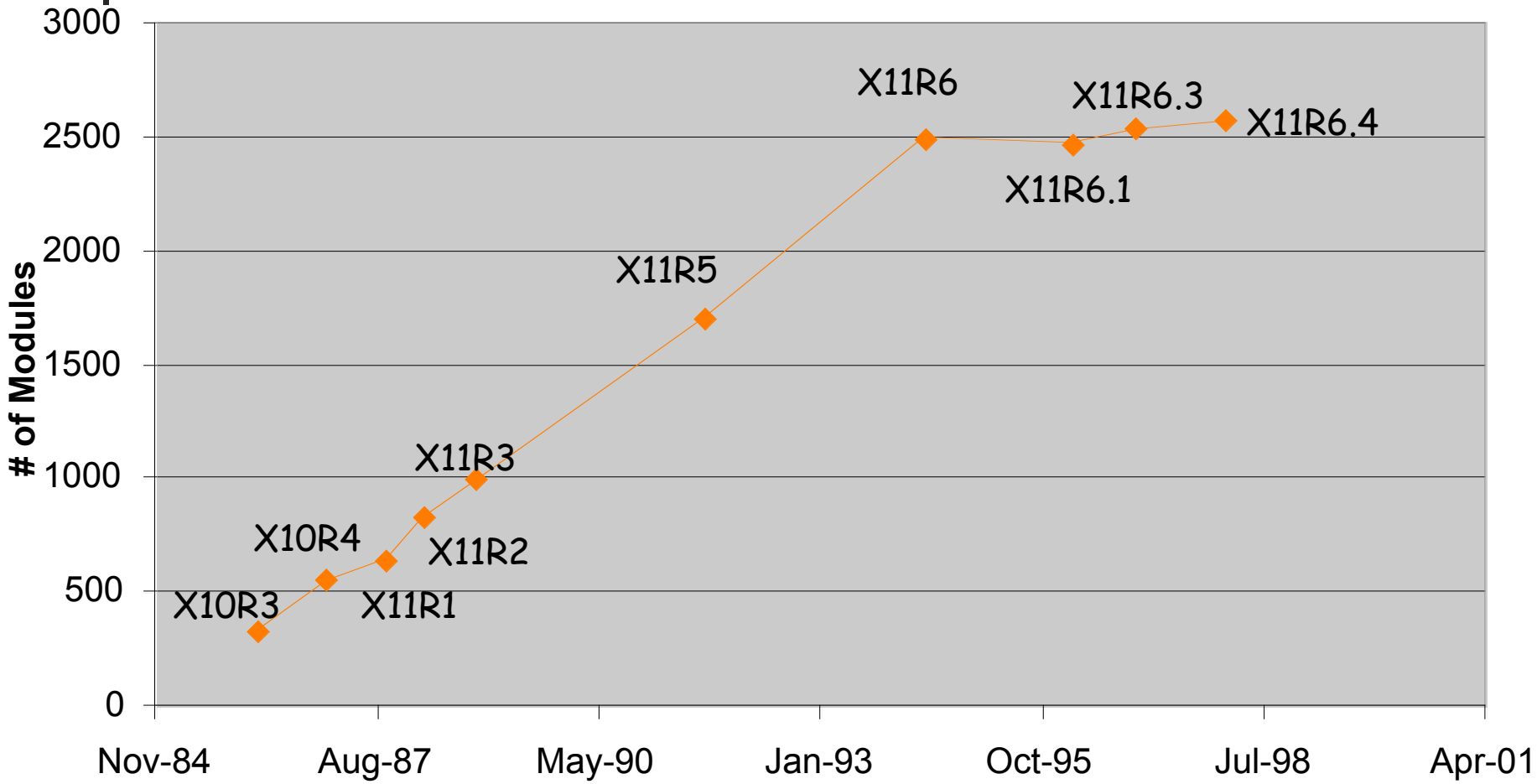
Growth of `fetchmail` [Raymond]



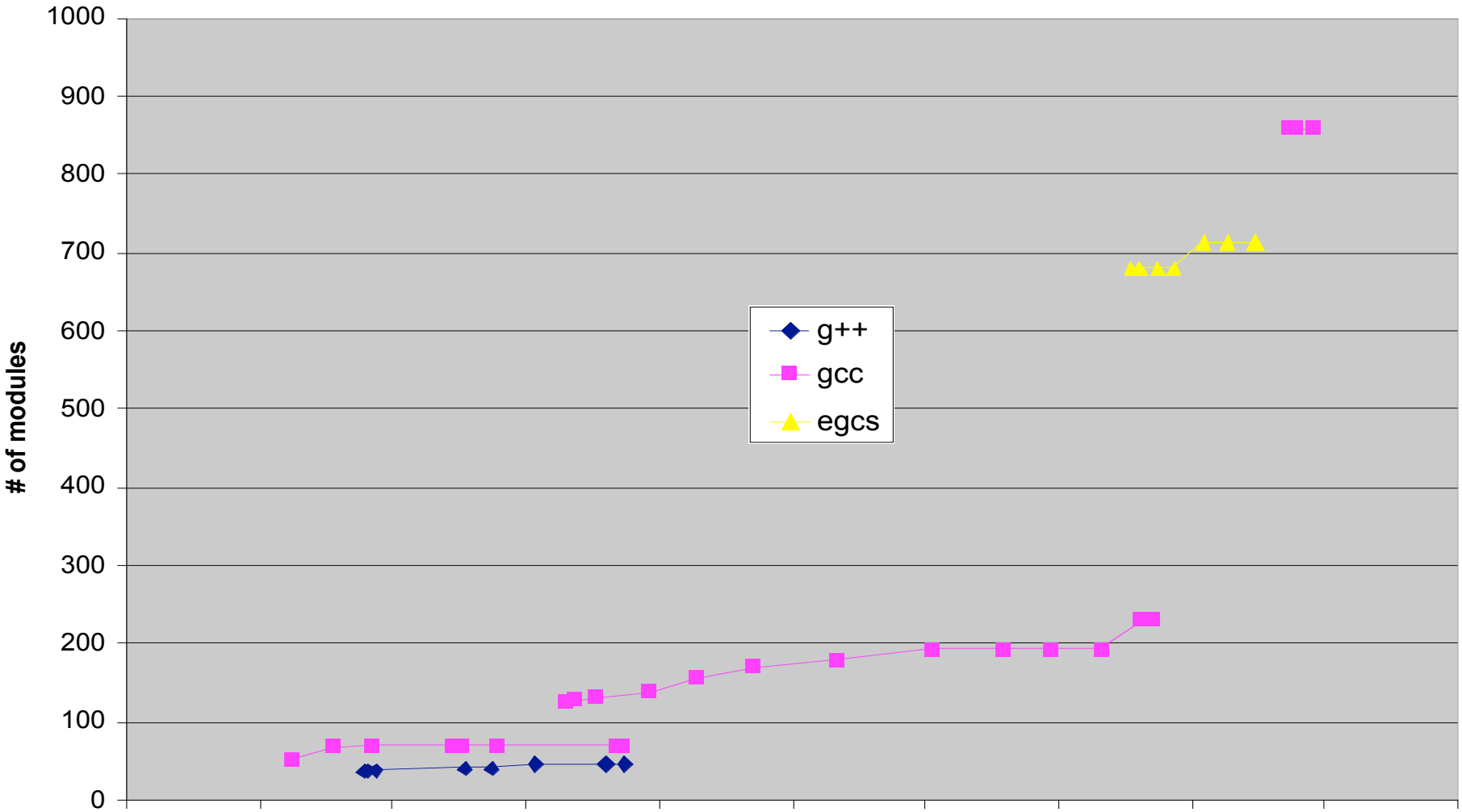
Growth of `pine` (email client)



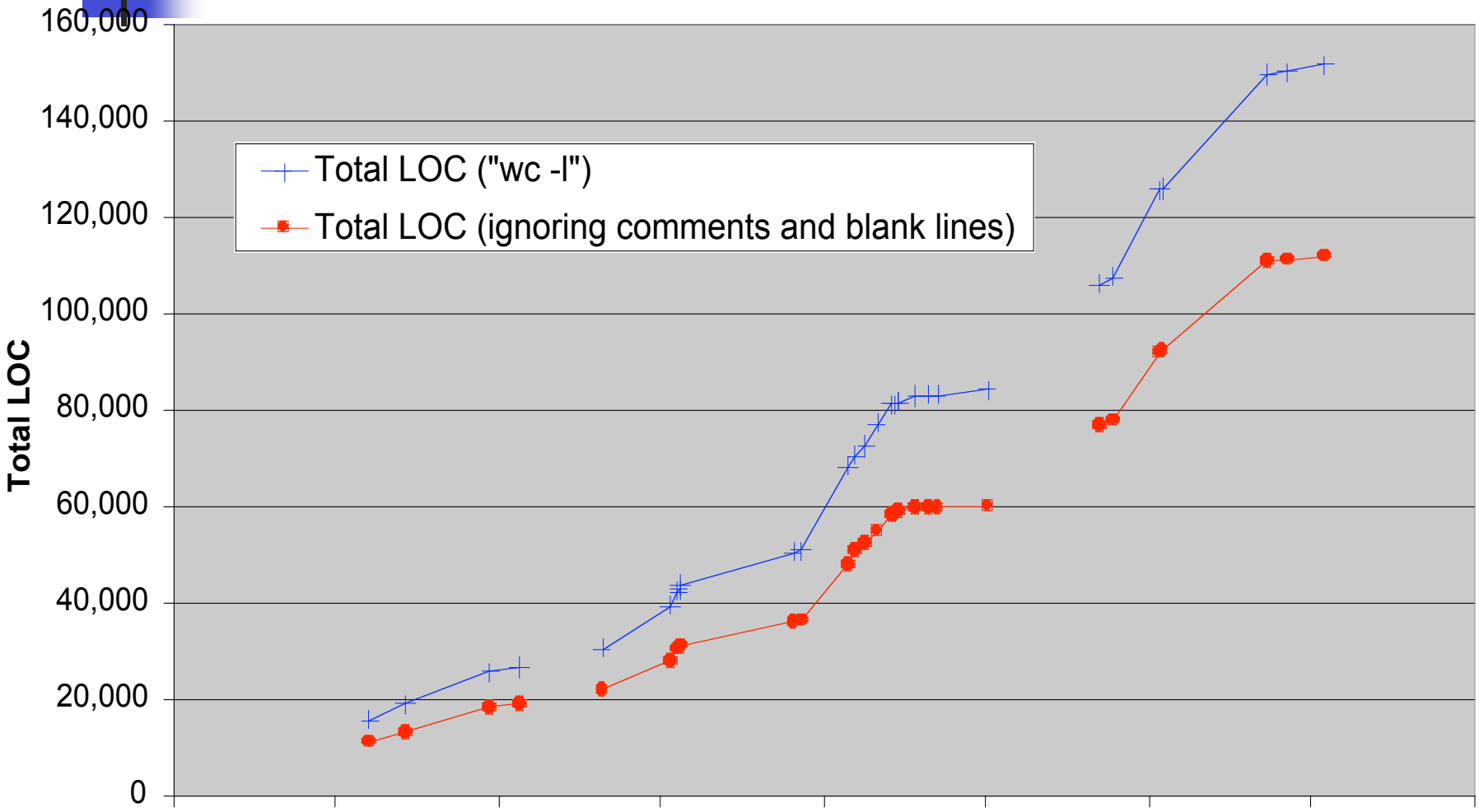
Growth of X Windows



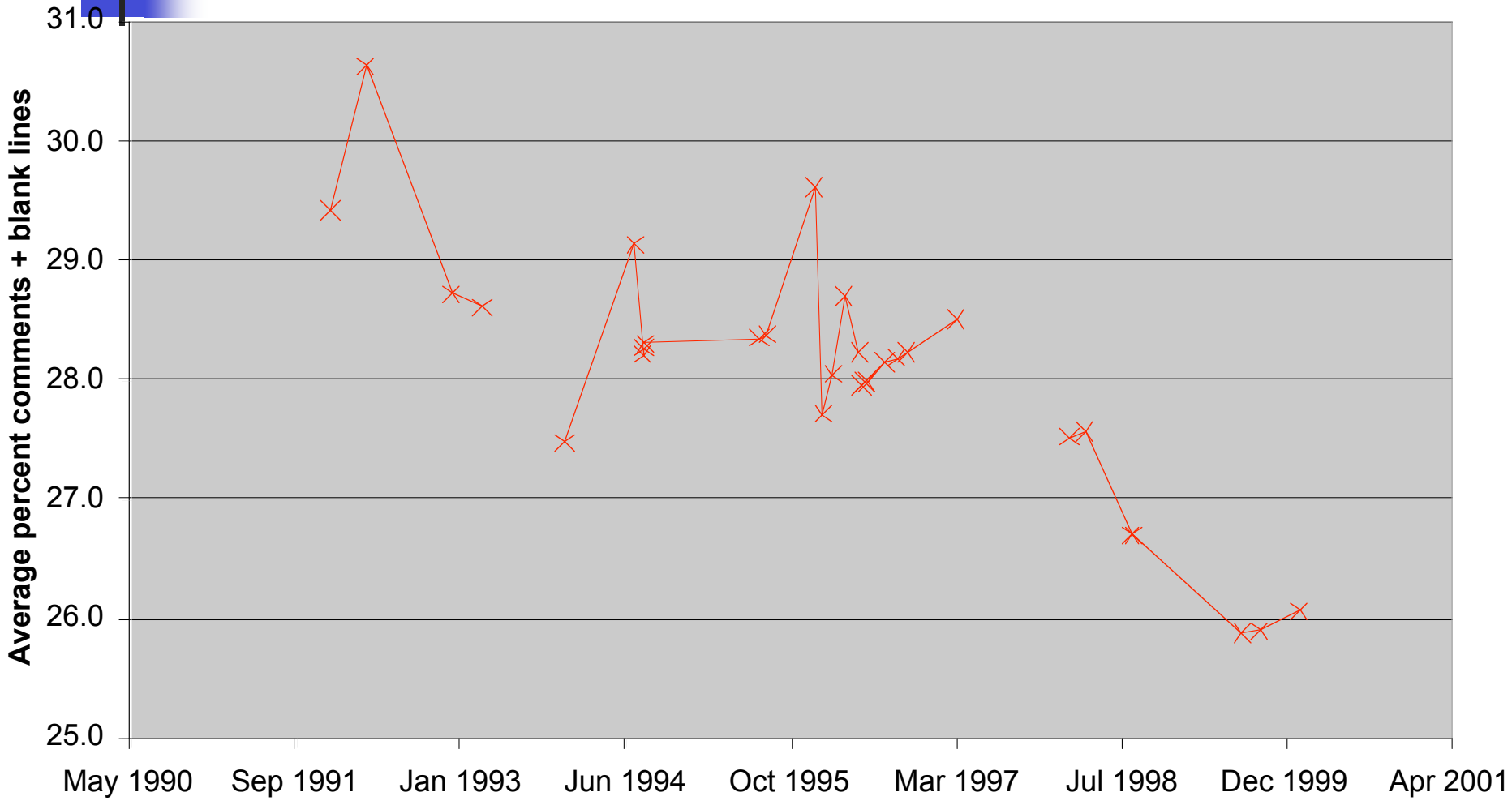
Growth of gcc/g++/egcs



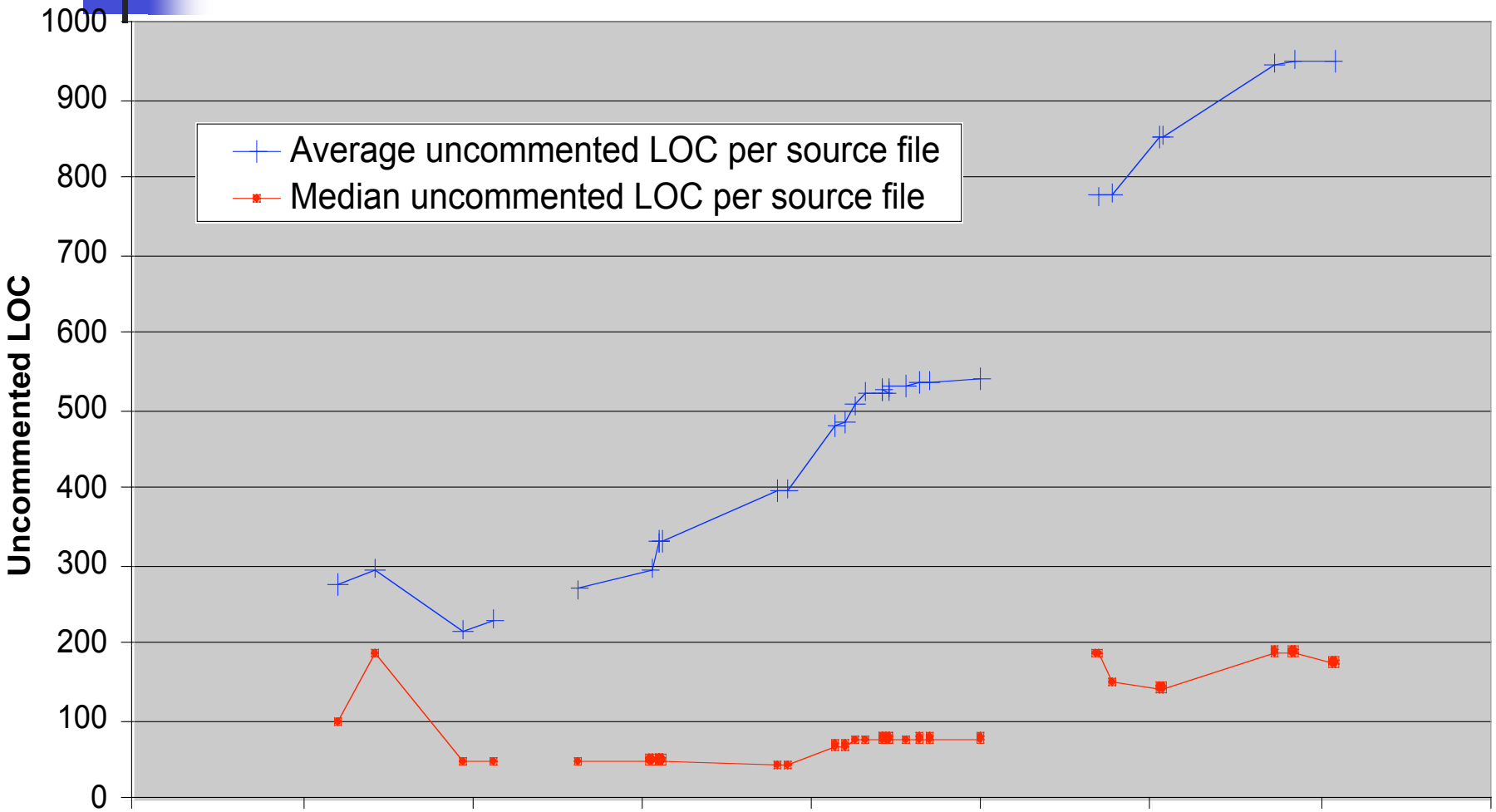
Growth of `vim` (text editor)



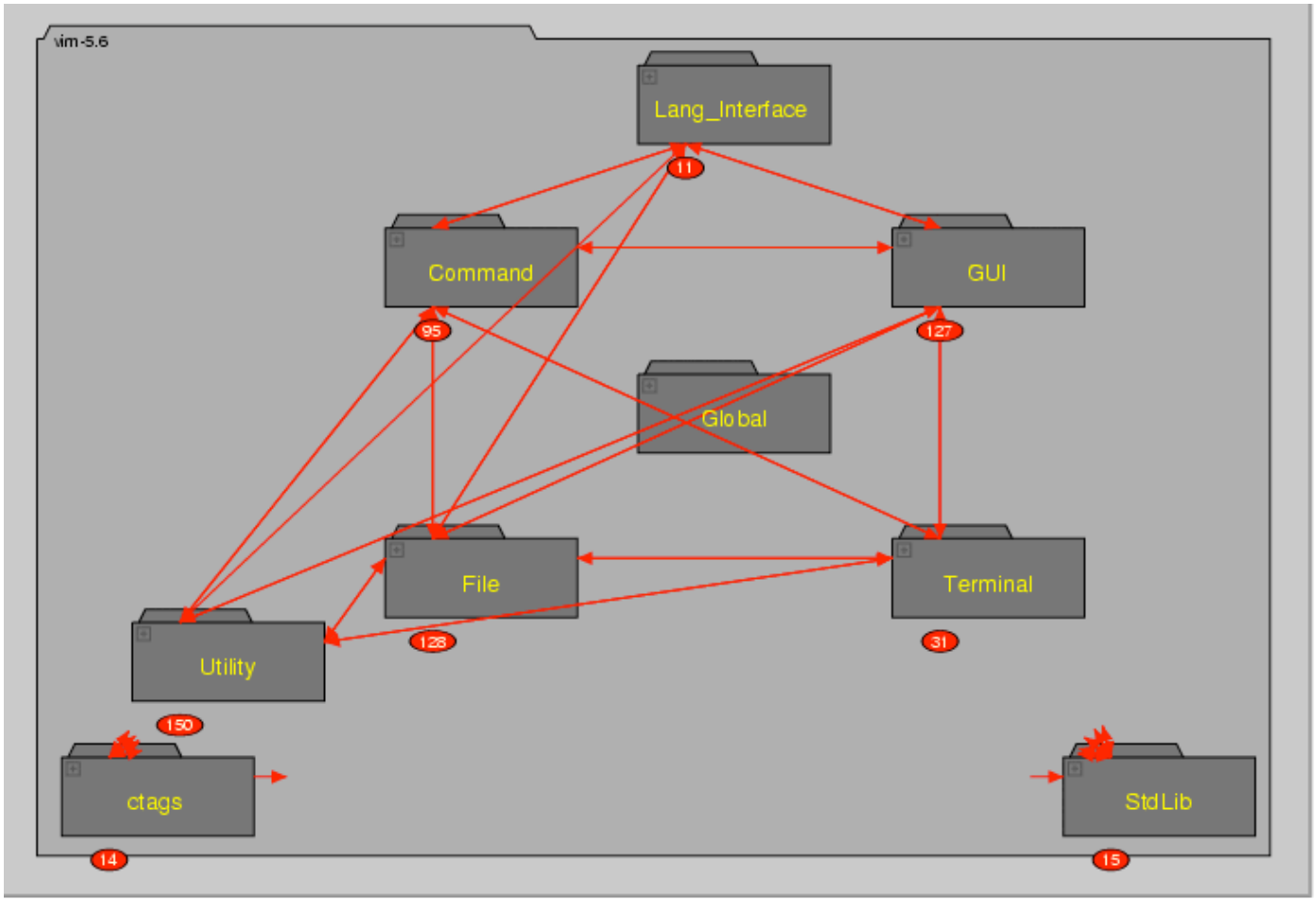
`vim` avg % comments and blank lines per file

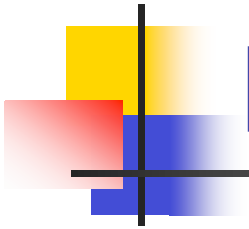


vim avg/median file size



vim's architecture





Hypotheses

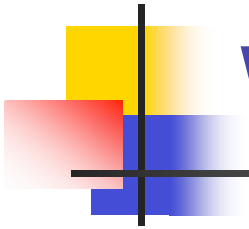
Factors affecting evolution include

- Size and age of system
- Use of traditional sw. eng. principles during development

PLUS

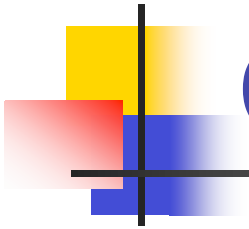
- Problem domain
  - Problem complexity, multi-platform, multi-features
- Software architecture
- Process model
- Sociology, market forces, and acts-of-God

Software evolution research: What next?



So far, we have examined only growth.

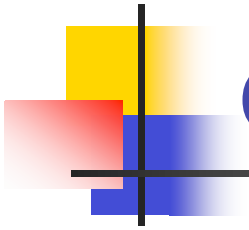
- More case studies needed
  - Qualitative and quantitative
  - Industrial and open source systems
  - Different problem domains, architectures
- Supporting tools to aid analysing, visualizing, and querying program evolution
  - More than just `RCS` and `perl`
  - Support for architecture repair
- Codified knowledge: Why and how does software change?
  - Build catalogue of *change patterns* and *evolutionary narratives*



Codified knowledge

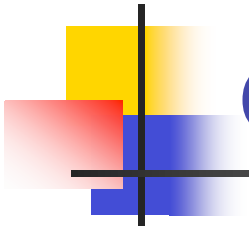
- Mature engineering disciplines codify knowledge and experience.
- Arguably, this is lacking in software engineering.
  - Software architecture styles [Shaw]
  - Design patterns [GoF]
- Codified knowledge of how and why programs evolve:
  - *Evolutionary narratives* [Godfrey]
    - Long term, coarse granularity
  - *Change patterns*
    - Short term, fine granularity

Change patterns and evolutionary narratives



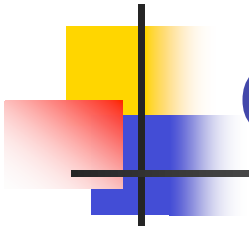
- Phenomena observed in Linux evolution
  - Bandwagon effect
  - Contributed third party code
  - “Mostly parallel” enables sustained growth
  - Clone and hack
  - Careful control of core code; more flexibility on contributed drivers, experimental features

Change patterns and evolutionary narratives



- Cathedral style [Raymond]
  - careful control and management
  - debugging done before committing code
  - evolution is slow, planned, rarely undone
- Bazaar style (OSD)
  - lots of low-level changes, frequent fixes
  - lots of "building around" rather than wholesale changing, occasional redesigns
  - creeping feature-itis, "complete" dependency graph

Change patterns and evolutionary narratives



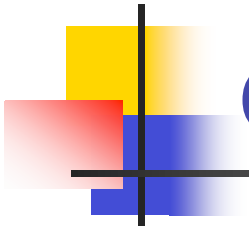
- Band-aid evolution (just add a layer)
  - quick & dirty way to add new functionality, esp. if system is not well understood

    e.g., Y2K fixing, adding portability, new features

- "Vestigial features"
  - design artifact persists after rationale dies

    e.g., whale fin bone structure resembles hand

Change patterns and evolutionary narratives



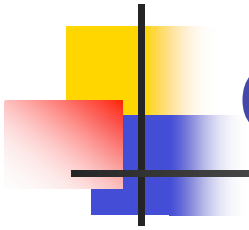
- "Adaptive radiation" [Lehman]
  - when conditions permit, encourage wild variation for a while.
  - later, evaluate and let "best" ideas live on.

    e.g., Linux kernel evolution

- "Convergent evolution"
  - compare similar systems to reference arch. (or to each other)

    e.g., everyone grows an XML generator in response to market pressure

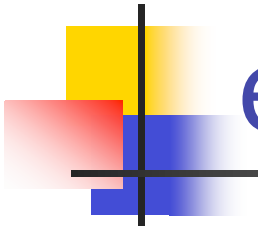
Change patterns and evolutionary narratives



- Radical redesigns (localized and global)
  - aka "refactoring"
  - little new functionality added, but structure changes significantly, legacy cruft dissipates
  - likely "goodness" (design metrics) improves
- Migration patterns
  - look out for known translation idioms, especially if migration is not one big bang

    e.g., procedural-to-OO idioms

Change patterns and evolutionary narratives



- OO evolutionary patterns
  - one recognizable design pattern transformed into another (or a variation of the original)
    - requires good OO extraction tools (dynamic binding, polymorphism, reflection, *etc.*)

- Reuse patterns
  - components are (re)used in different systems

    e.g., build COTS interface, throw out homebrew DB

(Fin)