Guiding Philosophy

- CPU cycles are Cheap,
  People cycles are not

2001-03-20 13:00-0500 P T Withington – Software Journeyman 1

Applies to Users, and Programmers

In runtime systems we are often slaves to cycle counts. We believe that if we take care of the pennies, the pounds will take care of themselves; so we spend hours laboring over algorithm choice and implementation to minimize cycle counts – at the expense of our own cycles that could be more productively used elsewhere. Often these cycle-driven choices result in an algorithm that is less general or less useful to our customer (the application designer), or an implementation that prohibits or overlooks corner cases that will not be exposed until our customer's customer (the end user) has been using the product for some time.

Jon Bently (of AT&T) in his "Programming Pearls" described a bug he discovered in a text layout program (nroff) that had lain dormant for years until a particular document took the algorithm into a branch of code that had apparently never been used. The programmer had clearly spent some time optimizing this particular case, with only one problem – he got it wrong. Bently speculates as to the cost of that optimization – consider the cycles the programmer put into designing and implementing it, add in the interest of many years, and then discover the payoff was negative!

As a consumer of runtime libraries, you should choose the library for what it does for you (and your customers) in terms of functionality, not the one that produces the best benchmark.

This same philosophy can be applied to compilers, languages, and applications.

Tune for people cycles, not CPU cycles.

Future Directions

- Safety, Reliability

2001-03-20 13:00-0500 P T Withington – Software Journeyman 2

It probably doesn't matter if my IM session hangs. Or if Netscape or Explorer "execute and illegal operation". If my email is held on a backed-up IMAP server, I probably don't care if my mail client "unexpectedly quits".

I get a little more anxious when Quicken or TurboTax crash though. I know they have a reputation for ensuring that my transactions are recorded to disk, but I get nervous when I see that the database is reindexed on the next launch. I get especially nervous when my historical stock data has disappeared, or a memorized transaction shows up in the wrong category.

Safety and reliability are boring. They don't sell.

But lacking them costs people cycles. I have to re-enter my Quicken data. I have to re-write my Word document. I have to re-draw my Illustrator diagram.

How many of these faults lie in the runtime that the application designer had no control over? How many of these faults are because the application designer did not have a reliable automatic memory management runtime and had to design his own manual system instead? How many of these faults are because the implementation language makes the application programmer declare types left and right but then permits non-type-safe values (I.e., null) to be assigned and passed?

Secrets

- Whatever is worth doing at all,
  is worth doing well

  –Earl of Chesterfield

2001-03-20 13:00-0500 P T Withington – Software Journeyman 3

This doesn't have to mean spending a lot of time creating a one-off solution. It should mean doing something well enough that your efforts can be repaid by subsequent users.

A solution that is done well will get re-used. It will benefit from use-testing, and longevity will cause it to be polished and refined.

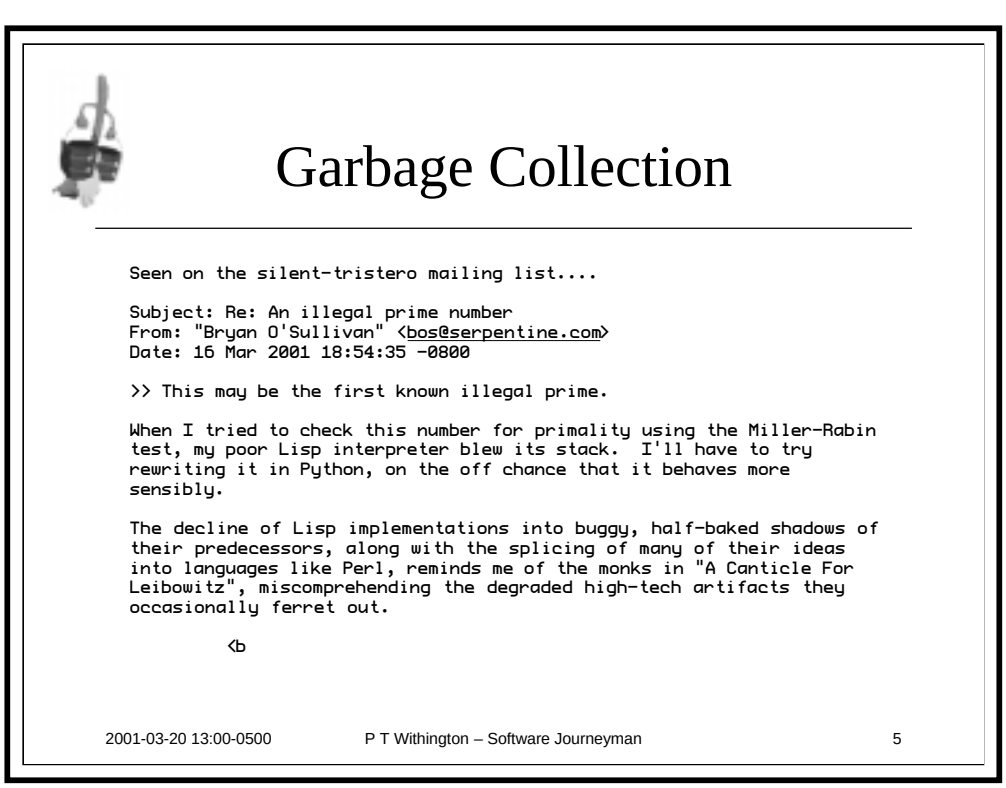
The ultimate extension of this is Open Source.

The power of Lisp and the Lisp Machines has been lost because of proprietary interests.

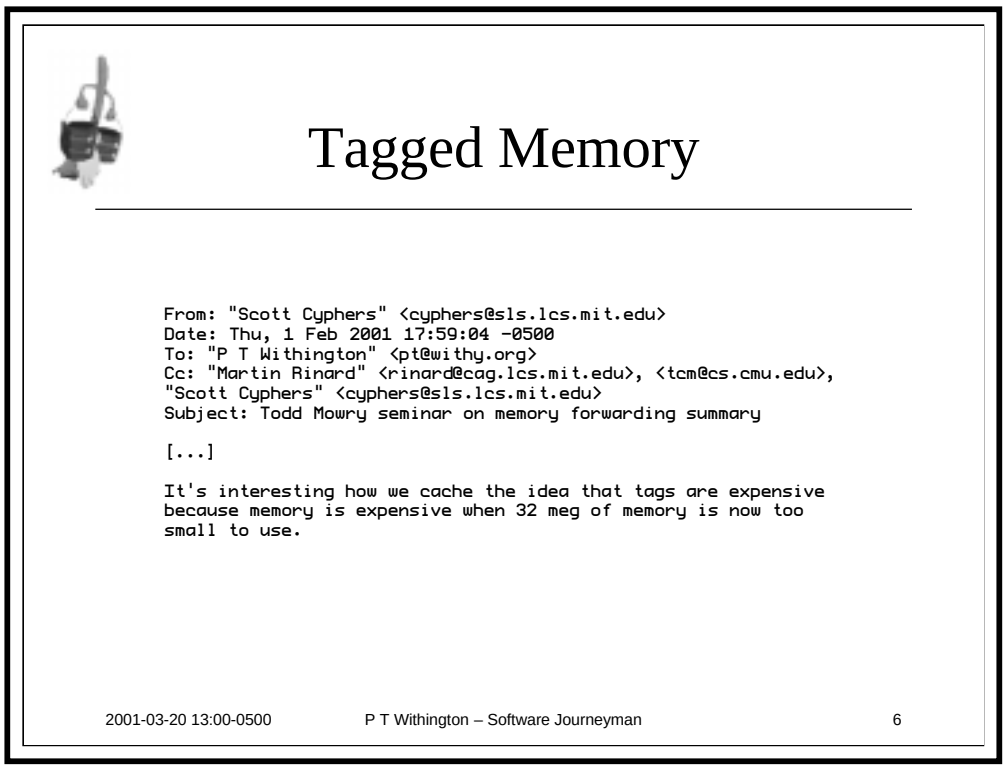
It's unfortunate that some great ideas are (still) secrets.

Good ideas

- Garbage Collection
- Tagged Memory
- Microprogramming



Garbage collection is finally coming back into vogue. Java has done much to legitimize garbage collection. When will operating system and hardware designers realize that this is a technique that is here to stay and support it in their domains?



Tagged memory was a popular hardware technique in the 70's that was used to support run-time type information. Eventually it was discarded because "memory was expensive", and compilers got better at optimizing away the need for type information at run time. Although you might not know it from some of today's languages.

The Standard Template Libraries, which represent an excellent example of something done well that has received significant polish through reuse, are generic through the use of templates. Each instantiation costs code bloat. How much could be saved if generic, run-time dispatch could be used instead?

Microprogramming

"Modern processors are like nitro-fueled funny cars – they excel at the 1/4 mile. Unfortunately modern languages are like Monte Carlo – full of twists and turns."

— Dave Ungar 1998

What's on your hard drive? JPEG's? MP3's? Code?

Microprogramming

	CPU Registers	L1 Cache	L2 Cache	Main Memory	Disk
Quantity	10^1	10^2	10^5	10^7	10^11
Speed	10^9	3*10^8	3*10^7	7*10^5	2*10^3

What's on your hard drive? How is it compressed? Is "micro-code" a form of code compression? (Think about how a single instruction in a virtual machine is expanded into many native CPU instructions. If those instructions fit into one of the CPU caches, are the a "micro-program"? Is an instruction cache like a "writeable control store"?

Pitfalls

*Rules of **Optimization**:*
Rule 1: Don't do it.
Rule 2 (for experts only): Don't do it yet.
—M.A. Jackson

"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity."
—W.A. Wulf

*"We should forget about small efficiencies, say about 97% of the time: **premature optimization** is the root of all evil."*
—Donald Knuth

"The best is the enemy of the good."
—Voltaire

This collection of thoughts originally from Jonathan Hardwick
http://www.cs.cmu.edu/~jch