Browsing Web 2.0 on 2.0 Watts

Why (mobile) web browsers must be parallel

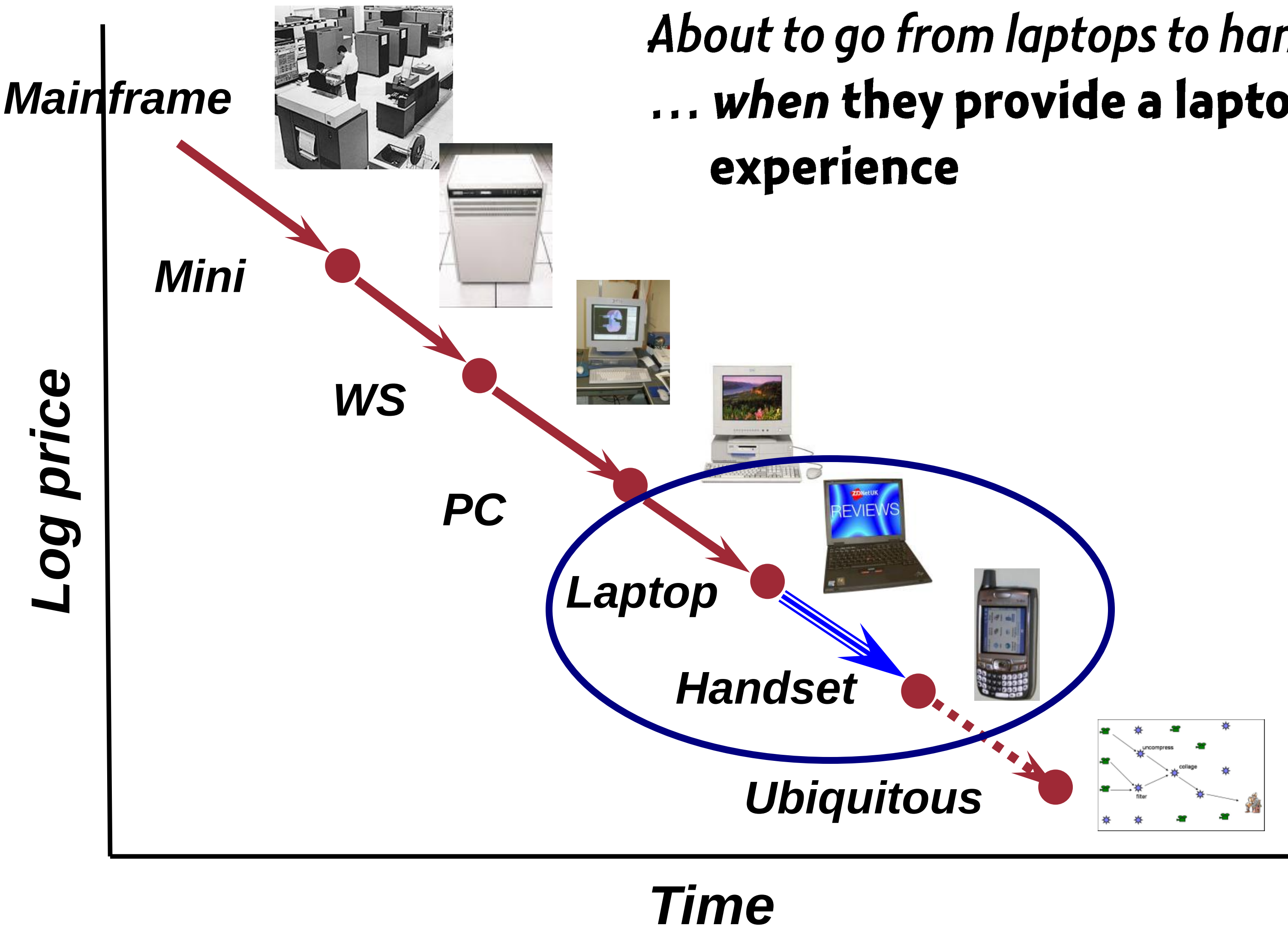
Chris Jones, Rose Liu, Leo Meyerovich, Ras Bodik
with Krste Asanovic and the rest of Par Lab

UC Berkeley

We Live in Exciting Times

About to go from laptops to handhelds…

… *when* they provide a laptop-quality experience



Do we have the technology?

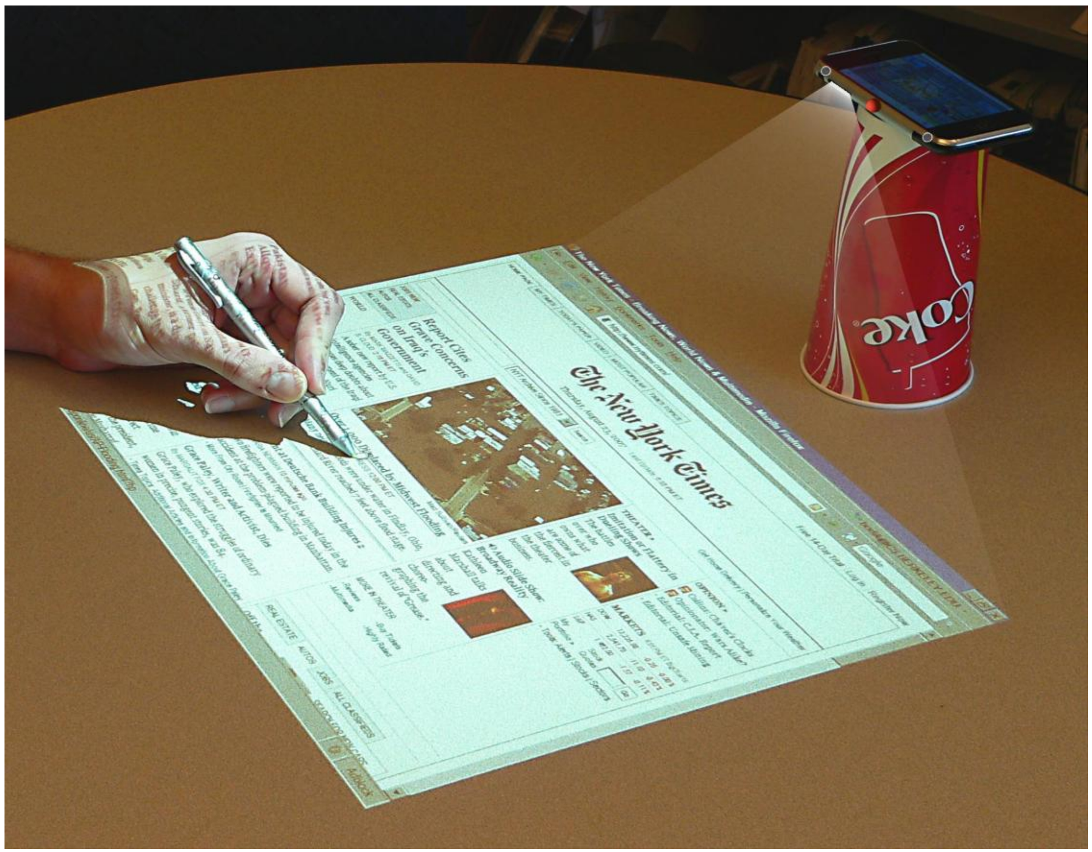
- For a laptop-quality browsing experience, we need
  - **network**: 50Mbps 

  - **display**: at least 1024x768
  - **input**: keyboard-like rate

- All three are forthcoming …

We can build it!

A guy walks into a bar, asks for a cup, and starts his browser.



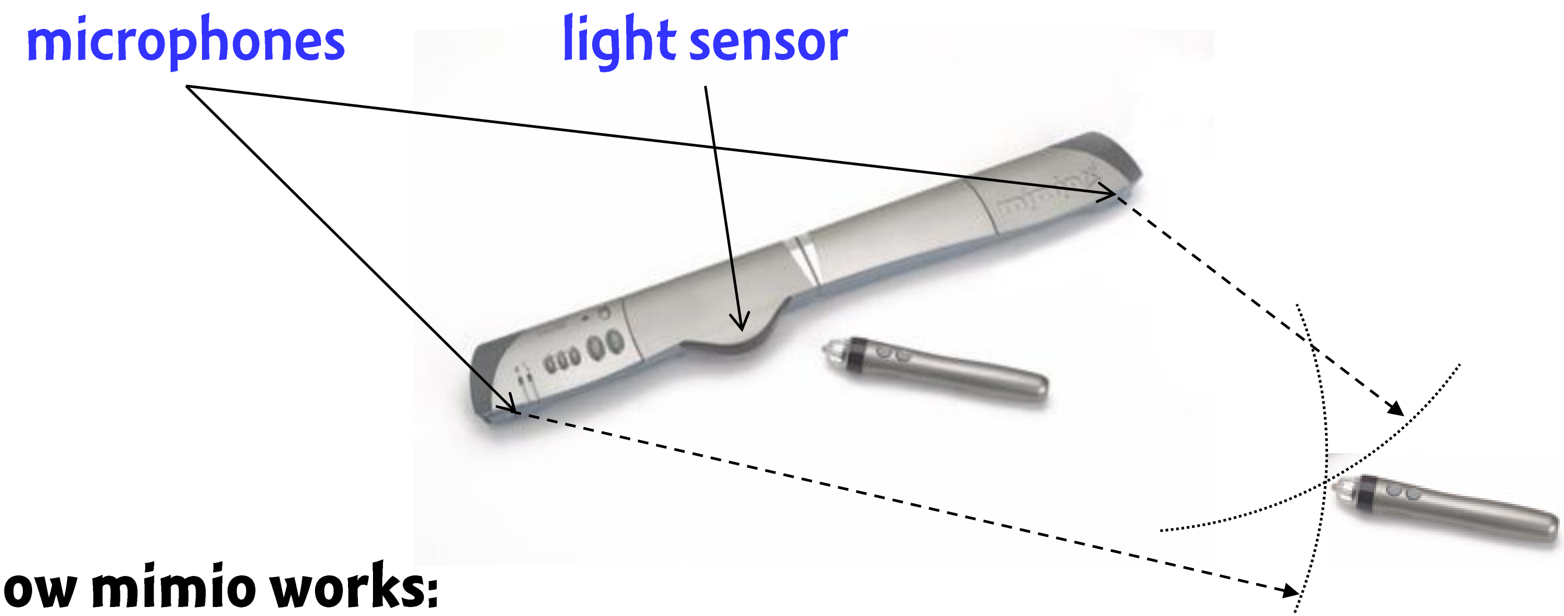
Let's see why this "tablet phone" may actually appear soon...

Display: cell phone projector or "wearable"

Texas Instruments, CES 2008

Input: idea for tablet input for a handheld

- **Inspiration:** mimio, a whiteboard recorder (mimio.com)

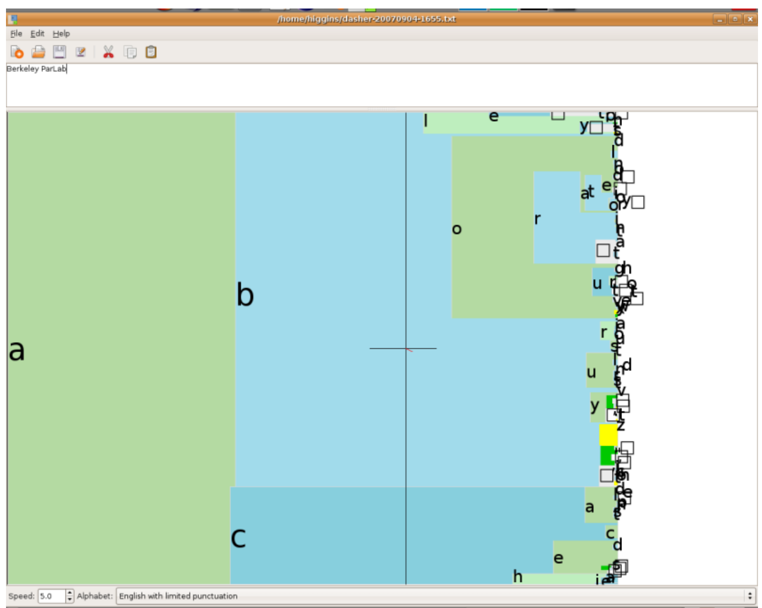


How mimio works:
triangulates in the same way that one measures lightning distance

1. marker simulates a lightning strike: simultaneously emits light and sound signals;
2. capture bar measures sound travel time: yields marker distance to each mic;
3. the two distances determine marker location on the whiteboard; goto step 1

Dasher + picomimio = keyboard-rate input

- Dasher: replacement for traditional keyboards
- Input rates up to ~30 words/minute
- Only needs 1 input axis (up/down) to work
  - can be controlled by picomimio, eyes, tilt sensor, …



See http://www.inference.phy.cam.ac.uk/dasher/ for more info, online demo

What about CPU performance?

- **Display:** many alternatives
- **Input:** half as many
- **Network:** plenty fast soon (if we get better providers)
- CPU speed no longer considered a reason to upgrade …

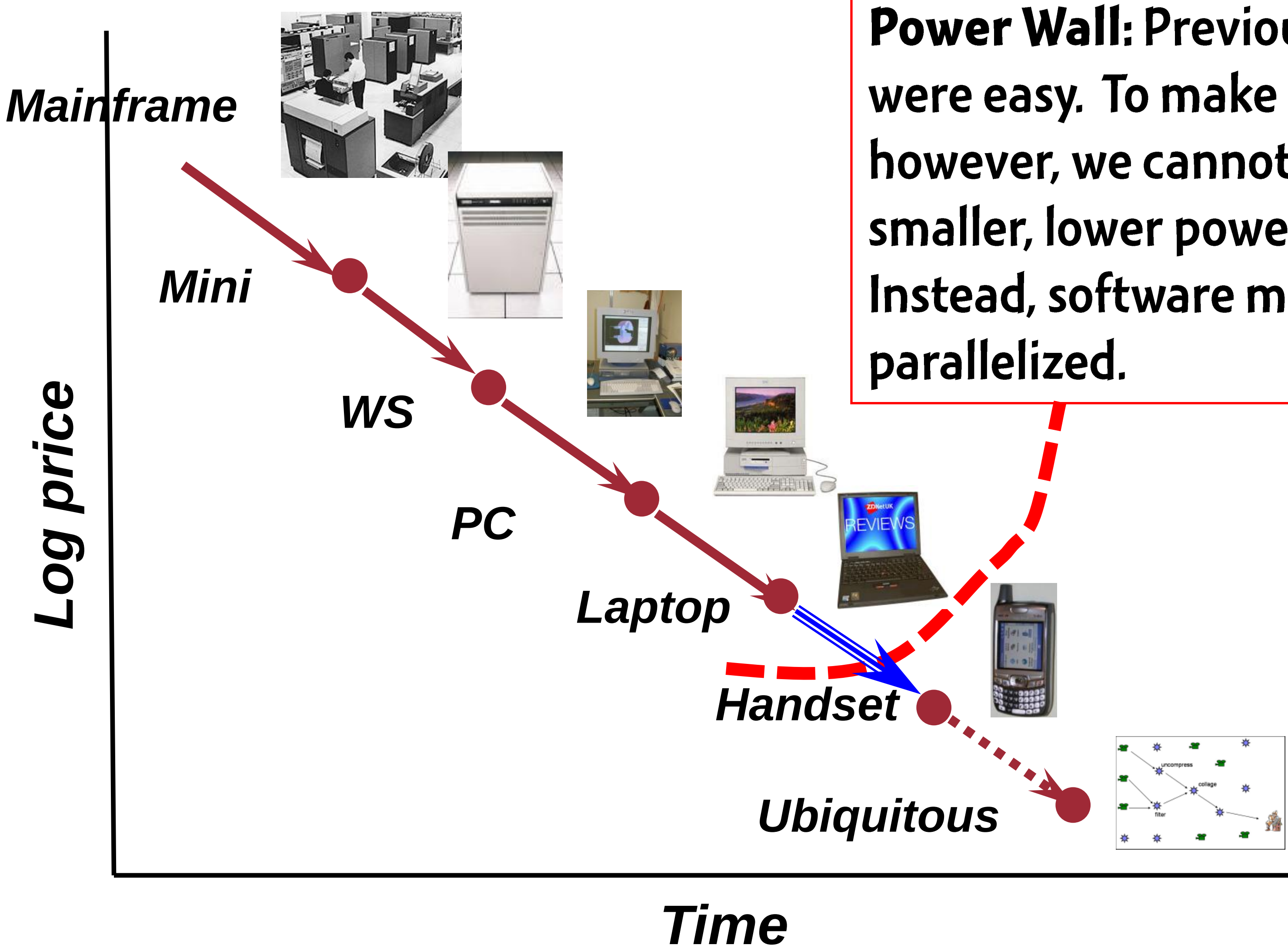
- Loading cnn.com on 1 Mbps and 2 Mbps network

T40 1.6GHz (a very old laptop; 2Mbps network)	7 sec
T40 1.6Ghz (laptop in battery mode, same network)	13 sec
iPhone 600MHz (1Mbps network)	40 sec
iPhone 600MHz (2Mbps network)	37 sec

☞ CPU speed is important

Transition to handhelds is not so easy

Power Wall: Previous Bell steps were easy. To make the net step, however, we cannot wait for smaller, lower power processors. Instead, software must be parallelized.



Key observations

- Current handheld browsers
  - far too slow to offer laptop experience
- But Bell's Law predicts fast, low-power handheld processors
  - does the prediction hold?
- Good news: we should be able to get 50GOPS at 2W
  - even in current 65nm technology (40pJ/op)
  - (compare with best laptops: ~20GOPS at 20W, more w/ SIMD)

☞ **Bad news: the 50GOPS will come from 10-100 cores**

☞☞☞ **Must build a parallel browser**

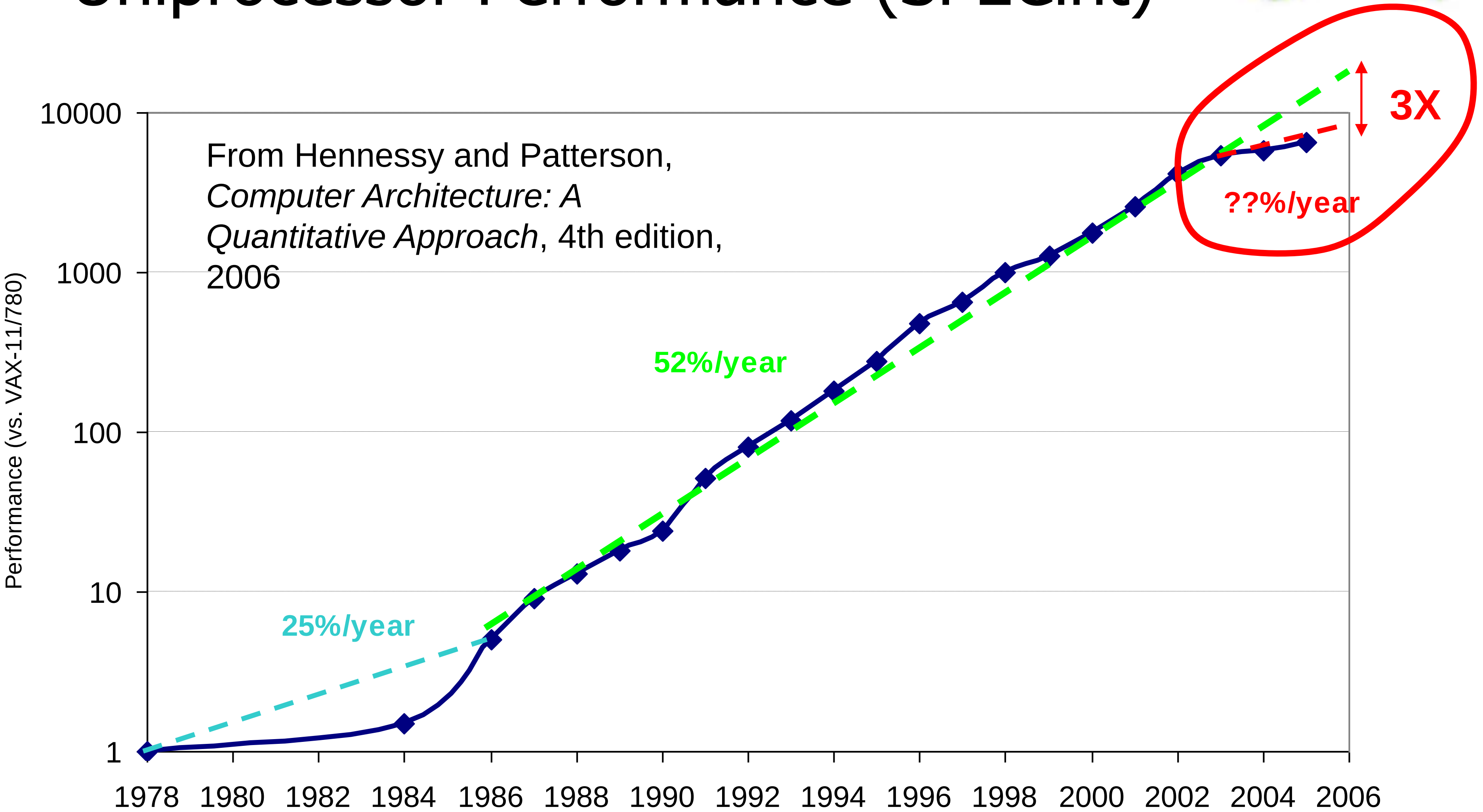
Talk Outline

▪ Browsers will be parallel	Chris
▪ Hardware trends	Rose
▪ Parallel Parsing	Chris
▪ Future Apps	Ras
▪ Parallel Browser Scripting	Leo
▪ Manycore OS	Rose

Hardware Trends

In industry and in the Berkeley Par Lab

Uniprocessor Performance (SPECint)



Conventional Wisdom in Computer Architecture

- **Old Conventional Wisdom:** Power is free, Transistors expensive
- **New Conventional Wisdom:** "Power wall"

  Power is expensive, Transistors free
  (Can put more on chip than can afford to turn on)

  ⇒ Sea change in chip design: multiple "cores"
  (2X processors per chip / ~ 2 years)
  - More, simpler processors are more power efficient

"We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

Paul Otellini, President, Intel (2005)

Multicores are Here

- Major microprocessor companies switch to Multicores
  - E.g. Intel Core 2 Duo (2 cores/chip, 1 thread/core)
  - Sun Niagra (8 cores/chip, 8 threads/core)

- Performance trends:
  - 2X CPUs / 2 yrs ➔ parallelism for performance
  - 2X sequential perf. / 5 yrs

- Given this new performance trend, in 10 years…
  - Uniprocessor performance improves 4X
  - Multicore performance improves 32x

Significant performance improvements continue as long as applications become parallel

Potential Core Designs

- **Fat cores**: (general purpose) Out-of-order superscalar for good single thread performance
  - E.g. Intel's Core2 Duo (Pentium pro) but **fat core may be even simpler in the future**
  - Consumes more power so can have less/chip
- **Thin cores**: (general purpose) In-order 1-2 issue cores with vector/SIMD units
  - E.g. simple RISC 5-stage pipeline, Larabee
  - Lower power so can have 100s/chip
- **GPU:** (programmable domain-specific cores, no cache coherence) Getting more programmable to support more app domains
  - E.g. NVIDIA G80
  - Very low power, 100s of cores/chip

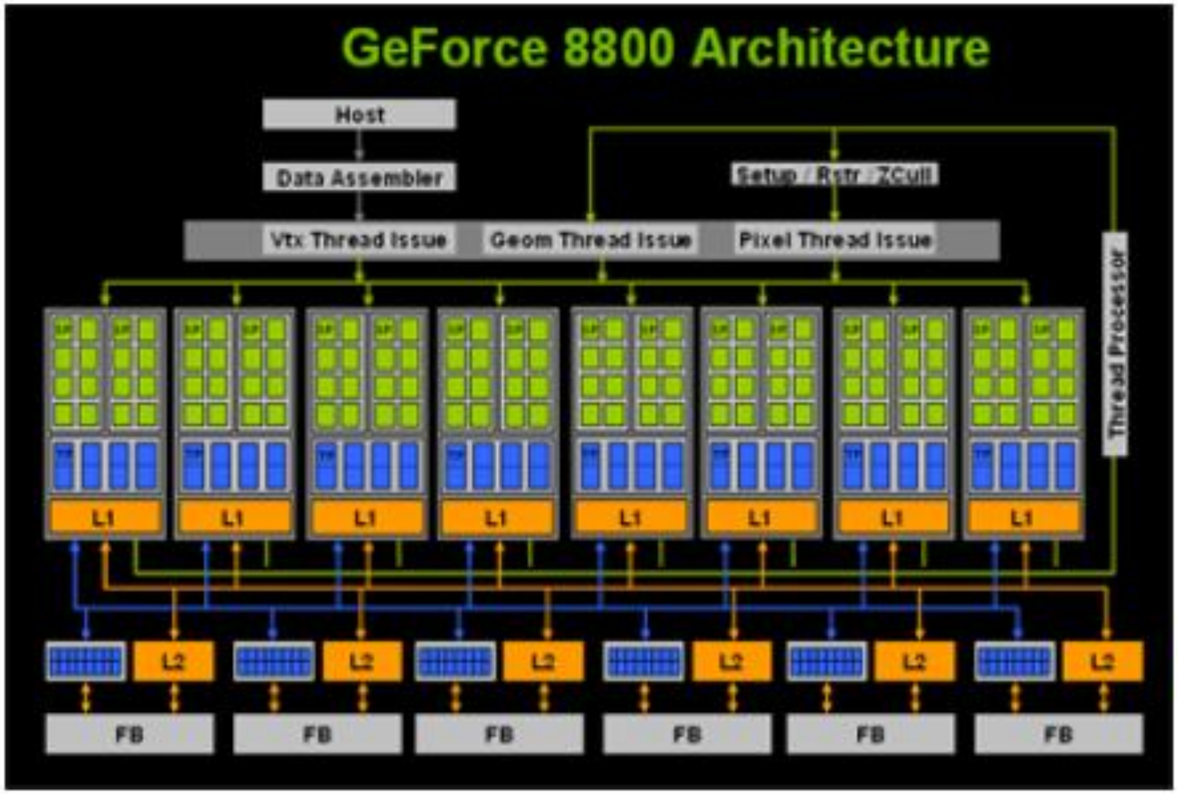
Heterogeneous Multicores in Future

- Heterogeneous architecture - mix of fat, thin, GPU in one machine

→ The more thin cores you can use, the lower power you consume

NVIDIA G80 GPU

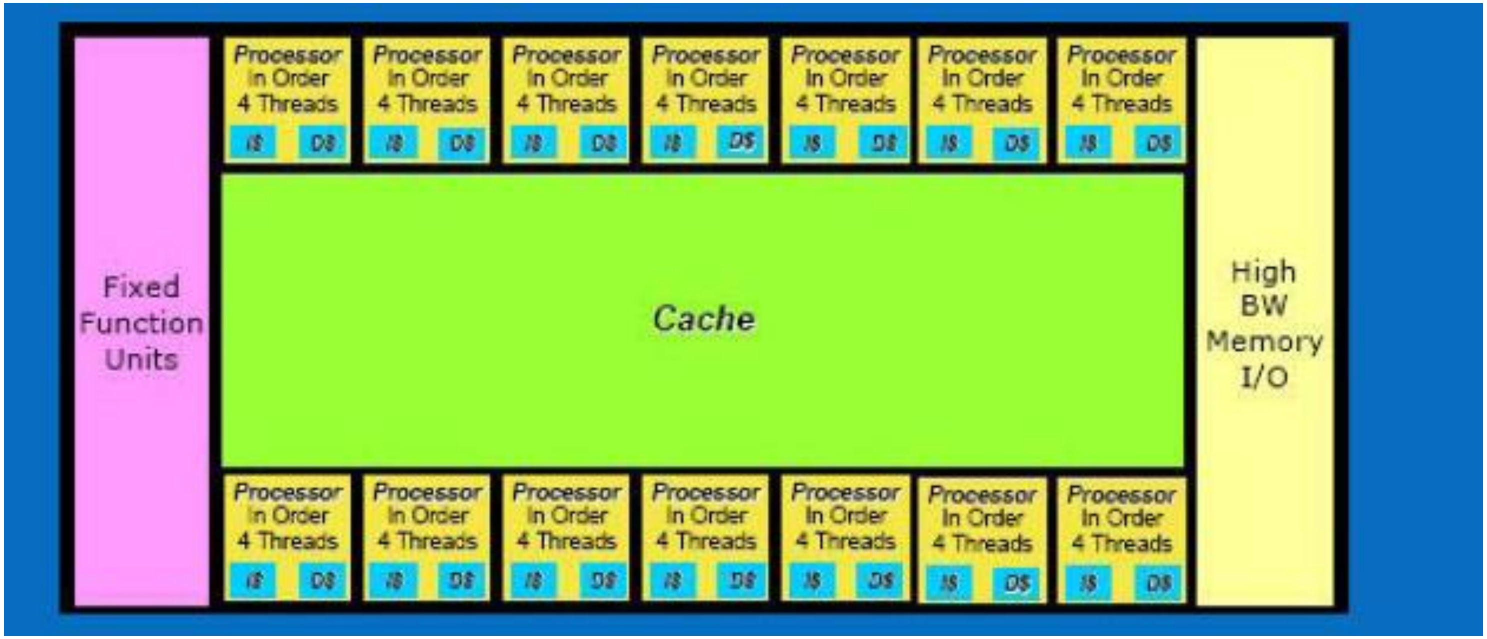
Figure Credits: NVIDIA



- 128 way parallelism in the form of 16 processors, each of which are 8 way SIMD
- High throughput: 128 * 1.35 GHz * 2 Flops/Hz = 346 GFlops (IEEE SP)
- 768 MB of memory, 6 channel GDDR3 => 86.4 GB/s
- 90 nm, 680M Transistors, 480 mm2, 200 W

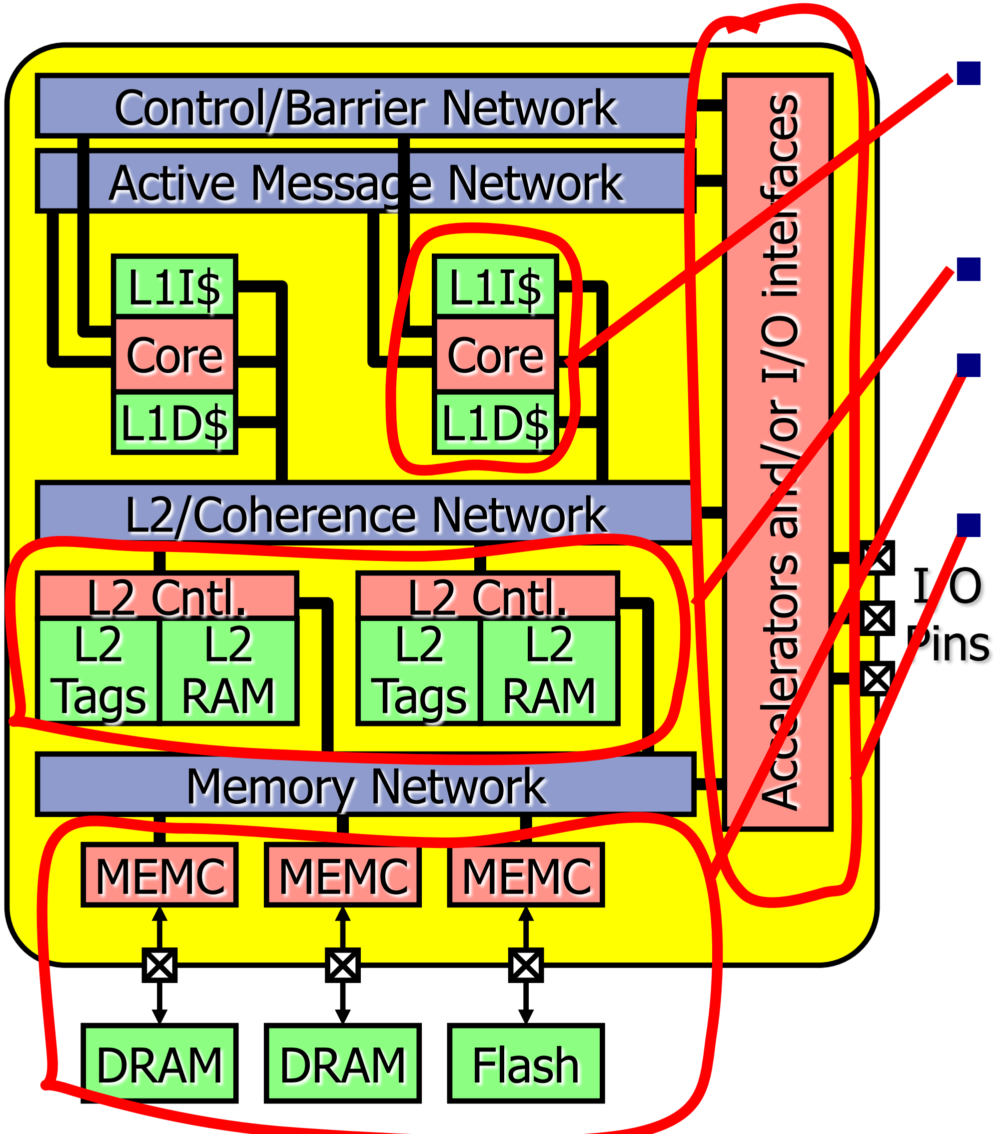
Intel's Larabee (potential design)

Figure Credits: Intel



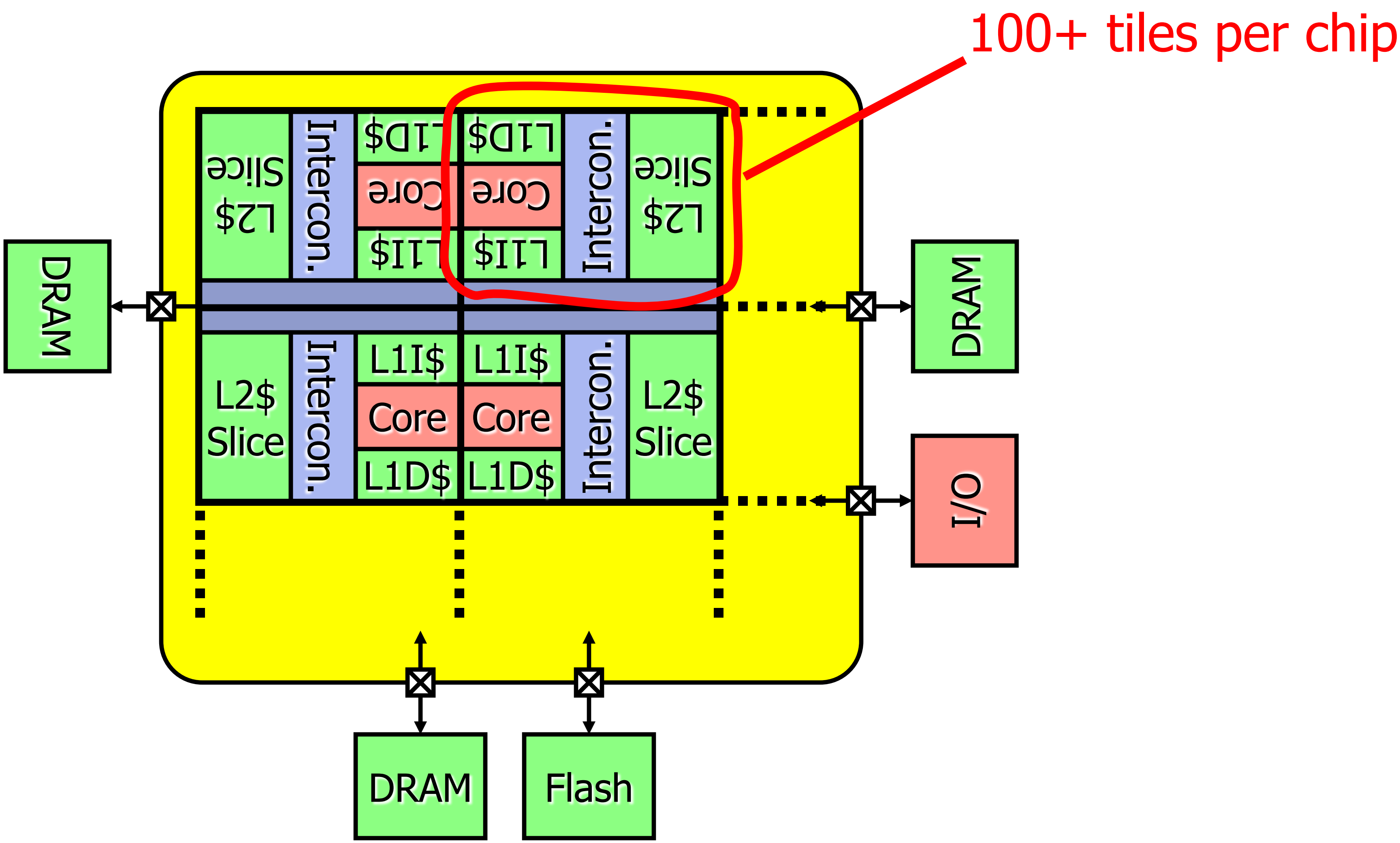
- Greater than 30 thin cores/die + a few fat cores + fixed function accelerators
- 4 threads per simple core
- Vector units (16-wide)
- VLIW instructions
- Cache coherent L1 caches
- L2 unified cache w/ dynamic cache partitioning for private caches
- Primitives for synch.

Logical View of ParLab Architecture

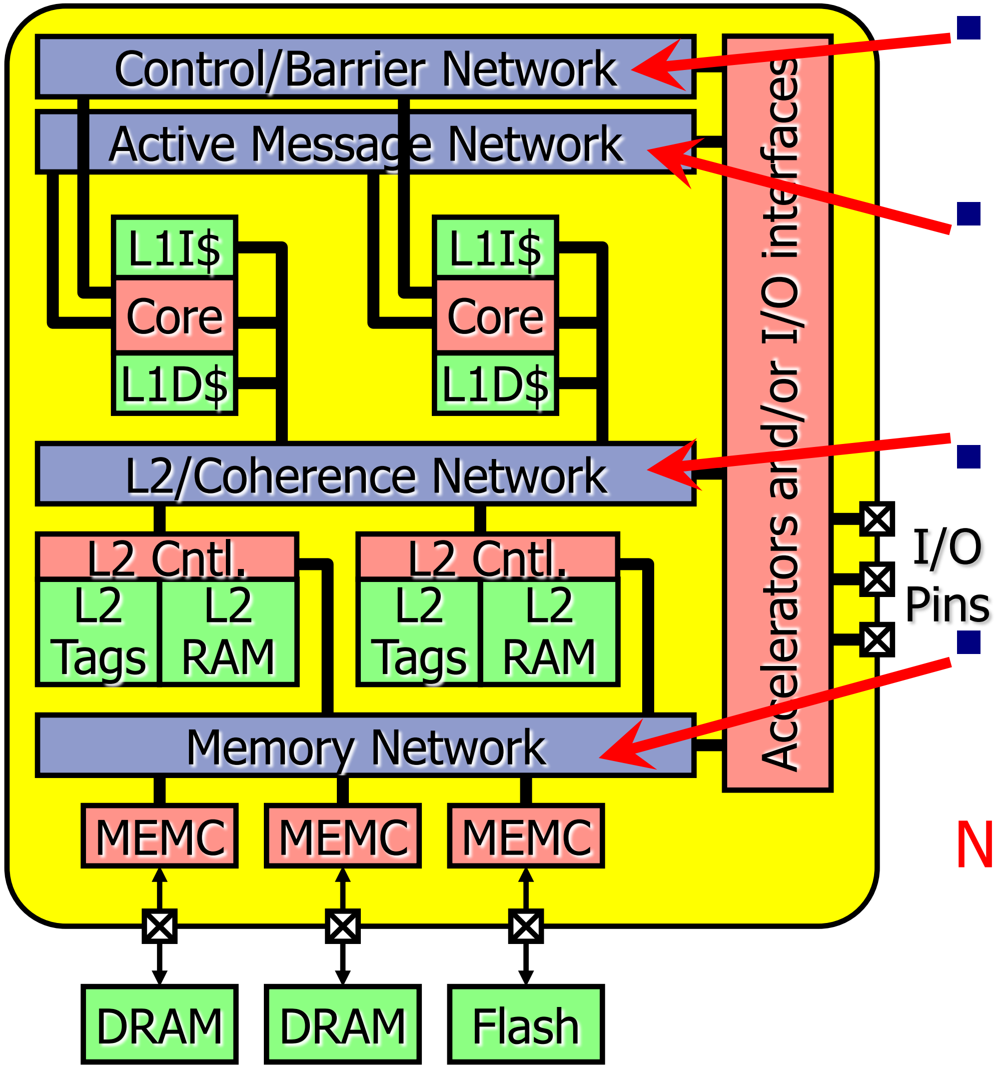


- Processing Element - core, private L1 instruction and data caches/RAM
- Unified L2 cache/RAM
- Unified physical memory, Flash replaces rotating disks
- Special function accelerators (e.g. FFT, image decompression) and I/O interfaces

Physical View of Tiled Architecture



Specialized On-chip networks

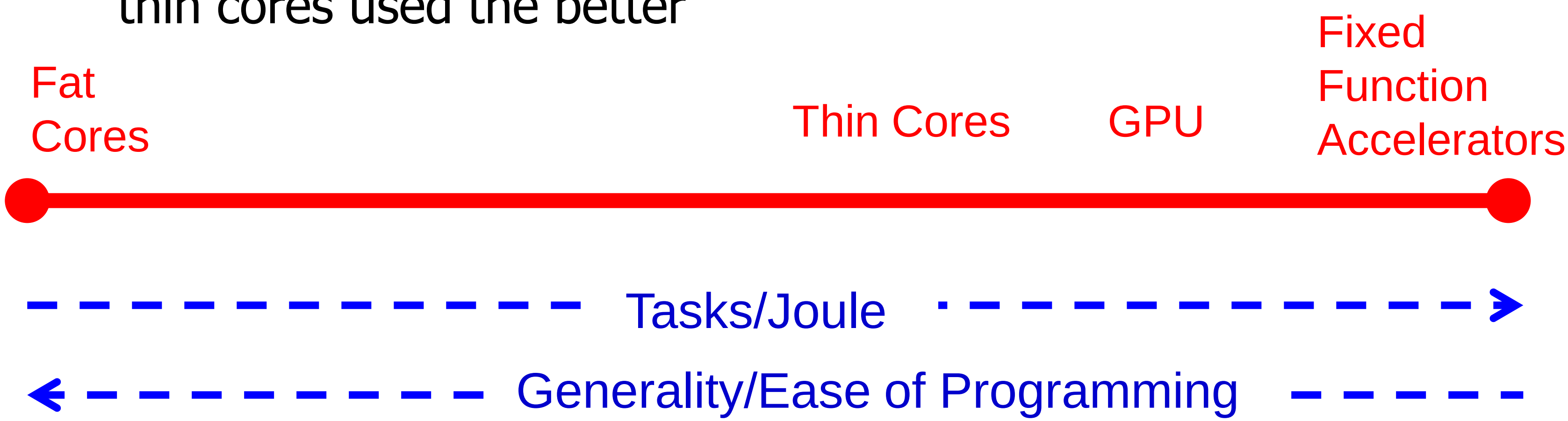


- Control networks combine 1-bit signals in combinational tree for interrupts & barriers
- Active message networks carry messages between cores (register-level RPC) – e.g. increment remote register
- L2/Coherence network connects L1 caches to L2 slices and indirectly to memory
- Memory network connects L2 slices to memory controllers
  - Flash memory

Networks provide quality of service (QoS) on bandwidth

HW Trend Takeaways

- Heterogeneous Design – mix of thin, fat, or GPU (special domain)
  - Even fat cores will be simpler than current cores
  - Parallelism will drive performance improvements -> the more thin cores used the better



- Spatial partitioning opens possibilities
  - less context switches/time multiplexing, more messaging
  - provides better isolation/protection/security
- QoS guarantees on resources (capacity, bandwidth)

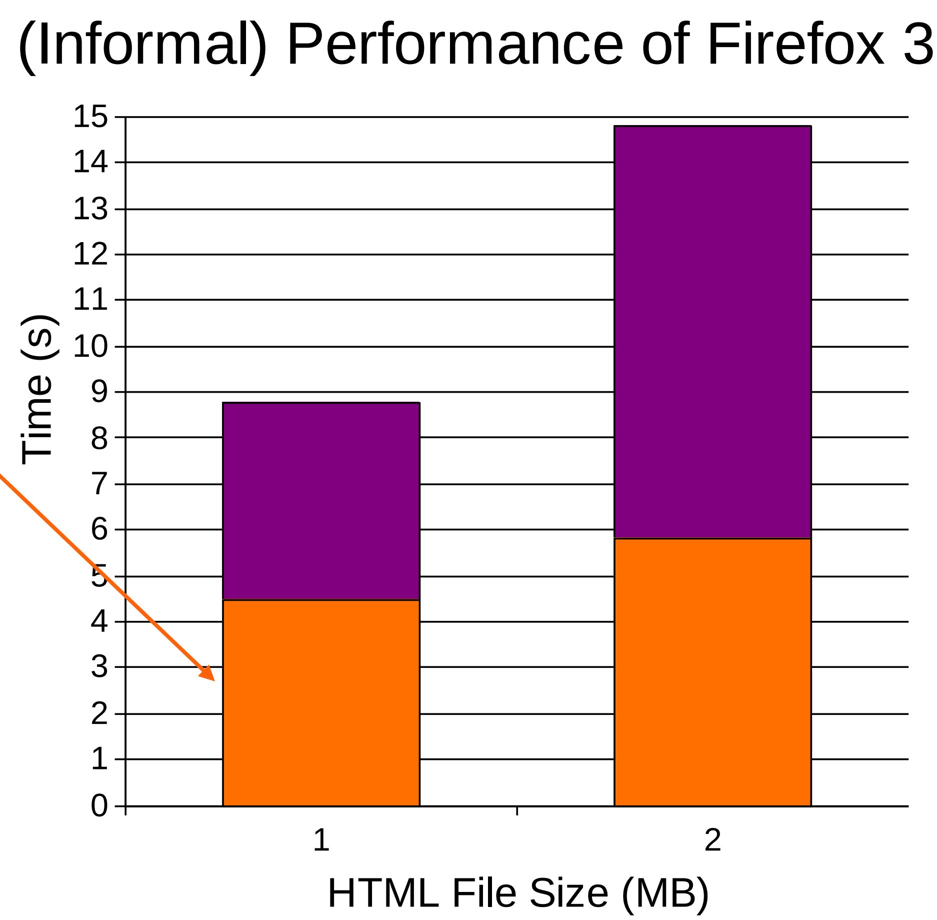
Parallel Lexing and Parsing

Initial results and future work

What fraction is HTML compilation?

- **10-40% of time spent in lexing, parsing, syntax-directed translation**
- (remainder in layout/rendering)

- loading a fark.com page from disk cache; little JavaScript
- on (old) debug build of FireFox 3



☞ **HTML compilation must be parallelized**

Preliminary work: parallel lexical analysis

- For the thin-core Cell processor on the Playstation 3
  - the Cell has 1 "fat" core and 8 "thin" cores
- The lexer is essentially a finite state machine
  - considered inherently sequential ("embarrassingly serial")

- **Goal**: efficiently run this "sequential" code on thin cores

- We parallelized lexing by algorithm-level speculation
- **Preliminary results**: ~linear speedup up to 6 cores
  - if lexing can be run in parallel on thin cores …

Lexing, from 10,000 feet

Goal: given lexical specification and input, find lexemes

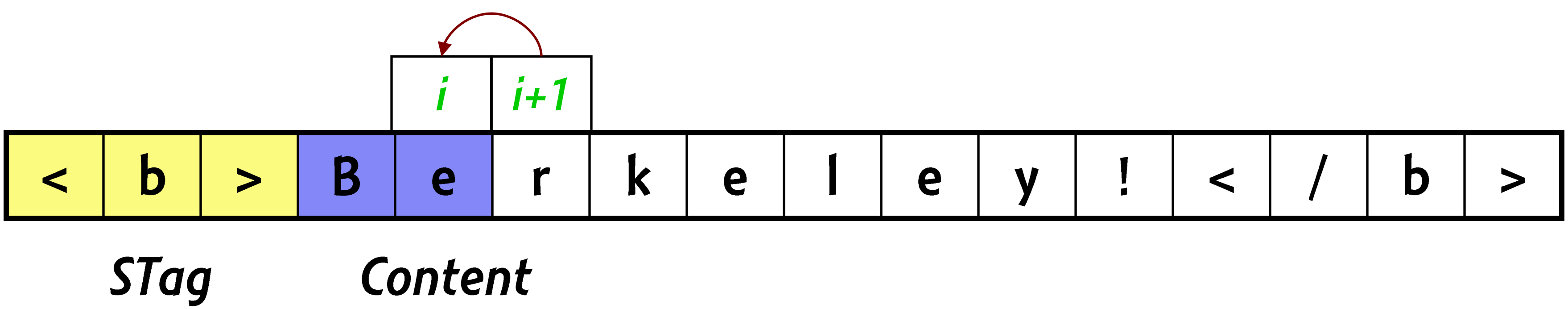
```
Content ::= [^<]+
ETag    ::= </[^>]*>
STag    ::= <[^>]*>
```

<	b	>	B	e	r	k	e	l	e	y	!	<	/	b	>

STag *Content* *ETag*

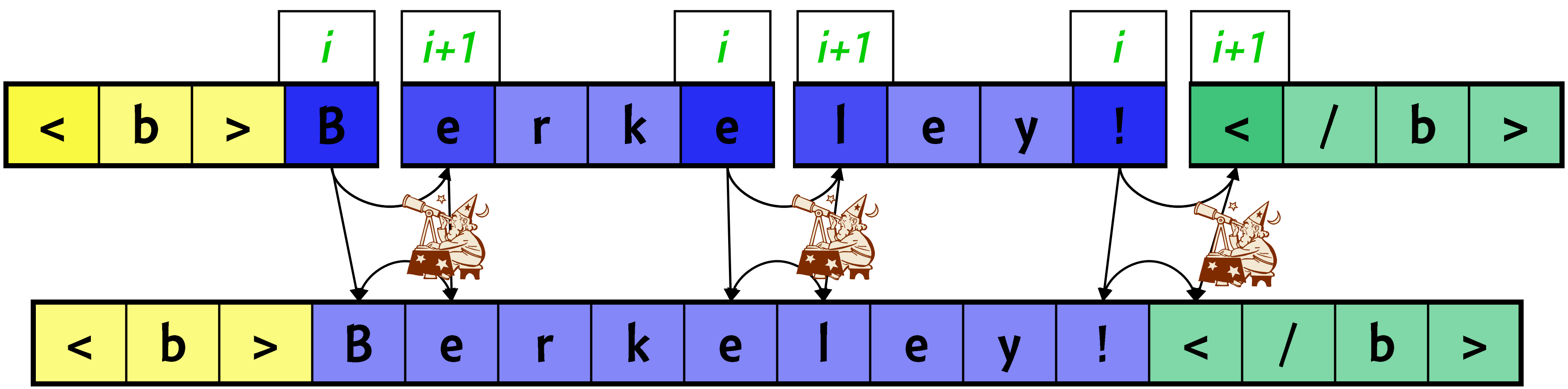
Problem: lexing *seems* “inherently sequential”

- To know automaton state at input position *i+1*
- We need to know the automaton state at position *i* !



Ideal solution

- Divide input among the processors
- For each processor starting at position *i+1*
  - Ask an **oracle** in which state the neighbor at *i* finished
  - Scan in parallel from next state, at *i+1*
- Finally, merge the results



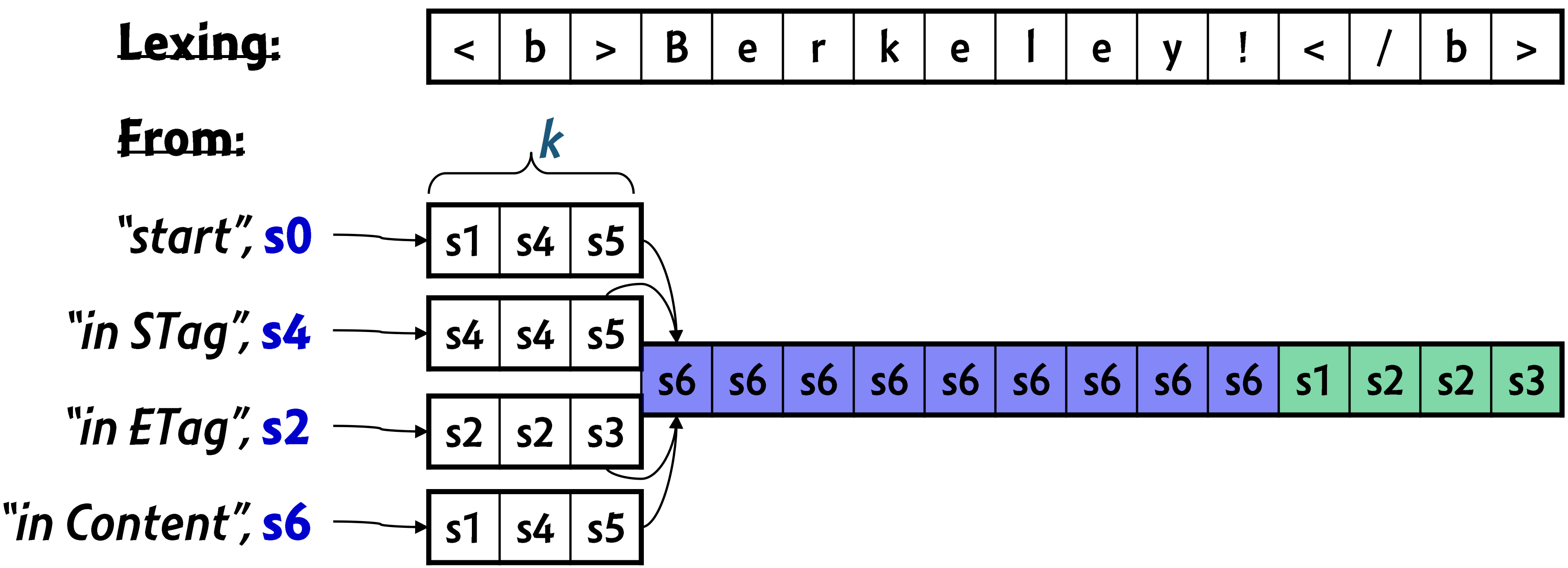
Practical solution: guess! (speculate)

How can we guess from position *i+1* the state at position *i*?

- (1) Could have been *every* automaton state
  - ☺ the "speculation" is always correct (not really a guess)
  - ☺ can yield O (log *n*) algorithm [Hillis and Steele] …
  - ☹ … but prohibitively expensive in practice
- (2) Was one of a "likely set" of automaton states
  - ☺ can be more efficient than algorithm (1)
  - ☺ can fine-tune speculation based on language and workload
  - ☹ speculation can be wrong
  - ☹ still can be expensive (memory overhead, bad guesses)
- ***But we can do better …***

Our solution (1/2)

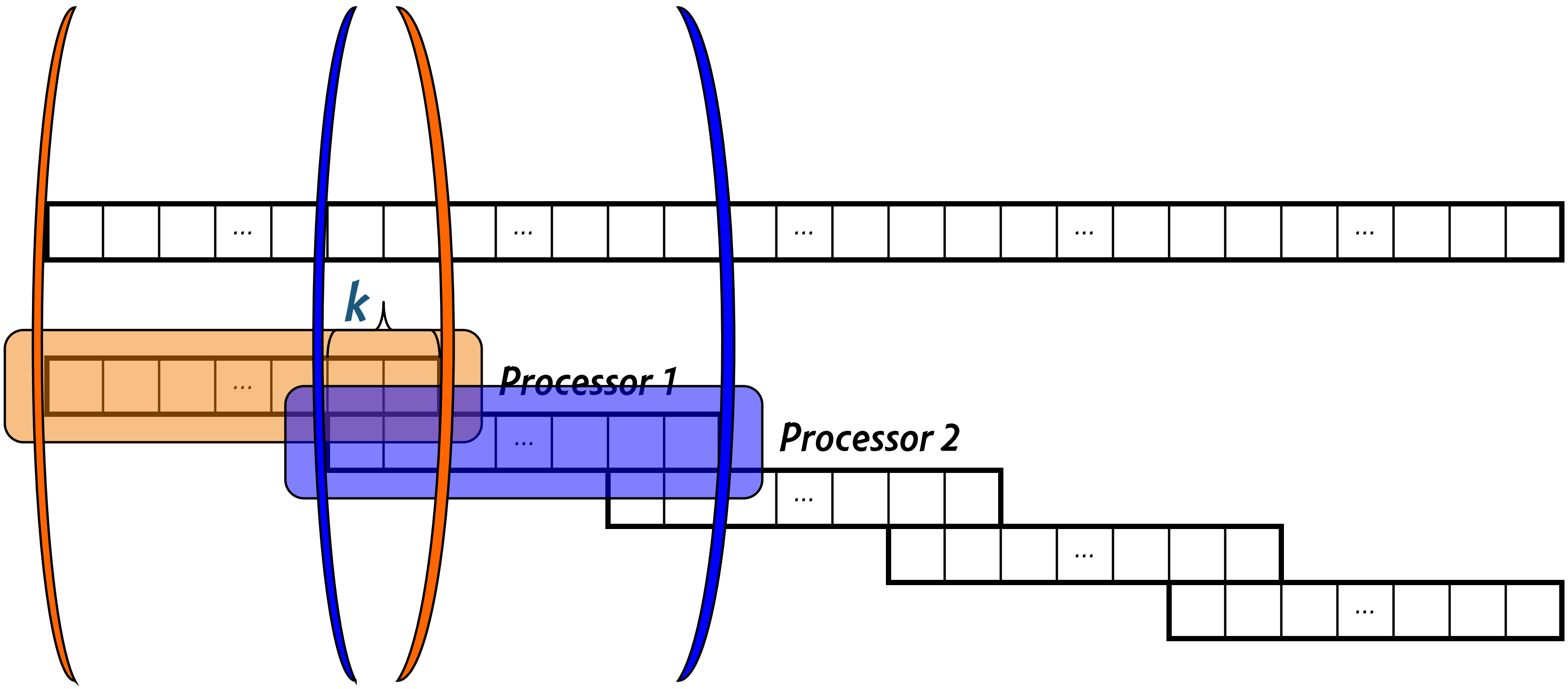
Observation: in "real" lexers, the DFA converges to a *stable, recurring* state (think "start state"), from multiple initial states, after a small number *k* of characters



☞ Only need to follow one DFA path instead of several

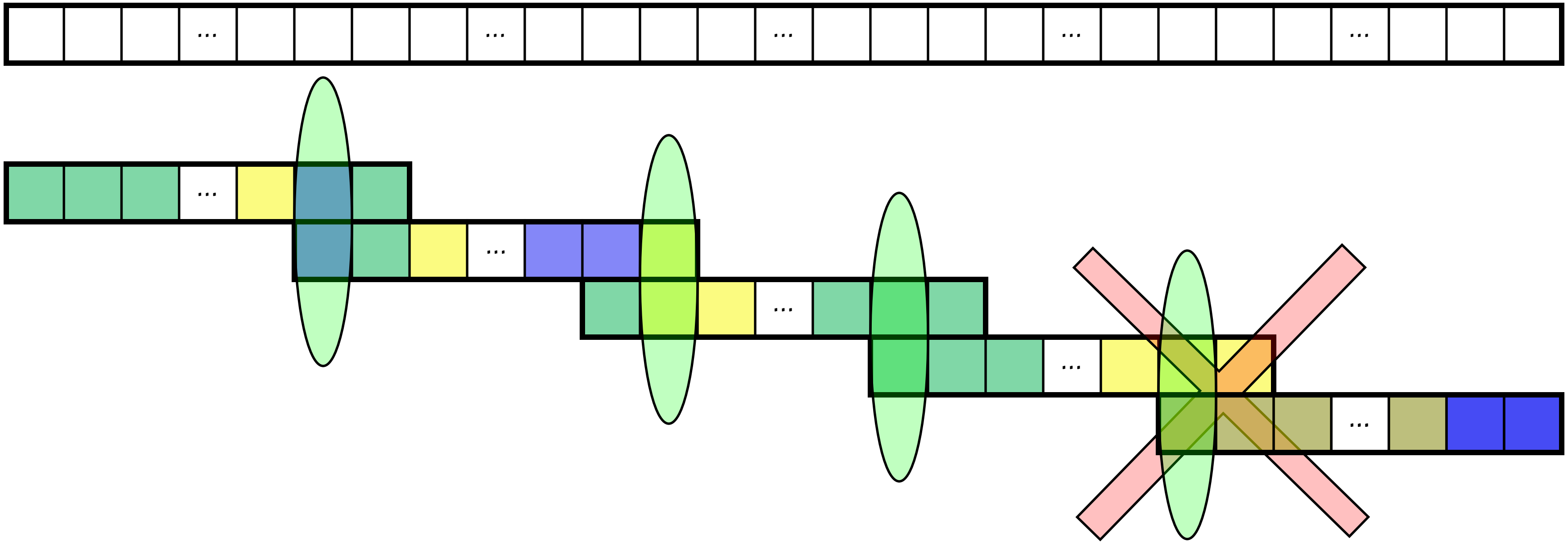
Our solution (2/2)

- **Sketch of our algorithm:**
  - split input into blocks with *k*-character overlap
  - scan blocks in parallel, each starting from "good" initial state



Our solution (2/2)

- **Sketch of our algorithm:**
  - split input into blocks with *k*-character overlap
  - scan blocks in parallel, each starting from "good" initial state
  - find if blocks converge: expected in *k*-overlap
  - speculation may fail; if so, block is rescanned

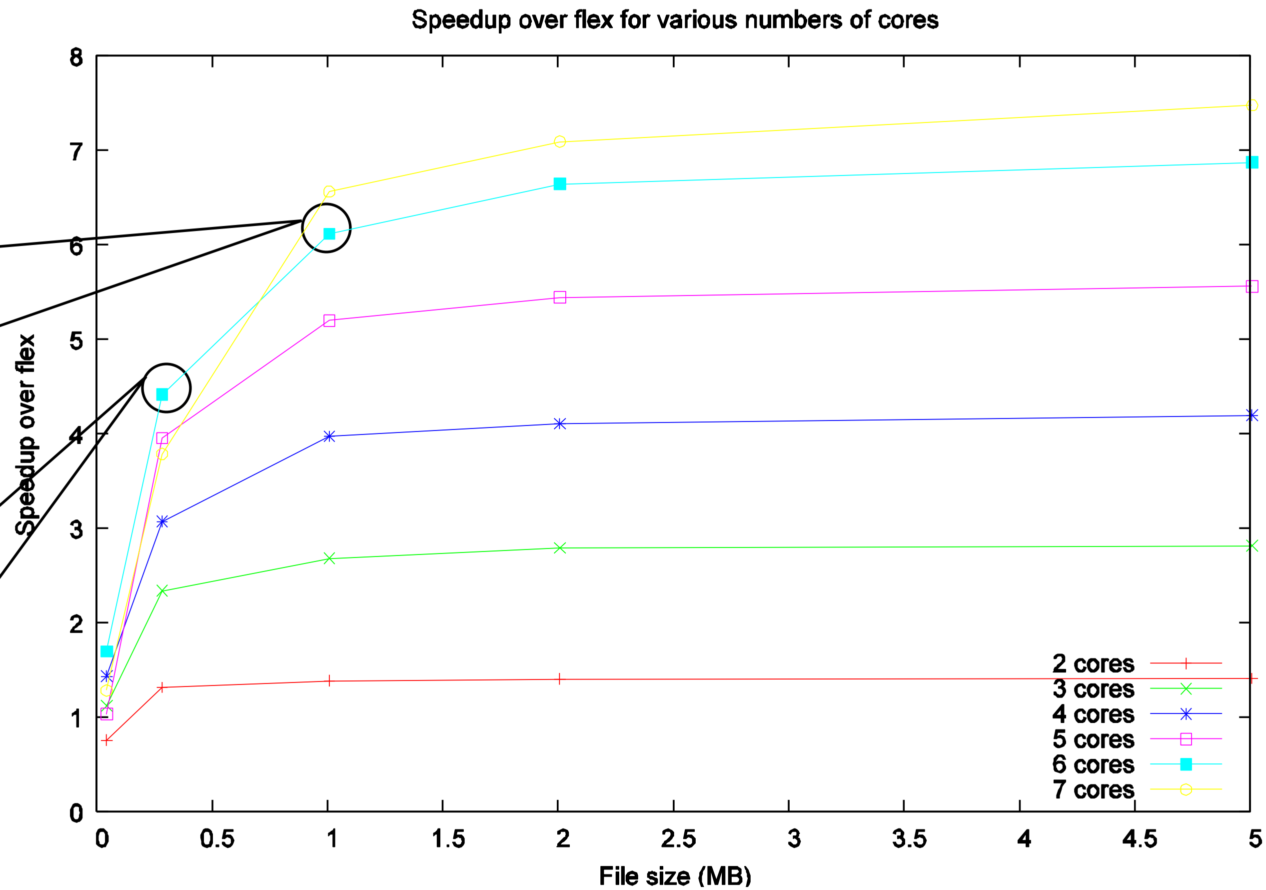


Preliminary results: speedup over flex

- **flex**: optimized, single-thread lexer on fat Cell core
- Speedup computed by *flex time / cellex time*

future page sizes: 5 cores are 6x faster than flex

today's page sizes: 5 cores are 4.5x faster than flex



The parser

- Harder than Finite State Machine computation
  - lexer: FSM
  - parser: "FSM" where states are stack configurations
- Hence, we can't directly reuse lexer parallelization
- But we have ideas on parallelizing these algorithms:
- CYK
  - ☺ general: handles all context-free languages
  - ☹ dynamic programming → $O(n^3)$ time, $O(n^2)$ space
- Packrat
  - ☹ less general: like CYK, but with some restrictions
  - ☺ restrictions + DP → $O(n)$ time and space

Lexing/parsing summary

- Lexing seems sequential …
  - … but can be parallelized by algorithm-level speculation
  - and the parser appears amenable to the same
- Parallel lexing performs well:
  - when designed for "thin-core" platform (Cell)
- We will apply lessons learned from lexing to parsing
  - And target GPUs

☞ **Parallel algorithms for thin core = high performance**

Characteristics of Future Web Applications

Why speculate about application domains?

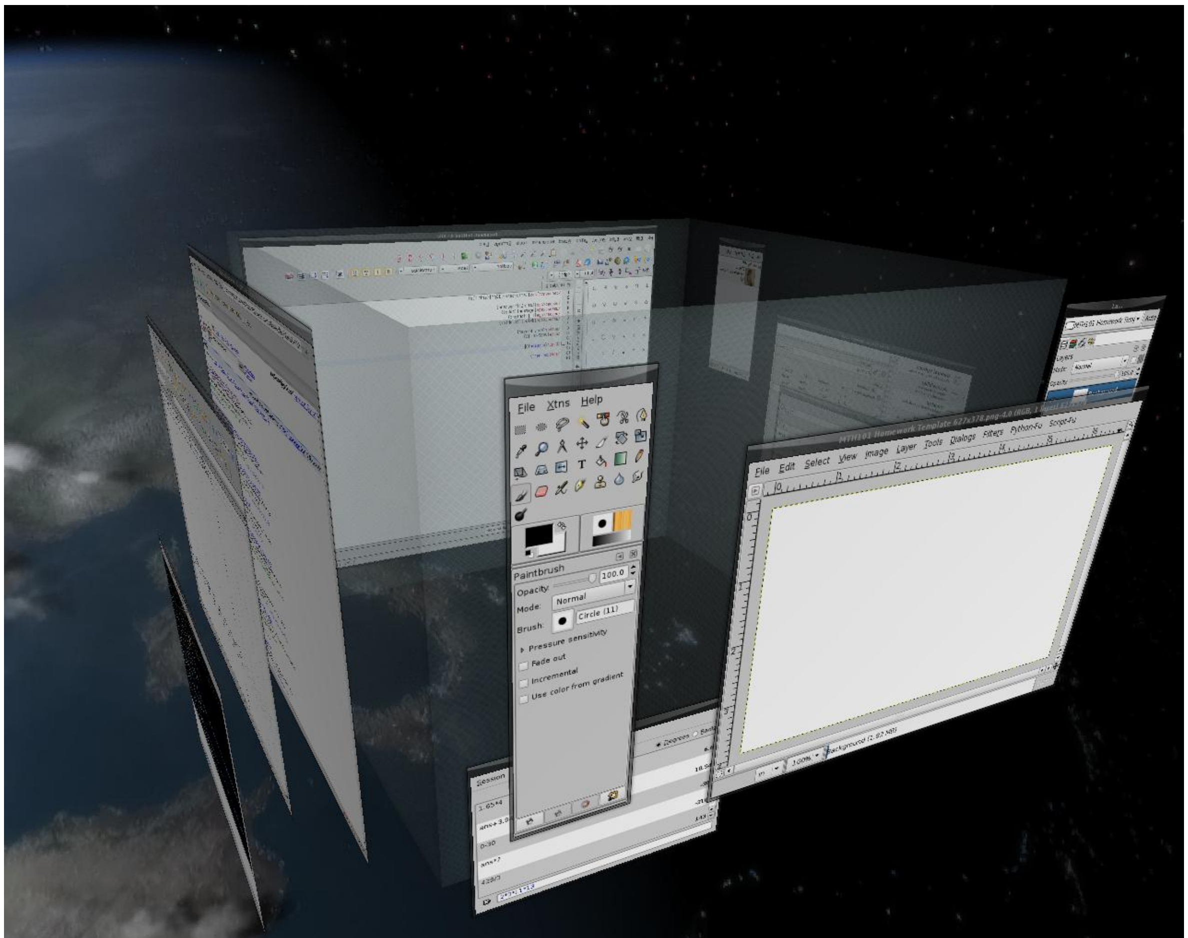
- Performance needs
  - influences HW architecture, compilation, plugin architecture
- Programming abstractions
  - DOM + JS originally intended for mostly static 2D documents
  - is this model suitable for future apps?

Future apps

- Future web apps will be like desktop apps and more …
  - browser = the new windows manager → tabs outdated
  - browser = new OS (local storage, refined security policies)
  - new usage modes (multi-touch, camera-based input, data)
- We want to identify domains that a browser can support
  - hypertext documents and media
  - office suites
  - simpler games
  - rich visualization, for data presentation (eg search results)

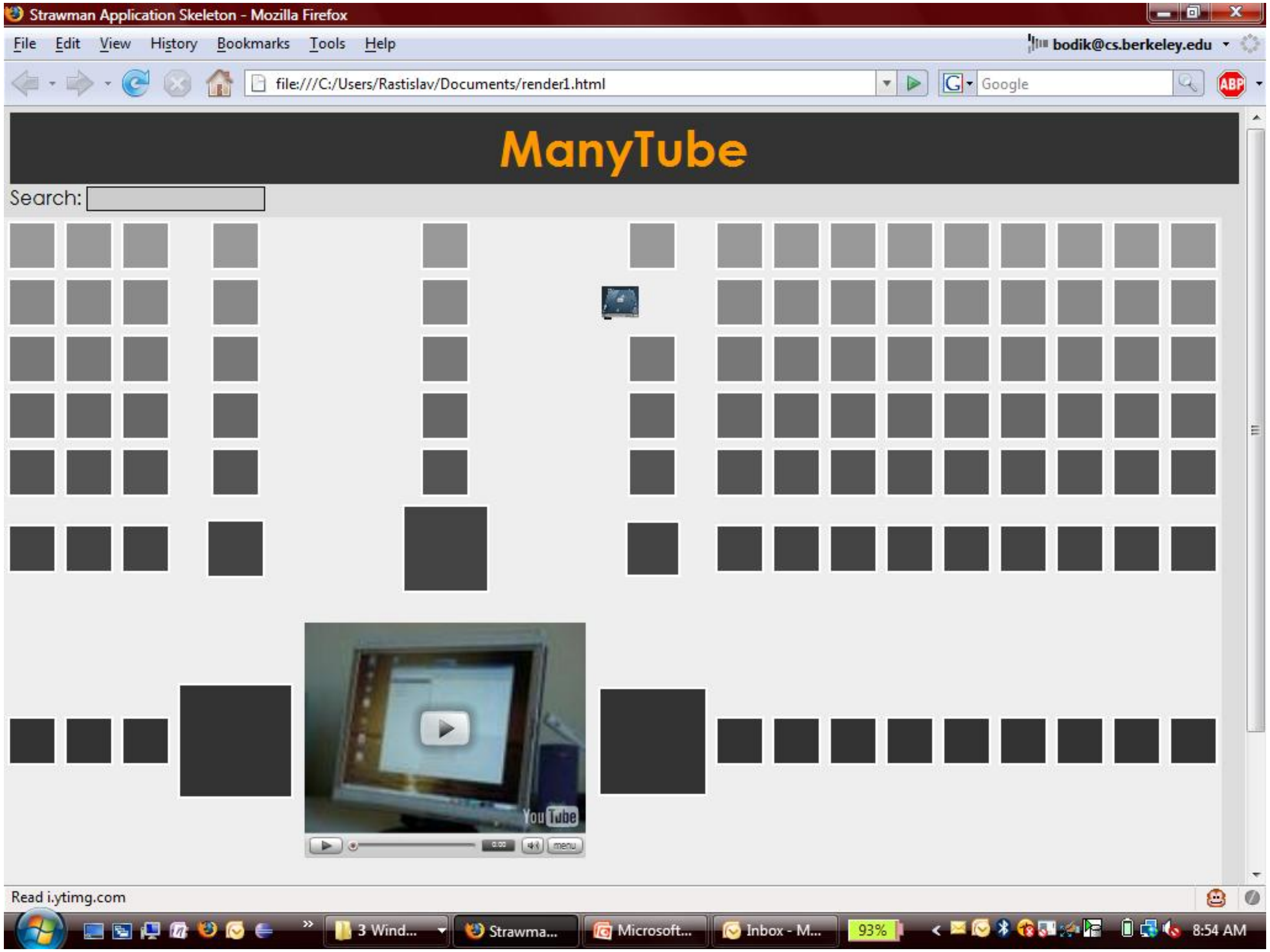
Example 1: Baryl Desktop Manager

- 3D desktop with physical properties



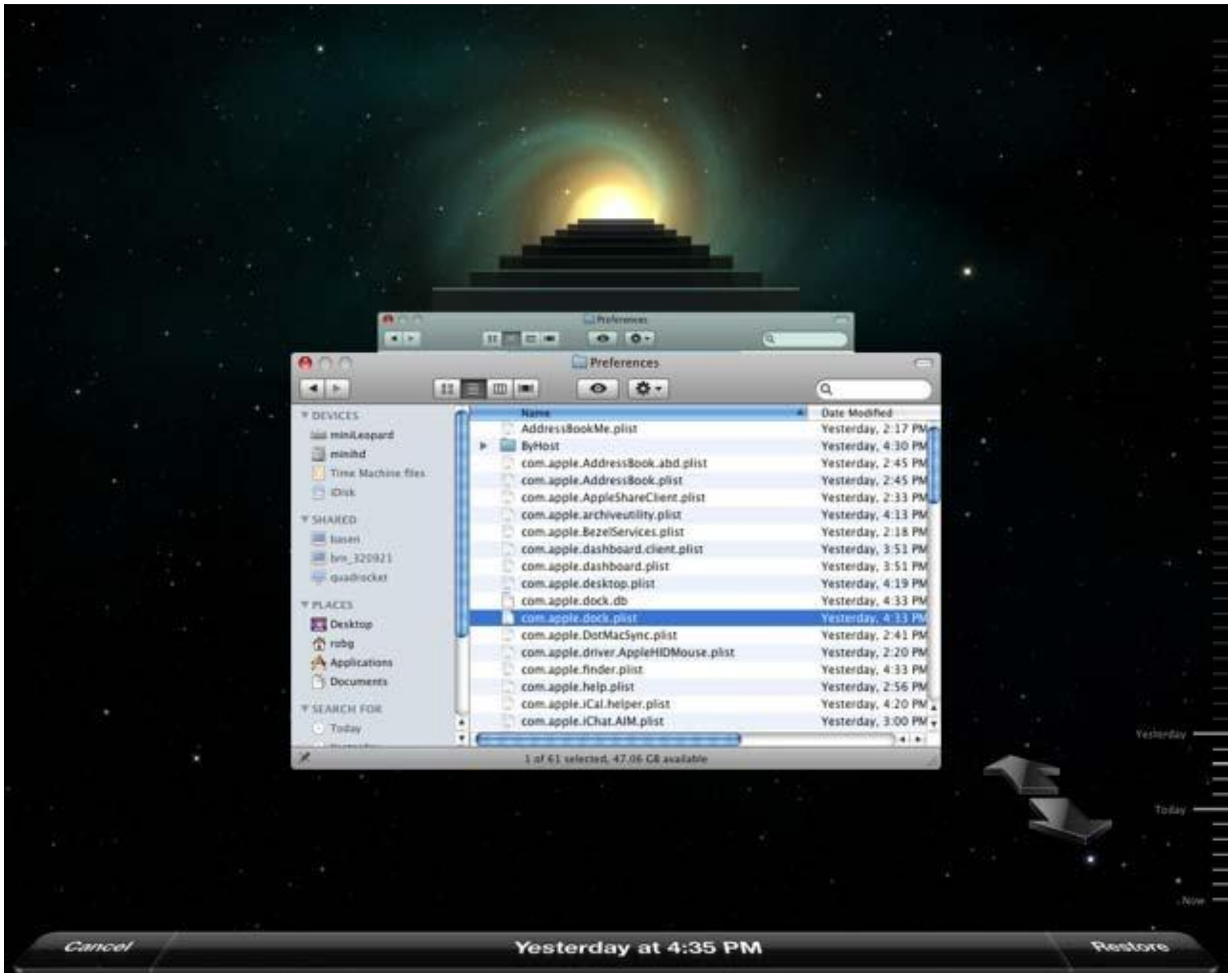
Example 2: ManyTube mockup demo

- Example of a new media app



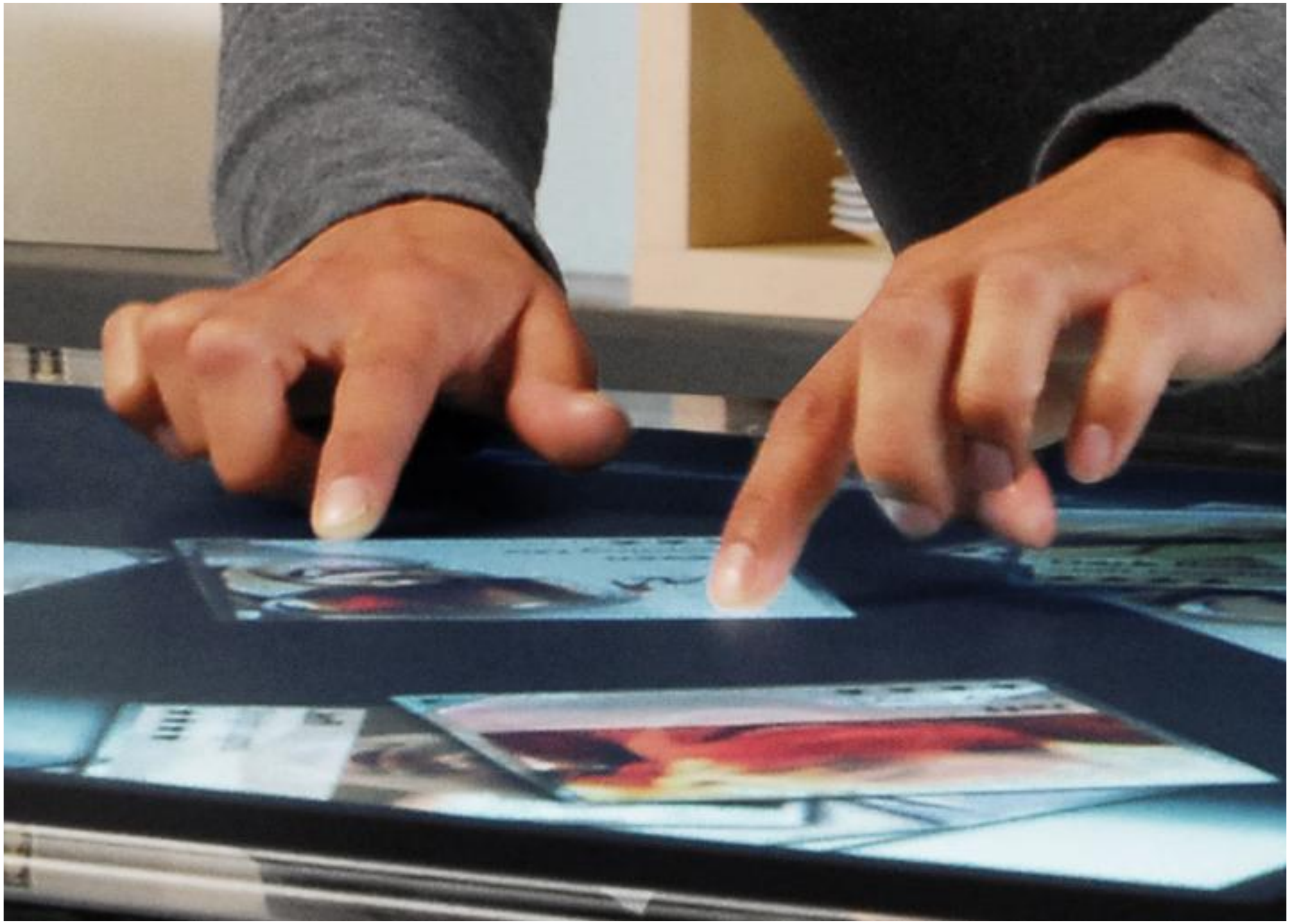
Example 3: OS X Time Machine

An early example of visualizing time-varying data



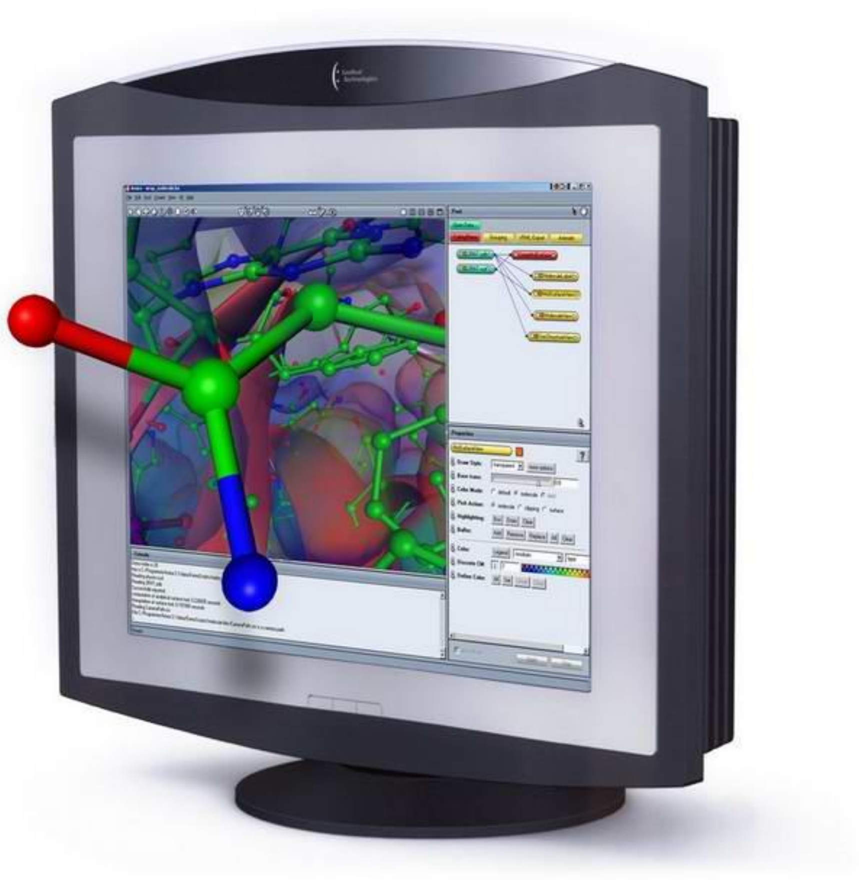
Example 4: Multi-touch interfaces

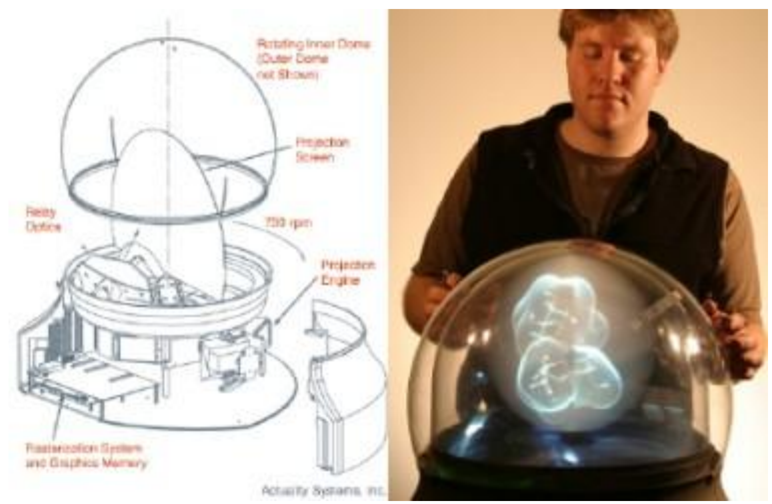
- http://www.youtube.com/watch?v=ysEVYwa-vHM



Example 5: Stereoscopic displays (VR)

- May force us to rethink the desktop metaphor





What should the programming model support?

- 2.5D and/or 3D
  - web page = logical structure + script-produced 3D view
    - What will a 3D nytimes.com look like? The 3D will ease browsing.
  - Q: how to project a part of 3D scene for 2D viewing/reading?
- Animation with physical properties (both GUI and games)
  - property changes over time, stated **declaratively**
  - **trajectories**: how to declare them?
  - **physical properties**: stretching, gravity, friction, but maybe also flow, fracture
- QoS:
  - latency specifications for GUI responsiveness
  - video frame rate, etc

Parallel Browser Scripting

Implicitly Parallel Web Apps

or

Web Designers Don’t Do Semaphores

[Strawman]

Ideal Parallelization

Plugins: Independent video playback

Scripts: Internal component animations

- Resizing of movies
- Fish eye menu in video

Layout: Resizing of table based off all movies

Is Parallelism Exposed Today?

Plugins: Independent video playback

YES, but annotation must be trusted

Scripts: Internal component animations

– Resizing of movies

MAYBE, with loop dependence analysis

– Fish eye menu in video

NO, pointer alias analysis

Layout: Resizing of table based off all movies

– **MAYBE**, with optimistic concurrency

Goals for **Parallel** Web Language

Implicit Parallelism:

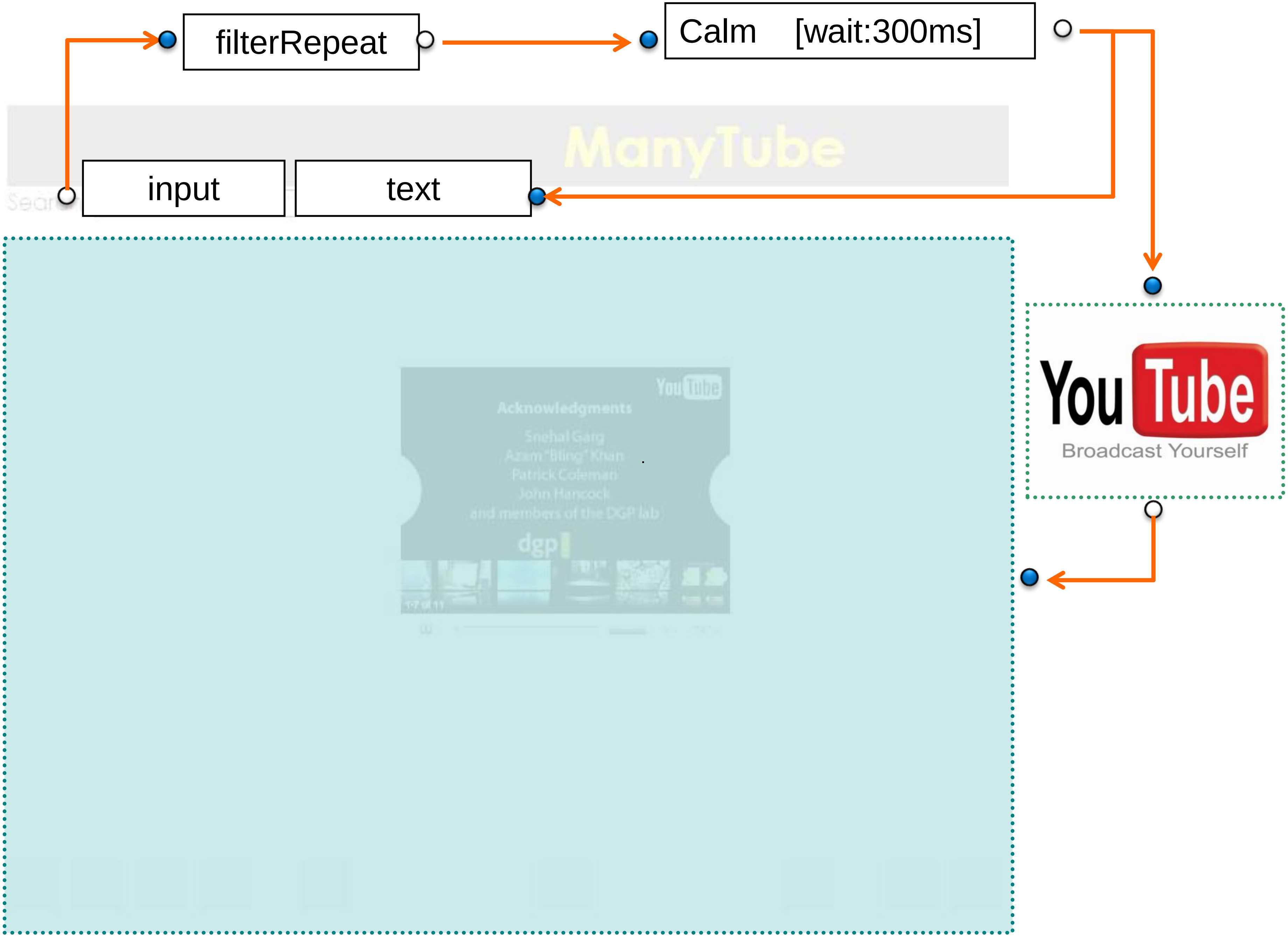
- sequential reasoning, but expose parallelism

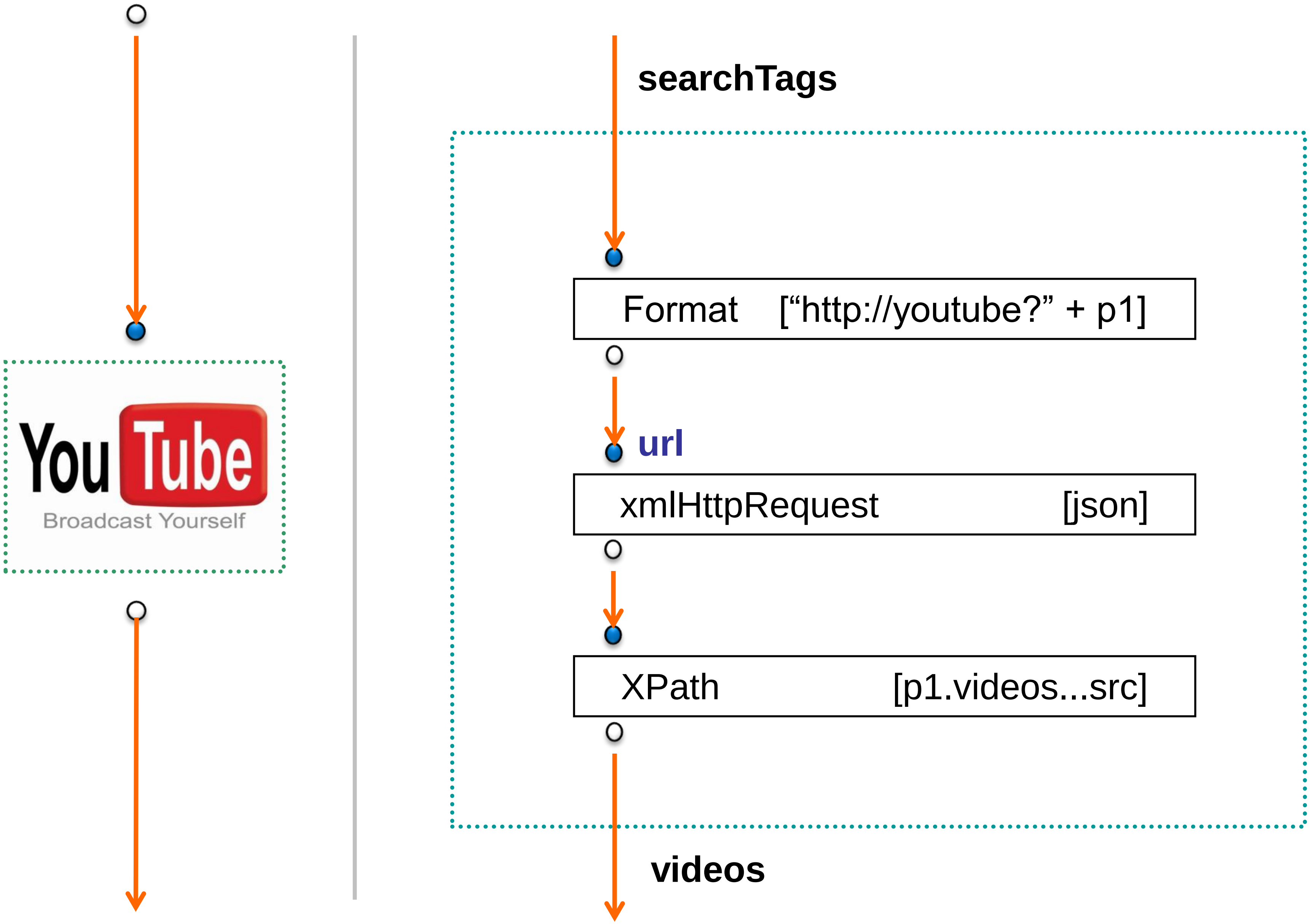
Abstractions for Web Apps:

- abstractions over time for animation
- abstractions for writing asynchronous code

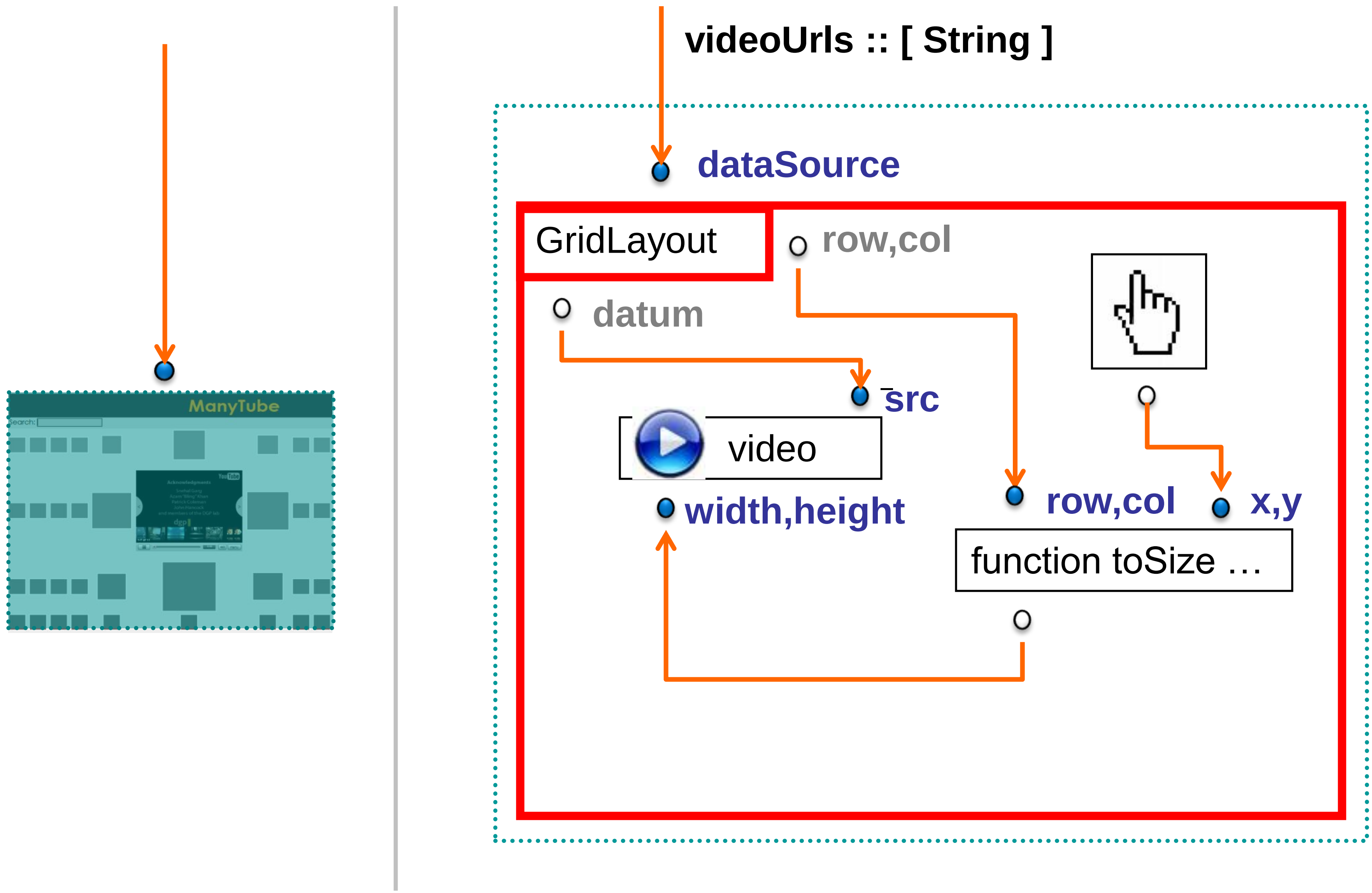
Declarative QoS:

- ex: grid is smooth, videos quality proportional to size


filterRepeat
Calm [wait:300ms]
ManyTube
input
text
YouTube
Broadcast Yourself



YouTube
Broadcast Yourself
searchTags
Format [“http://youtube?” + p1]
url
xmlHttpRequest [json]
XPath [p1.videos...src]
videos


ManyTube
videoUrls :: [String]
dataSource
GridLayout
row,col
datum
src
video
width,height
row,col
x,y
function toSize …

Benefits

Expressive
- asynchronous flows clearly connected
- rich yet static enough to be visualized
  - animation, tangible values
- composition

Implicit Structure Aiding Performance
- parallelization: state, if any, localized to node
- DOM writes: single write stream!
- scheduling:

Other Concerns (another day)

- Data
  prefetching, sharing, consistency
- Security
  policies, capabilities, delegation, anonymity (e-cash)
- Adoption
  standards, virtual machines, ES4
- Sequential Optimization
  types, partial evaluation, runtime tricks

Inspiration

Flapjax (flapjax-lang.org)

functional reactive programming (more dynamic, text based, compiled in JS)

Max/MSP

data flow system for live music synthesis & manipulation

More event & web languages

Flex, ES4, FrTime, LabVIEW, Esterel, …

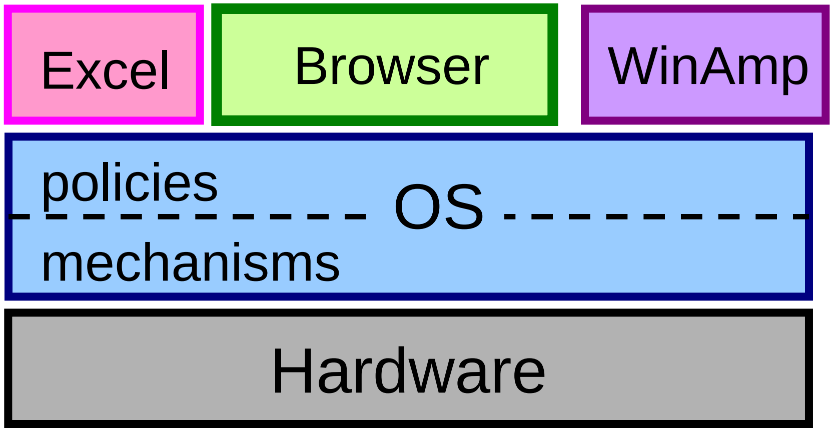
Mitosys: many-core OS

Current Platforms

OS is in charge of

1. resource mechanism (to securely multiplex apps onto resources)
2. resource policies (How to use resources –
   When to run threads and which to run together,
   which pages swapped to disk)

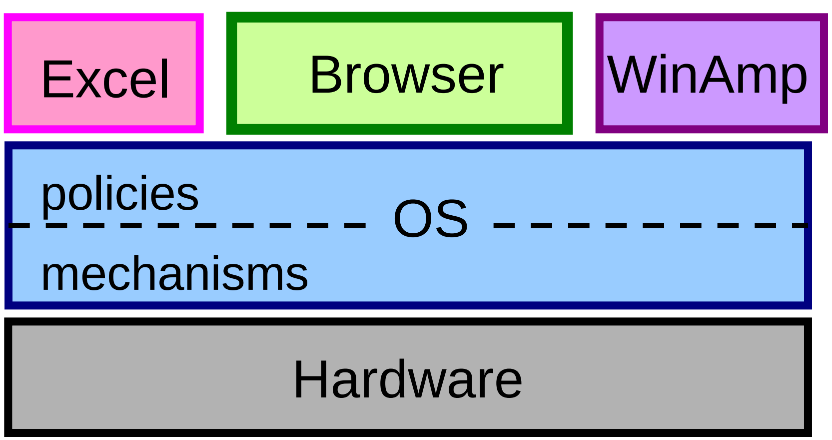
→ Monolithic OS is large and complex



Currently, Apps have limited control…

How can a browser easily specify and obtain the following?

- Browser wants 30 % of cpus regardless of what dvd ripper/virus scanner does.
- Browser wants some threads to be scheduled regularly (eg. Mouse event, decoder – run every frame)
- Browser wants threads to be always scheduled on same cores to find data in caches

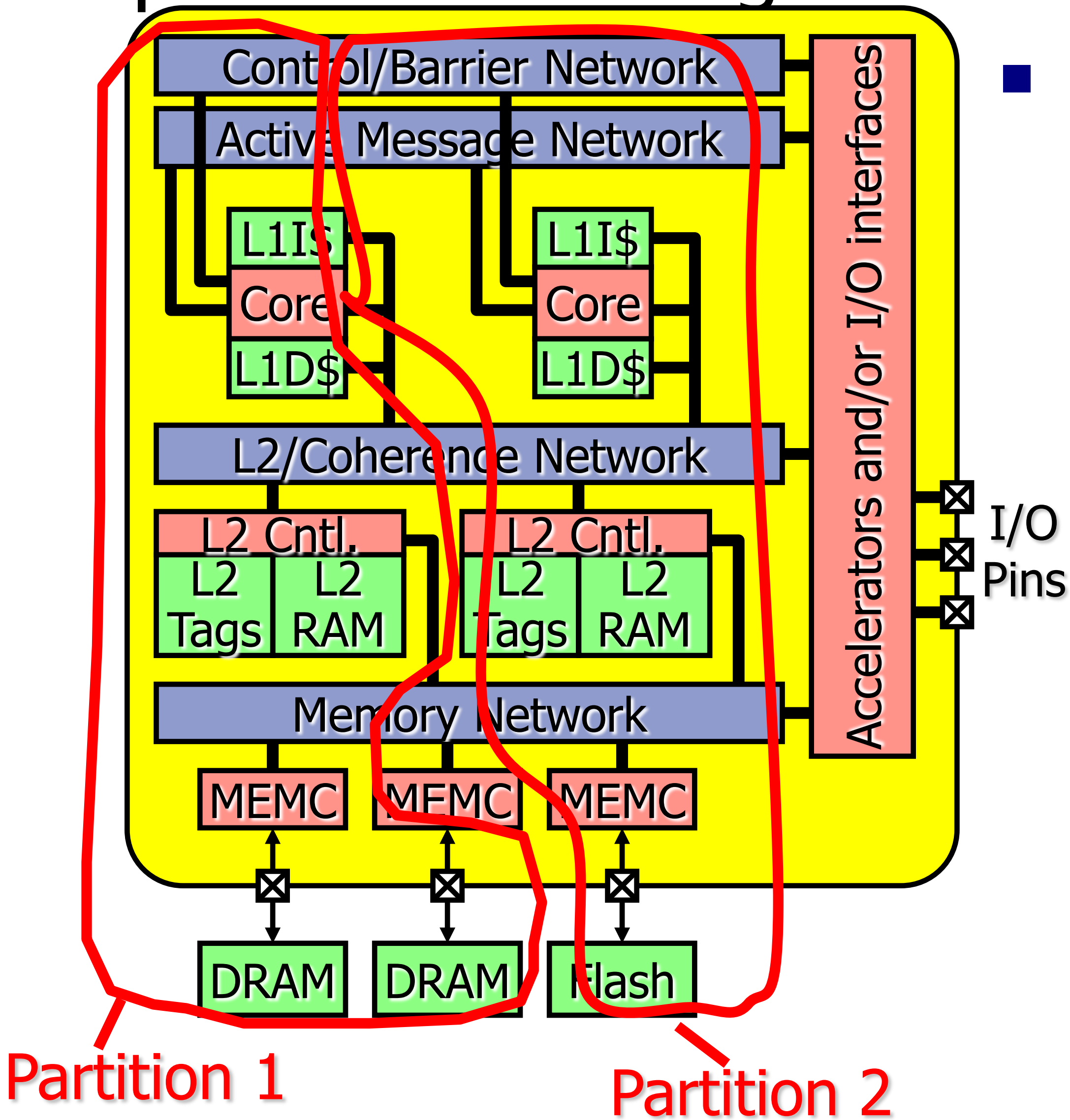


We can modify existing OS to do this but its messy….

Let's re-think the OS architecture

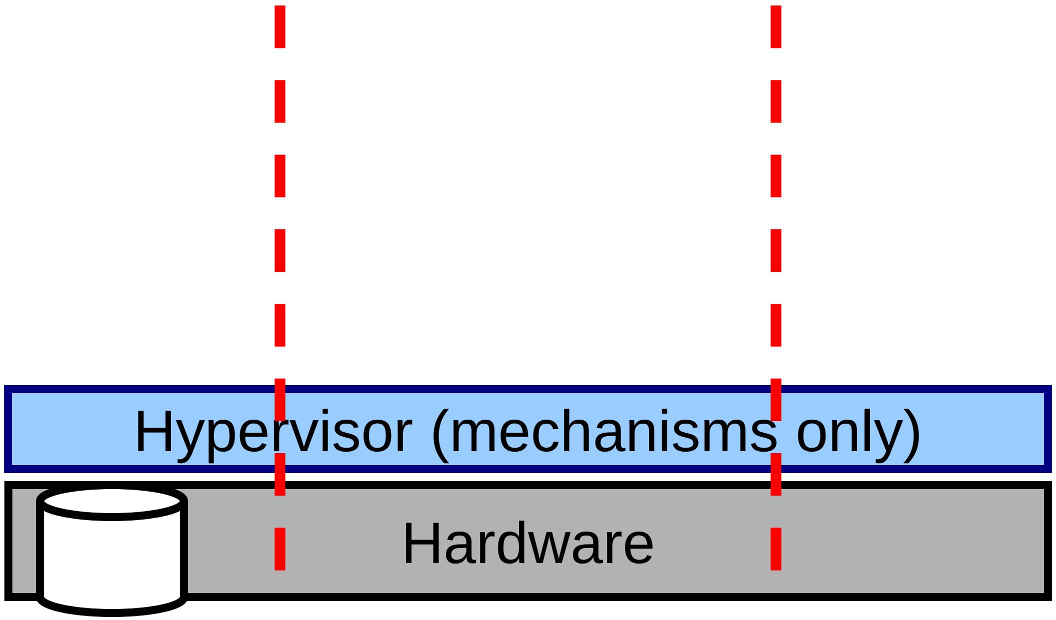
- flexible policies to suite app needs
- simplify OS to improve security
- scale OS for multicores

Recall: Future HW supports Spatial Partitioning

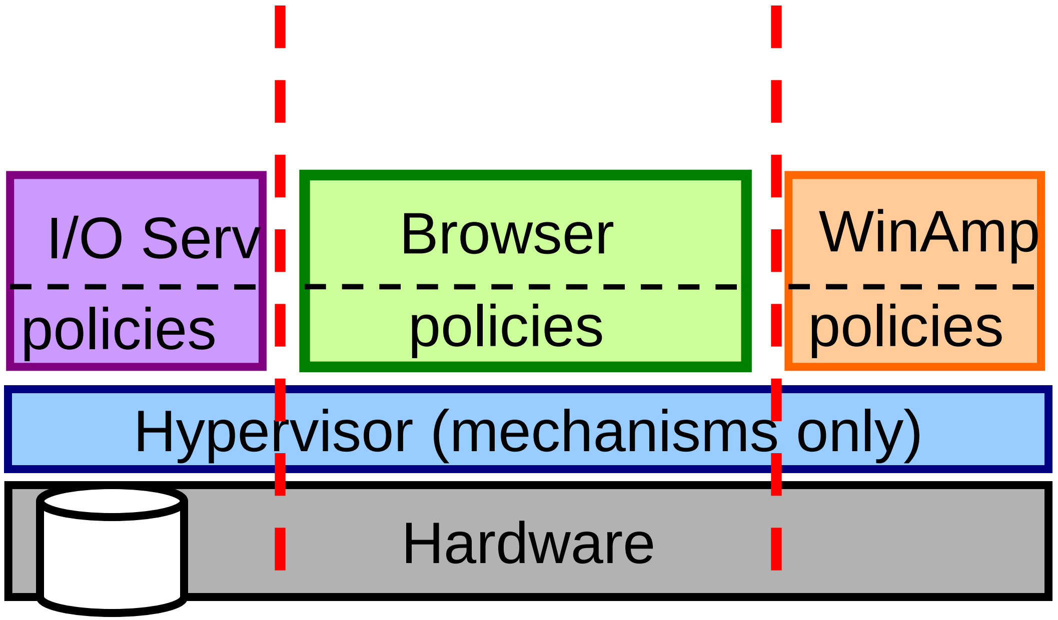


- Software specifies how resources are partitioned
  - Each resource can be partitioned independently of others
  - Partition allocation can be changed without re-starting app

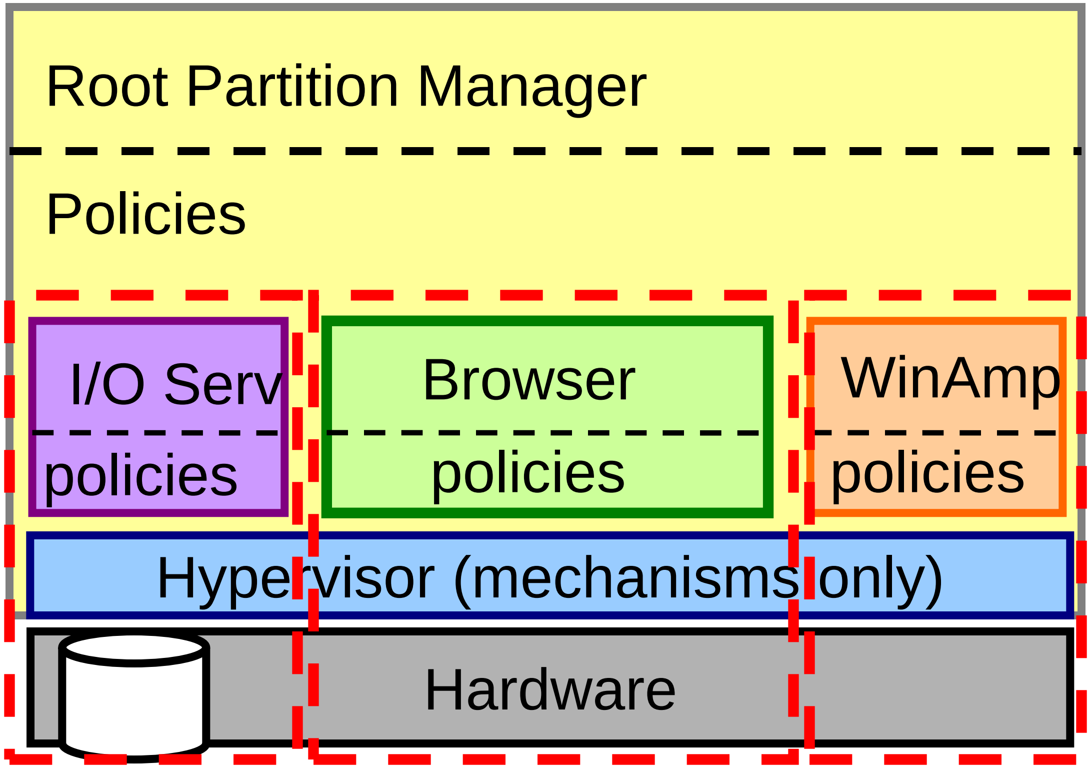
ParLab OS Architecture



ParLab OS Architecture

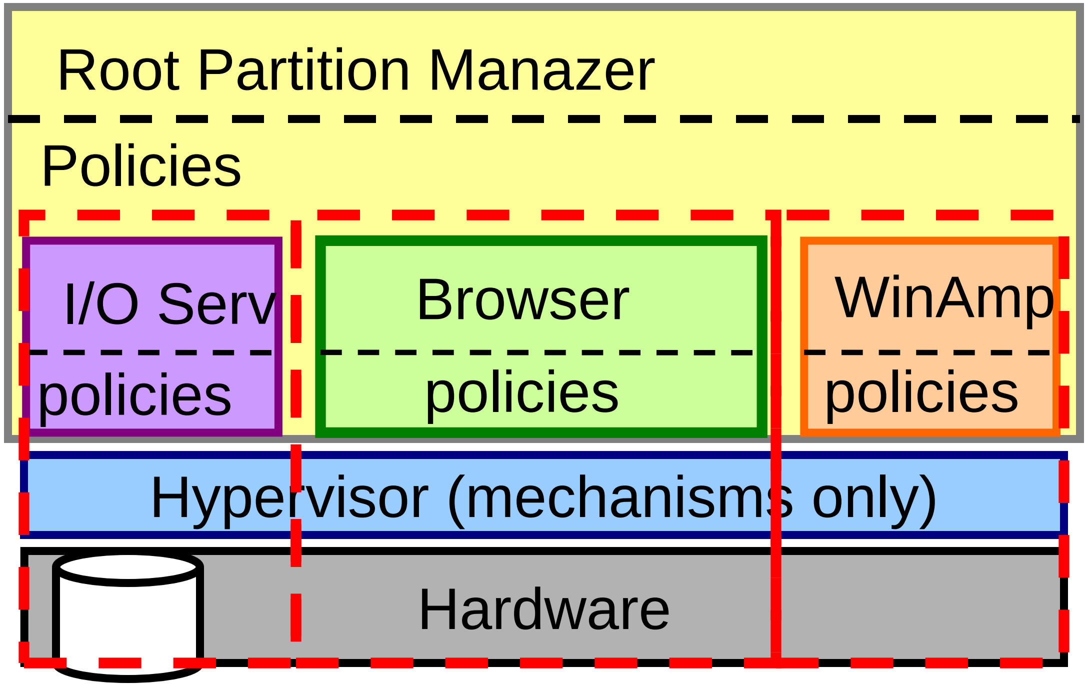


ParLab OS Architecture



ParLab OS Architecture

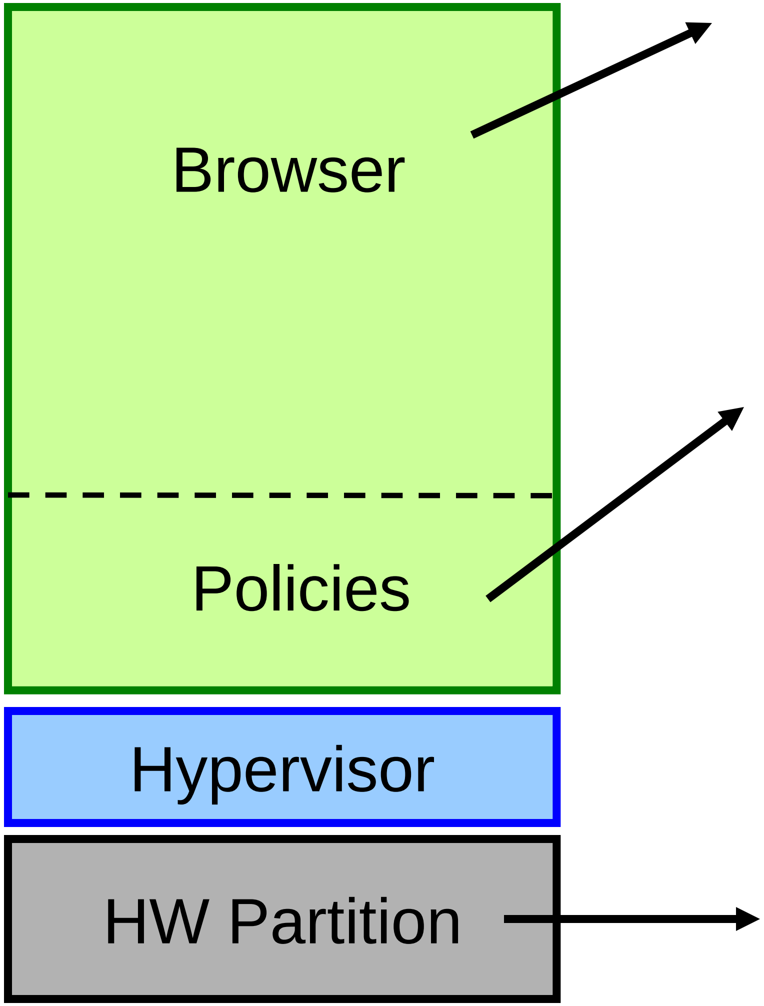
Applications implement policies on resource management and usage



Partitioning provides Quality of Service (QoS) Guarantees for App

1. Capacity (how much)
2. Latency (when)
3. Throughput (bandwidth)

Inside a partition



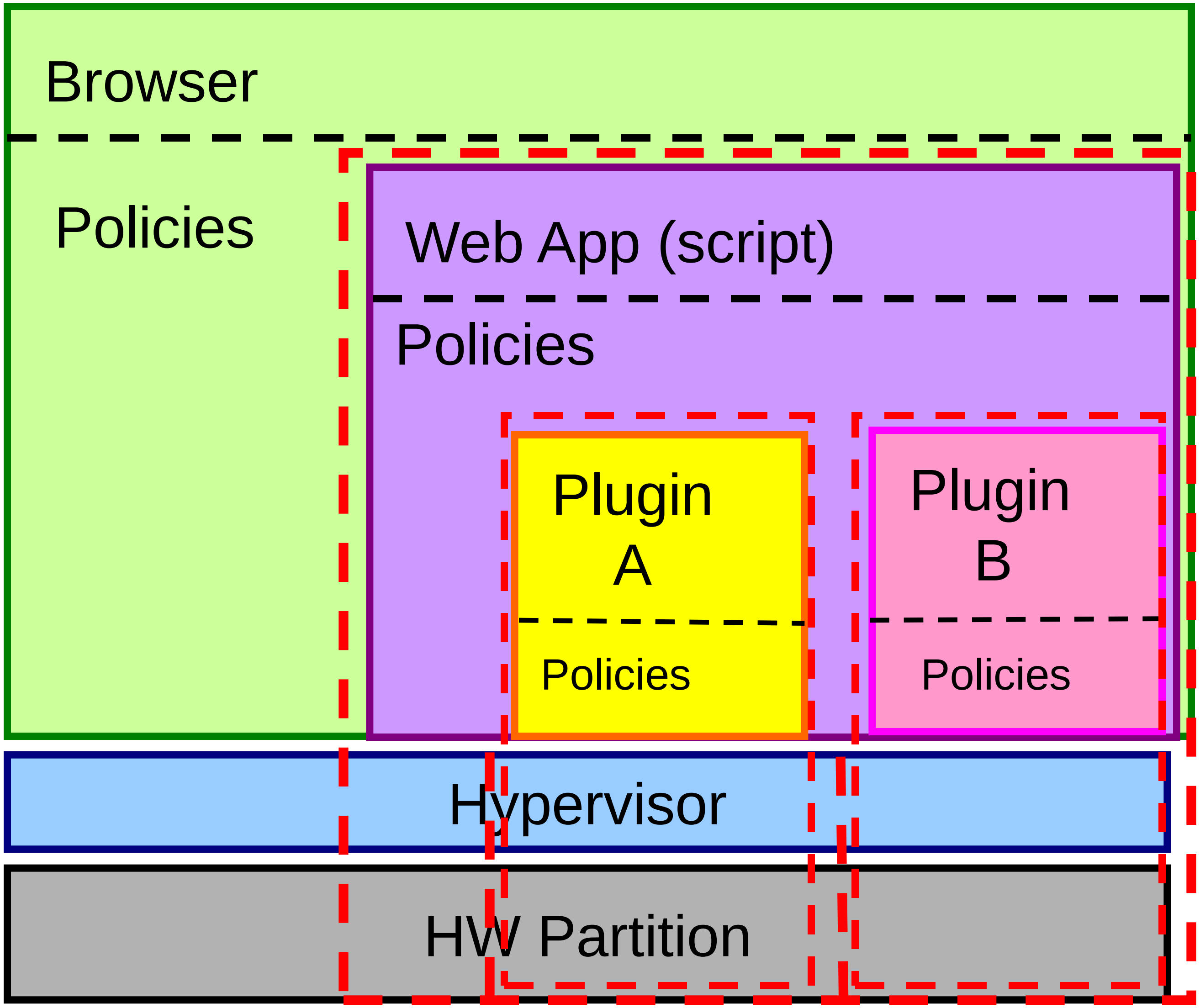
Bare-metal execution provides optimized and predictable performance

Domain Specific Resource Management Libraries:

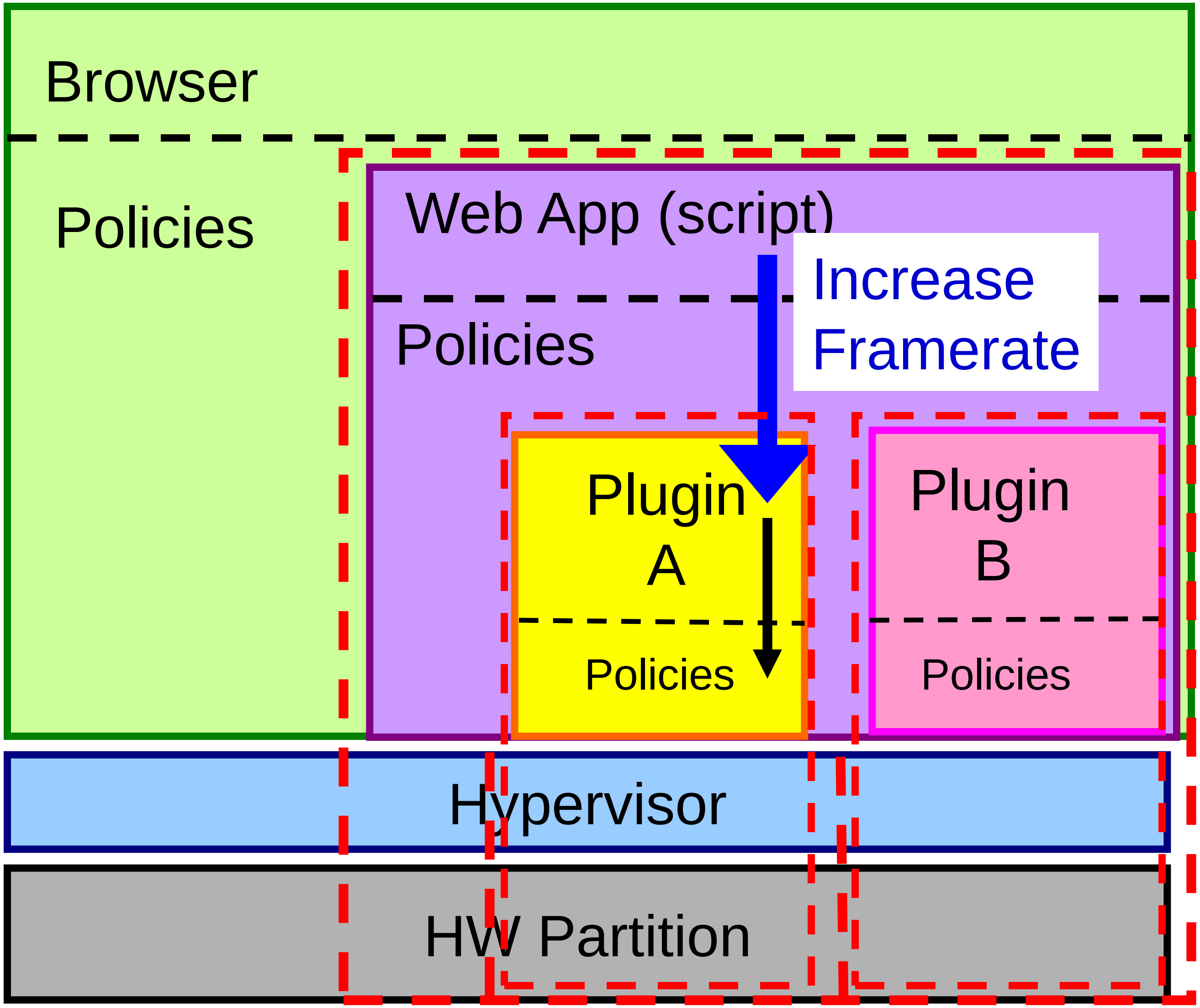
- Thread management
- Memory management
    - virtual-phys mapping
    - swapping pages to disk

Partition - Cores, Memory, Memory bandwidth allocation

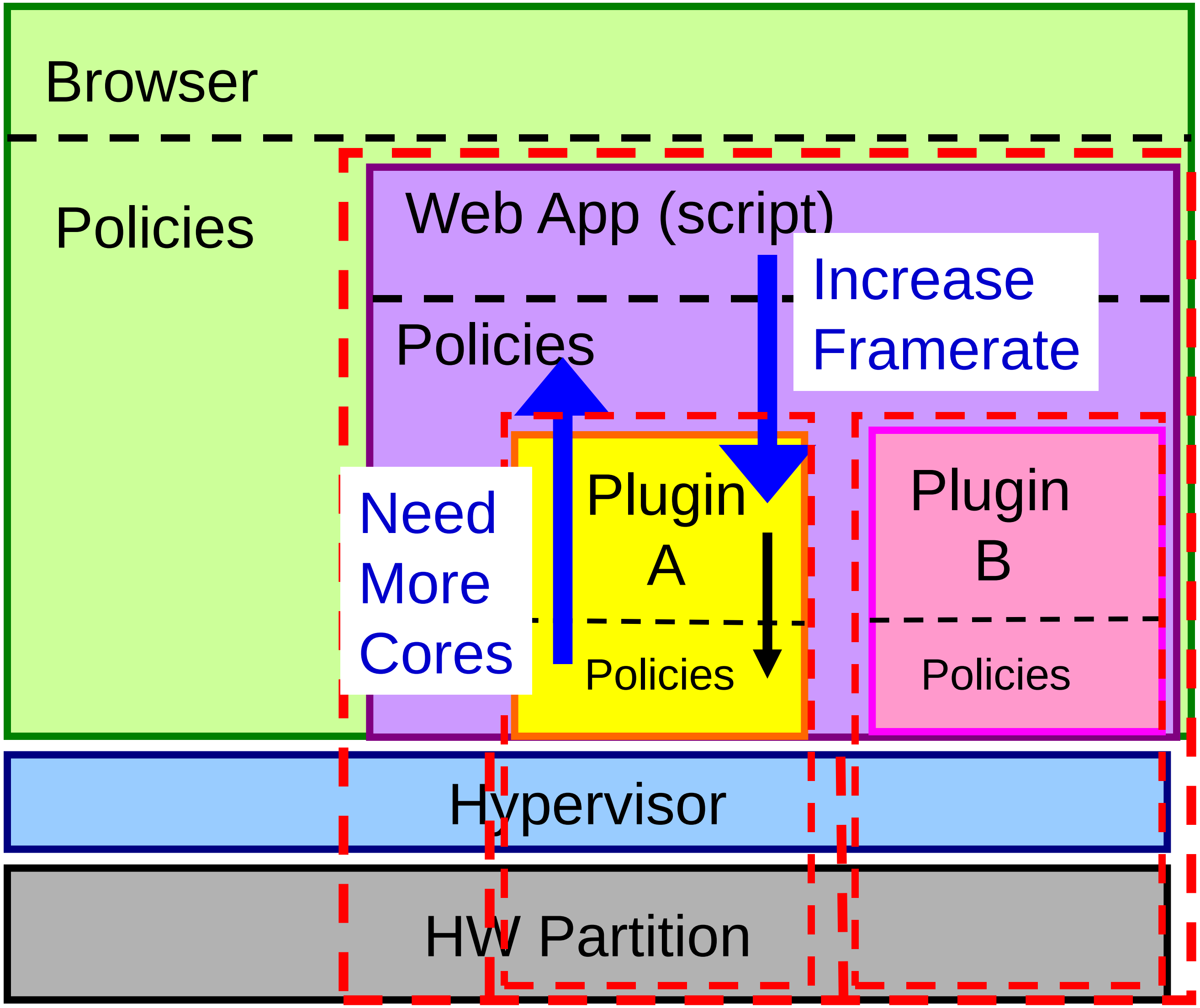
Support for hierarchical partitions



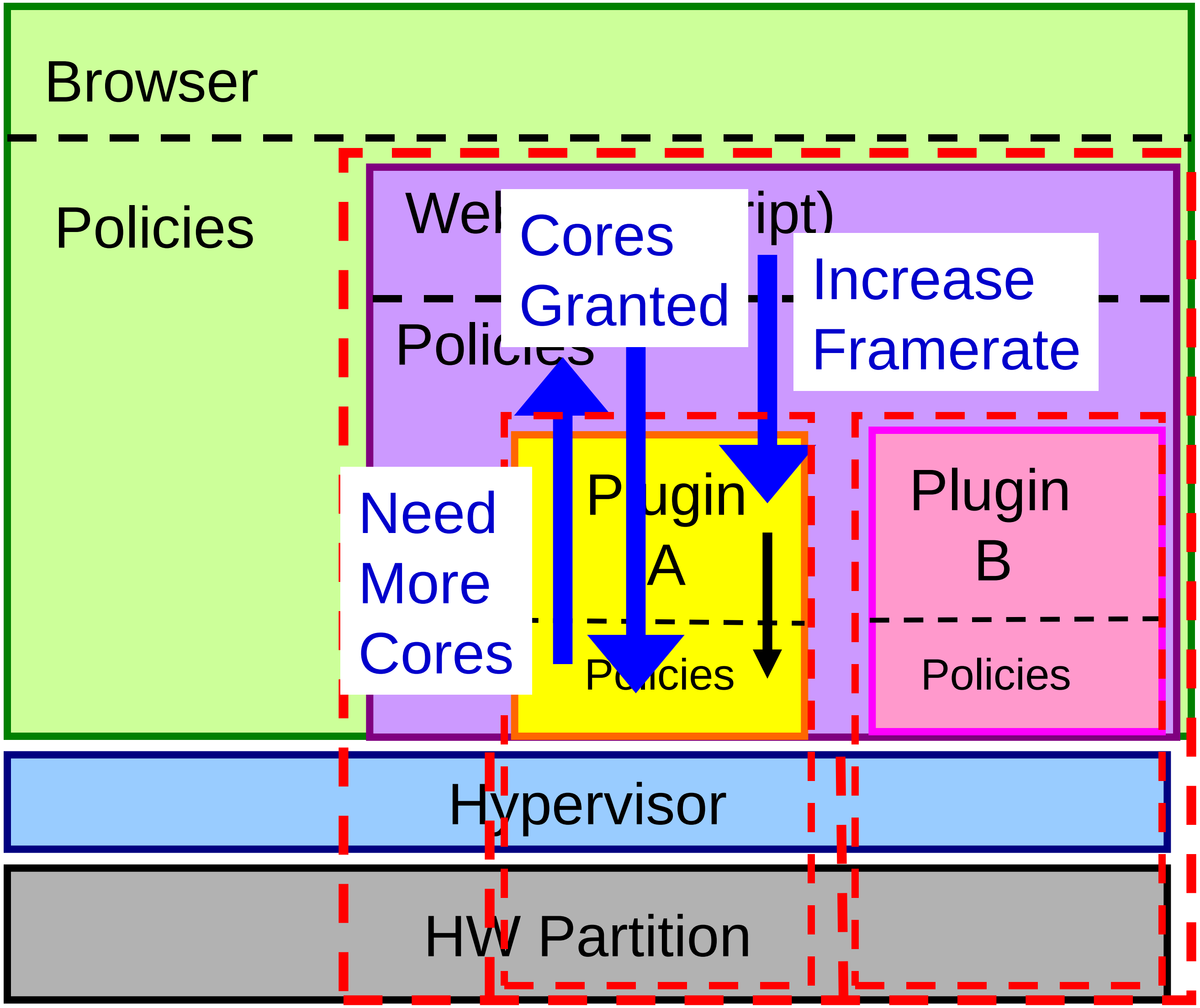
Support for hierarchical partitions



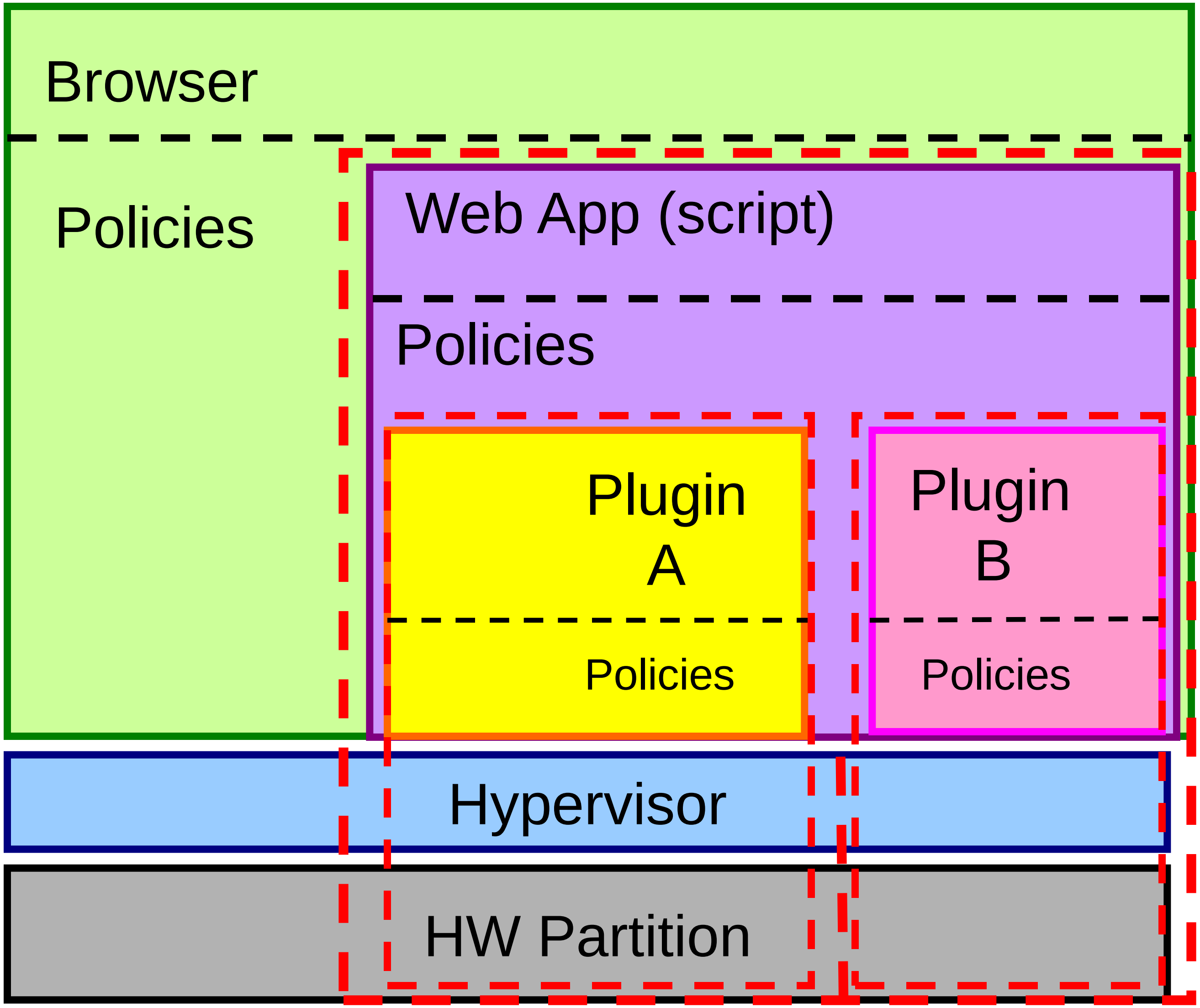
Support for hierarchical partitions



Support for hierarchical partitions



Support for hierarchical partitions



OS Takeaways

- Partitioning brings opportunities
  - Better QoS guarantees on resources
  - Better isolation/protection/security – codec crashes but web page Ok.
  - Simplifies hypervisor → fewer bugs, more secure
- Application will have better control over resource management and usage →supported by domain specific resource management libraries
- New communication mechanisms
  - Between partitions
  - Across cores within partition
  - Synchronization mechanisms