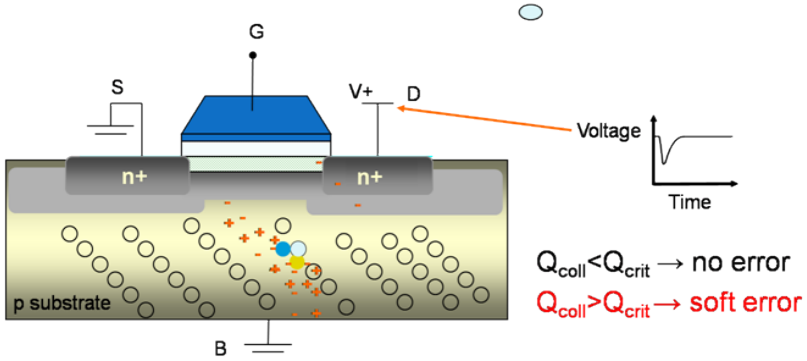
“Good Enough” Systems: Tolerating (most) Hardware Errors in Software

Karthik Pattabiraman
University of British Columbia (UBC)

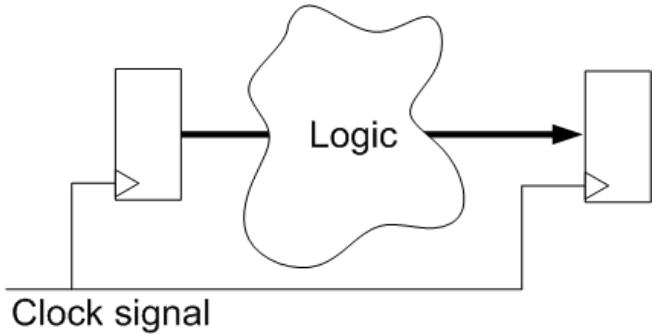
Joint work with Jiesheng Wei (UBC), Song Liu (Northwestern), Thomas Moscibroda (MSR), and Benjamin Zorn (MSR)

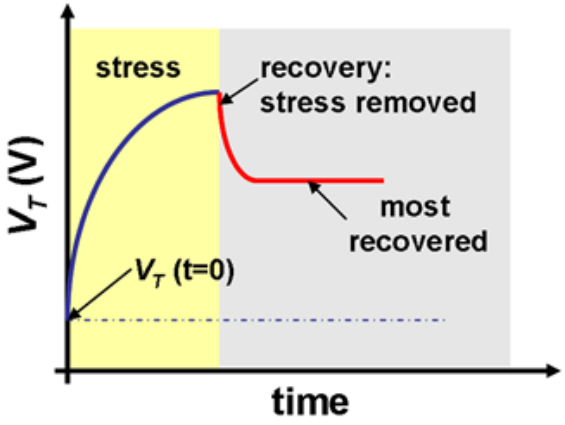
Motivation: Hardware Errors

Soft-errors

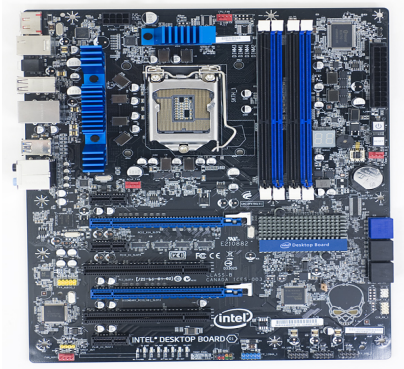


Timing errors





Device wear-outs (e.g., NBTI)

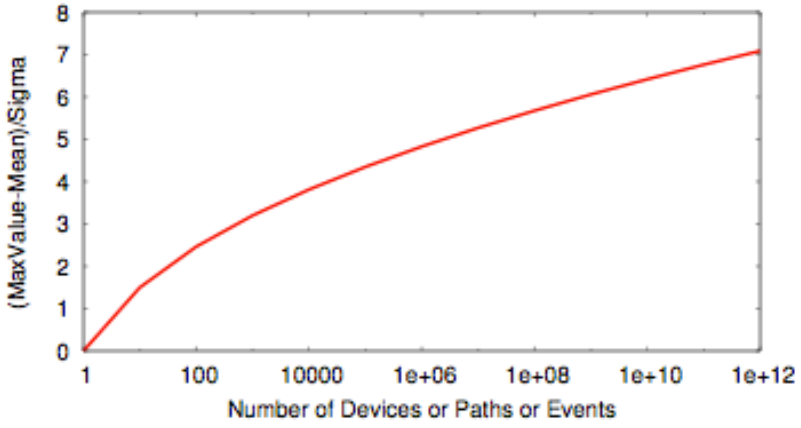


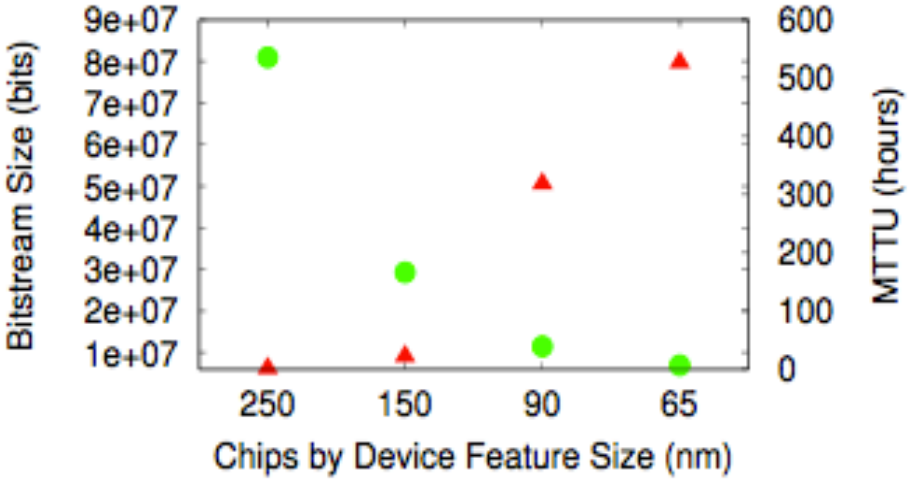
Intel Sandy Bridge chipset bug, 2011

Manufacturing/Design defects

Motivation: Variations and Errors

- Variation of device times
  - Higher spread of device variations for future generations of technology
- Feature size Vs MTTU
  - Increase in number of bits correlated with decrease in MTTU of the chip





Source (CCC study on cross-layer reliability): www.relxlayer.org (March 2011)

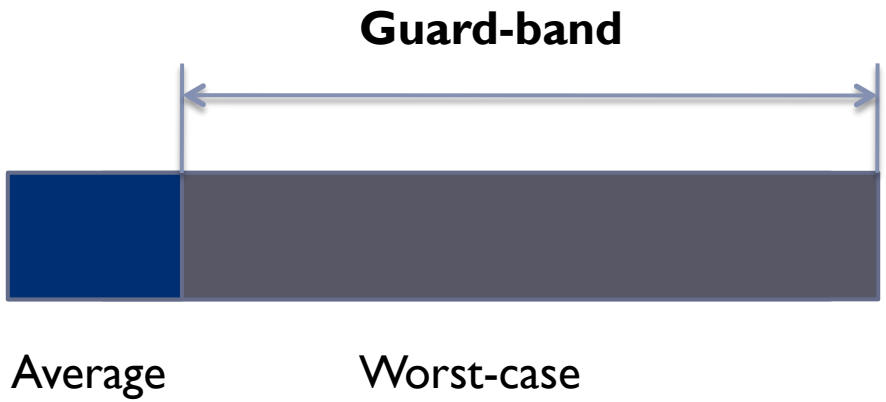
Hardware Errors: Traditional "Solutions"

- **Guard-banding**

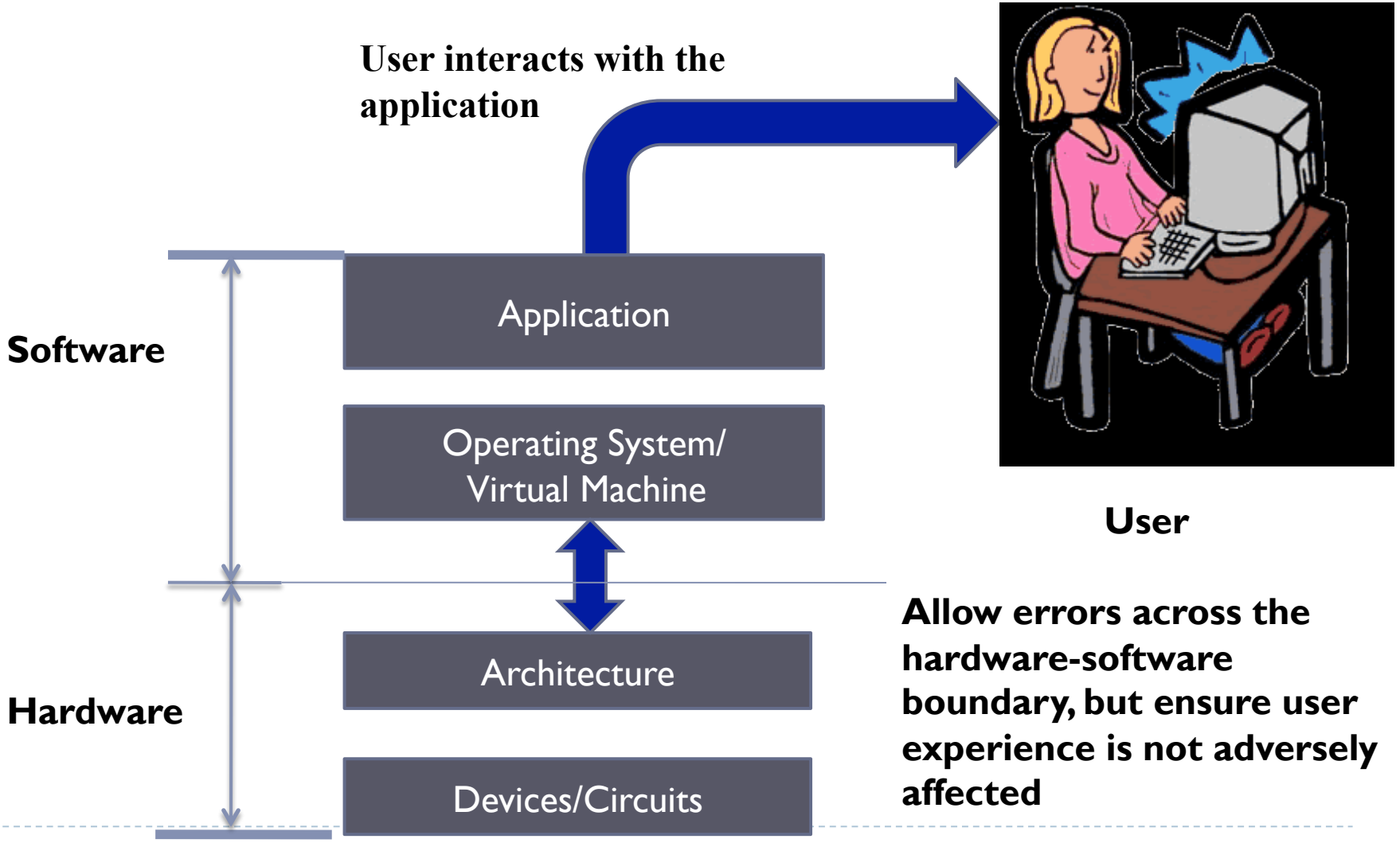
  Guard-banding wastes power and performance as gap between average and worst-case widens due to variations

- **Duplication**

  Hardware duplication (DMR) can result in 2X slowdown and/or energy consumption

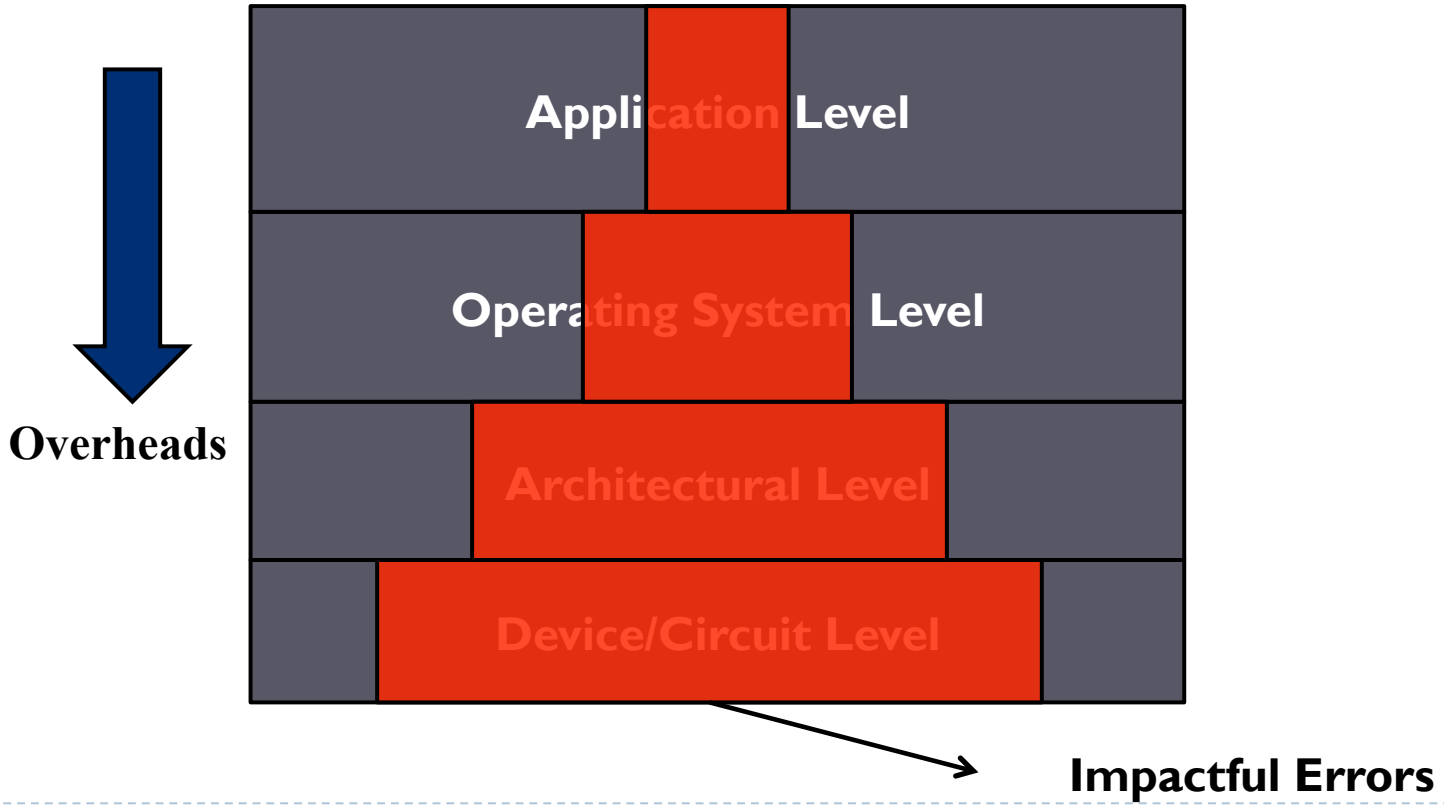


Our approach



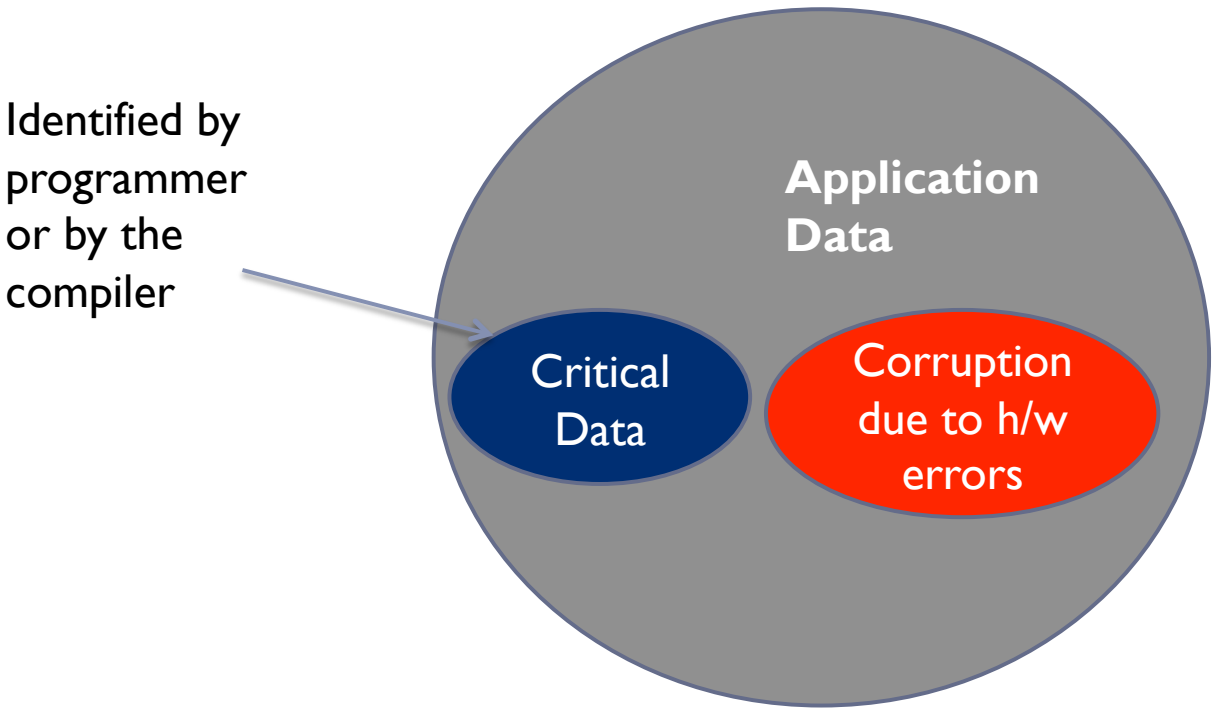
Why Software ?

Errors get progressively filtered as we go up the system stack



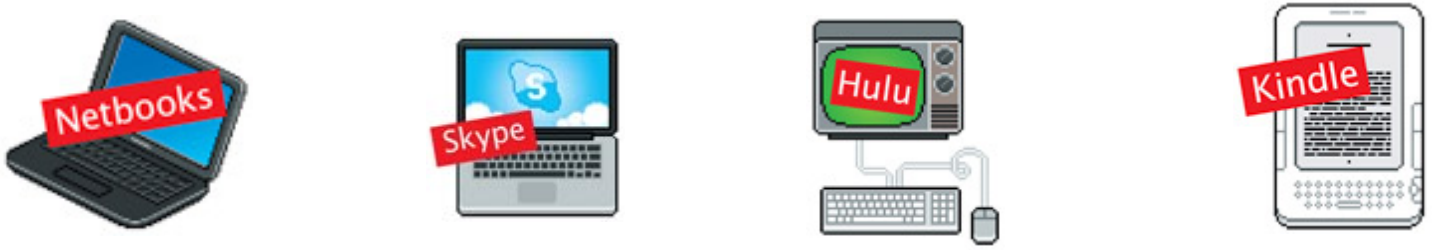
Critical Data

- **Software has high-level redundancy in data**
  - Can tolerate limited amounts of data corruption
  - Provided certain critical data is not corrupted



The "Good Enough" Revolution

Source: WIRED Magazine (Sep 2009) – Robert Kapps
http://www.wired.com/gadgets/miscellaneous/magazine/17-09/ff_goodenough



People prefer "cheap and good-enough" over "costly and near-perfect"

“Good Enough” Computer Systems

- **Just reliable enough to get the job done**
  - Do not provide the illusion of perfection to end user
  - But do not fail catastrophically or cause severe errors
  - Depends on the application and its context of use

Talk Outline

- Motivation and Approach

- Good Enough Software Systems
  - Flikker [ASPLOS 2011] with S.Liu, T. Moscibroda and B. Zorn
  - BlockWatch [submitted] with J. Wei

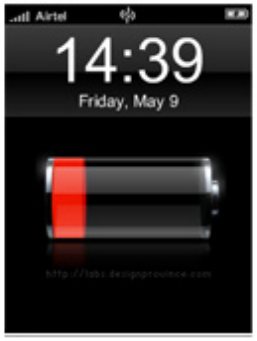
- Future Work and Conclusions

Flikker: Smartphones

Smartphones becoming ubiquitous

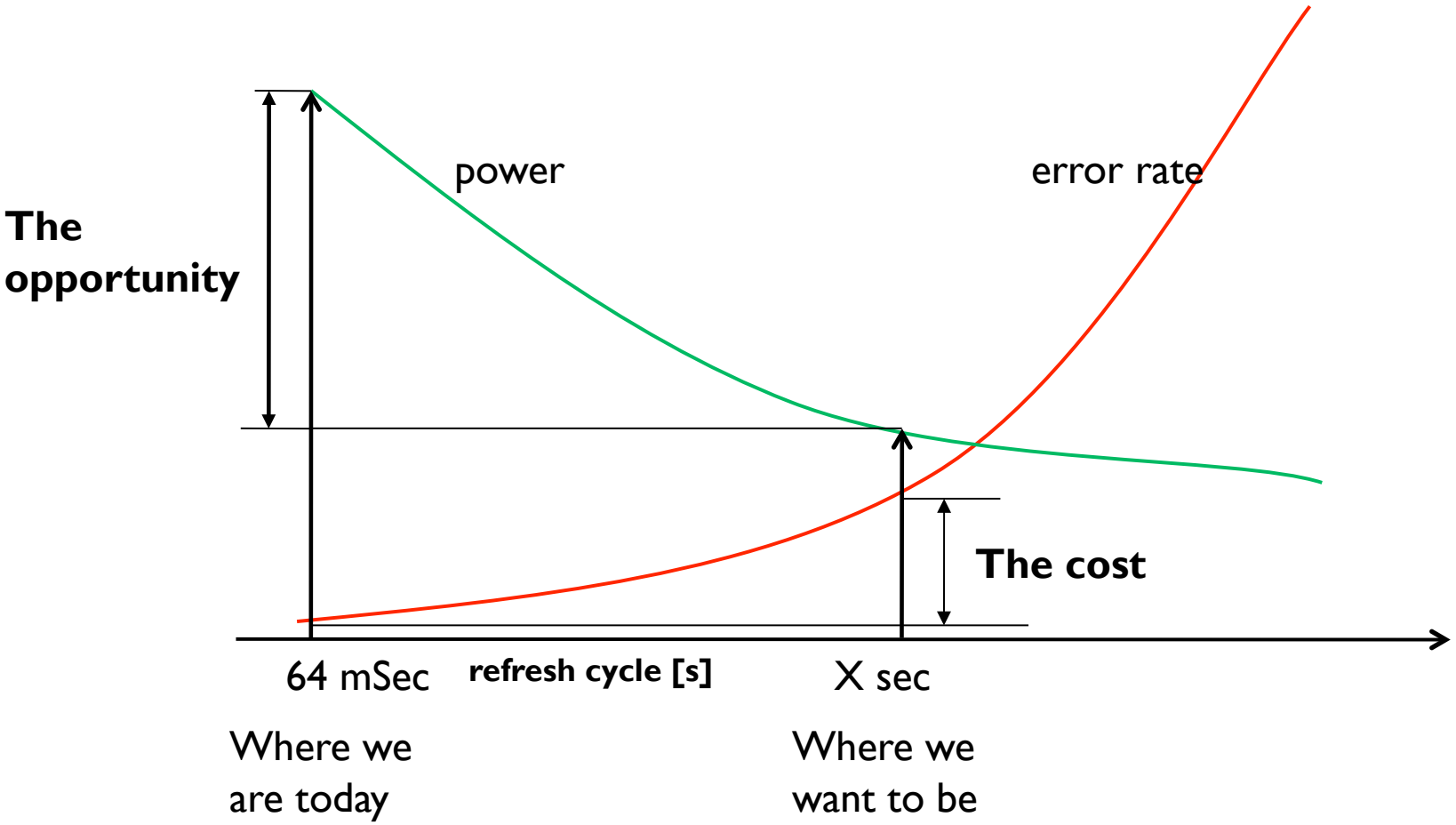
Responsiveness is important

DRAM Memory consumes up to 30% of power



Can drain the battery even when idle

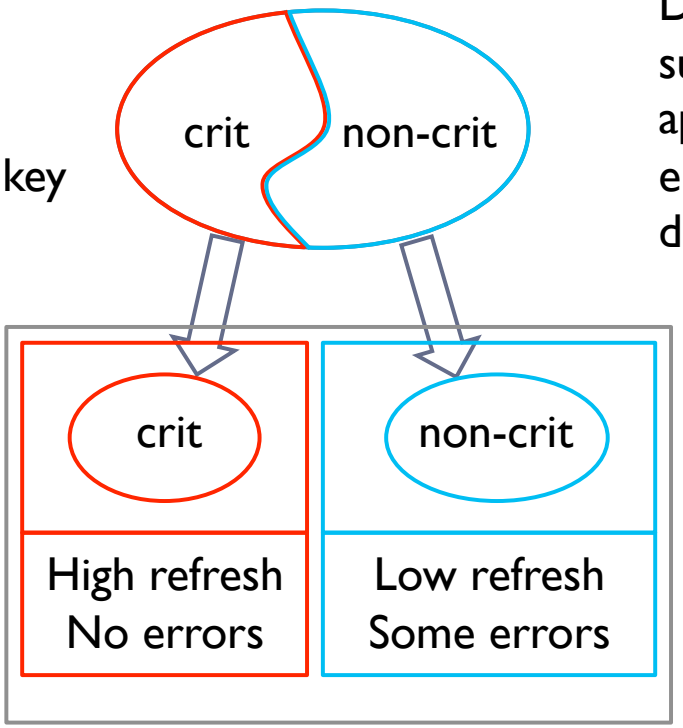
Flikker: DRAM Refresh



Flikker: Approach

- **Critical / non-critical data partitioning**

Important for application correctness e.g., meta-data, key data structures

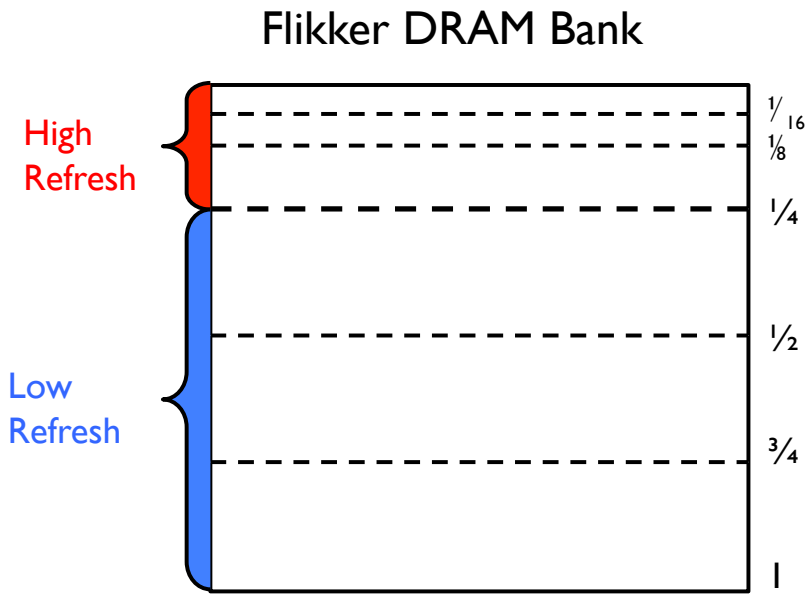


Does not substantially impact app correctness e.g., multimedia data, soft state

Flikker DRAM

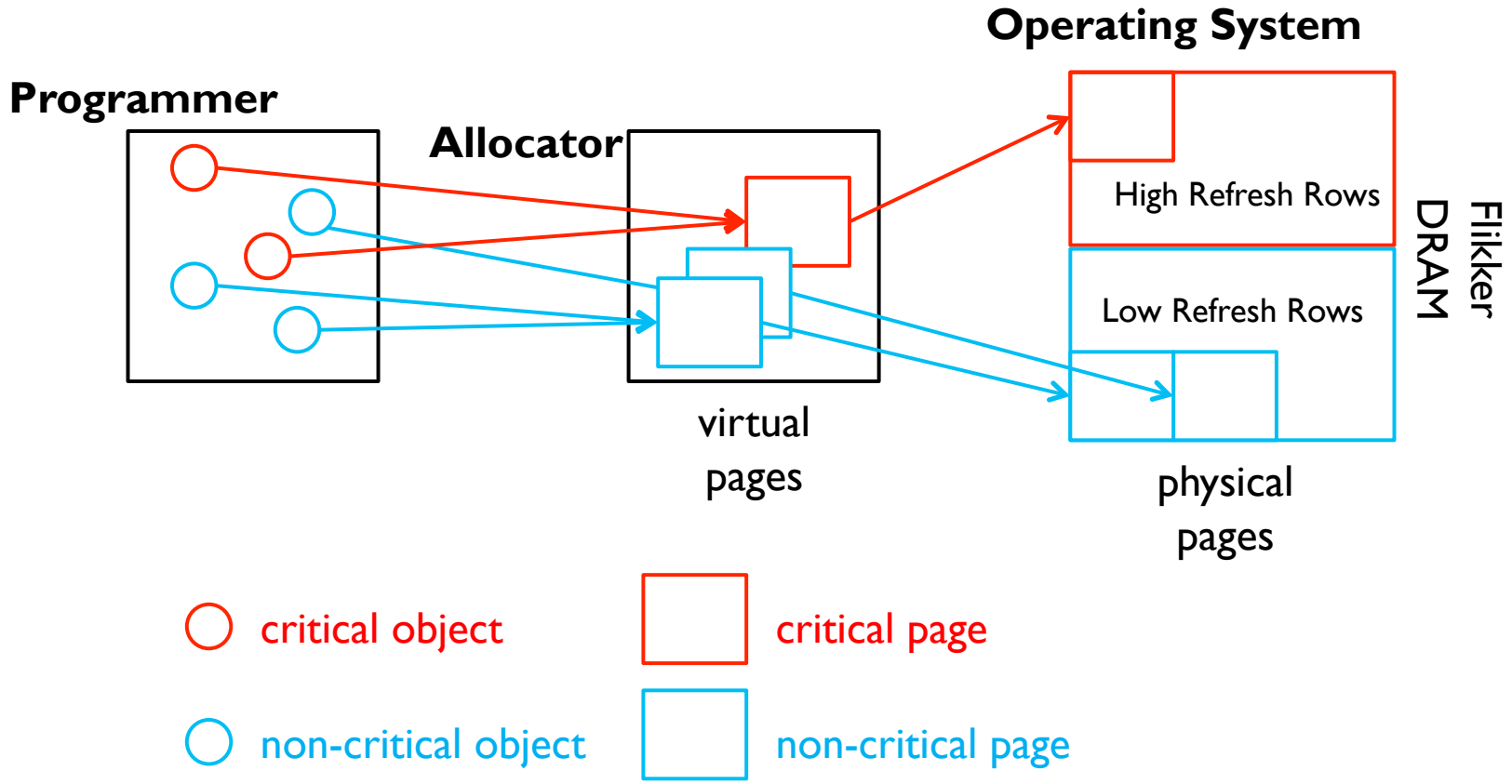
Flikker: Hardware Implementation

- Divide memory bank into high refresh part and low refresh parts
- Size of high-refresh portion can be configured at runtime
- Small modification of the Partial Array Self-Refresh (PASR) mode



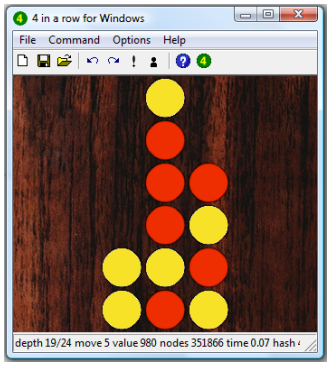
Flikker: Software Implementation

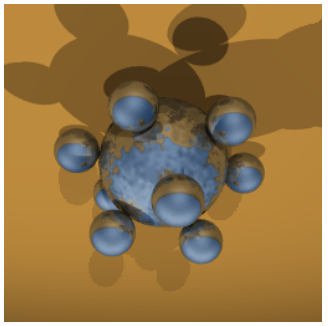
Minor changes to the memory allocator and the Operating System (OS)



Flikker: Mobile Applications

- mpeg2 (video decoding)
- c4 (connect 4, four-in-a-row)
- rayshade (ray-traced images)
- vpr (Stochastic optimization)
- parser (Natural-language processing)



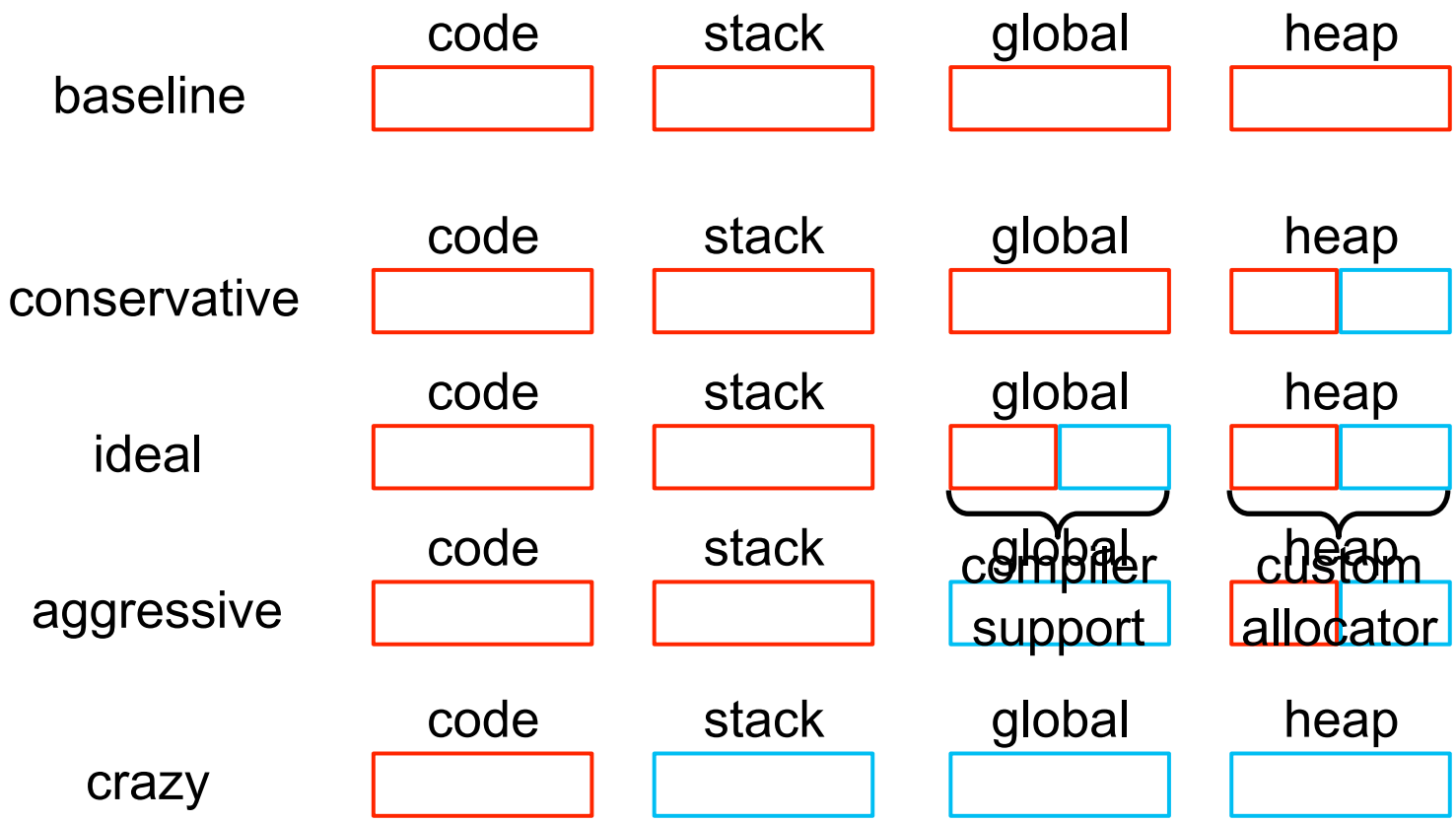


Application	No. of lines	Input	Metric
mpeg2	10.0k	mei16v2.m2v	output SNR
c4	6.1k	N/A	saved moves
rayshade	24.2k	balls.ray	output SNR
vpr	24.6k	ref/test	output file
parser	11.5k	ref/test	output file

Flikker: Experimental Setup

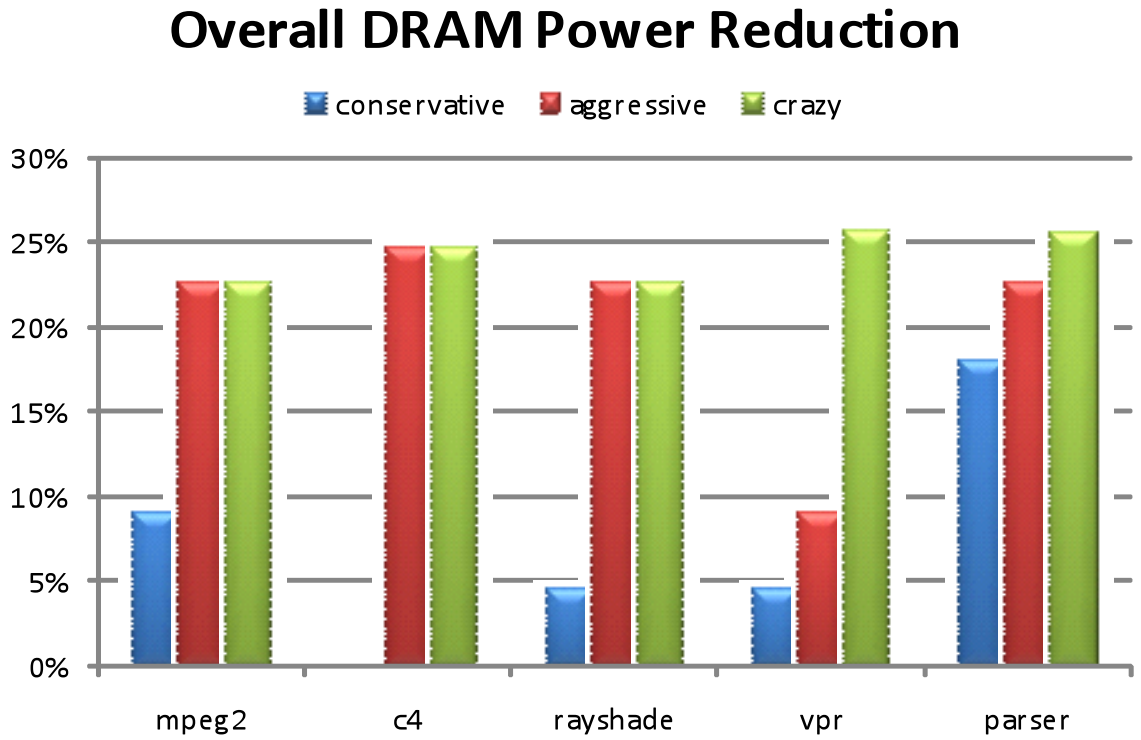
- Performance (architectural simulator)
  - Impact of data partitioning (loss of locality) < 0.5%
  - Took less than **one day** to partition each application
- Overall DRAM power (simulator, model)
  - Active power, Idle power
  - Usage profile (95% idle, 5% active) [Karlson'09]
- Fault injection simulation (Pin)
  - Simulate a self-refresh period, and inject errors corresponding to DRAM error model [Venkatesan-05]

Flikker: Configurations



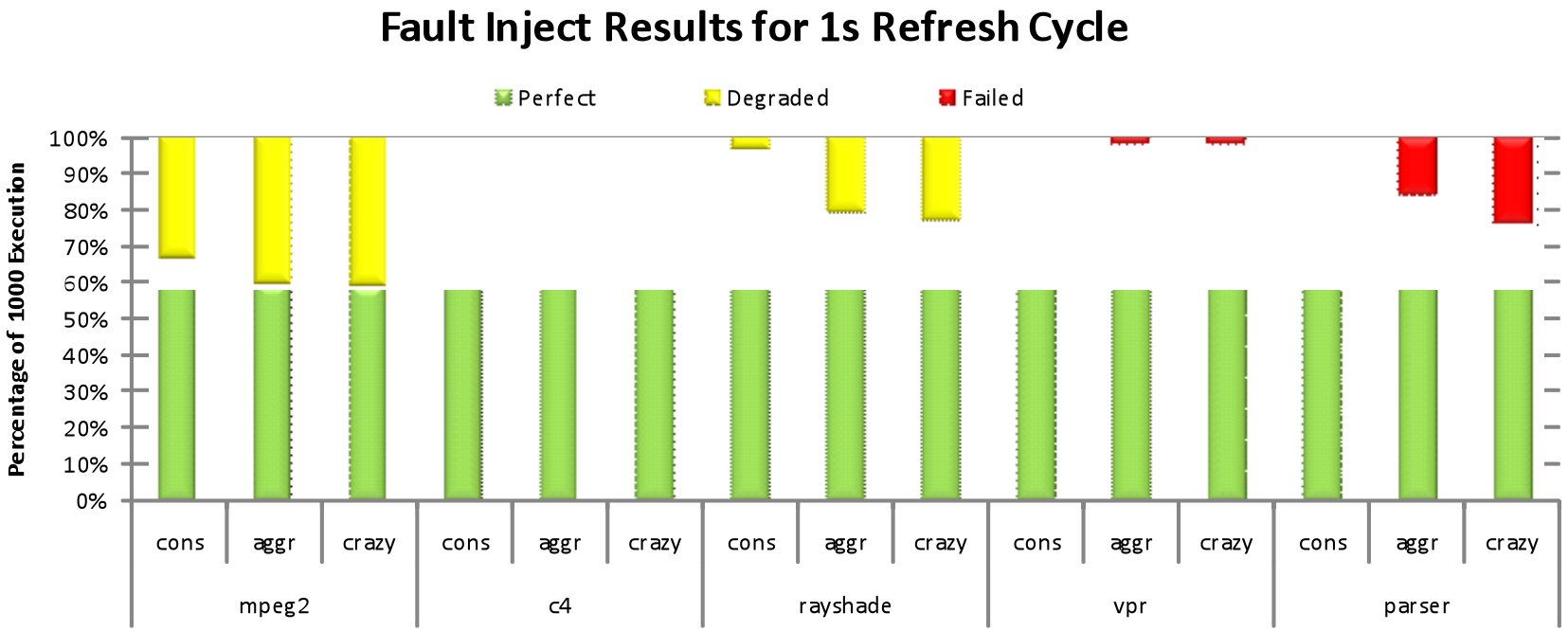
Flikker: Power Reduction Results

- Estimate the portion of high refresh part based on the percentage of critical pages in application
- Overall savings: 20 to 25% of memory power

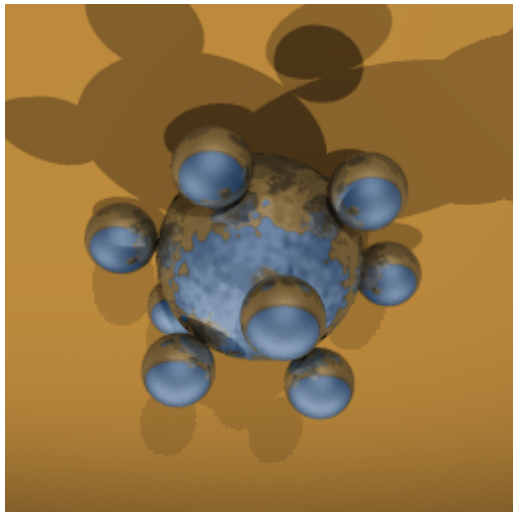


Flikker: Fault-injection Results

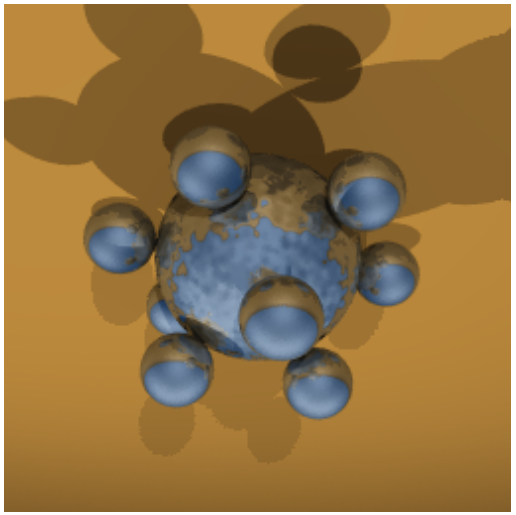
- c4: always perfect
- mpeg2, rayshade: some degraded output
- vpr, parser: some failed in aggressive and crazy



Flikker: Rayshade Degraded SNR



Original



Flikker - 78.9dB

2 X Zoom

Flikker: Summary

- **First software technique to intentionally lower hardware memory reliability for energy savings**
  - Minimal changes to hardware – based on PASR mode
  - Minor changes to applications to identify critical data
- Reduced the overall DRAM memory power by **20-25%** with negligible loss of reliability and performance
- **Future work:**
  - Extension to data center applications (e.g., Internet Search)
  - Extension to faulty processor components

Talk Outline

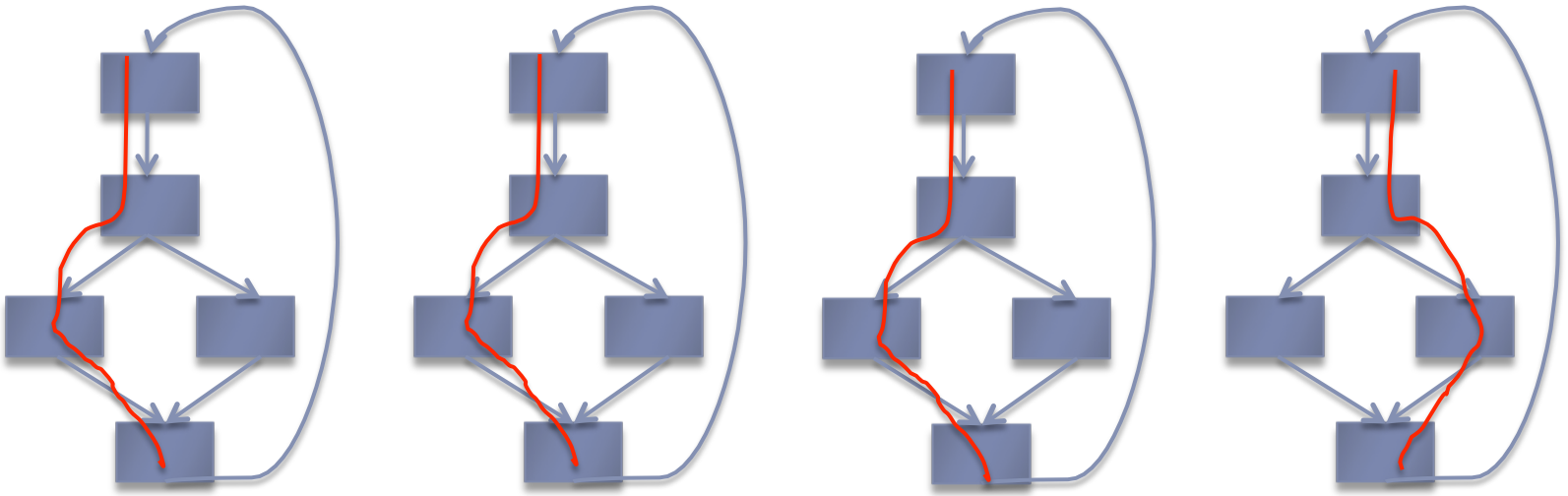
- Motivation and Approach

- Good Enough Software Systems
  - Flikker [ASPLOS 2011] with S.Liu, T. Moscibroda and B. Zorn
  - BlockWatch [submitted] with J. Wei

- Future Work and Conclusions

BlockWatch: Motivation

- Software will become more parallel (due to multi-cores)
- Can we leverage the parallel nature of software to provide error checking for free (or nearly free) ?
  - Idea: Exploit similarity in control data of parallel programs
  - Arises as a result of high-level models (e.g., SPMD)



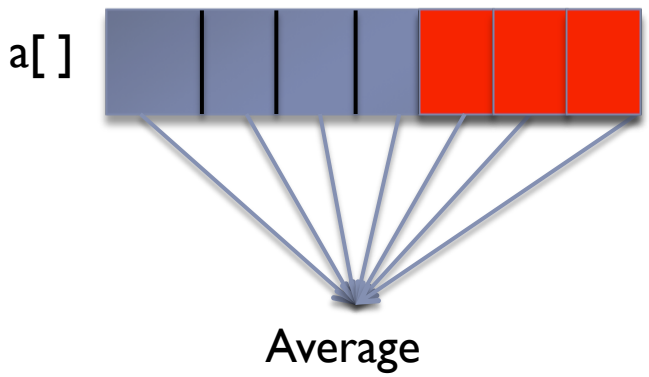
Error ?

BlockWatch: Why Control Data ?

- **Control-data**: Any data that influences a branch decision, i.e., backward slices of condition variables

- Errors in control-data are more likely to lead to egregious outputs and catastrophic failures [Thaker-IISWC-2006]

```
int findAverage(int a[], int n) {
        int sum = 0;

        for (int i = 0; i < n; ++i) {
                sum = sum + a[i];
        }
        return (sum / n);
}
```



BlockWatch: Approach

- **Identify patterns of control-data similarity in parallel programs**
- **Extract the similarity through static analysis**
  - No false-positives (and hence no spurious detection)
  - Insert instrumentation to check the similarity
- **Check similarity at runtime**
  - Monitor executed in a separate thread
  - In case of error, halt program and restart

BlockWatch: Example

```
long im = DEFAULT_N;
void slave() {
    int i, private, procID;
    //procid is the thread id
    if (procID == 0) {
            …
    }
    for (i = 0; i <= im - 1; i ++) {
       …
    }

   if (gp[procid].num>im-1)
         private = 1;
   else
         private = -1;

   if (private >0){
         …
}
```

Invariant: Exactly one thread takes the branch (thread 0).

BlockWatch: Example

```
long im = DEFAULT_N;
void slave() {
    int i, private, procID;
    //procid is the thread id
    if (procID == 0) {
            …
    }
    for (i = 0; i <= im - 1; i ++) {
       …
    }

   if (gp[procid].num>im-1)
         private = 1;
   else
         private = -1;

   if (private >0){
         …
}
```

Invariant: All threads either take the branch (OR) do not take the branch, i.e., they execute the same number of loop iterations.

BlockWatch: Example

```
long im = DEFAULT_N;
void slave() {
    int i, private, procID;
    //procid is the thread id
    if (procID == 0) {
            …
    }
    for (i = 0; i <= im - 1; i ++) {
       …
    }

   if (gp[procid].num>im-1)
        private = 1;
   else
        private = -1;

   if (private >0){
        …
}
```

BlockWatch: Example

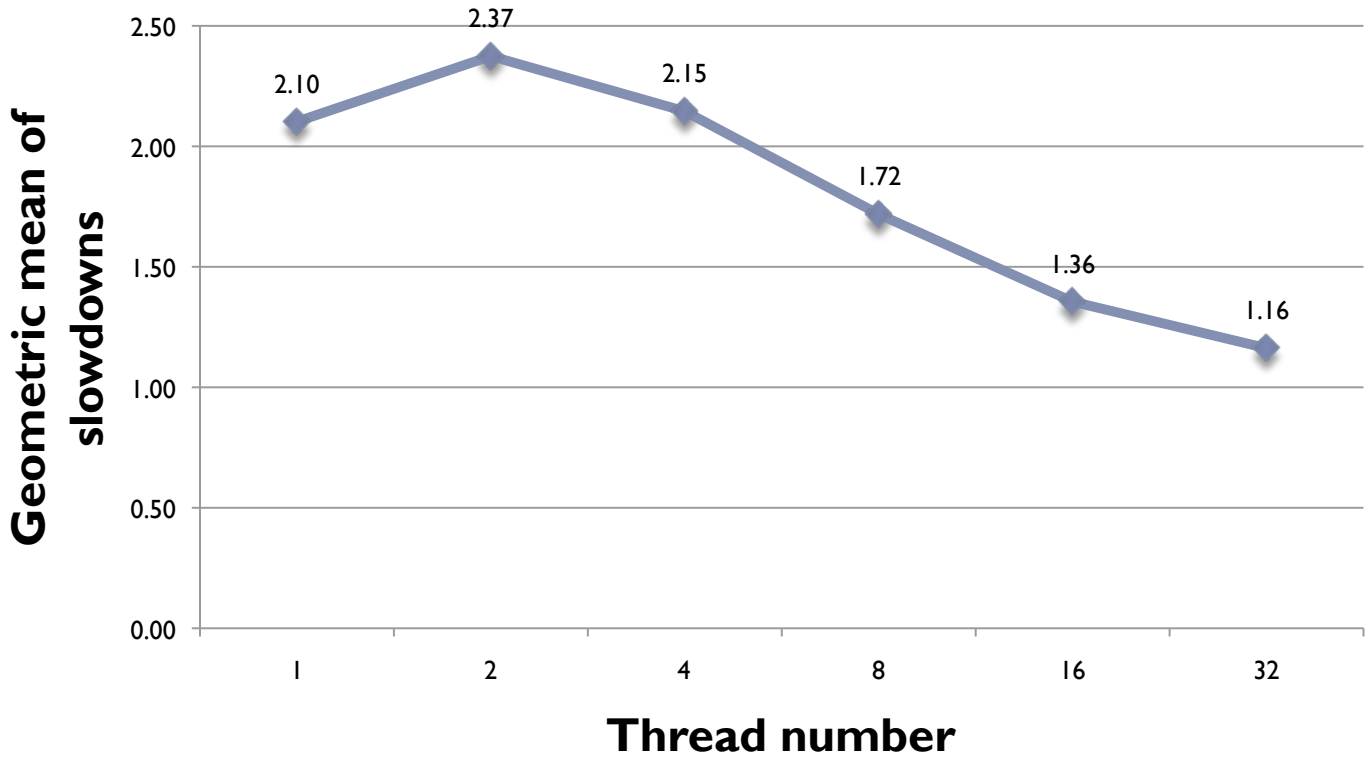
```
long im = DEFAULT_N;
void slave() {
    int i, private, procID;
    //procid is the thread id
    if (procID == 0) {
            …
    }
    for (i = 0; i <= im - 1; i ++) {
       ...
    }

   if (gp[procid].num>im-1)
         private = 1;
   else
         private = -1;

   if (private >0){
         ...
}
```

Invariant: All threads which have the same value of *private* will take the branch, while others will not.

BlockWatch: Experimental Setup

- Implemented using the LLVM compiler
  - Two passes: one for analysis and one for instrumentation
  - Monitor implemented using a lock-free queue/hash-table
- Evaluated on seven SPLASH2 benchmark programs
  - Range from 1000 to 11000 lines of C code
  - Between 50 and 95% of the branches exhibit similarity
- 32-core machine (four eight core nodes) machine
  - AMD Opteron 6120 processors at 2 Ghz each

BlockWatch: Performance Results



- Average overhead is about 16% for 32 threads on 32 cores

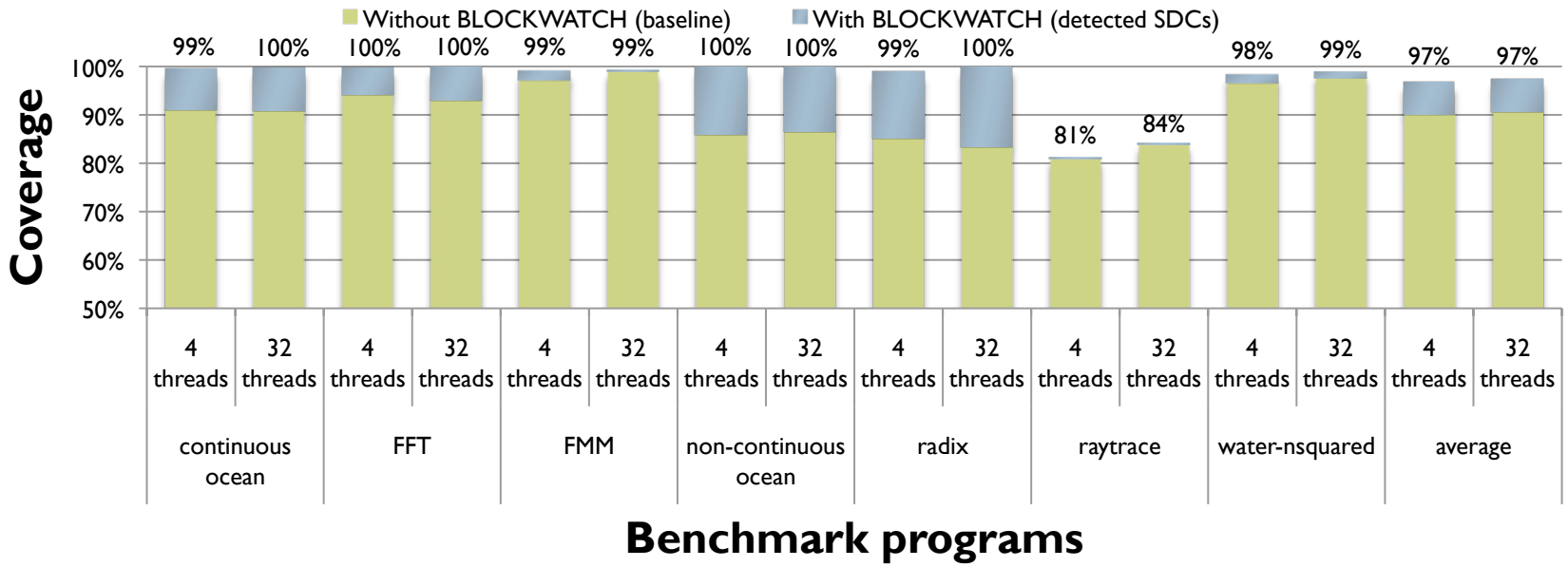
BlockWatch: Coverage Evaluation

- **Built a fault-injector using the PIN tool from Intel**
  - Injected faults in **all** branches executed by the program
  - Uniformly over the number of executed conditional branches
  - Faults = single bit-flip in branch condition variable

- **Monitored program after injecting fault for SDCs**
  - **Coverage = 1 – Prob. of SDC**

- Measured false-positives by executing without faults
  - **No false-positives observed for any benchmark**

BlockWatch: Coverage Results



- SDC coverage goes up from 85-90% without BlockWatch to 99-100% with BlockWatch (for 32 threads)
  - For all applications except Raytrace (81% to 84%)

BlockWatch: Summary

- **BlockWatch leverages similarity in parallel programs for detecting errors in control data**
  - Identifies 3 kinds of similarity in control data
  - Extracts the similarity through static analysis
  - Dynamically checks similarity through a monitor

- **Evaluated on a 32 core system with SPLASH2**
  - Performance overhead is about 16% for 32 threads
  - Error coverage is between 98 and 100% for 32 threads
  - No false-positives incurred for any of the programs

Talk Outline

- Motivation and Approach
- Good Enough Software Systems
  - Flikker [ASPLOS 2011] – with S.Liu, T. Moscibroda and B. Zorn
  - BlockWatch [submitted] – with J. Wei
- Conclusions and Future Work

Conclusions

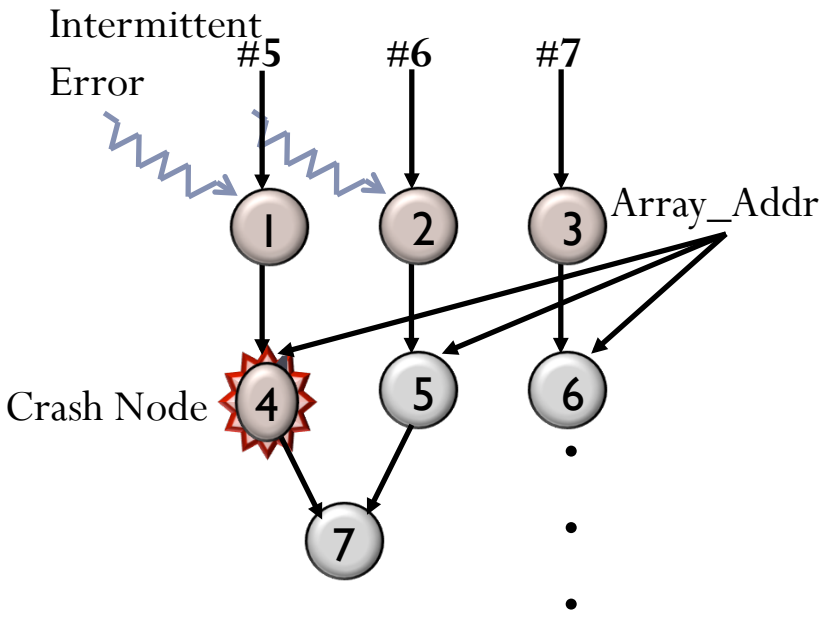
- **"Good Enough Software Systems" is a promising approach for dealing with hardware errors**
  - Software needs to be engineered to deal with hardware errors
  - Only need to be good enough to satisfy user's requirements
  - Can achieve substantial power and performance benefits

- **Two systems based on critical data in programs**
  - **Flikker:** Leverages slack in DRAM refresh rates
  - **BlockWatch:** Leverages parallel program's similarity

Future Work: Identifying Critical Data Automatically

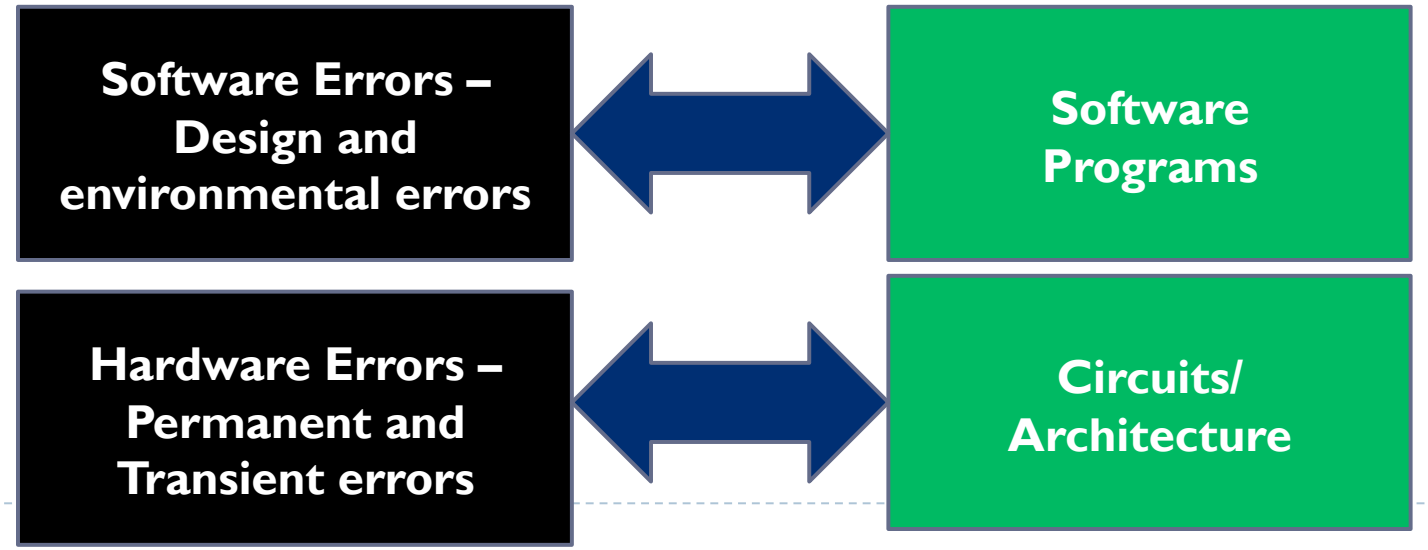
- Based on Dynamic Dependence Graph (DDG)
  - Use of heuristics to estimate error propagation
  - Critical data to minimize error propagation [PRDC 2010]
  - Algorithms for static analysis and error containment

Code Fragment	Node
mov R1, #5	1
mov R2, #6	2
mov R3, #7	3
ld R4, R1, Array_Addr	4
ld R5, R2, Array_Addr	5
ld R6, R3, Array_Addr	6
mult R7, R5, R4	7



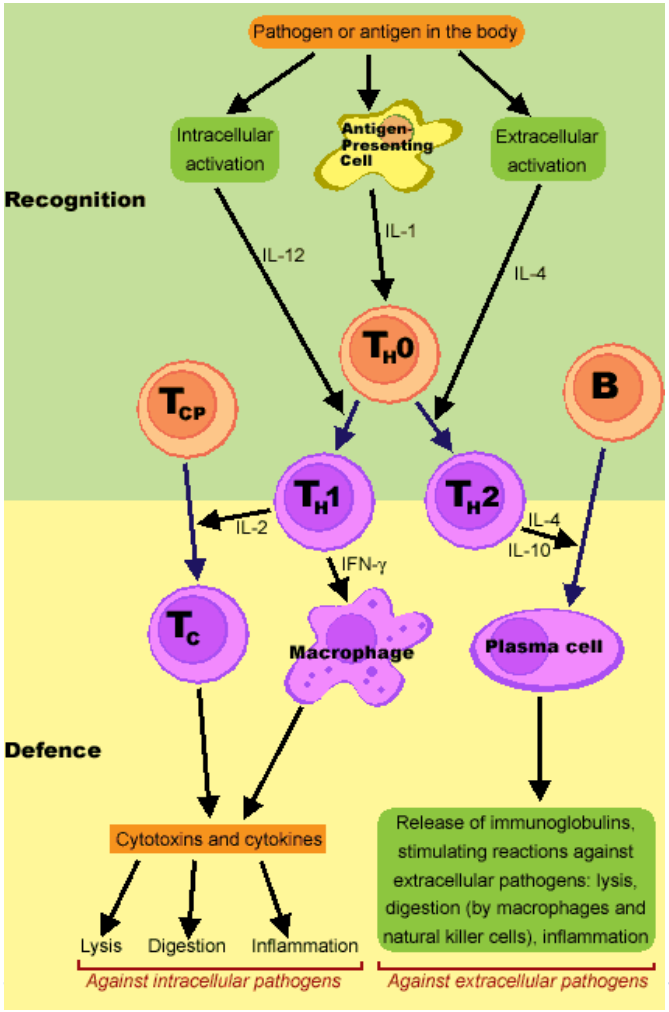
Future Work: Reasoning about resilience

- Formal verification techniques for software typically assume that the hardware is error free
- Need techniques to abstract hardware errors to software
- Use of model-checking [DSN'08], Hoare logics [CSF'11]



Vision: Software as Immune system

- **Software systems that anticipate and handle hardware errors**
  - Detect and diagnose source of the errors
  - Recover from errors by reconfiguring the software
    - JIT recompilation
    - OS scheduling
    - Algorithmic resilience



Source: mcld.co.uk

Thank you

http://www.ece.ubc.ca/~karthikp

Contact: karthikp@ece.ubc.ca

Partial Array Self Refresh (PASR)

- Self-refresh: low power, keep the data
- PASR: only refresh part of the memory array, configured among discrete levels [Samsung], [Micron]
- Cons: less DRAM available in idle periods

42

DRAM Error Rate

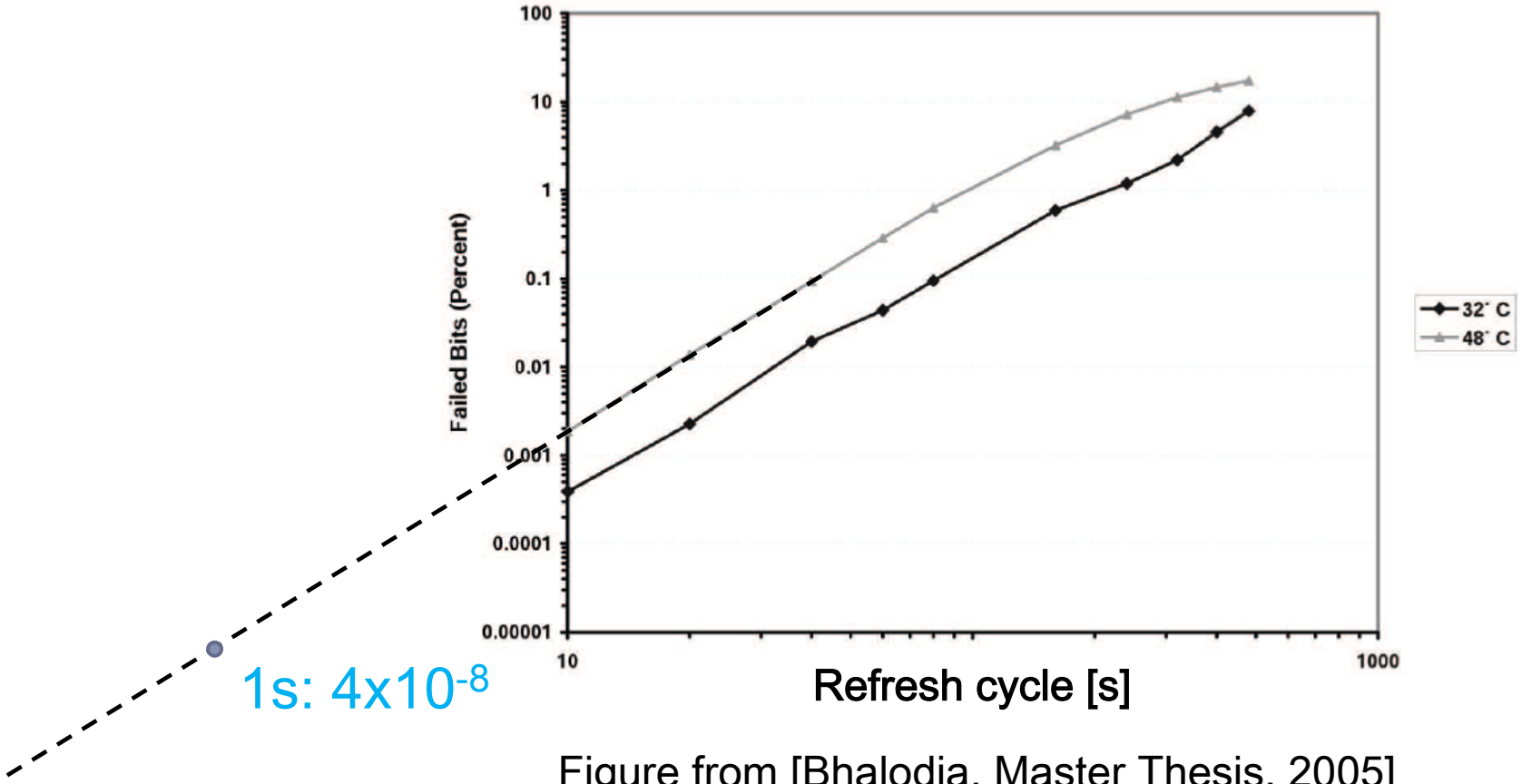


Figure from [Bhalodia, Master Thesis, 2005]

Fault-injection Result: SNR

- Signal-to-Noise-Ratio (SNR): the ratio of signal energy and noise energy
- SNR in logarithm scale: 3dB means double the ratio
- mpeg2 encoder -> decoder: 35 dB
- Flikker yields very high SNR

Configuration	mpeg2	rayshade
conservative	95.48	101.1
aggressive	88.34	72.84
crazy	88.04	73.63

Average SNR of degraded output of mpeg2 and rayshade [dB].

The impact of Flikker is negligible.

DRAM Refresh is Expensive

- Refresh power consumption
- Performance penalty
  - Refresh penalty increases with capacity [Stuecheli, MICRO'10]
- Variation in retention time [Venkatesan, HPCA'06]

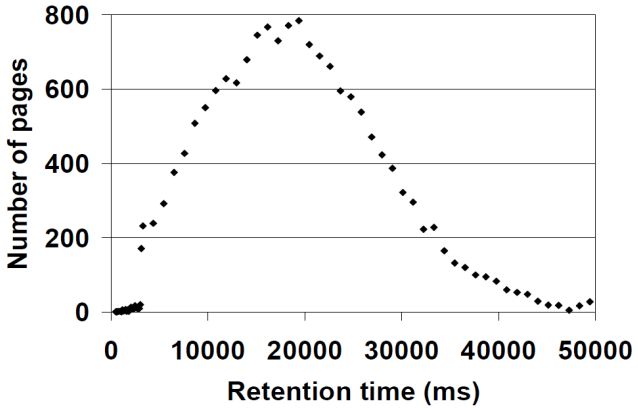
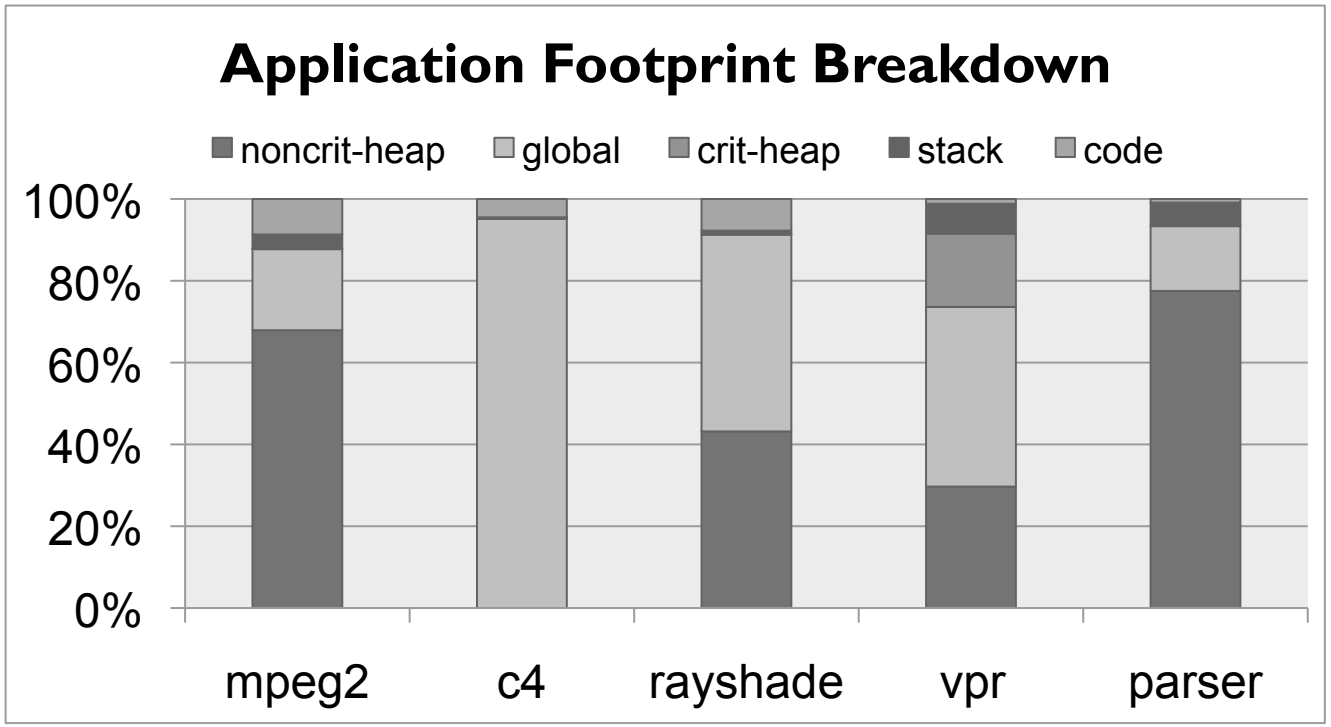


Figure from [Venkatesan, HPCA'06]

Memory Footprint Breakdown

- Global data is not partitioned



Self-refresh Power Model

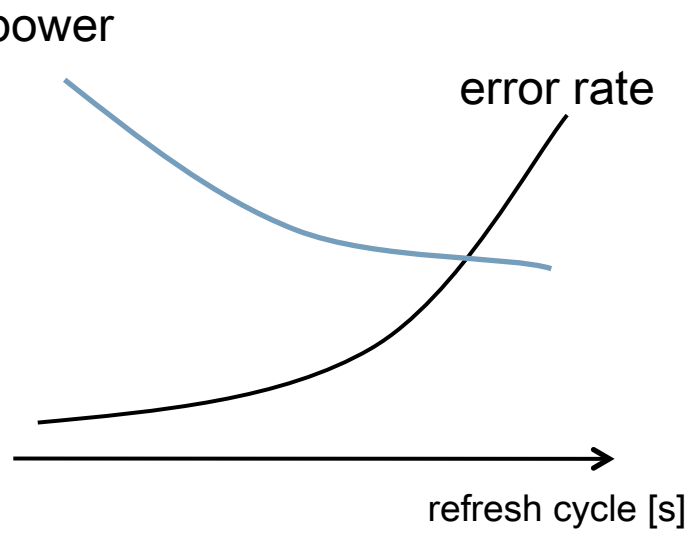
- Self-refresh power is not just power spent on refresh

$$P_{self\text{-}refresh} = P_{refresh} + P_{other}$$

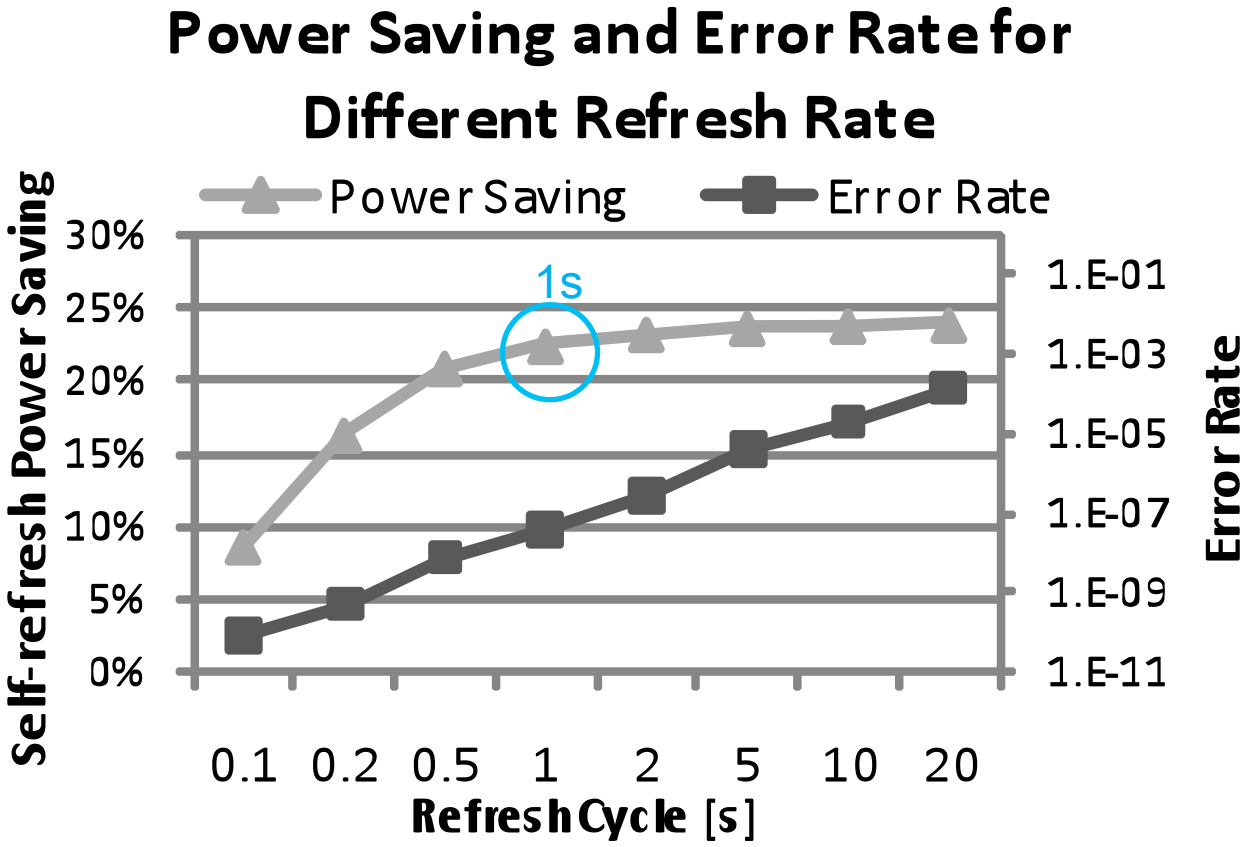
- Assume $P_{refresh}$ is proportional to refresh rate

High Refresh Part Size	Self-Refresh Current [mA]			
	PASR	Flicker		
		1s	10s	100s
1	0.5	0.5	0.5	0.5
3/4	0.47*	0.4719	0.4702	0.4700
1/2	0.44	0.4438	0.4404	0.4400
1/4	0.38	0.3877	0.3807	0.3801
1/8	0.35	0.3596	0.3509	0.3501
1/16	0.33	0.3409	0.3310	0.3301

* This value is derived from linear interpolation of full array (1) and half array(1/2) cases.



Power Saving vs. Error Rate



¼ array high refresh

BlockWatch: Static Analysis

- Used SSA-based analysis to identify similarity types
- Context sensitive analysis to track similarity types
  - Using dynamic context to resolve the runtime checks

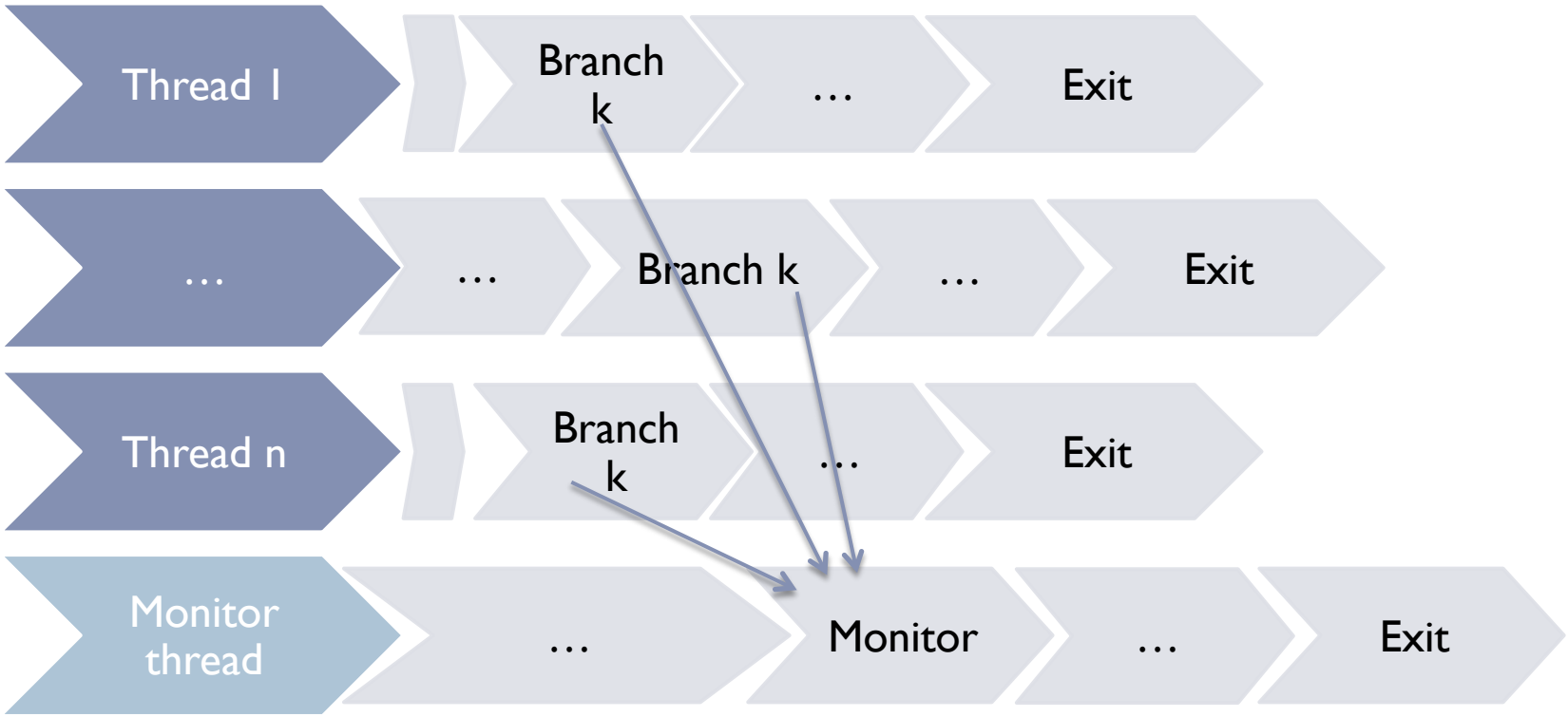
```
global g;
void foo() {
        goo(1);
        if (test)
                goo(2);
}

void goo(int callSite) {
        for (i = 0; i < 5; ++i)
                if (i < g)
                  checkBranch(TAKEN, i, callSite);
}
```

inserted

inserted

inserted

BlockWatch: Monitor Implementation



Uses a lock free queue and hash-table to check branches - Asynchronous