“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**

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

**Timing errors**

- Manufacturing/Design defects

- Intel Sandy Bridge chipset bug, 2011
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](http://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

User interacts with the application

- Application
- Operating System/Virtual Machine
- Architecture
- Devices/Circuits

User

Allow errors across the hardware-software boundary, but ensure user experience is not adversely affected
Why Software?

Errors get progressively filtered as we go up the system stack

- Device/Circuit Level
- Architectural Level
- Operating System Level
- Application Level

Overheads

Impactful Errors
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

The opportunity

The cost

Where we are today

Where we want to be

64 mSec

refresh cycle [s]

X sec

power

error rate
Flikker: Approach

- Critical / non-critical data partitioning

<table>
<thead>
<tr>
<th>Critical (crit)</th>
<th>Non-critical (non-crit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High refresh</td>
<td>Low refresh</td>
</tr>
<tr>
<td>No errors</td>
<td>Some errors</td>
</tr>
</tbody>
</table>

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)**

- **Programmer**
- **Allocator**
- **Operating System**

- **DRAM**
  - **High Refresh Rows**
  - **Low Refresh Rows**

Virtual pages:
- **critical object**
- **non-critical object**

Physical pages:
- **critical page**
- **non-critical page**
Flikker: Mobile Applications

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

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

- **Baseline**
  - Code
  - Stack
  - Global
  - Heap

- **Conservative**
  - Code
  - Stack
  - Global
  - Heap

- **Ideal**
  - Code
  - Stack
  - Global
  - Heap

- **Aggressive**
  - Code
  - Stack
  - Global
  - Heap

- **Crazy**
  - Code
  - Stack
  - Global
  - Heap

- **Critical**
  - Blue

- **Non-critical**
  - Red
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

Fault Inject Results for 1s Refresh Cycle
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)
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]

```c
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
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

```c
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.
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

```c
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

<table>
<thead>
<tr>
<th>Code Fragment</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov R1, #5</td>
<td>1</td>
</tr>
<tr>
<td>mov R2, #6</td>
<td>2</td>
</tr>
<tr>
<td>mov R3, #7</td>
<td>3</td>
</tr>
<tr>
<td>ld R4, R1, Array_Addr</td>
<td>4</td>
</tr>
<tr>
<td>ld R5, R2, Array_Addr</td>
<td>5</td>
</tr>
<tr>
<td>ld R6, R3, Array_Addr</td>
<td>6</td>
</tr>
<tr>
<td>mult R7, R5, R4</td>
<td>7</td>
</tr>
</tbody>
</table>
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].

Software Errors – Design and environmental errors

Hardware Errors – Permanent and Transient errors

Software Programs

Circuits/Architecture
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: mcl.d.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
DRAM Error Rate

1s: $4 \times 10^{-8}$

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**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>mpeg2</th>
<th>rayshade</th>
</tr>
</thead>
<tbody>
<tr>
<td>conservative</td>
<td>95.48</td>
<td>101.1</td>
</tr>
<tr>
<td>aggressive</td>
<td>88.34</td>
<td>72.84</td>
</tr>
<tr>
<td>crazy</td>
<td>88.04</td>
<td>73.63</td>
</tr>
</tbody>
</table>

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

![Application Footprint Breakdown](image-url)
Self-refresh Power Model

- Self-refresh power is not just power spent on refresh
  \[ P_{\text{self-refresh}} = P_{\text{refresh}} + P_{\text{other}} \]
- Assume \( P_{\text{refresh}} \) is proportional to refresh rate

<table>
<thead>
<tr>
<th>High Refresh Part Size</th>
<th>Self-Refresh Current [mA]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PASR</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>3/4</td>
<td>0.47*</td>
</tr>
<tr>
<td>1/2</td>
<td>0.44</td>
</tr>
<tr>
<td>1/4</td>
<td>0.38</td>
</tr>
<tr>
<td>1/8</td>
<td>0.35</td>
</tr>
<tr>
<td>1/16</td>
<td>0.33</td>
</tr>
</tbody>
</table>

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

Power Saving and Error Rate for Different Refresh Rate

Self-refresh Power Saving

0% 5% 10% 15% 20% 25% 30%

Error Rate

0.1 0.2 0.5 1 2 5 10 20

Refresh Cycle [s]

1s


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

```c
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);
}
```
BlockWatch: Monitor Implementation

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