BLOCKWATCH: Leveraging Similarity in Parallel Programs for Error Detection

Jiesheng Wei and Karthik Pattabiraman
University of British Columbia (UBC)
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.
Why Software?

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

Overheads

Impactful Errors
Software Development Trend

Parallel programs become ubiquitous and essential

How to protect parallel programs from hardware faults at software level?
How to detect faults propagated to parallel programs at software level?
Leverage the similarity across threads of the parallel programs for error checking

Why similarity? High-level program models (e.g., SPMD)

What data? Control data in this study

Errors in control-data are more likely to lead to silent data corruptions (SDC) [Thaker-IISWC-2006]

```c
int max(int a[], int n) {
    int max = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
```
BlockWatch

- Leverage the similarity across threads of the parallel programs for error checking
  - Why similarity? High-level program models (e.g., SPMD)
  - What data? Control data in this study
  - What faults? Transient hardware faults in computation
Why Static Analysis?

- No false positives, unlike dynamic techniques

### Diagram:

- **Thread 1**, **Thread 2**, **Thread 3**, **Thread 4**

  - Shared

  - Error!
Outline

- Motivation
- BlockWatch Approach
- Experimental Setup and Evaluation
- Conclusion
BlockWatch Approach

- **Identify control-data similarity patterns in parallel programs**
  - Focus on shared-memory programs in this study

- **Extract the similarity through static analysis**
  - Instrument error detection code to check the similarity

- **Check similarity at runtime through a monitor**
Example: similarity pattern

```java
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)
        ...
}
```

**Var. characteristics**: Depends on thread ID

**Invariant**: Exactly one thread takes the branch (thread 0).
Example: similarity pattern

```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){
        ...
    }
}
```

**Var. characteristics**: Depend on shared variables

**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){
        ...
    }
}

Var. characteristics : Depends on local variables
Example: similarity pattern

```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)
        …
}
```

**Var. characteristics:** Depends on a subset of shared variables

**Invariant:** All threads which have the same value of `private` will take the same decision at the branch.
Compiler-based Static Analysis

Similarity Category Summary

<table>
<thead>
<tr>
<th>Similarity category</th>
<th>Variable characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>All operands are shared vars</td>
</tr>
<tr>
<td>ThreadID</td>
<td>One operand depends on thread ID; the remaining are shared vars</td>
</tr>
<tr>
<td>Partial</td>
<td>Local vars that are assigned with a small subset of shared vars.</td>
</tr>
<tr>
<td>None</td>
<td>Local vars</td>
</tr>
</tbody>
</table>

Algorithm

- Study the propagation of shared vars and thread ID
  - Iterative data flow algorithm
- Get the similarity categories of the branches
Example: static analysis

```c
long im = 0;
void foo(int direction) {
    ...
    if (direction > 0) { //branch 1
        ...
    } else {
        ...
    }
}

void parallel_start() {
    int dir;
    int procID; // procID is thread ID
    ...
    if (procID == im) //branch 2
        dir = 1;
    else
        dir = -1;
    foo(dir);
}
```

<table>
<thead>
<tr>
<th>1st iteration:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>im</td>
<td>NA</td>
</tr>
<tr>
<td>direction</td>
<td>NA</td>
</tr>
<tr>
<td>branch 1</td>
<td>NA</td>
</tr>
<tr>
<td>procID</td>
<td>NA</td>
</tr>
<tr>
<td>branch 2</td>
<td>NA</td>
</tr>
<tr>
<td>dir</td>
<td>NA</td>
</tr>
</tbody>
</table>
Example: static analysis

```c
long im = 0;
void foo(int direction) {
    ...
    if (direction > 0) { //branch 1
        ...
    } else {
        ...
    }
}

void parallel_start() {
    int dir;
    int procID; // procID is thread ID
    ...
    if (procID == im) //branch 2
        dir = 1;
    else
        dir = -1;
    foo(dir);
}
```

2nd iteration:

<table>
<thead>
<tr>
<th></th>
<th>shared</th>
<th>partial</th>
<th>threadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>im</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>direction</td>
<td></td>
<td></td>
<td>(NA)</td>
</tr>
<tr>
<td>branch 1</td>
<td></td>
<td>(NA)</td>
<td>(NA)</td>
</tr>
<tr>
<td>procID</td>
<td></td>
<td>threadID</td>
<td></td>
</tr>
<tr>
<td>branch 2</td>
<td></td>
<td>threadID</td>
<td></td>
</tr>
<tr>
<td>dir</td>
<td></td>
<td>partial</td>
<td></td>
</tr>
</tbody>
</table>
Example: static analysis

```c
long im = 0;
void foo(int direction) {
    ...
    if (direction > 0) { //branch 1
        ...
    } else {  
        ...
    }
}

void parallel_start() {
    int dir;
    int proclID; // proclID is thread ID
    ...
    if (proclID == im) //branch 2
        dir = 1;
    else
        dir = -1;
    foo(dir);
}
```

3rd iteration:

<table>
<thead>
<tr>
<th></th>
<th>im</th>
<th>shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>direction</td>
<td>partial</td>
<td></td>
</tr>
<tr>
<td>branch 1</td>
<td>partial</td>
<td></td>
</tr>
<tr>
<td>proclID</td>
<td>threadID</td>
<td></td>
</tr>
<tr>
<td>branch 2</td>
<td>threadID</td>
<td></td>
</tr>
<tr>
<td>dir</td>
<td>partial</td>
<td></td>
</tr>
</tbody>
</table>
Example: static instrumentation

```
long im = 0;
void foo(int direction) {
    ...
    if (direction > 0) {   //branch 1
        ...
    } else {
        ...
    }
}

void parallel_start() {
    int dir;
    int procID; // procID is thread ID
    ...
    if (procID == im) //branch 2
        dir = 1;
    else
        dir = -1;
    foo(dir);
}
```

sendBranchAddr(2, TAKEN)

sendBranchAddr(2, NOTTAKEN)
Example: runtime check

Thread 1

... Branch 2 ...

TAKEN

Exit

Thread i

... Branch 2 ...

NOT TAKEN

Exit

Thread n

Branch 2 ...

NOT TAKEN

Exit

Monitor ...

Check ...

✓

Thread ID Invariant: Exactly one thread takes the branch

branch 2  thread ID
Outline

- Motivation

- BlockWatch Approach

- Experimental Setup and Evaluation

- Conclusion
Experimental Setup

- Implemented using the LLVM compiler
  - Two passes: one for analysis and one for instrumentation

- Evaluated on seven SPLASH-2 benchmark programs
  - Between 50% and 95% of the branches exhibit similarity

- 32-core (four eight core nodes) machine
  - AMD Opteron 6120 processors at 2 Ghz each

- Built a fault-injector using the PIN tool from Intel
  - Injected faults in all branches in the parallel section
  - Faults = single bit-flip in branch condition variable in one thread
Performance Evaluation

slowdown = \frac{\text{execution time with BlockWatch}}{\text{execution time without BlockWatch}}

Average overhead is about 16% for 32 threads on 32 cores
Coverage Evaluation

Monitored program after injecting fault for Silent Data Corruptions (SDC)

\[ \text{Coverage} = 1 - \frac{\text{Number of SDCs}}{\text{Number of activated faults}} \]

SDC coverage goes up from 90% without BlockWatch to 98 - 100% with BlockWatch
Outline

- Motivation
- BlockWatch Approach
- Experimental Setup and Evaluation
- Conclusion
Conclusion

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

- **Future work**
  - Extend BlockWatch to other classes of parallel programs
  - Extend BlockWatch to other program data
  - Further reduce the performance overhead
Thank you

- **http://www.ece.ubc.ca/~jwei**
- **Contact: jwei@ece.ubc.ca**
Why Control Data?

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

```c
int max(int a[], int n) {
    int max = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
```
Runtime Monitor

- Design goal
  - Asynchronous
  - Lock freedom
  - Fast lookup

- Architecture
Critical Data

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

<table>
<thead>
<tr>
<th>Identified by programmer or by the compiler</th>
<th>Application Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical Data</td>
<td>Corruption due to h/w errors</td>
</tr>
</tbody>
</table>
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)