BLOCKWATCH: Leveraging Similarity in Parallel Programs for Error Detection

Jiesheng Wei and Karthik Pattabiraman
University of British Columbia (UBC)

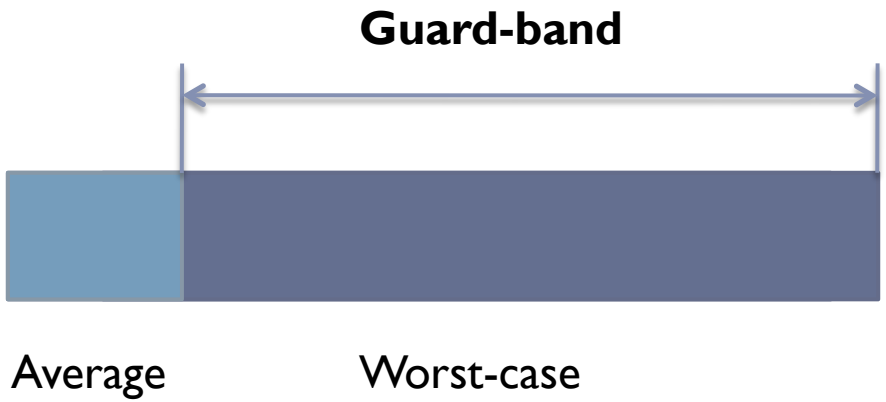
Hardware Errors: Traditional "Solutions"

- **Guard-banding**

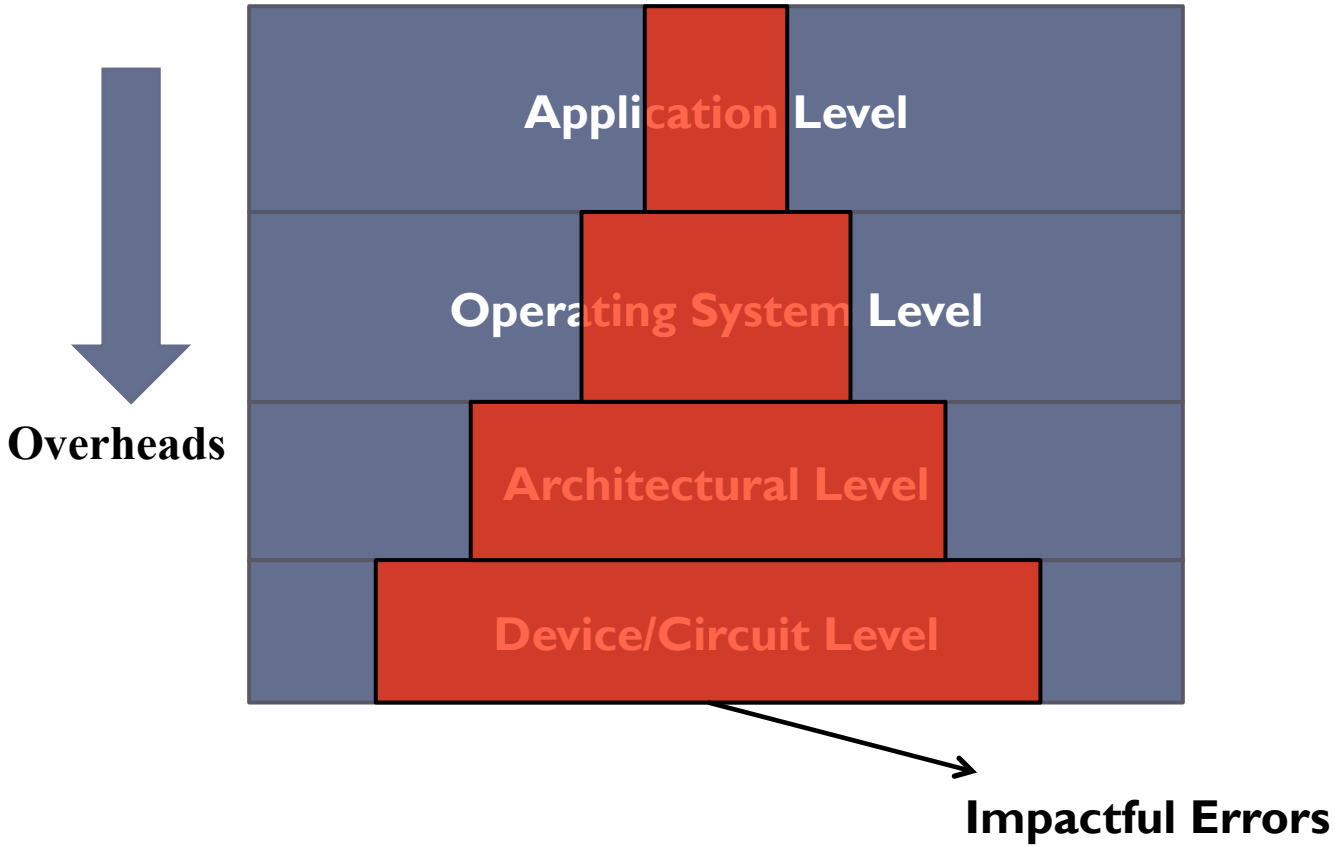
  Guard-banding wastes power and performance as gap between average and worst-case widens due to variations

- **Duplication**

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

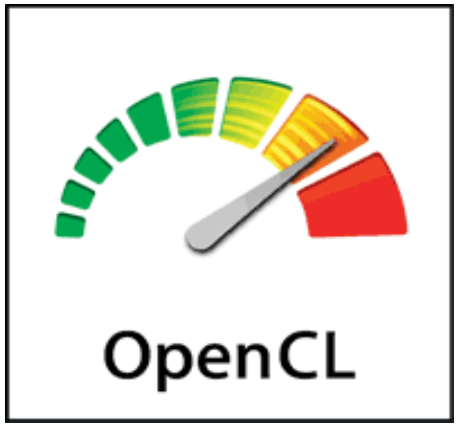


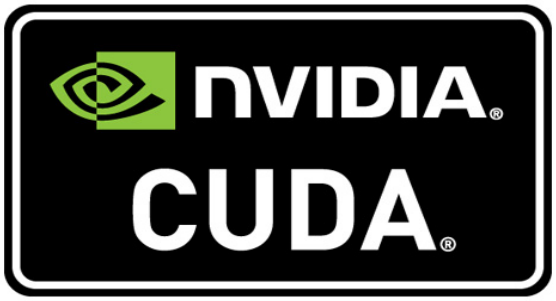
Why Software ?



Software Development Trend

Parallel programs become ubiquitous and essential





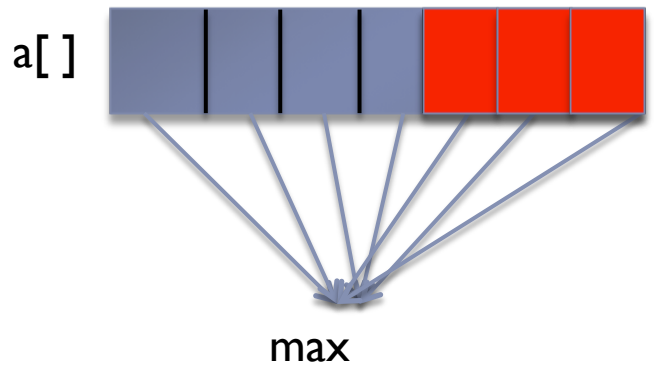
How to protect parallel programs from hardware faults *at software level*?

How to detect faults propagated to parallel programs *at software level*?

BlockWatch

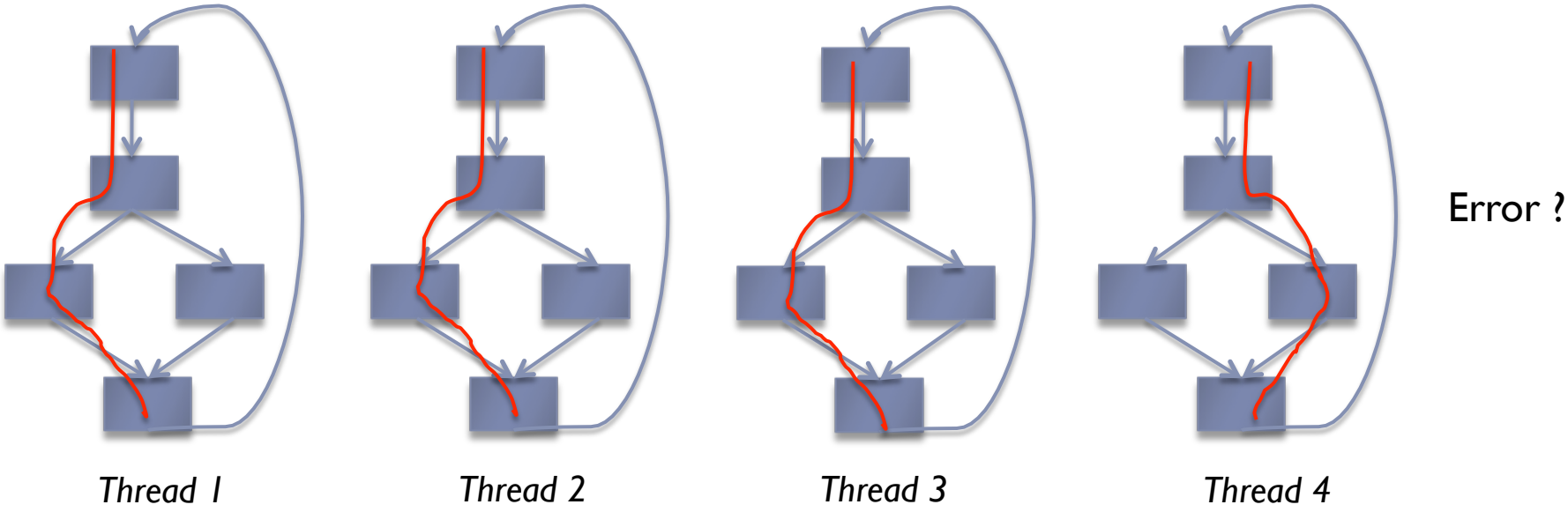
- Leverage the similarity across threads of the parallel programs for error checking
  - Why similarity? High-level program models (e.g., SPMD)
  - What data? Control data in this study
    - Errors in control-data are more likely to lead to silent data corruptions (SDC) [Thaker-IISWC-2006]

```
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;
}
```



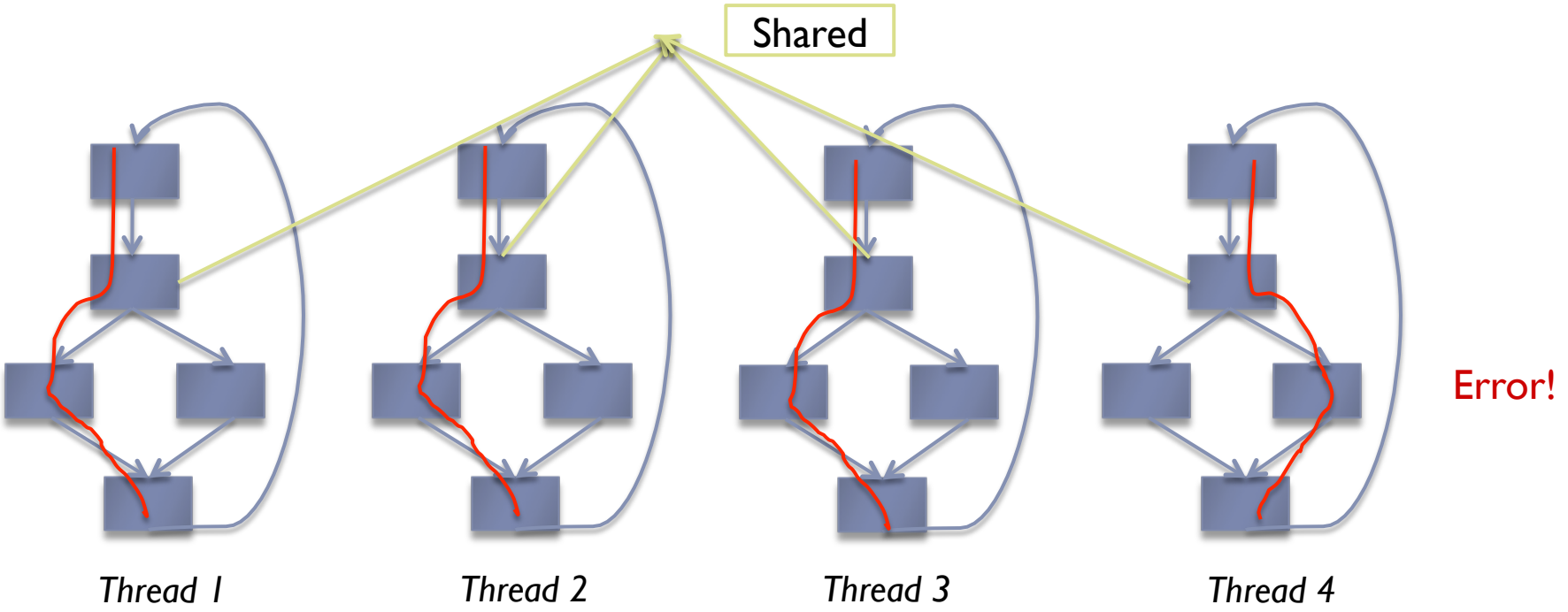
BlockWatch

- 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



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

```
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

```
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.

Example: similarity pattern

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

None

Var. characteristics : Depends on local variables

Example: similarity pattern

```
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

Similarity category	Variable characteristics
Shared	All operands are shared vars
ThreadID	One operand depends on thread ID; the remaining are shared vars
Partial	Local vars that are assigned with a small subset of shared vars.
None	Local vars

Algorithm

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

Example: static analysis

```
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);
}
```

1st iteration:

im	NA *shared*
direction	NA
branch 1	NA
procID	NA *threadID*
branch 2	NA *threadID*
dir	NA *shared*

Example: static analysis

```
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:

im	*shared*
direction	~~*partial*~~ NA
branch 1	~~*partial*~~ NA
procID	*threadID*
branch 2	*threadID*
dir	*partial*

Example: static analysis

```
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);
}
```

3rd iteration:

im	*shared*
direction	*partial*
branch 1	*partial*
procID	*threadID*
branch 2	*threadID*
dir	*partial*

Example: static instrumentation

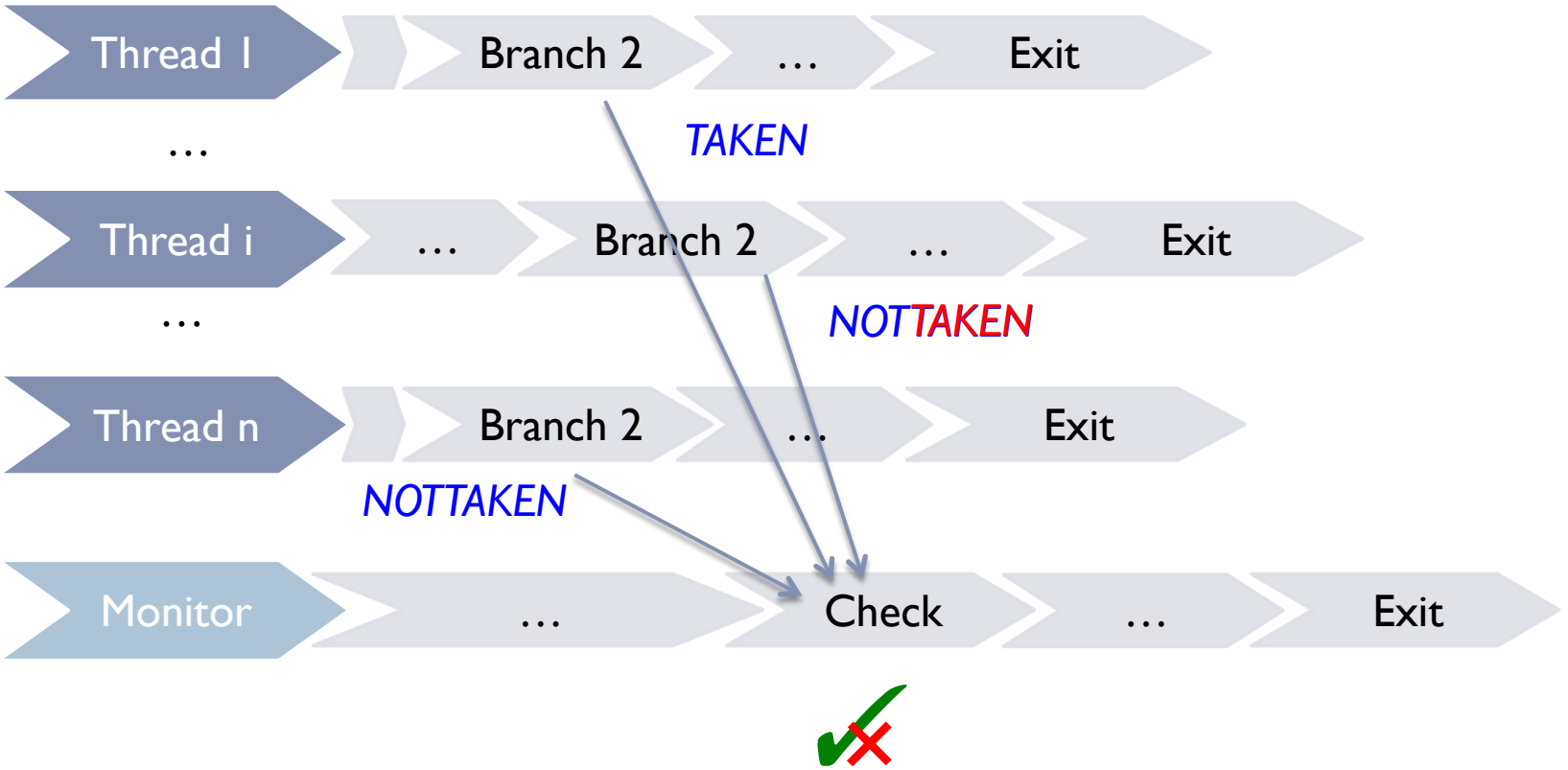
```
long im = 0;
void foo(int direction) {
   …
   if (direction > 0) { //branch 1
      …
   } else {
      …
   }
}

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)

branch 2	*threadID*

Example: runtime check



***ThreadID* Invariant:** Exactly one thread takes the branch

branch 2	*threadID*

Outline

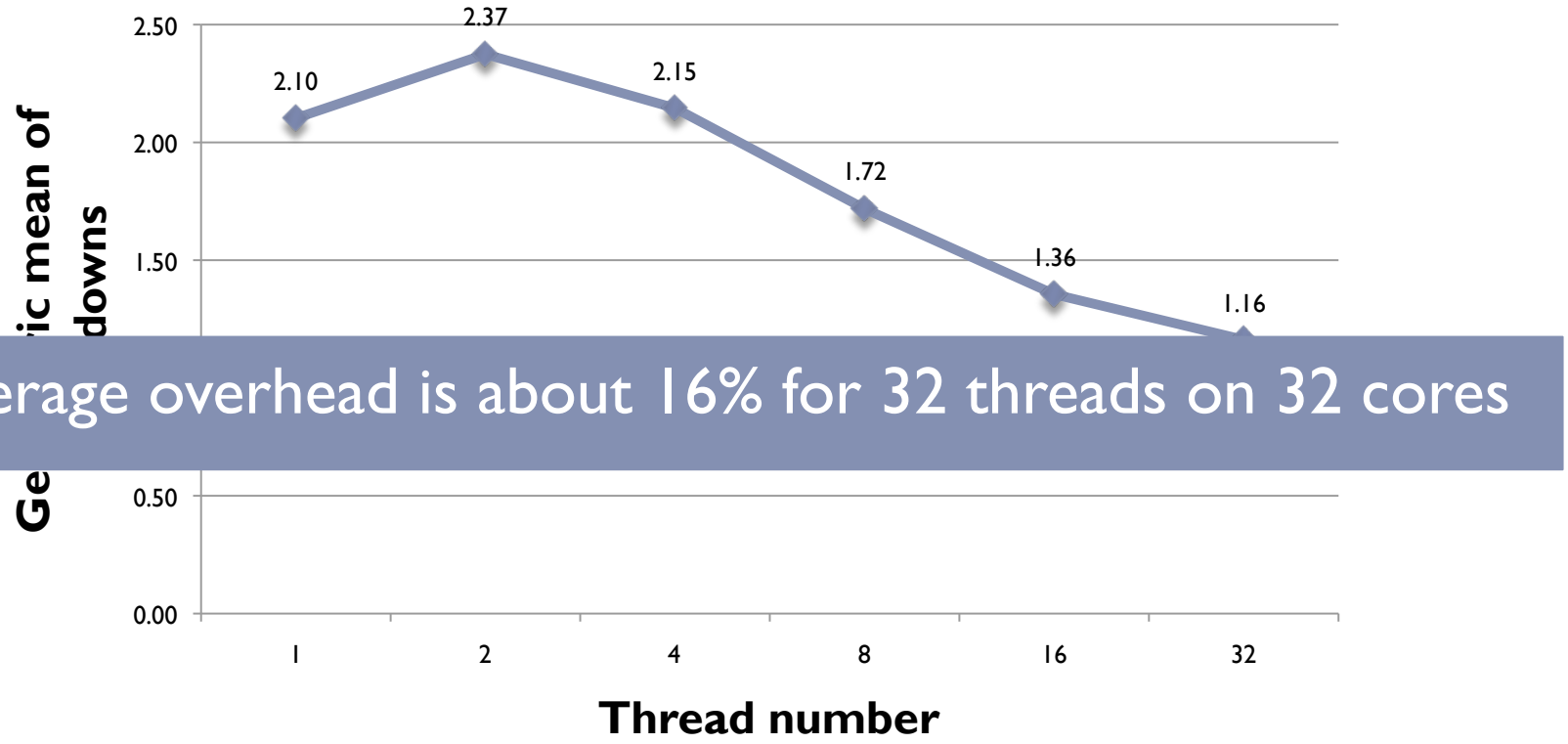
- Motivation
- BlockWatch Approach
- Experimental Setup and Evaluation
- Conclusion

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

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

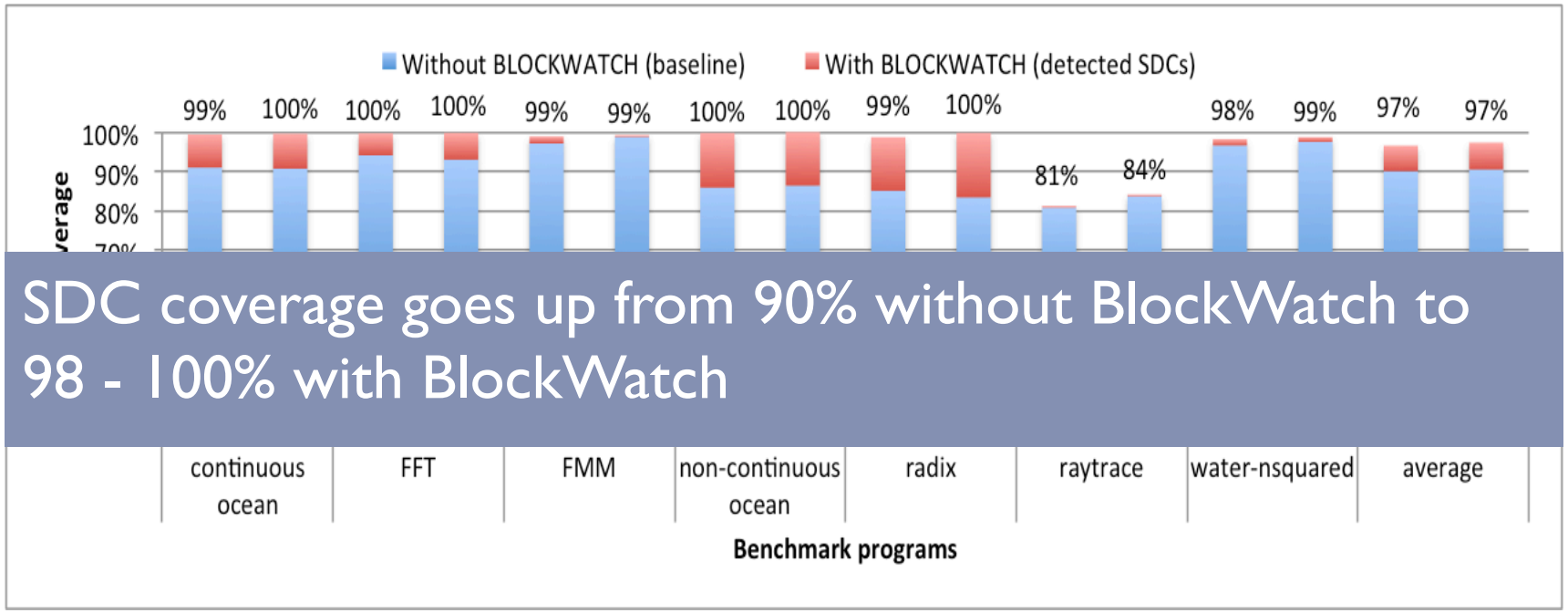


Average overhead is about 16% for 32 threads on 32 cores

Coverage Evaluation

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

$$Coverage = 1 - \frac{Number\ of\ SDCs}{Number\ of\ activated\ faults}$$



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

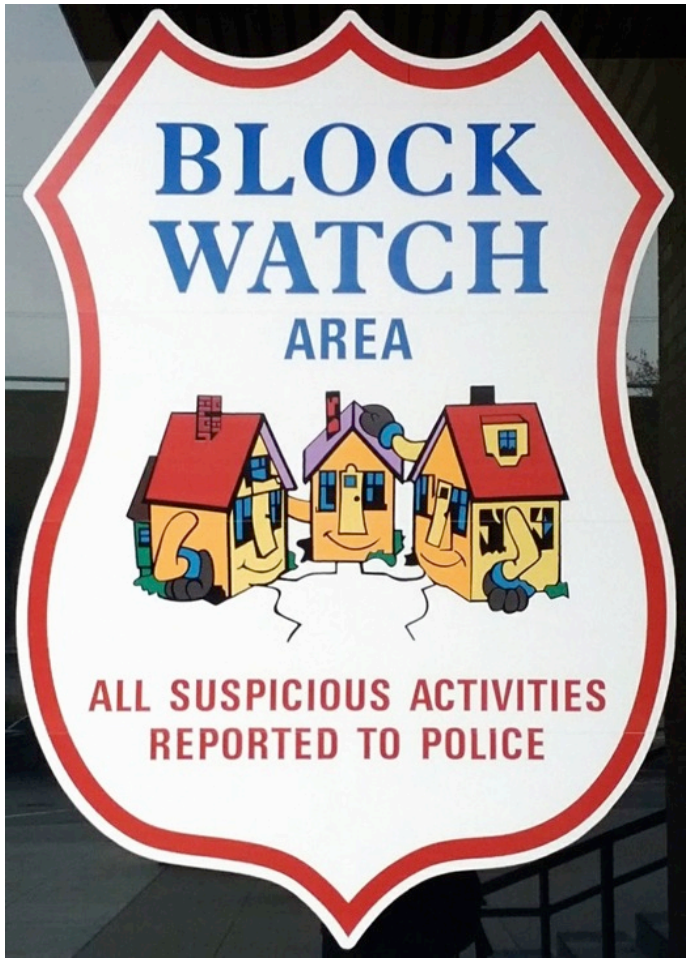
Outline

- 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



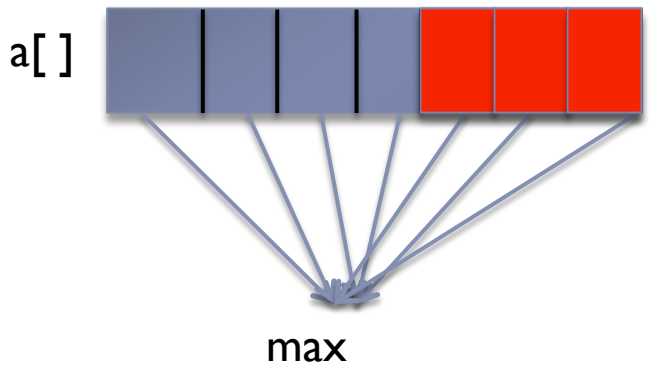
Thank you

- http://www.ece.ubc.ca/~jwei
- Contact: jwei@ece.ubc.ca

Why Control Data ?

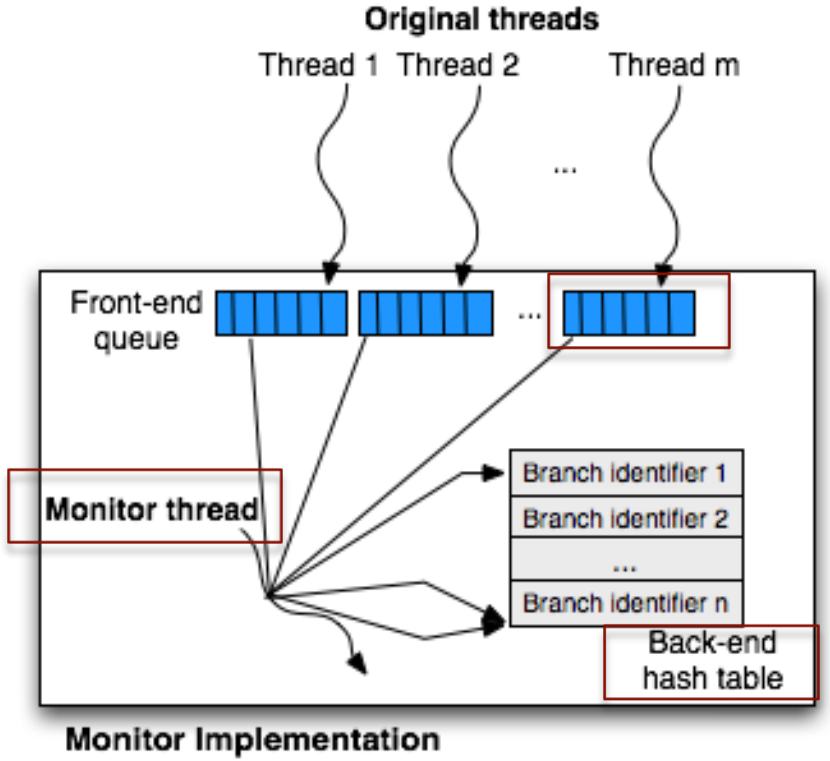
Errors in control-data are more likely to lead to egregious outputs and catastrophic failures [Thaker-IISWC-2006]

```
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;
}
```



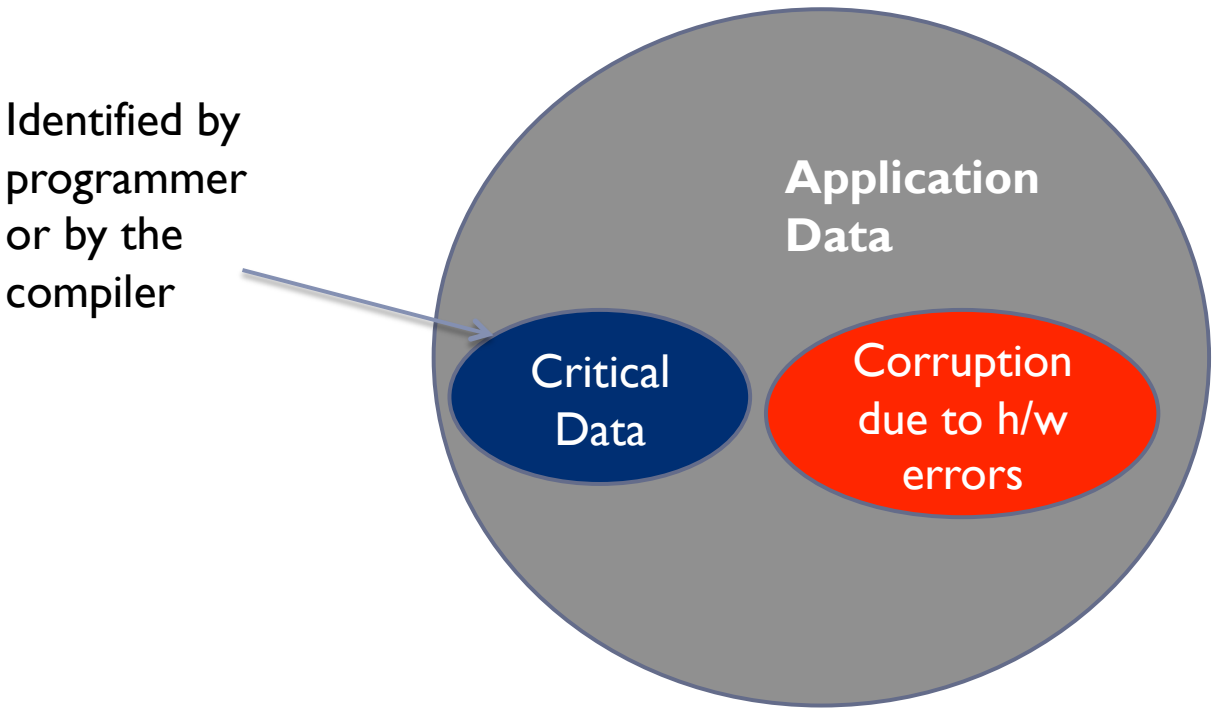
Runtime Monitor

- Design goal
  - Asynchronous
  - Lock freedom
  - Fast lookup
- Architecture



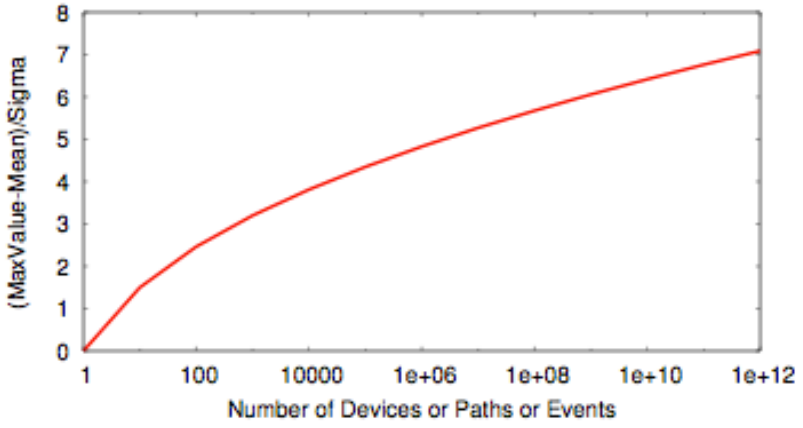
Critical Data

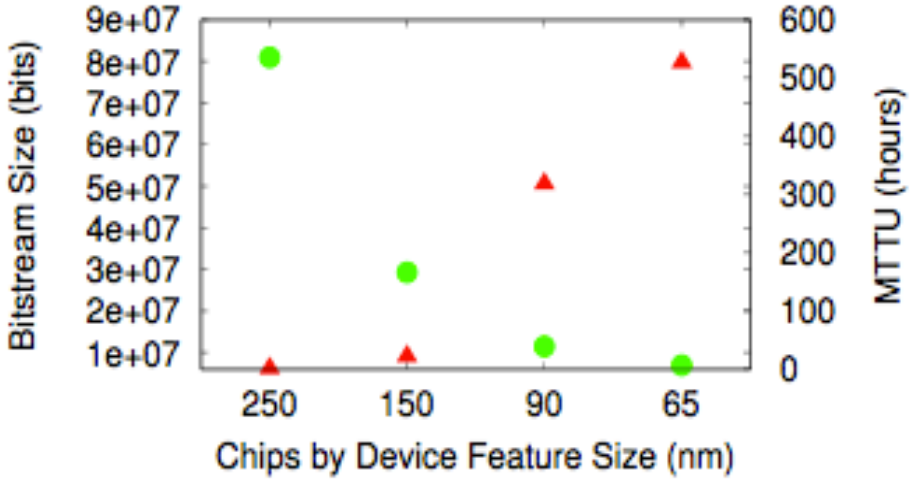
- **Software has high-level redundancy in data**
  - Can tolerate limited amounts of data corruption
  - Provided certain critical data is not corrupted



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)