iWatcher: Efficient Architectural Support for Software Debugging

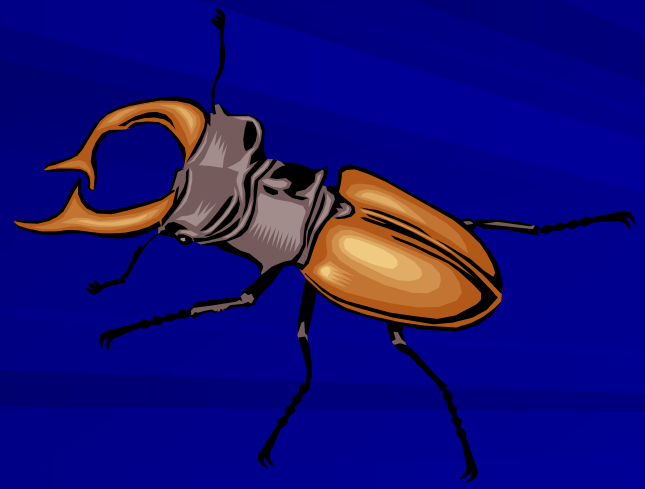
Yuanyuan (YY) Zhou
yyzhou@cs.uiuc.edu

Pin Zhou, Feng Qin, Wei Liu
Yuanyuan Zhou and Josep Torrellas

Department of Computer Science, UIUC

Software Bugs are Expensive!

- Software bugs
  - Counts for 40% system failure
  - Costs 59.5 billion dollars annually, 0.6% gross domestic product!
  - 50-75% of software development cost



Software Debugging is Hard!

- Too many bugs and keep coming
  - On average,100 defects per thousand lines of code
- Hard to test everything
  - Many bugs are hidden, “transient bugs”
- Some bugs are dynamic (hard to reproduce)
  - Configuration and running environment dependent
  - Time-dependent: a particular inter-leaving sequence
  - Unexpected user interaction
- Hard to find root causes
  - Root causes may be far away from where bugs occur

Existing Support for Debugging

- Interactive debugger (e.g. gdb)
  - Pros: program-specific
  - Cons: Time and effort-consuming
- Static checking
  - Find bugs using static analysis or model checking
  - Pros: No execution overhead
  - Cons: Need specification, annotation, and
    No accurate information
- Dynamic monitoring
  - Monitoring execution dynamically
  - Pros: Have more accurate information
  - Cons: Impose run-time overhead

Classification of Dynamic Monitoring

- Code controlled monitoring (CCM)
  - Monitoring is done at special code locations
  - Examples:
    - Assertions
    - instrumentation-based dynamic checkers (e.g. DIDUCE)
- Location-controlled monitoring (LCM)
  - Monitoring is associated with memory locations
  - Examples:
    - Watchpoints
    - iWatcher

Advantages of LCM

- No aliasing problem
- Can catch all accesses to a watched location
- Monitor only “true” accesses
- Example: CCM may catch an error much later

```
int x, *p;
                    /* invariant: x=0 or 1 */
...
p = foo();        /* a bug: p points to x incorrectly */
*p = 5;           /* line A: corrupt x */
...
assert( (x==0) || (x == 1));    /* an invariant check*/
z = Array[x];
...
```

Limitations of Watchpoints

- Inflexible for dynamic monitoring
  - Limited number of watchpoints (4 in x86)
  - Trigger an OS exception upon an access to a monitored location
- Inefficient for dynamic monitoring
  - OS exceptions are expensive

Our Contribution

- iWatcher, a flexible and efficient location-controlled monitoring approach
  - Use architectural support
  - Language-independent
  - Cross-module and developers
  - Work for low-level code

Outline

- Motivation
- Background
- Main idea of iWatcher
- Architectural design of iWatcher
- Evaluation results

Thread-Level Speculation (TLS)

- Speculative parallelization of sequential code
  - Multiple simultaneously executed microthreads
  - Maintaining sequential semantics
  - Detecting true data dependence violations between microthreads
  - Buffering speculative state so that the side effects made by a mocrothread can be discarded

iWatcher: Main Idea

- Associate monitoring functions with watched memory locations
  - At a triggering access to a watched location, the associated monitoring functions are triggered and executed by hardware
- Use TLS for reducing overhead and rollback
  - Execute the main thread speculatively in parallel to monitoring functions
  - Use TLS to buffer states for rollback in case of errors reported by monitoring functions

iWatcher User Interface

- Turn on/off monitoring for a memory location
  - iWatcherOn (MemAddr, Length, WatchFlag, ReactMode, MonitorFunc, Param1, Param2, …, ParamN)
  - iWatcherOff (MemAddr, Length, WatchFlag, MonitorFunc)
  - A global switch
    - EnableiWatcher
    - DisableiWatcher

An Example

- No need to insert invariant checks at accesses

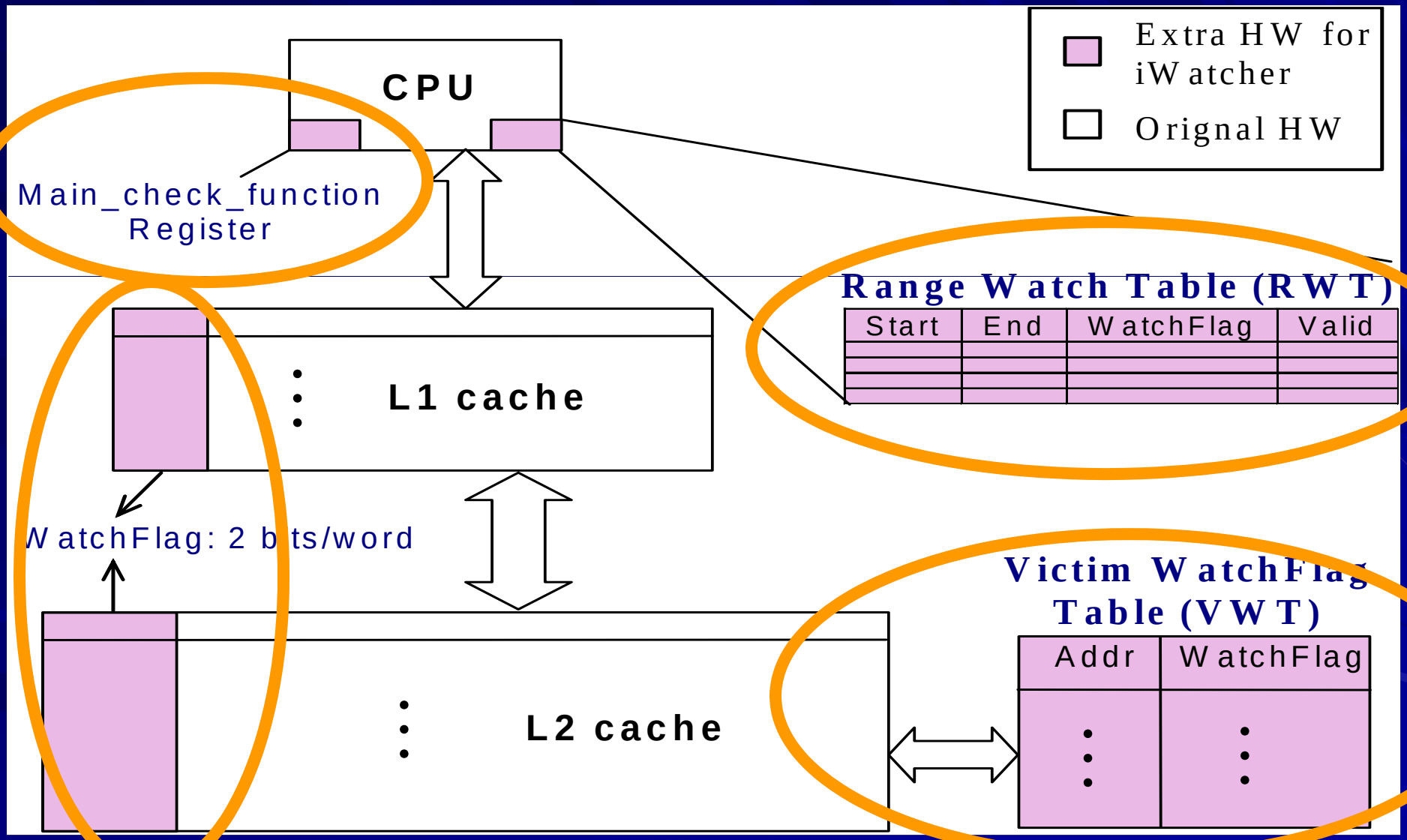
```
int x, *p;                    /* invariant: x=1 */
iWatcherOn(&x, sizeof(int), READWRITE, BreakMode, &MonitorX, &x, 0,1);
...
p = foo();        /* a bug: p points to x incorrectly */
*p = 5;           /* line A: a triggering access, BUG Detected! */
...
z = Array[x];   /* line B: a triggering access */
...
iWatcherOff(&x, sizeof(int), READWRITE, &MonitorX);

bool MonitorX(int *x, int value1, int value2) {
  return (*x==value2 || *x==value2);
}
```

The bug is caught at line A by iWatcher !

iWatcher Design Overview

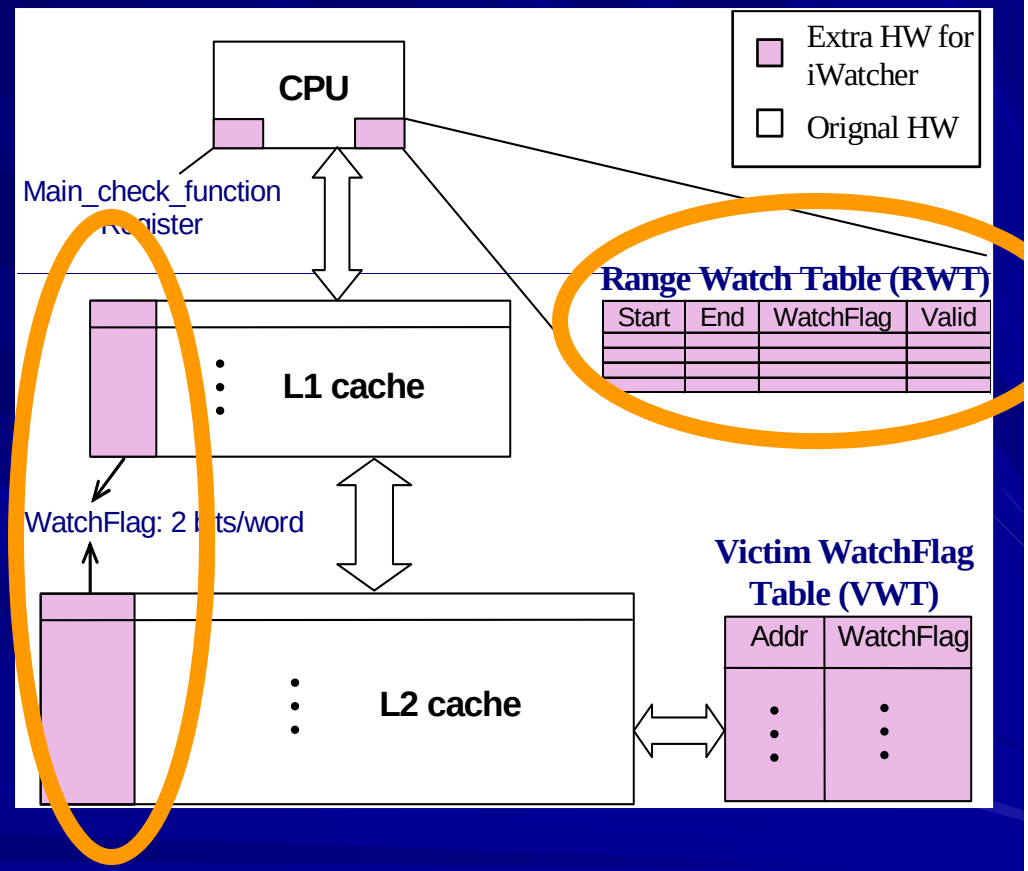
- Hardware:
  - Detecting triggering accesses
  - Triggering the main monitoring function
- Software
  - Manage associations between watched locations and monitoring functions
  - Call the appropriate monitoring function upon a triggering access

iWatcher Hardware Overview



Watch a Memory Region

- Hardware
  - Large region: allocate an RWT entry
  - Small region: set WatchFlags in caches
- Software
  - Add monitoring function info to check table

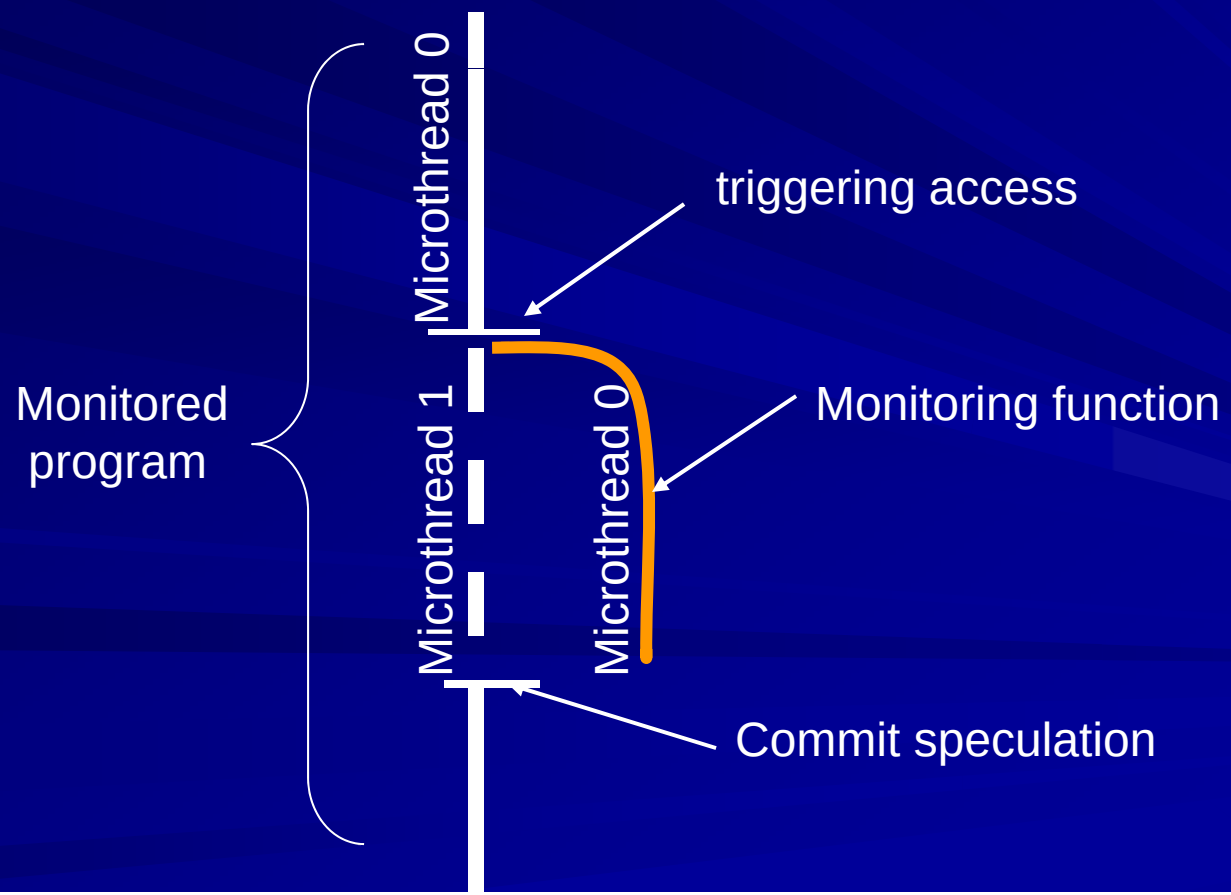


Detect Triggering Accesses

- When to detect?
  - Reads: during the read operation
  - Stores: during the pre-touch operation
- How to detect?
  - Checking RWT in parallel with TLB lookup
  - Checking WatchFlags in load/store queues
  - Checking WatchFlags in the caches
- When to trigger (executing the monitoring function)?
  - At the retirement of the triggering access
  - Use a *Trigger* bit in ROB

Executing Monitoring Functions

- Triggered by hardware and executed in parallel with main program by TLS
- Monitoring functions can make any side-effects



Different Reaction Modes

- Reactions when a monitoring function returns FALSE (indicating an error):
  - ReportMode: report the error and continue
  - BreakMode: pause right after the triggering access
  - RollbackMode: rollback to the most recent checkpoint (need checkpoint support)

Evaluation Methodology

- Execution-driven simulator
- Compare with Valgrind
  - A software debugging tool
  - We extend it to take options for fair comparison
    - Detecting memory-leaks only
    - Detecting buffer-overflows only
    - Detecting memory-corruptions only
    - Detecting all possible bugs

Evaluated Applications

App	Bug Class	Monitoring Type
gzip-STACK	stack smashing (security)	General (monitoring function is not program-specific)
gzip-MC	memory corruption	
gzip-BO1	dynamic buffer overflow	
gzip-ML	memory leak	
gzip-COMBO	combination of MC, BO1, ML	
gzip-BO2	static array overflow	
gzip-IV2	invariant violation	Program-specific
gzip-IV1	invariant violation	
cachelib-IV	invariant violation	
bc-1.03	outbound pointer	

Examples of Monitoring Functions

- Stack smashing
  - After entering a function: iWatcherOn() on the returnAddress
  - Before leaving a function: iWatcherOff() on the returnAddress
- Buffer overflow
  - Padding the two ends
  - iWatcherOn() on the two ends
- Accessing freed memory
  - iWatcherOn() on freed memory
- Memory leak
  - iWatcherOn() all dynamic memory objects
  - Any access updates the timestamp with the accessed object
- Uninitialized variable
  - iWatcherOn() on all variables
  - iWatcherOff() at first writes
- Invariant violation
  - iWatcherOn() on related memory locations and check for invariants

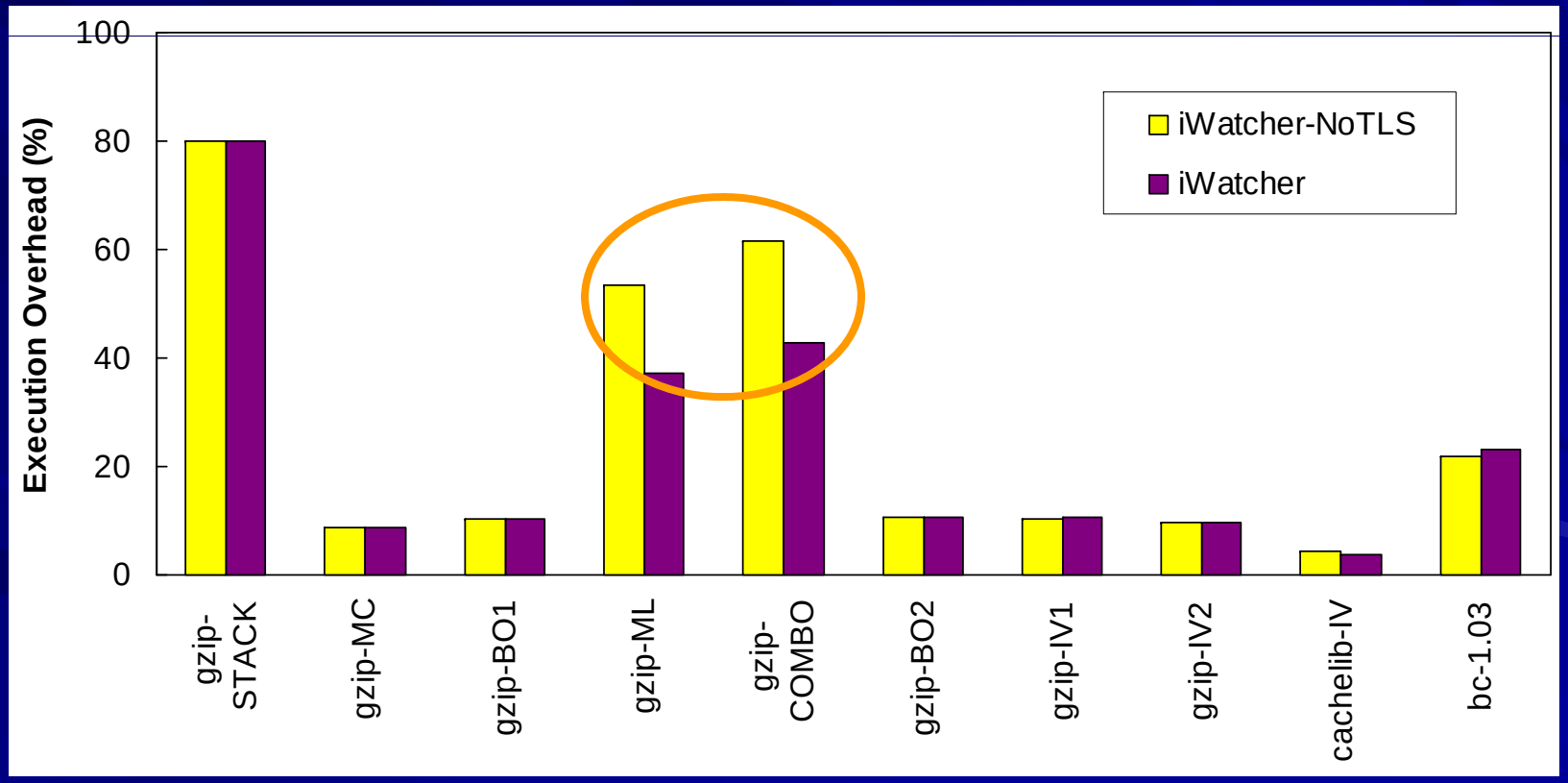
Overall Results

- iWatcher detects more bugs than Valgrind
- IWatcher adds only *4-80%* overhead (a factor of *25-169* smaller than Valgrind)

Application	Valgrind		iWatcher	
	Bug Detected?	Overhead (%)	Bug Detected?	Overhead (%)
gzip-STACK	No	-	Yes	80.0
gzip-MC	Yes	1466	Yes	8.7
gzip-BO1	Yes	1514	Yes	10.4
gzip-ML	Yes	936	Yes	37.1
gzip-COMBO	Yes	1650	Yes	42.7
gzip-BO2	No	-	Yes	10.5
gzip-IV1	No	-	Yes	10.5
gzip-IV2	No	-	Yes	9.6
cachelib-IV	No	-	Yes	3.8
bc-1.03	No	-	Yes	23.2

Benefits of TLS

- For applications with substantial monitoring, TLS is effective at reducing overheads
- Note: TLS is also used to support the rollback reaction mode



Summary: Advantages of iWatcher

- Provides location-controlled monitoring
  - Can detect “sneaky” accesses
- Low overhead (4-80%)
  - Only monitor true accesses
  - Trigger monitoring functions by hardware
  - Leverage TLS
- Flexible and extensible:
  - Program-specified monitoring functions
  - No need to instrument every places
  - Support three reaction modes
- Cross-module, Cross-developer, Language independent

Future Work

- Build automatic tools that use iWatcher to detect bugs
- Further reduce overhead in iWatcher
- Use iWatcher for
  - Profiling
  - Performance debugging