Contech Part 2

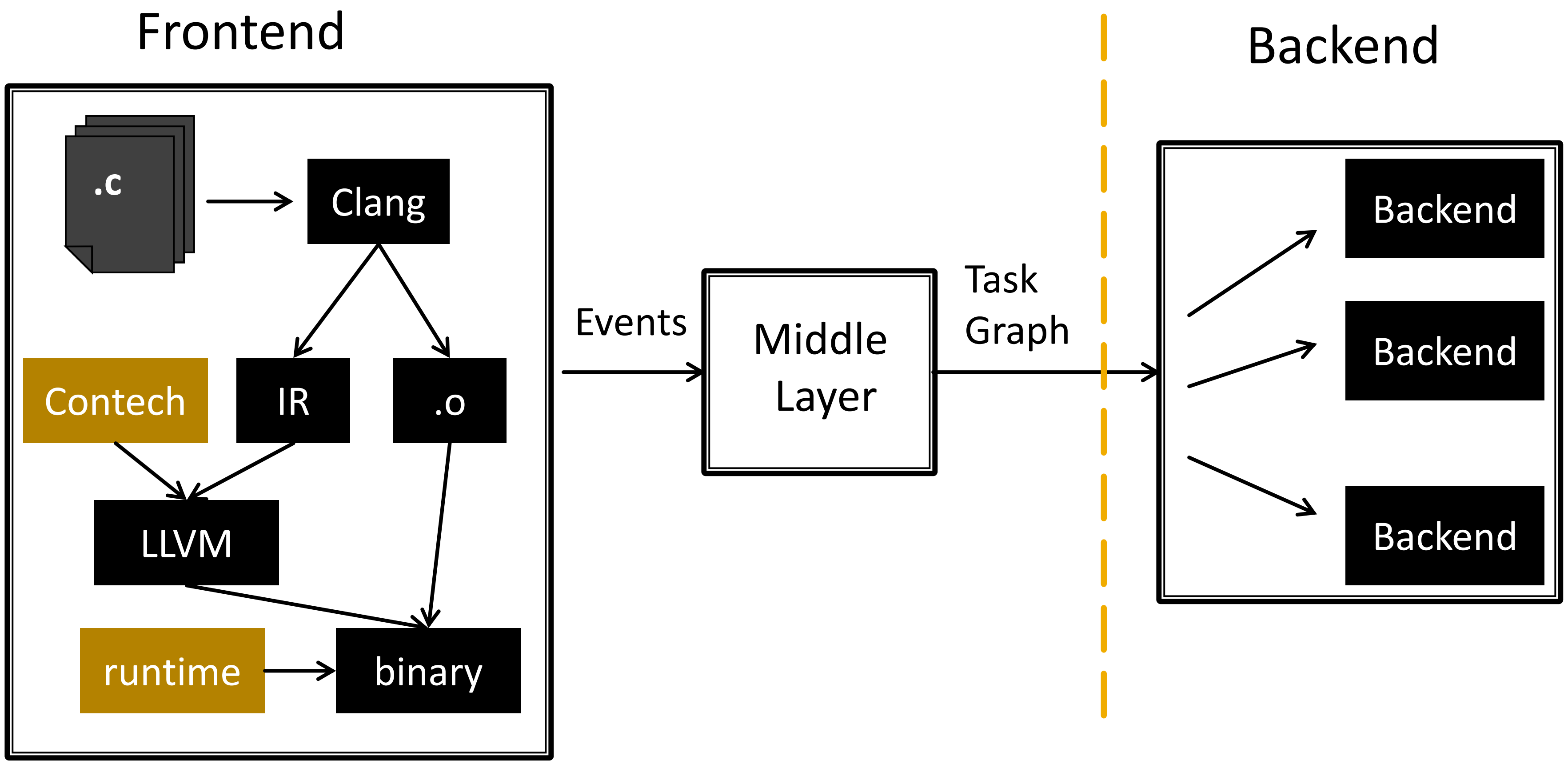
Outline

- Introduction
- Contech’s Task Graph Representation
- **Parallel Program Instrumentation**
- (Break)
- Analysis and Usage of a Contech Task Graph
- Hands-on Exercises

What is Contech's Instrumentation

- Contech is
  - An LLVM compiler pass to instrument programs
  - A runtime library to emit a trace from instrumented programs

Overview of Contech



Compiler Wrapper

- Pass the source file to the appropriate compiler
  - C -> clang
  - C++ -> clang++
  - Cilk -> clang-cilk
  - MPI -> Link in Contech MPI support
  - Fortram -> gfortran + DragonEgg
- Default clang compiler is assumed to have OpenMP support
  - http://clang-omp.github.io/

Compiler Wrapper cont.

- Clang emits an intermediate representation (IR)
- LLVM executes passes on the IR
  - Contech LLVM pass instruments IR of interest
- Link parallel program with Contech runtime

- Consequently, compile time is increased

Contech Runtime

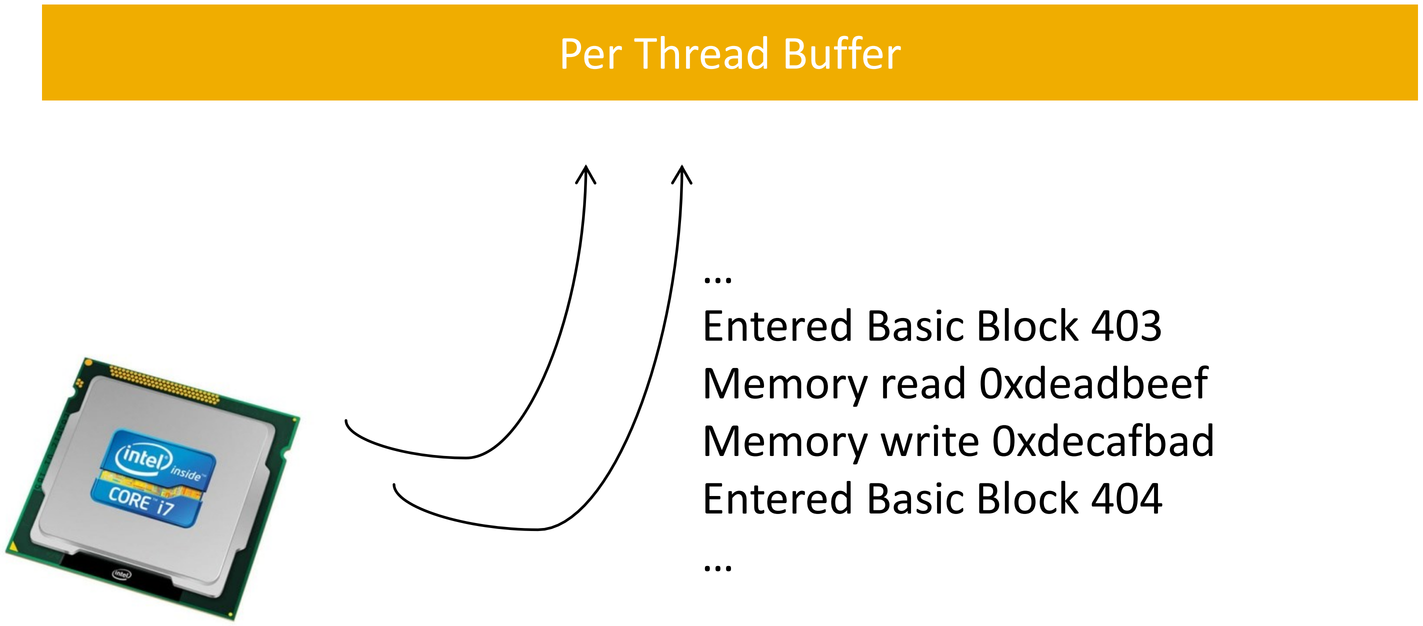
- Set of support routines for instrumentation
  - Linked into every instrumented program

- Many correspond to specific parallel routines or events
  - In lieu of modifying the parallel runtimes

Running a Parallel Program

- Parallel Programs emit events
  - Tens of millions per second per hardware context
  - Nearly all events are basic block events
    - Median of 20k per task across 40 parallel benchmarks
- Other events (15 in total):
  - Context create / created
  - Synchronization action
  - …

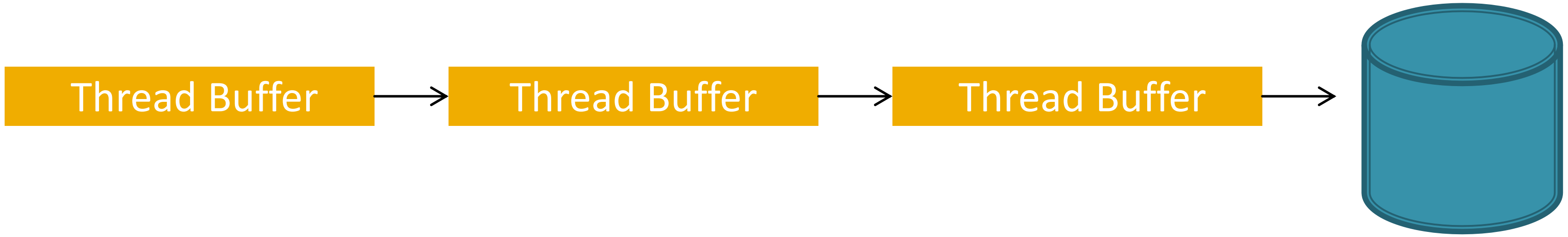
Event Collection



Per Thread Buffer

…
Entered Basic Block 403
Memory read 0xdeadbeef
Memory write 0xdecafbad
Entered Basic Block 404
…

Background writing



Middle Layer

- Does the following in parallel:
  - Consumes the events
  - Produces a Contech Task Graph

- Calculates a breadth-first traversal of the graph

Contech Features

- Tested Support for:
  - C, C++, Fortran
  - x86, ARM
  - PThreads, OpenMP, MPI, Cilk

Outline

- Parallel Program Instrumentation
  - **Instrumentation Design**
  - Generating a Task Graph
  - Performance Lessons Learned
  - Extending the Instrumentation

Instrumenting a Program

- Contech LLVM pass instruments IR of interest
  - Every basic block
  - Loads / Stores
  - Calls to functions of interest
    - Memory management (malloc, free, new, delete, memcpy, etc)
    - Pthreads (pthread_create, pthread_mutex_lock, etc)
    - OpenMP, …

Basic Block Normalization

- LLVM defines a basic block based on having ONE TerminatorInst
  - Function calls are not TerminatorInst
- Contech normalizes the basic blocks to consider function calls as terminating
  - Temporary transformation
  - Clang will restore / reoptimize the instrumented IR

Functions of Interest

- Each function is primarily identified by name
  - Names map to classifications
  - Each classification corresponds to a transformation approach

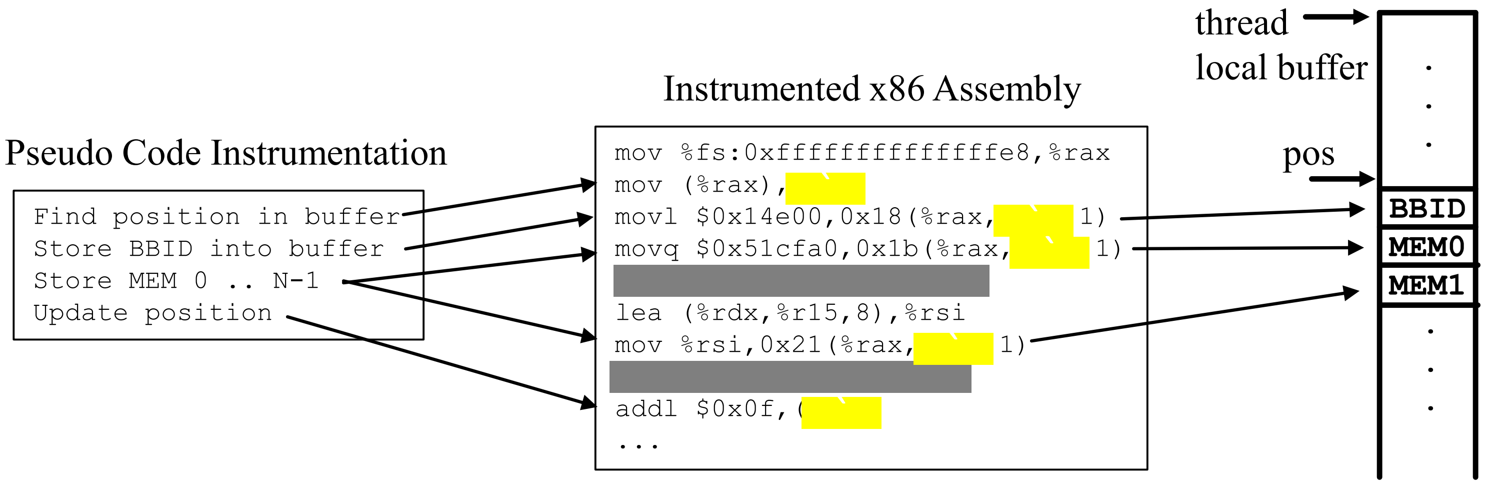
- SYNC_ACQUIRE:
  `pthread_mutex_lock, pthread_mutex_trylock, pthread_spin_lock, pthread_spin_trylock`

- BARRIER_WAIT:
  `pthread_barrier_wait, MPI_Barrier`

Instrumenting IR of Interest

- Call Contech instrumentation routines (~40 in number)
  - For example `__ctStoreBasicBlock(i32 474, i32 %bufPos3, i8* %bufPos2)`
  - Instrumentation written in C
    - Architecture independent (32- / 64-bit x86, 32-bit ARM)
  - Instrumentation routines are co-designed
- Use Clang’s link time optimizer (LTO)
  - Inline these calls into short assembly sequences

Instrumentation Design



Pseudo Code Instrumentation

```
Find position in buffer
Store BBID into buffer
Store MEM 0 .. N-1
Update position
```

Instrumented x86 Assembly

```
mov %fs:0xfffffffffffffe8,%rax
mov (%rax),
movl $0x14e00,0x18(%rax,        1)
movq $0x51cfa0,0x1b(%rax,        1)

lea (%rdx,%r15,8),%rsi
mov %rsi,0x21(%rax,        1)

addl $0x0f,(
...
```

Basic Block Instrumentation

- Prologue:

```
Buffer = __ctGetBuffer()
Buffer Position = __ctGetBufferPos()
fence singlethread acquire
*Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)
```

- Body:

```
__ctStoreMemOp(Addr, Number, *Buffer Position)
```

- Epilogue:

```
New Pos = __ctStoreBasicBlockComplete(Number of MemOps,
                                      Buffer Position, Buffer)
fence singlethread release
__ctCheckBufferSize(New Pos)
```

Aggressive Inling

- **Buffer = __ctGetBuffer()**

  ```
  mov %fs:0xffffffffffffffe8,%rax
  ```

- **Buffer Position = __ctGetBufferPos()**

  ```
  mov (%rax),%ecx
  ```

- **fence singlethread acquire**

  ```
  // Compiler directive
  ```

- ***Buffer Position = __ctStoreBasicBlock(BBID, Buffer Position, Buffer)**

  ```
  movl $0x14e00,0x18(%rax,%rcx,1)
  ```

- **__ctStoreMemOp(Addr, Number, *Buffer Position)**

  ```
  movq $0x51cfa0,0x1b(%rax,%rcx,1)
  ```

Compiler Shortcomings

- The compiler’s optimizations do not always align with the instrumentation architecture
  - The fence instructions prevent rare reorderings
  - Buffer and buffer position are passed between calls as the compiler would not apply common subexpression elimination to the calculations

Contech Statefile

- Contech instrumentation numbers basic blocks
  - Each basic block contains a static set of memory operations
  - Each memory operation has static properties:
    - Load / Store, Size
  - This information is stored in the statefile and included in the event trace
  - Used to reconstruct the events

Memory Operation Instrumentation

- Given the static properties, memory operations only store addresses
  - Some address calculations are static offsets of other calculations
  - Contech stores the offsets in the statefile
  - Elides the duplicate memory operation addresses
- Reduces trace size and lowers instrumentation overhead
  - Results discussed later today

Complex Instrumentation

- OpenMP – parallel regions
  - Each region is transformed into a function
  - OpenMP assigns threads to call the function
- Contech adds instrumentation into the caller and callee
  - Store create / join events into thread-local buffers
  - Assign and preserve the Context IDs

Complex Instrumentation cont.

- Cilk inlines much of its continuation management
  - Contech must detect not just a function, but a CFG signature indicating a cilk-spawn or cilk-sync
- (Almost) every cilk support routine can steal work

Outline

- Parallel Program Instrumentation
    - Instrumentation Design
    - **Generating a Task Graph**
    - Performance Lessons Learned
    - Extending the Instrumentation

From Instrumentation to Contech Task Graph

- Contech is part of program startup
- Instrumented program generates millions to billions of events
- Contech delays the program’s shutdown to finish writing out events

- Middle layer reads event list and generates a task graph

Instrumented Program Startup

- When the instrumented program launches, Contech will:
  - Initialize its internal structures
    - Create the first thread-local buffer
    - Determine its memory limit
  - Spawn the background writing thread
  - Transfer control to the original program

Instrumented Program Shutdown

- Contech must trap calls to exit
  - Ensure that all threads have terminated
  - All thread local buffers have been written to disk
  - Program will now exit

Thread-local Buffers

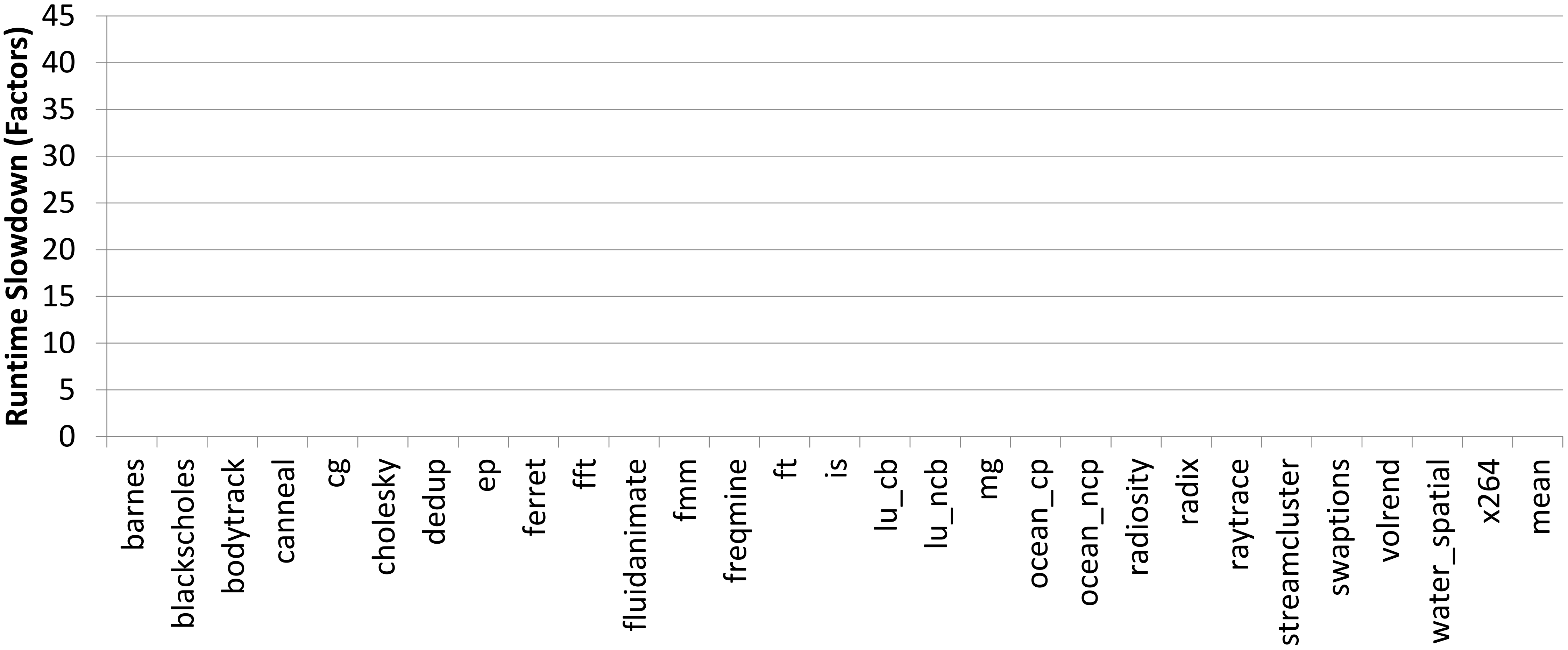
- Each thread has its own buffer, using thread-local storage
  - Technically, buffers are Context local
    - Threads that switch Context IDs (Cilk and OpenMP) refresh their buffers
  - Events are written into the buffer and the buffer position updated

- Buffers are queued into a global queue for writing to disk

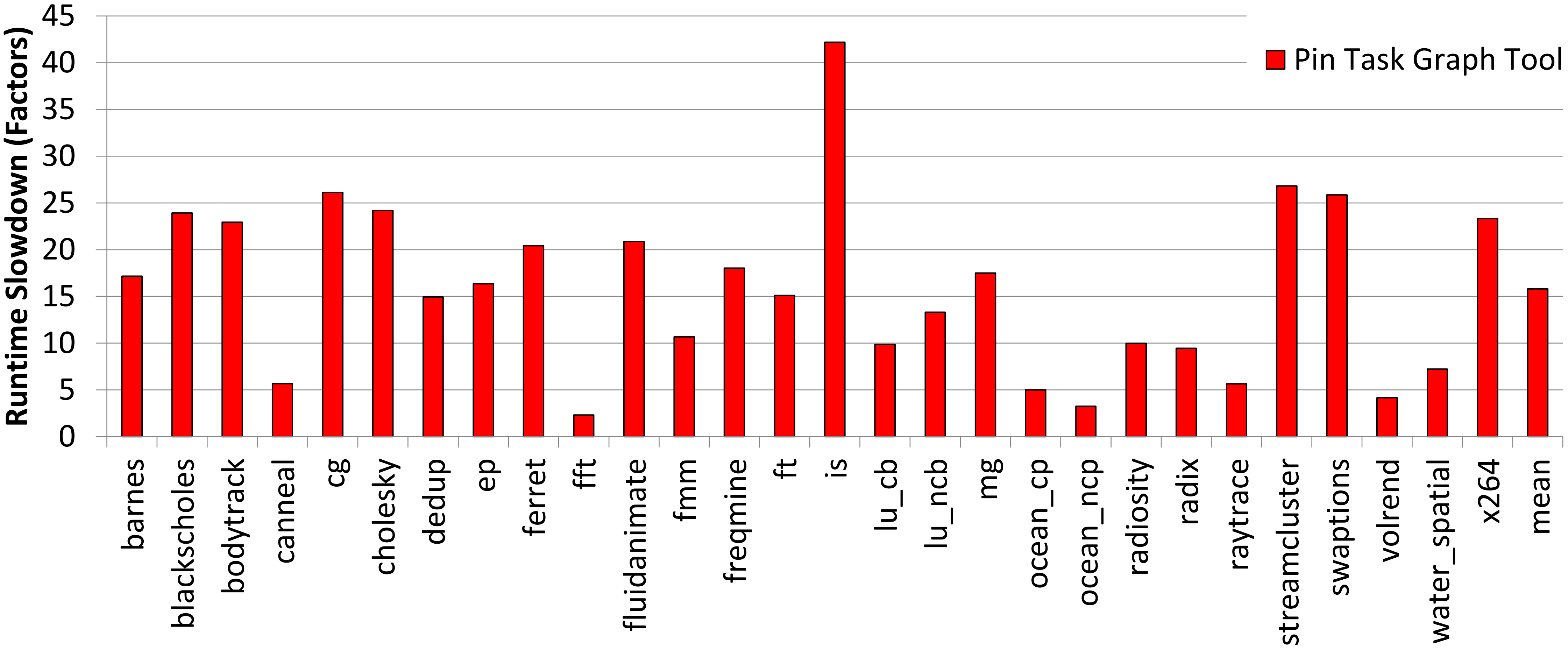
Buffer Overflow Checks

- Placed by the compiler pass
    - Follows a heuristic
    - Large basic blocks are always checked
- Each check verifies that at least 1KB of space is available
    - Events do not check for space

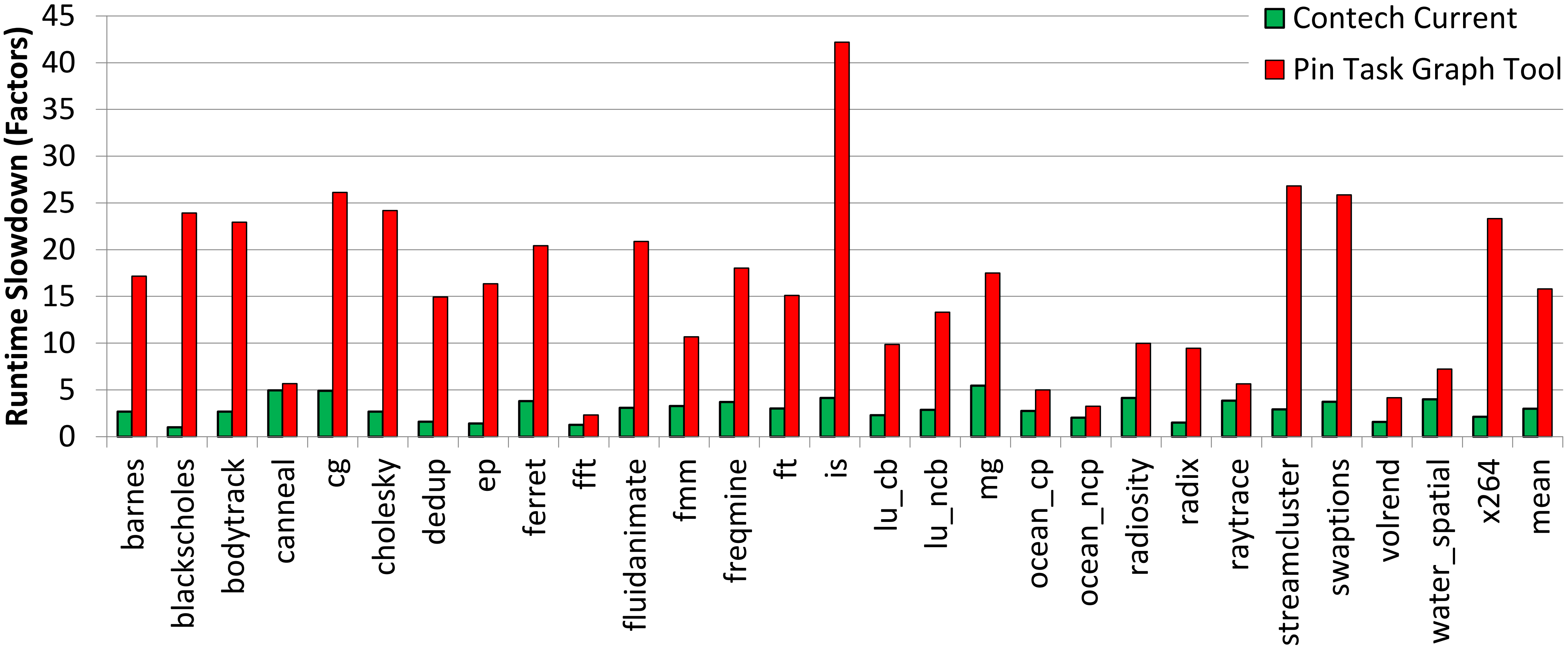
Slowdown (CPU Overhead)



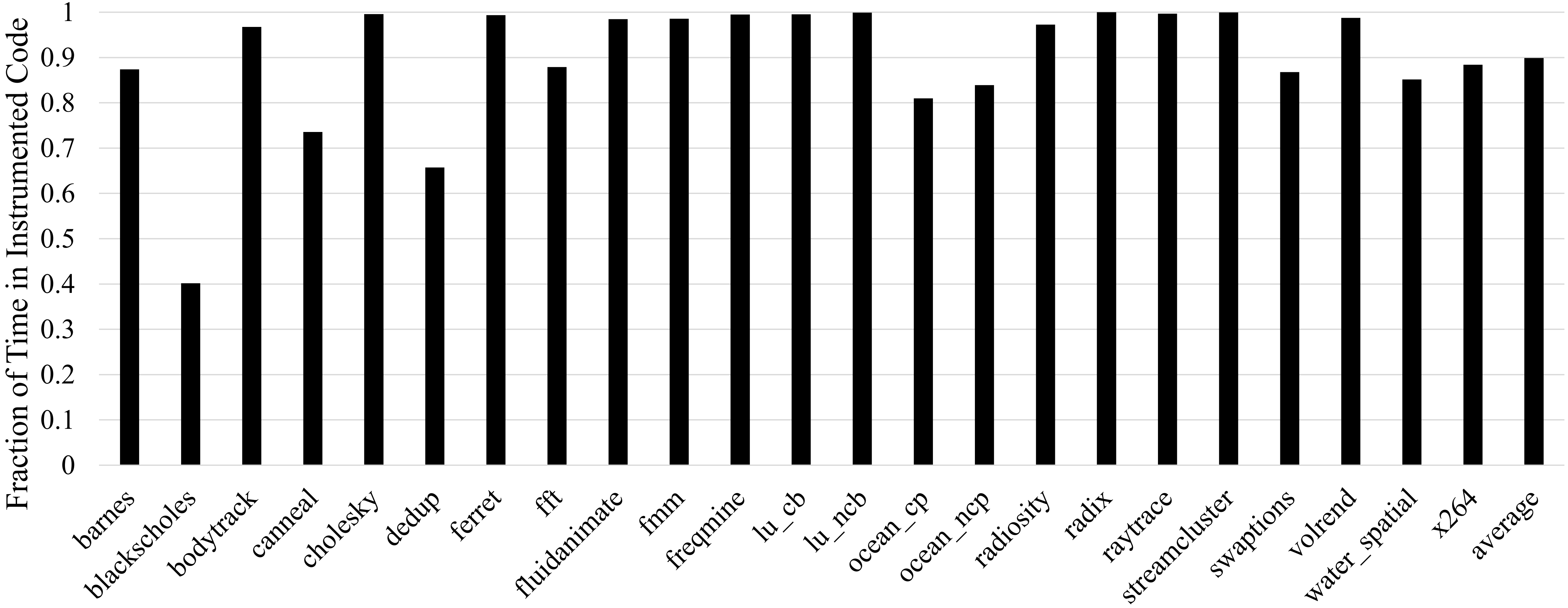
Slowdown (CPU Overhead)



Slowdown (CPU Overhead)



Code Coverage



Outline

- Parallel Program Instrumentation
  - Instrumentation Design
  - Generating a Task Graph
  - **Performance Lessons Learned**
  - Extending the Instrumentation

Contech’s Overhead and Mitigations

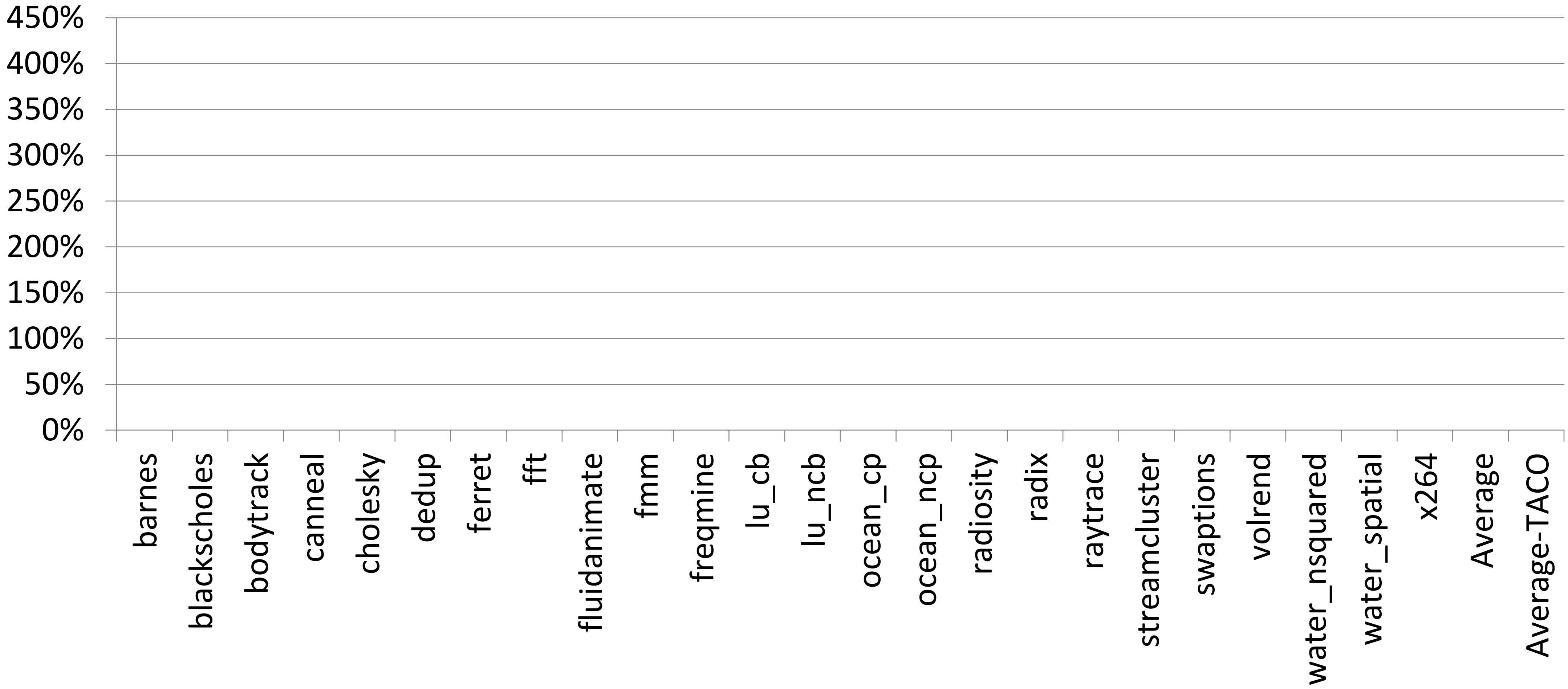
- Benchmark Overhead:

PARSEC + SPLASH:	2.80x
NAS:	3.79x
Rodinia:	2.70x

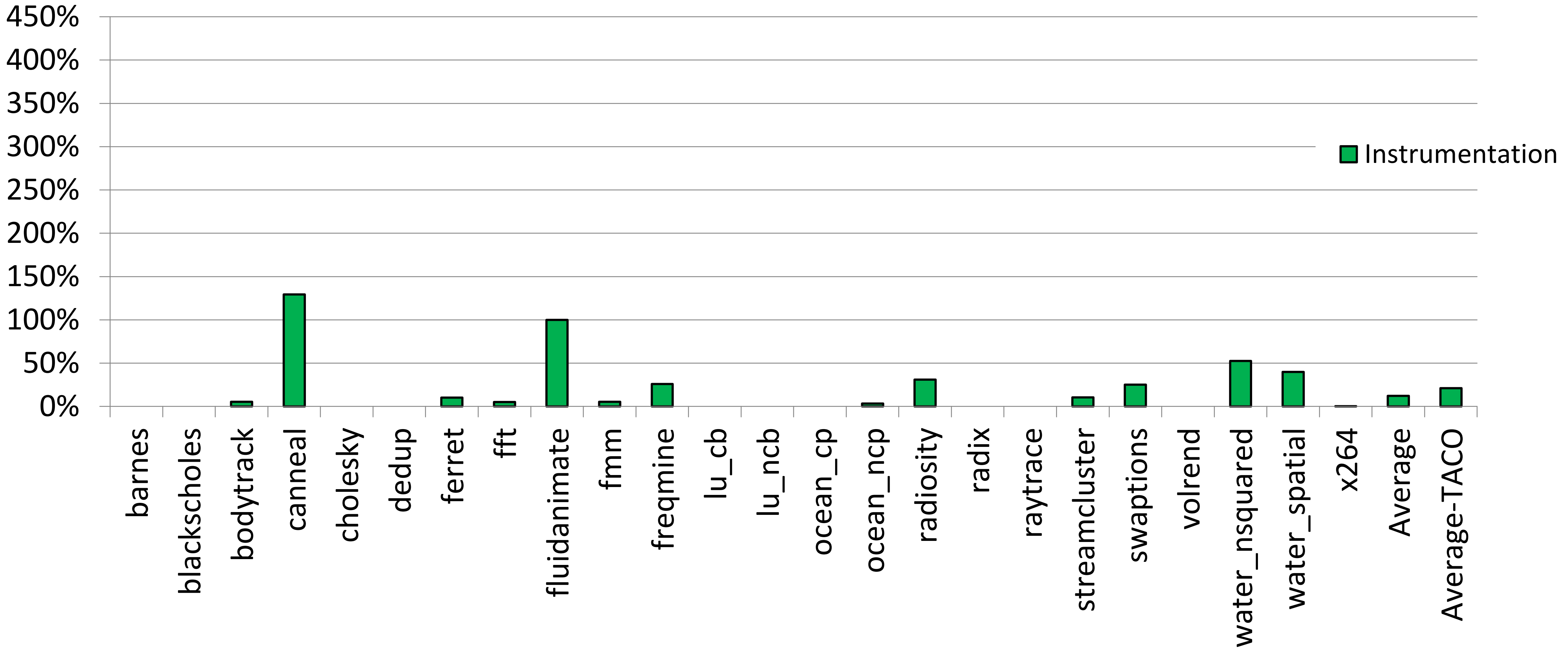
- Exceptions not included:

Water_nsquared:	8.9x
Lulesh:	10x

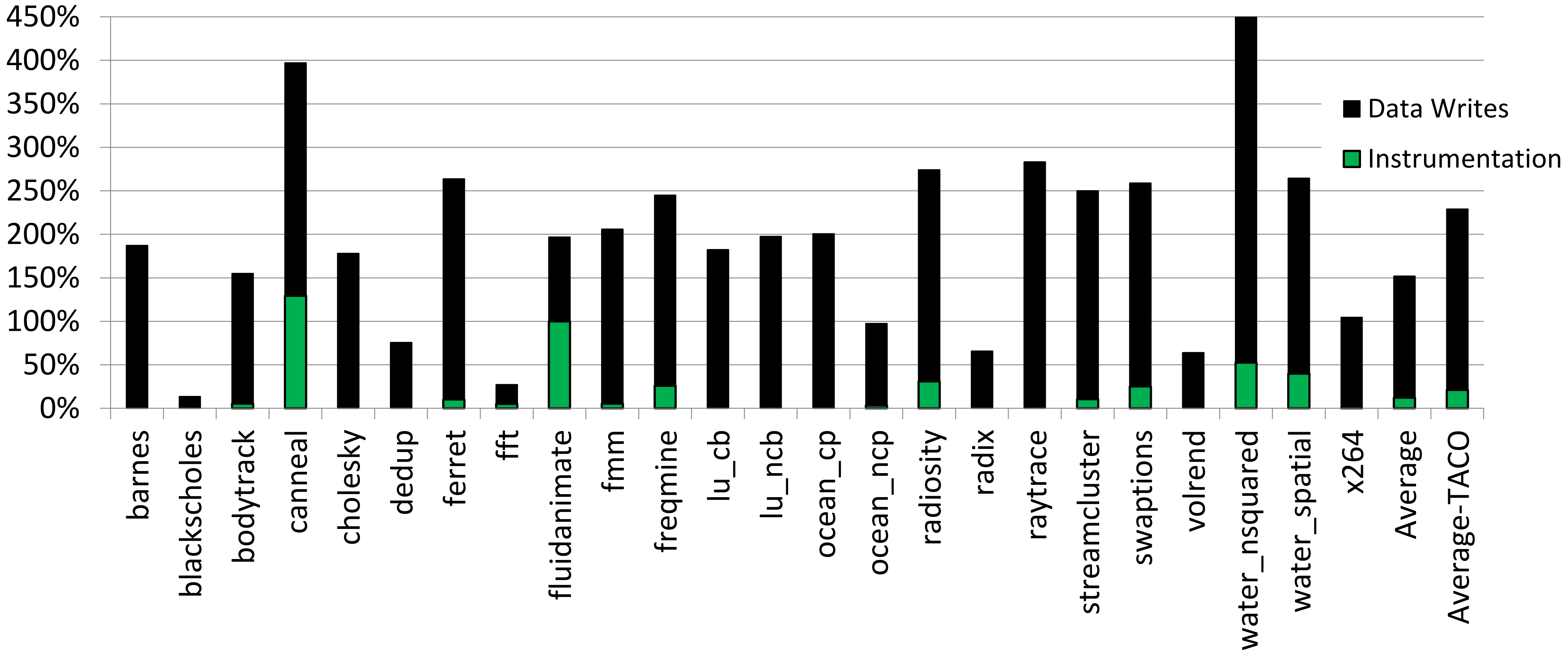
Instrumentation Overhead



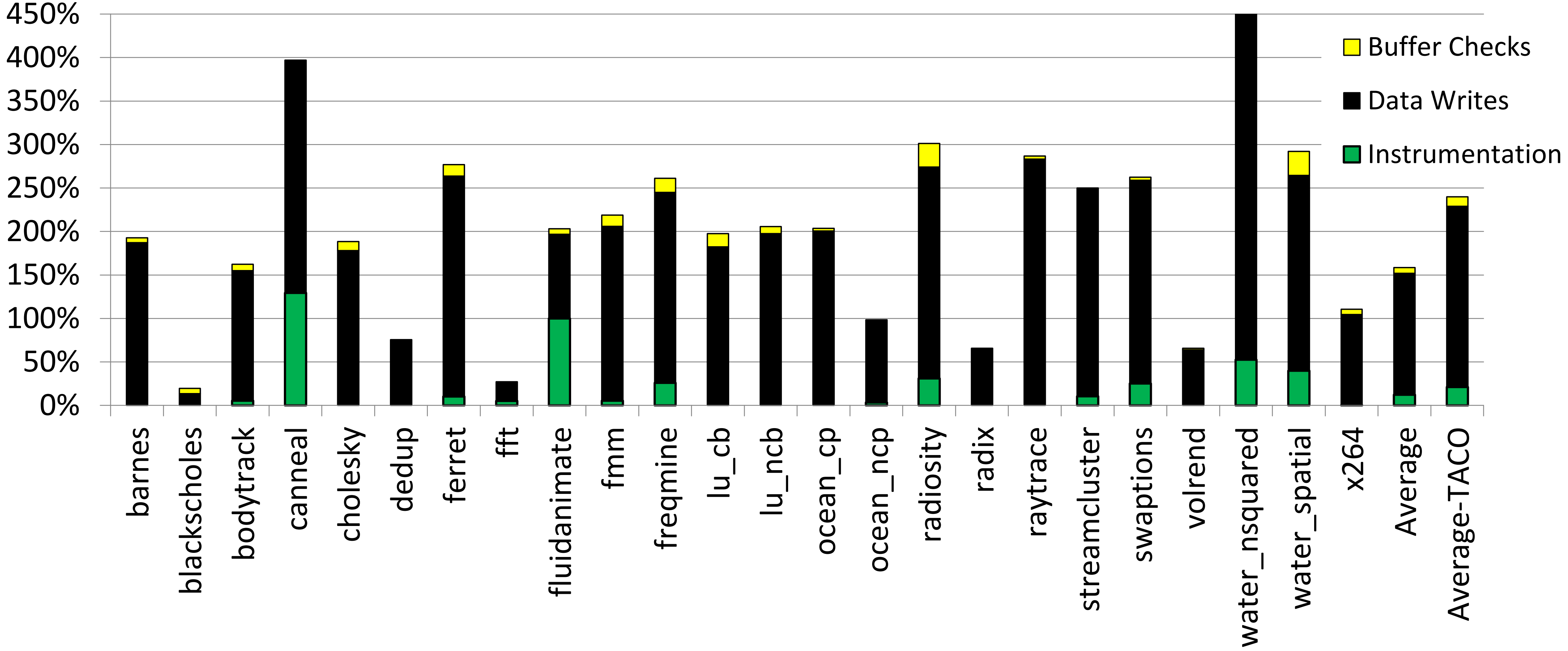
Instrumentation Overhead



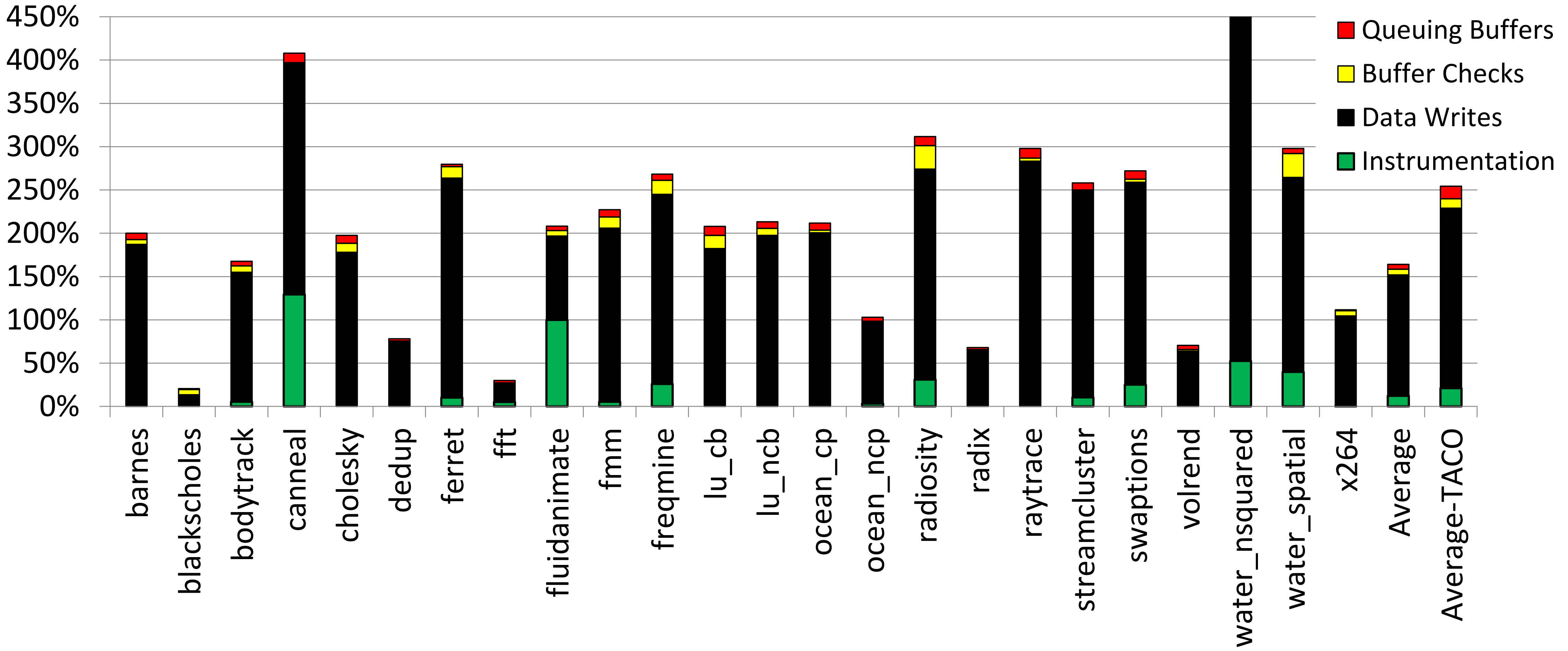
Instrumentation Overhead



Instrumentation Overhead



Instrumentation Overhead



Major Overheads Summarized

- Instrumentation
  - (see compiler section)
- Quantity of Data Generated
  - Compact Basic Block IDs
  - 6 Byte Memory Addresses
  - Redundant Memory Addresses
- Queuing and Allocating Buffers
  - Synchronization and Barrier Tickets
  - Small Buffer Copy

Compact Basic Block Event

- Basic Block IDs are 23-bit values
  - First byte identifies the event type
    - If high bit is 0, then a basic block event and remaining bits are part of ID
    - Else, one of the 14 other event types
- Virtual Memory Addresses are 6-byte values
  - Given the endianness, overlap writing the addresses

```
00 01 02 03 04 05  00 01 02 03 04 05  00 01 02 03 04 05 06 07
```

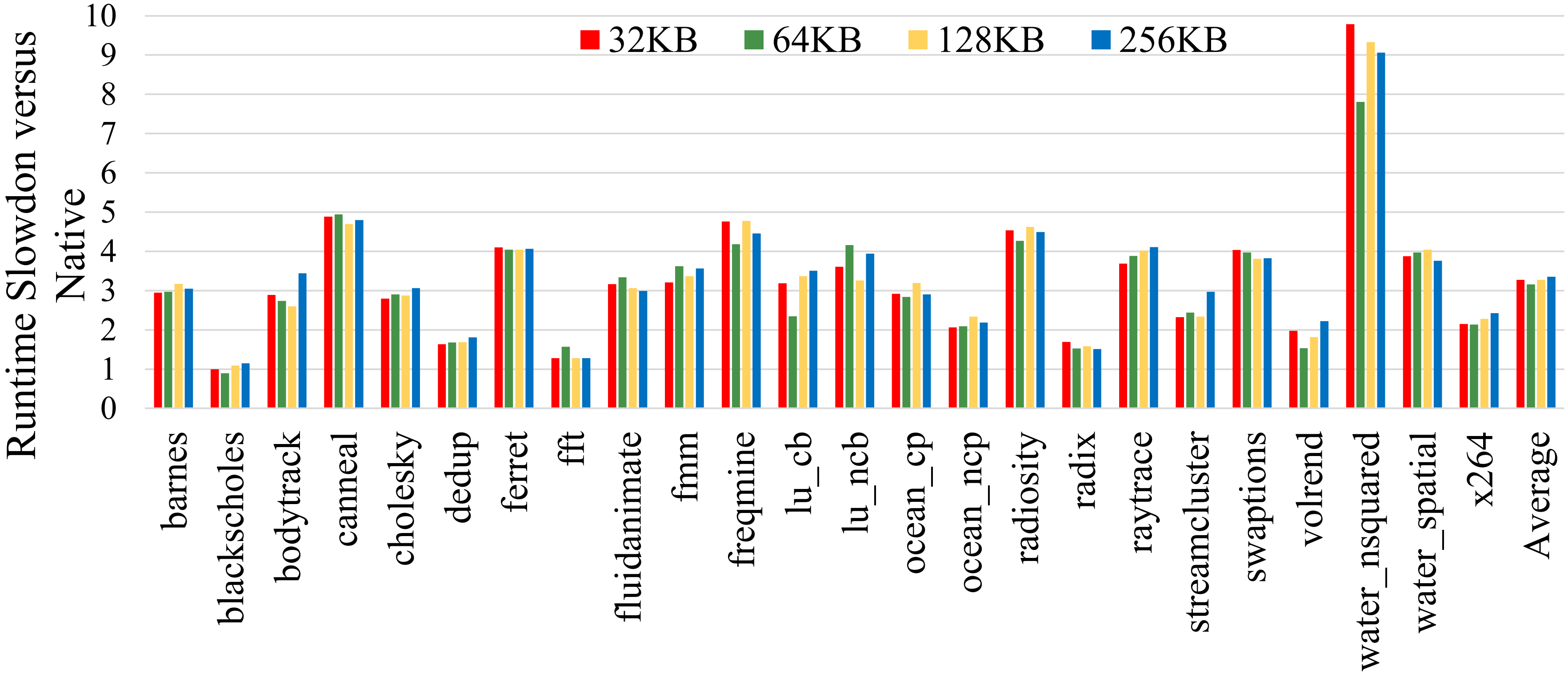
Synchronization and Barrier Tickets

- Syncs, and Barrier events have certain ordering requirements
  - Originally queued to ensure the ordering
  - Don’t queue, instead place an ordering identifier (aka, a ticket) into the events

- Ordering information used by middle layer to associate events from different Contexts with each other

Small Buffer Copy

- Some actions still require buffers to be queued early
  - Rather than allocate a new 1MB buffer, copy the data into a smaller-sized buffer



Lessons Learned

- Generating GB/s is expensive
  - Identifying static redundancies is vital
- Communication and Allocation costs are low
- Inlined instrumentation for minimal perturbation
  - Co-designed with the compiler for improved code generation

Outline

- Parallel Program Instrumentation
  - Instrumentation Design
  - Generating a Task Graph
  - Performance Lessons Learned
  - **Extending the Instrumentation**

Extending the Instrumentation

- Requires knowledge of LLVM
- Various levels of extension
  - Alternate Support Routine (e.g., custom allocator or lock)
  - Custom event
  - New Parallelism APIs (beyond today’s scope)

Adding a Routine

($CONTECH_HOME/llvm/lib/Transforms/Contech/Contech.cpp)

- Table of functions to instrument
  - Add new routine name into table
  - Increment size of table
  - Potentially add new type (SYNC_ACQUIRE, etc)

```
#define FUNCTIONS_INSTRUMENT_SIZE 57
llvm_function_map functionsInstrument[FUNCTIONS_INSTRUMENT_SIZE] = {
               {STORE_AND_LEN("main\0"), MAIN},
               {STORE_AND_LEN("MAIN__\0"), MAIN},
               {STORE_AND_LEN("pthread_create"), THREAD_CREATE},
               {STORE_AND_LEN("pthread_join"), THREAD_JOIN},
               {STORE_AND_LEN("pthread_barrier_wait"), BARRIER_WAIT},
               {STORE_AND_LEN("pthread_mutex_lock"), SYNC_ACQUIRE},
...
```

Custom Event

- Add function to table in Contech.cpp
- Add type and handler in Contech.h
- Event Serializer in ct_runtime.c
  - Add hook to serialization routine in _ConstantsCT in Contech.h
  - Initialize routine constant in Contech.cpp
- Event Type in ct_event_st.h
- Deserialization in ct_event.cpp
- Handle event in middle layer