CHAPERONE

RUNTIME SYSTEM FOR TRACKING AND MANAGING RUNNING APPLICATIONS VIA PARTIAL BINARY INSTRUMENTATION

GADI HABER – INTEL, HAIFA DEVELOPMENT CENTER

OPHIR ZISKIND – CS, TECHNION

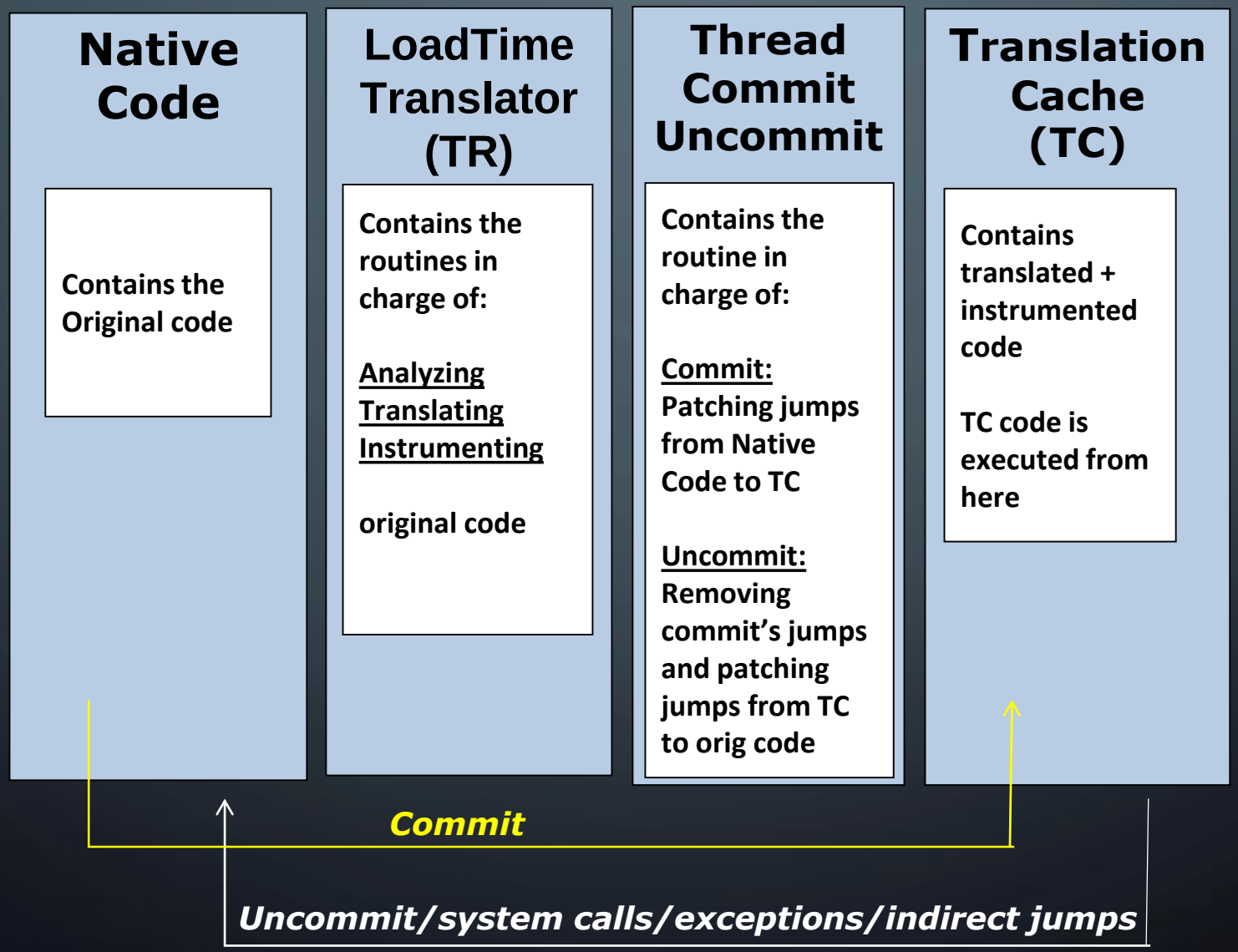
PROBLEM STATEMENT

- In the future, compiling a program and running it as standalone will not suffice:
  - Platforms are becoming complex with many cores and multiple Heterogeneous PEs
  - Running applications are prune to attacks
  - Over time, running applications can degrade performance and increase power
    - memory leaks
    - suboptimal load balancing
    - changing availability of resources in the system

WHAT IS CHAPERONE?

- CHAPERONE is platform for implementing efficient runtime subsystems via binary instrumentation
- Can be used for efficient monitoring and tracking of a running application
- CHAPERONE uses partial binary translation to switch between instrumented code and the original code while application is running
- CHAPERONE technology can be used for a variety of usages:
  - Collect profiling statistics which require code instrumentation
  - Identify cases of illegal memory accesses
  - Identify and alerting attacks
  - Identify illegal/unexpected behavior of the application
  - Apply fault recovery when possible
  - Alerting when entering rarely executed code area
  - Tuning applications resources at runtime for improved load balancing

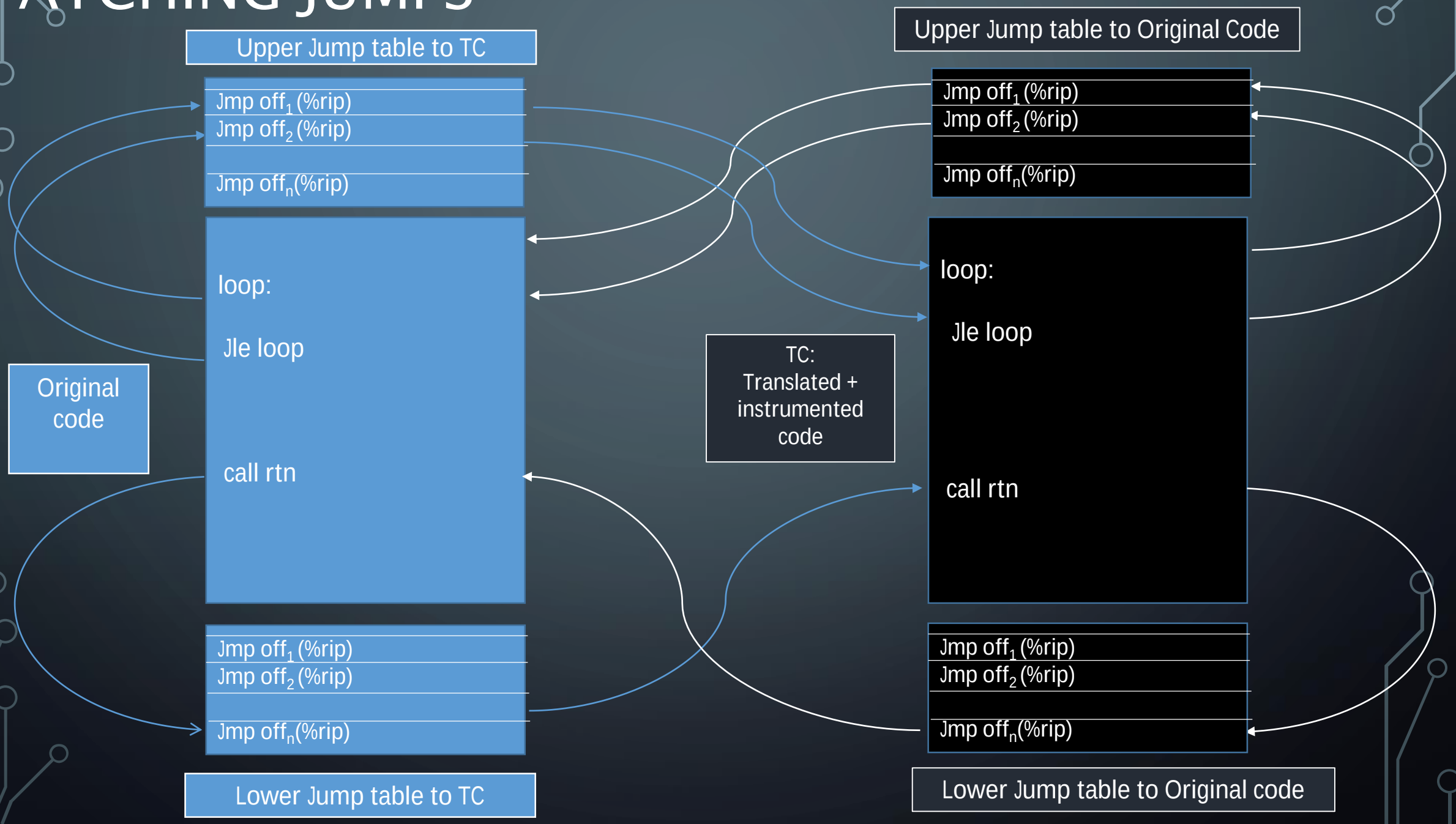
CHAPERONE – MAIN LAYOUT



TRANSLATOR (TR) TASKS

- The Translator contains all the analysis +instrumentation routines
- It applies the following steps at load time just before starting the application's execution:
  1. Checking execution environment
  2. Allocating required memory
  3. Analyzing original code using incremental disassembly following control flow
  4. Generating translated code along with instrumentation stubs
  5. Chaining targets of direct jumps and calls in data structure
  6. Encoding translated code + populating the patching list & jump tables
  7. Create and kick start the Commit-Uncommit thread

PATCHING JUMPS

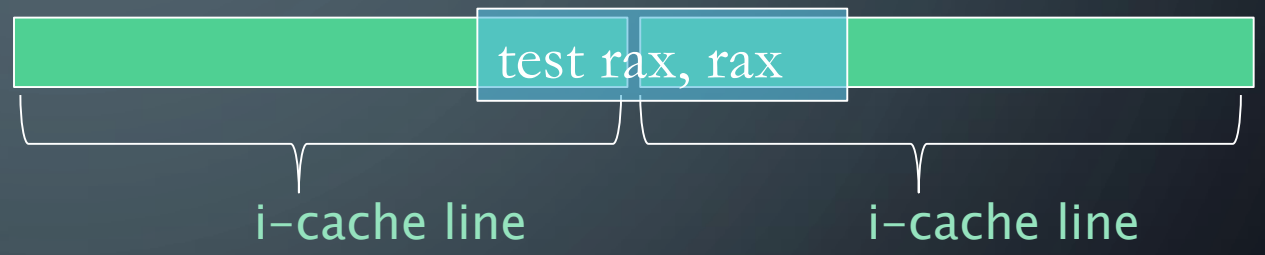


THE COMMIT/UNCOMMIT THREAD – MAIN FLOW

```
void commit_uncommit_thread(void *v)
{
     // Busy wait until translation completes:
     while (!enable_commit_uncommit_flag);

    sleep(1); // wait for 1ms before starting the 1st commit

    while (true) {
        // Commit translated code:
        commit_translated_code();

        // Wait 1ms for the main application to kick start.
        sleep(1);

        // Uncommit translated code:
        uncommit_translated_code();

        // Translated code is uncommitted. Original code is now running
         // wait a longer interval of 10ms before applying the commit again:
        sleep(10);
    }
  }
```

SELECTING INSTRUCTIONS FOR PATCHING

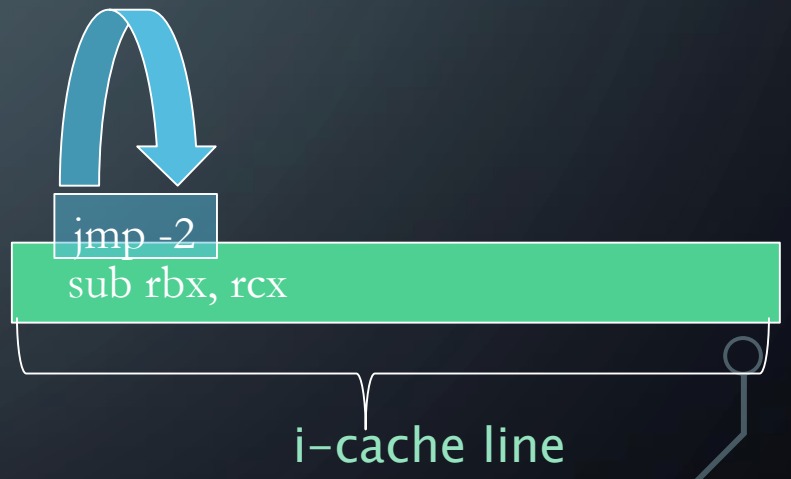
- The following instructions' addresses are added to the patching list:
  - Address of every direct backward jump ≥ 5 bytes not crossing i-$ line
    - Patching direct jump instr < 5 bytes cannot guarantee that the jump scope will reach the jump table. (5 bytes cover ± 2GB scope)
    - Patching memory that crosses i-$ line cannot be done atomically
    - Patching multiple instructions cannot be done atomically either



  - Address of every direct call instruction (not crossing i-$ line)
    - Direct call instructions in x86 = 5 bytes long.
  - For backward jumps < 5 bytes (or crossing an i-$ line):
    - Address of its target if the instruction in the target ≥ 5 bytes
    - If target instruction < 5 bytes then check the proceeding instruction candidate
    - (Very rare for all instructions in the loop's basic blocks < 5bytes)

SAFE CODE PATCHING AT RUNTIME

- Code patching is done atomically at runtime while the application is running:
  - Instructions up to 8 bytes (not crossing i-$ line) are patched using a single x86 *mov* instruction which is known to be atomic in x86
  - Instructions larger than 8 bytes (or crossing i-$ line) are patched in 3 steps:
    1. First 2 bytes are overwritten by a direct jump to itself:
       "jmp -2" = 0x eb fe
    2. Rest of the bytes are overwritten with the corresponding bytes of the patching bytes
    3. First 2 bytes are replaced by the corresponding patching bytes



THE CODE PATCHING PHASE

- The patching phase is done by overwriting the new instructions over the existing running code in 3 stages:
  1. Code to be patched is saved into a patching map located at the TR static memory.
  2. Target of the code to be patched is restored back from the patching map.
  3. All direct jumps to the jump tables are written on the instructions in the patching list.

MEMORY CHECKER CHAPERONE

- Checks out-of-bounds memory accesses in allocated heap memory
  - Occur when the application

    dynamically allocates a block of memory,

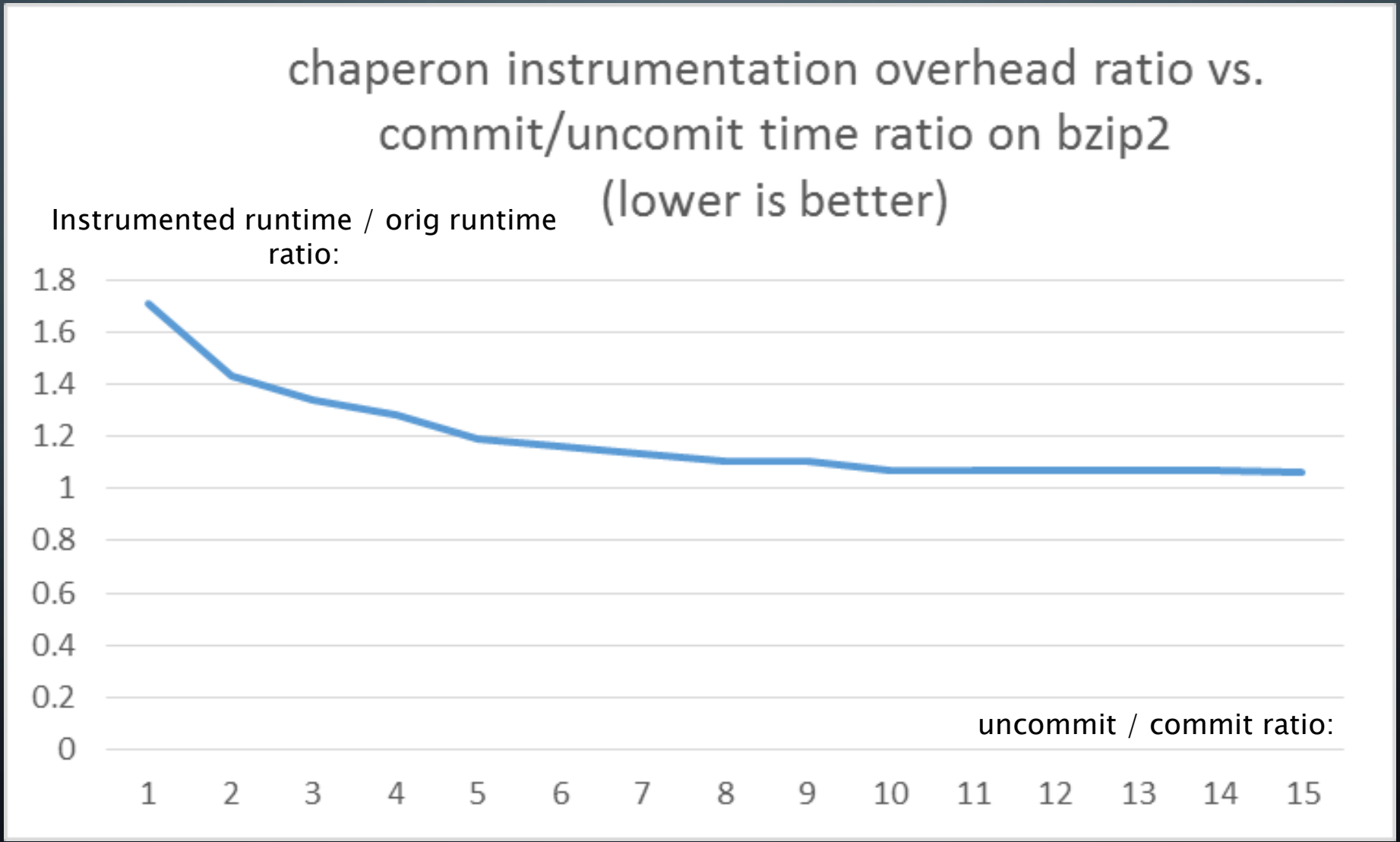
    iterates on the allocated block

    falsely accesses memory outside the bounds of the originally allocated memory

MEMORY CHECKER CHAPERONE – METHOD

- Instrumenting every call to *malloc* and *free*
  - On each call to *malloc:* additional wrapping memory allocation is made before and after the requested allocated memory area
  - On each call to *free:* the wrapping memory is released in addition to freeing requested memory.
- Instrumenting **every** instruction that references the memory by a call to an instrumenting routine that checks if the memory address falls in the wrapping area
- Thread committer/uncomitter is set to a ratio of 1ms in commit and 10 ms in uncommit

MEMORY CHECKER CHAPERONE PERFORMANCE RESULTS ON I7



OMP TUNER CHAPERONE

- Dynamically tunes the running application for the optimal number of OMP threads for reduced power and best performance and pins them to the CPUs.
- CHAPERONE
  1. Dynamically tries out different number of OMP threads
  2. Based on measured elapsed time, converges to optimal number of OMP threads
  3. Sets threads affinity based on CPUs availability and the state of each thread

OMP TUNER CHAPERONE – METHOD

- At init image load time, OMP Tuner:
  1. Retrieves current_threads_num – current num of threads using omp_get_max_threads() OMP api
  2. Retrieves available_processors using get_nprocs() system call.
- At the beginning of each omp parallel for:
  1. Checks if there is a recommended num of OMP threads optimal_num_threads that differs from current omp threads num
     If so, it modifies it using omp API: omp_set_num_threads()
  2. Starts a clock time measurement for the real_time_interval – elapsed time interval
- At the end of each omp parallel for:
  1. Stops the clock time measurement of the above time interval.
  2. Calculates average num of omp threads for all measured intervals executed with current_threads_num so far. Maintains result in: avg_real_time[current_threads_num]
  3. Searches and sets optimal_num_threads based on avg_real_time[]
     Allowing 10% perf penalty in favor of reduced power

CALCULATING RECOMMENDED NUM OF THREADS

```
// Loop searches for optimal number of threads by applying logartmic search:

double min_real_time_interval = avg_real_time[current_threads_num];

int optimal_num_threads = current_threads_num;

for (i = available_processors; i >= 2; i/=2) {

        if (min_real_time_interval  * perf_penalty_for_power  >= avg_real_time[i]) {

                min_real_time_interval = avg_real_time[i];

                optimal_num_threads = i;

        }

}
```

```
// search sequentially for further reduced number of threads for reduced power consumption:

int optimal_num_threads_for_reduced_power = optimal_num_threads;

for(i=optimal_num_threads-1; i > optimal_num_threads/2; i--){
  if(avg_real_time[i] <= min_real_time_interval * omplb_perf_penalty_for_power){
     optimal_num_threads_for_reduced_power=i;
  }
}
optimal_num_threads = optimal_num_threads_for_reduced_power;
```

CHAPERONE OMP TUNER RESULTS ON I7 FOR NASA NPB 3.1 OMP