Chaperone - Runtime System for Instrumenting Applications via Partial Binary Translation

Gadi Haber
Intel
Haifa, Israel
gadi.haber@intel.com

Coby Tayree
Intel
Haifa, Israel
coby.tayree@intel.com

ABSTRACT

We propose Chaperone, a new runtime system which implements an innovative binary-level instrumentation technique that bounds performance degradation of around 14% on average of the total applications execution time, hence it can be used by users in production deployments. Our technique is called Partial Binary Translation. It consists of translating the entire code image and inserting required instrumenting stubs in the translated code at loadtime, and at runtime, differently from normal instrumentation, switching the execution dynamically between original and instrumented code. Using this technology we demonstrate two different use cases of the Chaperone system. The first is a runtime memory checker that detects memory accesses outside the legally allocated heap memory bounds. The second is a dynamic performance tuning of OpenMP applications by automatically setting the optimal number of active OpenMP threads and affinitizing them to the available cores. In both cases we measured low overhead of Chaperone. Moreover, Chaperone shows minimal performance degradation for the Memory Checker and improved performance for OpenMP based applications.

1 INTRODUCTION

In the future, we predict that most applications will be run under the management and the tracking of some runtime system. Managed applications written in Java and .NET are running today under a runtime system and already benefit from useful runtime services such as the memory garbage collection, profiling services and even runtime optimizations. Programming frameworks such as OpenMP or OpenCL already include a runtime library that manages the distribution of tasks to the different processing units. The LLVM [10] compilation strategy is designed to enable staged optimizations starting from compile-time through link-time and finally at runtime by supporting profile-driven optimizations. However, applications are becoming more and more complex and so are the requirements from these runtime systems along with the requirement to not cause any noticeable performance regression while monitoring the execution. A subsystem that is able to efficiently monitor and trace a running application without degrading its performance will be able to provide new valuable functionalities. It will able to inform the user about issues during the application execution such as exceeding legal memory bounds used by hacking systems to attack the application. It will be able to detect and warn when entering inefficient computation phases such as long busy wait loops, or alternatively entering rarely executed code areas indicating some possible new untested input scenario was provided. It will also be able to record history for post-processing analysis, actively apply performance tuning or even apply fault recovery when possible. Consequently, applications will be less vulnerable to attacks, more fault tolerant and more energy efficient.

In this work we introduce the Chaperone system that uses static binary translation to generate a translation of the entire executable code that resides in a specific memory area called Translation Cache (TC). Chaperone implements Partial Binary Translation that enables runtime binary instrumentation with minimal overhead. It performs the complete analysis of the executable code once at loadtime and the generates the translated code along with the appropriate instrumentation stubs.

In the next sections we describe the design of the Chaperone system and its components followed by a description of two different use cases of the Chaperone technology implemented in two different ways. The first is a memory checker tool that detects illegal memory accesses at runtime implemented using the Intel Pin

SYSTOR, 2018, Haifa, Israel

ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

SDK [6] that provides convenient interface to instrument executables at runtime. The second use case is a runtime tuning subsystem for OpenMP applications. In here we used the *LD_PRELOAD* Linux mechanism to preload the TR library and run its main binary translation routine before the application's libraries are loaded and run.

2 DESIGN

The Chaperone introduces a technology called Partial Binary Translation (PBT) in which the execution is diverted from the original code to the translated/instrumented code at runtime for only part of the time before returning execution back to the original code. In PBT, both the original code and the translated code execute periodically, as opposed to regular Binary Translation [7] where only the translated code or the original code is executed for a given code area throughout the application run. To enable the PBT mode, the Chaperone system, creates a thread (in addition to the applications thread), which is in charge of periodically patching the original code with direct jumps to a jump table consisting of indirect jumps to the corresponding locations at the translated code and vice versa, i.e., translated code is patched by returning jumps to corresponding original code sites via the indirect jump tables. We refer to the process of injecting direct jumps to the translated code as the *Commit* step and the opposite process of inserting jumps back from translated code to original code as the *Uncommit* step. In both cases, the application continues to run while the jump instructions are patched atomically to the code.

Figure 1 shows the memory organization of the Chaperone system allocated at load time. The memory area on the left is the original image of the application comprising of the binary code, its static data and its dynamically linked libraries. The next memory area from left is the TR's memory containing all the routines in charge of analyzing, translating and instrumenting the original code and placing it in the TC on the right. The next memory area from the left is used by the thread committer-uncommitter (TCU) containing the code of the routines in charge of the commit and uncommit steps. All translated and instrumented code are executed from the TC shown on the right. The arrows at the bottom represent the control flow between the memory areas at runtime.

There are several motivations for using PBT to support the Chaperone implementation. Most importantly, there is no overhead at all and no modifications to the running binary when Chaperone is disabled. In this mode the

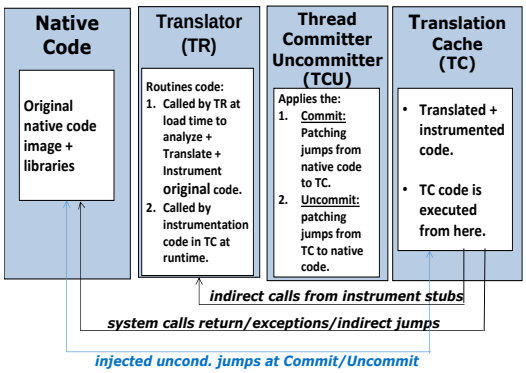

Figure 1: Logical memory organization of the Chaperone system.

commit-uncommit thread is put on a sleep state right after applying the uncommit phase. In this mode it is waiting for an external event to wake it up. Another motivation is the fact that there is no need to manage the TC as it is set once to a fixed size at load time and does not require a protocol for selecting which translated code to flush out at runtime due to space limitations.

The main drawback of a PBT system over a FBT system is the introduction of a new overhead type called *Transition* time which is the time required for the execution to change from the original code to the TC and back. In our case, the Transition time is immediate as we patch the code with required direct jump instructions at runtime without stopping the application. Another disadvantage is the limitation on the type of instrumentation that can be supported by a PBT system as it is applied only part of the total execution time. This means that instrumentation that requires full execution time such as emulation cannot be supported by it. In addition, in order for the proposed Chaperone system to be able to manage a running application there are two main requirements that need to be met:

- The instrumentation code stubs must be persistent and consistent, i.e., *mappable* to every corresponding instruction in the original code. Otherwise the commit-uncommit phases will cause an inconsistent state during execution.
- The *text* size of original code must not exceed a 4GB of size and the expanded instrumented code must not exceed 8GB in order for the patched direct jump instructions to reach the jump tables. This is an architectural limitation that is defined by the maximal range of 4GB that a direct long jump in x86 can reach.

In order to minimize the amount of code patching during runtime, the commit-uncommit thread injects only direct unconditional jumps - each of which is 5 bytes long covering up to 4GB of code size, from the executing code to a corresponding jump table located above or below the code image. Figure 2 shows the jump tables and the unconditional jumps that are injected by the commit-uncommit thread in the original code for the case of the commit stage and in the translated code for the uncommit stage.

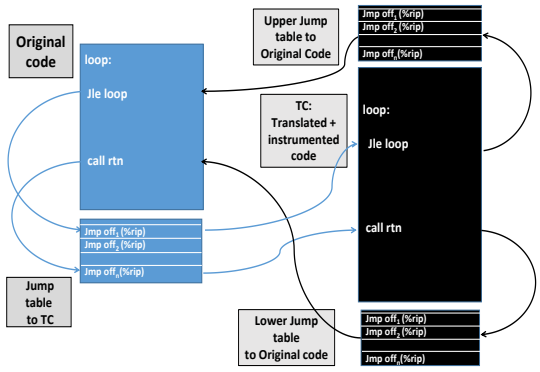

Figure 2: The Commit and Uncommit steps

3 THE TRANSLATOR (TR)

As mentioned in the introduction, the TR operates only at loadtime after the application image was loaded and just before it starts to execute, The translator is in charge of allocating the memory for the TC and the jump tables, analyzing the executable code, translating the code along with the needed instrumentation stubs, placing it in the TC and populating the jump tables with appropriate indirect jumps. In traditional BT systems where the TC size is limited, the TR is also responsible for the management of the TC and to determine which translation should be removed to make room for a new one. However, in the Chaperone system we can compute the needed maximal amount of instrumented code at load time thus we can rely on the fact that the TC is sufficiently large and its size is fixed. The TR applies 7 sequential steps before returning control to the loader in order to start executing the application:

Step 0: Checking the execution environment. In this step the TR checks for the number of available cores and the CPU ID for the supported instruction set features the size of the data cache line. It also checks the image permissions and modes as they may need to be modified to Writable when patching the jump instructions at the commit stage. In this step the TR checks for any available profile file in case that previous executions of the applications instrumented code had been recorded.

Step 1: Allocating required memory. This step requires a calculation of the size of the executable sections along with required instrumentation code in order to allocate the needed TC and the needed jump tables. The jump tables are allocated as close as possible to the application image that is to be translated and instrumented. If the size of the image is larger than 4GB, then 2 jump tables may be allocated to cover it - one before its top lowest address and one after its highest address. If no such memory is available then the TR exits with an error message and the original execution of the application continues without the Chaperone enabled.

Note that the TR is also responsible for allocating all data structure needed for the disassembly phase and for holding all instrumentation routines that are invoked from the TC.

Step 2: Analyzing original code. In this step the TR applies an incremental disassembly on the original code in order to dissect it into instructions, basic blocks and routines and separate code from data. In the analysis step all instructions are placed in an internal representation data structure. In our case we used a simple map array called $InstructionMap$ for all the disassembled instructions and maintained additional field for each direct jump or call instruction to hold the target array entry of the jump/call. The disassembly starts by decoding the instruction starting at the entry point of the program as provided by the image file format. In our implementation we use the x86 Encoder Decoder (XED) SDK [5] to decode the x86 instructions. The disassembly process then proceeds according to the control flow as follows:

(1) If the decoded instruction is not a control transfer instruction, then continue to decode the next instruction at current IP + size of previous decoded instruction.
(2) If the instruction is a conditional branch, add the target address into a queue of unvisited entry points and continue to decode the fall through instruction.
(3) If the decoded instruction is a *Call* instruction, add the target call address into the entry points queue and continue to the next instruction.

(4) Unconditional branches (direct or indirect) including *ret* instructions (with the exception of *call* instructions) stop the disassembly as we cannot guarantee that the following instruction is indeed code. In this case, extract the first entry point address from the queue of entry points and continue analyzing from there.
(5) Invalid decoded instructions naturally stop the disassembly at the previous instruction and the disassembly continues from next address in the entry points queue.

At the end of the process if uncovered code exceeds a threshold of 5% of the entire code size, then the Chaperone finishes with an error message of a non fully analyzed binary and the application execution continues without it.

Routines and Basic Block are also dissected during the analysis phase. For basic blocks every branch instruction terminates a basic block whereas every target of a direct jump/call starts a new one. Routine boundaries are mostly extracted using the symbolic and relocation tables in the file format, but also from targets of call instructions.

Step 3: Generating translated code. The disassembled code is placed in the internal representation in the required level: routine, basic block or instruction. The instrumentation routines of Chaperone are written in C++ as part of the TR source and are called directly to and from the TC memory area. In our experiments we used the Intel Pin SDK [6] to write the TR as a Pin tool and implemented the needed instrumentation functions in it. The instrumentation stubs were inserted into the translated code in the TC and consisted of:

(1) the instructions for saving the registers context into the stack (including the FLAGS and the stack registers). All saved registers are placed in an offset of -128 bytes from current stack pointer in order to avoid writing on the function's stack red zone.
(2) setting the needed values to be passed to the instrumentation fuction into registers
(3) an indirect call to the corresponding instrumentation function in the TR
(4) the instructions for restoring the context back.

Step 4: Chaining. In this step the TR goes over the $InstructionMap$ and checks for targets of direct jumps or calls that need to branch to a new target address. At this point only the internal data structure is modified to hold the correct target entry in the $InstructionMap$ for each direct jump or call.

Step 5: Encoding. In this step the instructions are encoded directly into the TC one after the other. All rip-based, direct branch and direct call displacements are updated in the encoded instructions based on the relocation of target instructions. For forward jump instructions that branch over code that is not yet encoded, the maximal displacement encoding is used. As a result, the step requires several iterations until the displacements of all long and short branches are fully resolved.

In this step the TR also populates the jump tables and prepares a patching list of all addresses that need to be patched with an unconditional jump instruction by the TCU. The list includes a pair of an original code address and its corresponding translated code address. There are two main considerations in selecting the instructions that are to be patched by the unconditional direct jumps. The first consideration is to minimize the transition time between the TC and the original code on every commit and uncommit step. The assumption is that frequently executed code is found in either loops or a chain of frequent routine calls. Therefore, direct backward jumps and direct call instructions make a sufficient coverage of these cases.

Another consideration is the ability to perform the patching atomically while the application is running and still be able to reach the jump table with a single direct jump. In x86 architecture, a *mov* instruction of 8 bytes is guaranteed to be atomic unless the memory address crosses a cache line boundary. Consequently, the following instructions addresses are added to the patching list provided a 32 byte d-cache line boundary does not cross them on the first 2 bytes:

- Address of every direct call instruction. Direct call instructions in x86 are 5 bytes long.
- Address of every direct backward jump that is of size larger or equals to 5 bytes. Instructions smaller than 5 bytes cannot be patched by a direct jump instruction that can reach the relevant entry in the jump table. At the same time, atomically patching two or more consecutive instructions is not possible either as the system cannot guarantee that the applications processes (or threads) will not attempt to execute one of partially patched instructions during the patching step.
- Address of a target of a backward jump that is smaller than 5 bytes provided the target instruction is of size larger or equal to 5 bytes. If the target instruction is shorter than 5 bytes then

the proceeding instruction in the basic block is checked as a candidate for patching.

Step 6: Kick start the TCU. The TR kick starts the TCU by setting the global volatile variable *enable_commit_uncommit_flag* to *true* and apply the x86 $mfence$ instruction to fulsh the store buffer.

4 THE THREAD COMMITTER-UNCOMMITTER (TCU)

As explained above the TCU is created together with the TR at load time. Once created, it is waiting in an endless loop that checks the global variable *enable_commit_uncommit_flag*. When set to true by the TR the TCU exits the busy wait loop and enters another loop that periodically applies the commit and the uncommit phases separated by the *usleep*() system calls. Naturally, a longer sleep interval is given to the uncommit stage over the commit stage where the translated and instrumented code is running, in order to reduce the performance regression. Below is the main routine body of the TCU written under the Intel Pin SDK [6]:

```
// Busy wait until translation completes:
while (!enable_commit_uncommit_flag);

// wait 1ms before applying 1st commit:
usleep(1);

while (true) {

  // Commit translated code:
  PIN_LockClient();
  commit_translated_code();
  PIN_UnlockClient();

  // Original code is committed.
  // Translated code is now running

  // Wait 1ms for the application start:
  usleep(1);

  // Uncommit translated code:
  PIN_LockClient();
  uncommit_translated_code();
  PIN_UnlockClient();

  // Translated code is uncommitted.
  // Original code is now running

  // wait a longer interval of 10ms
  // before applying the commit again:
  usleep(10);
}
```

Note that the routines that apply the commit and uncommit operations which are protected by the *Pin_Lock/Unlcok_Client* APIs, apply the $mfence$ x86 instruction just before returning. This is in order to flush any waiting stores in store buffer. The routines in charge of patching the jumps to and from the translated code go over the patching list that was prepared by the TR beforehand.

At the commit stage, the TCU is in charge of injecting the direct unconditional jumps from the original to the jump table and to undo the injected jumps from the translated code to the original code in the commit stage. At the uncommit stage, the TCU performs the reversed operation of injecting direct jumps from the translated code back to the original code via the jump tables and to undo the injected jumps in the original code. The injection step of the direct jumps (in both commit and uncommit stages) is performed by the TCU in 3 stages:

(1) The original code before being patched is saved into a patching map located in the TR data area. Note that only the 5 bytes of the encoded jump instruction that are patched need to be saved.
(2) The target of the code that is to be patched is restored back to the original code before being patched from the patching map. The restore is done by writing directly on to the code atomically for the patched 5 bytes. If the patched code is located in the original code then the page permissions containing it must be changed to R/W using the *mprotect*() system call. The actual patching is done by writing the encoded instruction in the patching list directly on the patched address using an atomic code update routine described below.
(3) The code is patched with the encoded direct jump in the patching list using the same atomic patching step.

Since the patching is done while the application is running and without stopping it, then it is imperative that steps 2 and 3 above are performed atomically. As explained in the TR section, in x86 architecture, a *mov* instruction of 8 bytes is guaranteed to be atomic unless the memory address crosses a cache line boundary. In order to patch instructions that cross a 32 byte boundary we use 3 steps:

(1) Replace the first 2 bytes of the instruction by the short $jmp\ -2$ instruction which in fact serves as an endless loop to itself in the event that the execution reaches the instruction while being patched.
(2) The remaining 3 bytes are patched by the needed encoding bytes.

(3) The first 2 bytes are replaced to the correct jmp encoding prefix.

Each of the 3 steps is performed atomically as illustrated in Figure 3.

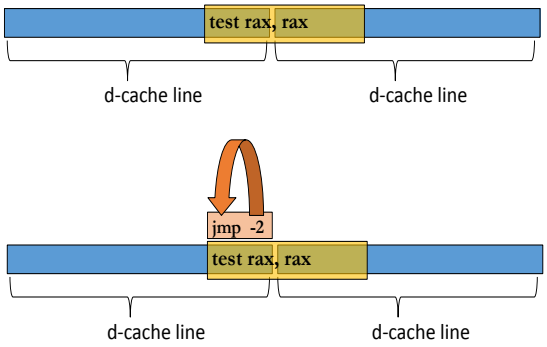

Figure 3: Atomic code update for instructions that cross d-cache line.

5 USE CASE 1: MEMORY CHECKER CHAPERONE

Out-of-bounds memory bugs occur when the application dynamically allocates a block of memory, iterates on the allocated block and falsely accesses memory outside the bounds of the originally allocated memory. In order to detect instructions that exceed a legal bound, the TR instruments every call to the $malloc()$ and $free()$ subroutines of the C runtime library. Each call to $(void*)addr\ malloc(size\_tsize)$ is diverted to allocate additional 16 bytes of memory which serves as a wrapper before and after the original allocated memory region. Consequently, the instrumented $malloc()$ returns $addr + 8$, whereas every call to $free(addr)$ de-allocates the original size along with the extra 16 bytes starting from $addr - 8$. The allocated addresses along with the extra allocated 16 bytes of memory are tracked via a global hash table called $AllocatedRegionsMap$ located in the TR static area.

Next, the TR instruments every instruction in the application code image that references the memory of the form:

$$op\ srcReg,\ displacement(baseReg,\ indexReg,\ scale)$$

e.g. $mov\ r8,\ 5(r9,\ r1,\ 2)$.

In order to decode all x86 memory-based instructions we use the Intel x86 Encoder Decoder (XED) SDK [5] which is also available as part of the Intel Pin SDK [6]. Specifically, we use the following integrated Pin APIs that rely on the XED library:

```
ADDRDELTA INS_MemoryDisplacement (INS ins)
REG       INS_MemoryBaseReg (INS ins)
REG       INS_MemoryIndexReg (INS ins)
UINT32    INS_MemoryScale (INS ins)
```

At each instrumentation stub injected before each memory instruction we check that the address calculated by $displacement + val(baseReg) + val(indexReg) * scale$ does not fall into any of the wrapping memory regions that were created at the calls to $malloc()$ or to non-allocated/de-allocated regions that are not covered by the $AllocatedRegionsMap$ table and if so, an $out-of-bound$ message is printed out to $cerr$. To limit the performance overhead to a minimum the commmit-uncommit ratio in the TCU is set to 1/15 i.e., 1ms the application runs in commit mode, i.e. instrumented, following by 15ms of application running in uncommit mode, that is original code.

As mentioned above, Memory Checker Chaperone was implemented using the Intel Pin SDK [6] on the Linux OS. The Intel Pin framework enables to write binary-level instrumentation tools in Probe mode where all the translation and the instrumentation is done only once at load time. In this mode the runtime VM of Pin is not activated and thus translation time overhead is limited to load time.

The performance results of the Memory Checker Chaperone are not comparable to those obtained by other known instrumentation tools. For example, the Valgrind [12] instrumentation-based memory checker runs between $5x$ to $55x$ slower than the original application due to its heavy instrumentation stubs. The Chaperone, however, reaches up to a 12% regression when setting the TCU frequency to the appropriate ratio.

Figure 4 shows the performance ratio of the Memory Checker Chaperone run time on the SPEC CPU 2006 benchmarks divided by the base run time vs. Valgrind ratio on the ref input. The runs were measured on an Intel Xeon machine with CPU model $E5-2699$ (Code Name Broadwell) of 2.20GHz frequency and a 128GB of RAM memory. For these measurements we set the TCU to a 1/15 ratio following preliminary measurements that showed it to produce best results and still identify out-of-bound memory operations.

Figure 5 shows the performance results when applying the Memory Checker Chaperone on the $bzip2$ application from SPEC CPU 2006 with different TCU frequencies, where 1 represents 1ms spent in commit and 1ms in uncommit whereas 10 represents 1ms in committed mode and 10ms in uncommitted mode, and so on.

Chaperone's main problem is the fact that it may miss illegal out-of-bound memory accesses since only part of

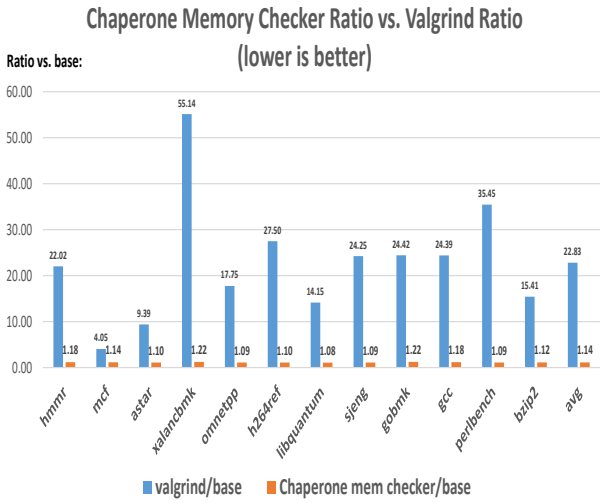

Figure 4: Chaperone MemChecker results on SPEC CPU 2006 vs. valgrind

the execution time is spent in the instrumented code. To check the effectiveness of the Chaperone in detecting memory bugs we inserted artificial illegal references in the *bzip*2 source code. The first bug was inserted in function *mainGtU* which is the hottest function of *bzip*2 taking 70% of the total execution time and located in the file *blocksort.c*. The bug was the following an illegal read access added at the top of the function:

```
UChar ctmp = block[-1];
```

where *block* is a pointer supplied to the function as a parameter. The above code did not cause *bzip*2 to fail and Valgrind managed to detect it correctly. Interestingly, the Chaperone managed to detect it as well for all the TCU frequencies between 1 through 50. The second bug which was artificially inserted was the following illegal write access planted in a rarely executed function *compressStream* in the file *bzip*2.*c*:

```
*((char *)bzfp - 1) = 0;
```

where *bzfp* is a pointer that points to an allocated memory. The above code is executed only once at the beginning of the program and while Valgrind managed to detect it correctly, the Chaperone failed to detect it even for the TCU frequency of 1 where 50% of the time is spent in the original code and 50% at the instrumented code, since the bug occurs immediately at the beginning of the run when the original (non-instrumented) code is running.

In general, there are ways to improve the coverage of buggy memory accesses by the Chaperone Memory Checker if we introduce a randomized TCU ratio. Another way is to use persistent memory where we log the collected profiling data into a file which is then read on future invocations of the application. The collected profile information from previous runs can help Chaperone instrument the rarely executed code areas. We actually implemented such a mode of persistent profiling in Chaperone using the *mmap* system call in Linux to map all profiled data into a file and then check its existence upon every invocation. This mode seems to be working well for cases where Chaperone was focused on collecting data in rarely executed code areas.

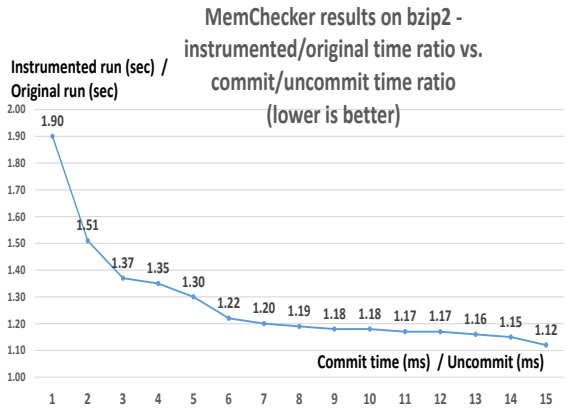

Figure 5: Chaperone MemChecker results on bzip2 with different instrumentation frequencies

6 USE CASE 2: OPENMP LOAD BALANCER CHAPERONE

OpenMP applications spawn threads at runtime in order to support the parallelization constructs of parallel loops and parallel blocks (see [2]). Usually, the user sets the maximal number of OpenMP threads that can be spawned by the OMP runtime ahead of time for all the parallel loops/blocks in the program by using the environment variable $OMP\_NUM\_THREADS$ or by calling the *omp_set_num_threads*() OMP API. This often leads to unbalanced allocation of threads, as heavy CPU-bound loops starve for OMP threads whereas non CPU-bound loops waste OMP threads and consume power. In addition, the maximal number of OpenMP threads also sets the chunk size of the loop iterations space that is given to each thread and thus impacts the cache conflict effects that occur between the running tasks that accessed shared data. Consequently, finding the optimal number of OpenMP threads for a given loop

ahead of time is considered hard and a tool that can find and set the optimal number of OpenMP threads at runtime, while taking into account cache conflict effects, can have a positive impact on performance and power consumption. Note that OMPLB does not change the scheduling policy and the size of the sequential $CHUNK$ of iterations of each parallel loop directly as dictated by the programmers using the OpenMP *schedule* directive (see [1]). Instead, it simply sets the maximal allowed spawned threads which indirectly dictates the $CHUNK$ size.

In addition to setting the optimal number of threads, the affinity of threads to available CPUs is also done mostly ahead of time by the user by setting global environment variables of $KMP\_AFFINITY$ or OMP_PROC_BIND protocols. To optimize the affinity of threads to CPUs, OpenMP Load Balancer (OMPLB) tracks the status of each thread and accordingly detaches waiting threads and attaches running threads to the available CPUs at runtime.

The OMPLB searches and then sets the optimal number of OMP threads and then affinities them to the available CPUs at runtime. It dynamically tries out different number of OMP threads, measures the elapsed time intervals for each parallel loop invocation and then, based on the collected time statistics, converges to the optimal number of OMP threads.

The OMPLB was implemented using the $LD\_PRELOAD$ Linux environment variable which guarantees that the shared library containing the TR code is loaded prior to any library that is dynamically linked to the OMP application. The pre-loaded library also contains instrumented redefinitions of the relevant OMP runtime library APIs that are called by the application before entering and after completing each parallel loop or parallel block in the program. The TR is implemented in the special routine $\_\_attribute\_\_((constructor))$ $void$ $init\_load(void)$ which the Linux loader always executes before applying the application's constructors. The TR uses the $libelf$ library in order to read the application's file format and uses the Intel XED SDK to decode the instructions and disassemble the code. Note that OMPLB does not require a TCU as it uses the calls from the application to the instrumented routines as a trigger to the instrumentation functions in the TR.

At load time, the TR retrieves the current maximal number of OMP threads ($current\_threads\_num$) by calling directly to the OMP library API $omp\_get\_max\_threads()$ along with the number of available processors in the system ($available\_processors\_num$) by calling the Linux system call $get\_nprocs()$. At the entry of every *omp parallel for* construct, the TR checks if there is already a pre-calculated number of recommended OMP threads that differs from the currently set number of maximal omp threads and if so, it modifies it by calling directly to the OMP API $omp\_set\_num\_threads()$. Then, it starts the clock for measuring the elapsed time interval $real\_time\_interval$ for the currently set maximal threads. Next, the original code of the *omp parallel for* executes without any interference from the TR and just before exiting the *omp parallel for*, the TR stops the clock measurement. It then calculates the average elapsed time of the executed loop for the currently set number of threads - $current\_threads\_num$.

The average interval value is placed in the array element $avg\_real\_time[current\_threads\_num]$ which is calculated from the division of: $real\_time\_sum[current\_threads\_num]$ by $first\_time\_sum[current\_threads\_num]$ where:

- $real\_time\_sum[current\_threads\_num]$ is the sum of all time intervals measured for $current\_threads\_num$ for all invocations of all parallel loops encounter so far .
- $first\_time\_sum[current\_threads\_num]$ is the sum of all time intervals measured for $current\_threads\_num$ for the first invocation of every parallel loop encountered so far.

Next, based on the above statistics, the TR searches for the recommended number of maximal omp threads for the next *omp parallel for* invocation by using two consecutive search loops when taking into account a 10% allowed performance regression in favor of reduced power by using the constant variable $ALLOWED\_PERF\_PENALTY$ which is set to 1.1:

```
// 1. Apply logarithmic search for the num
// of threads that give the shortest avg
// time interval:
//
double min_time_interval =
    avg_time_interval[current_threads_num];
int optimal_num_threads =
    current_threads_num;

for (i = available_processors; i >= 2; i/=2) {
  if (min_time_interval * ALLOWED_PERF_PENALTY
       >= avg_time_interval[i]){
     min_time_interval = avg_time_interval[i];
     optimal_num_threads = i;
  }
}
// 2. Search further in the range found
// by the logarithmic search above for
// smaller num of threads with shorter
```

```
// time interval to reduce power:
//
int further_optimal_num_threads =
    optimal_num_threads;

for(i = optimal_num_threads-1;
    i > optimal_num_threads/2; i--) {
  if(min_time_interval * ALLOWED_PERF_PENALTY
       >= avg_time_interval[i])
     further_optimal_num_threads = i;
}
optimal_num_threads =
    further_optimal_num_threads;
```

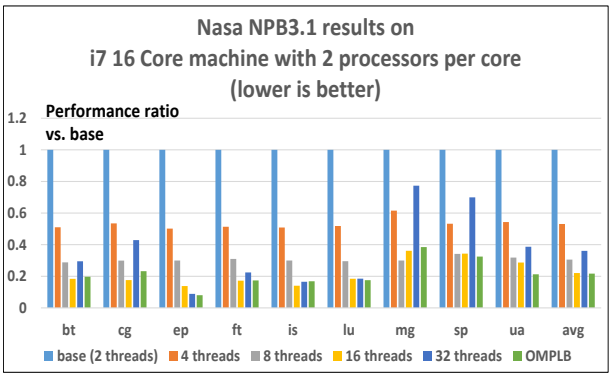

Figure 6: OMPLB measurements on NASA benchmark

Figure 6 shows the runtime results for the NASA benchmarks when setting the maximal number of threads manually via the $OMP\_NUM\_THREADS$ environment variable vs. the time measured when OMPLB is enabled. The results were measured on the same machine that was used for the Memory Checker Chaperone for the NASA benchmark compiled using the Class B input size. The baseline was the runtime of the executed benchmarks without the OMPLB.

For a small input size such as the Sample Input, the NASA benchmarks completed withing a matter of seconds and the OMPLB was actually running almost twice longer due to its initial constant overhead of applying its binary analysis. However, for the B class inputs, in most cases OMPLB manages to converge to the optimal result except CG - Conjugate Gradient benchmark which contains some degree of irregular parallel loops causing OMPLB not to converge to the optimal number of threads.

Figure 7 shows the corresponding power measurements on the smaller platform of 8 cores using a sampling of the power/energy events in the CPU by the $perf$ Linux utility. The Linux $perf$ package provides profile sampling abilities based on supported hardware performance counters and performance events of the CPU. For our measurements we used the following $perf\ stat$ command $perf\ stat\ -a\ -e\ power/energy-cores/k$ to collect the needed results. Unfortunately, collecting power statistics in this mode is limited but can give a rough estimate of the power consumption trend.

Table 1: Converged num of threads by OMPLB:

bt	13
cg	15
ep	29
ft	14
is	30
lu	31
mg	13
sp	8
ua	17

OMPLB was able to show a stable reduction in power consumption of 10% compare to the usual case of setting the maximal number of OMP threads to the number of available cores which in this case equals to 8. The reduction in power is due to the fact that OMPLB has converged to a number of threads that permits up to 10% regression. Table 1 lists the converged number of OMP threads by OMPLB per benchmark.

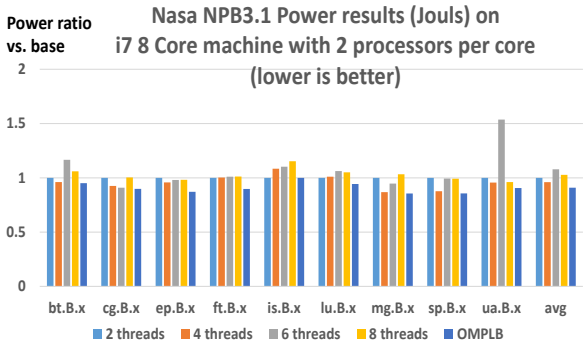

Figure 7: NASA Benchmark Power measurements

7 RELATED WORKS

There are two main modes of binary instrumentation - static and dynamic. The static binary instrumentation produces the complete instrumented code before starting to run it. Tools such as the IBM FDPR [9] used to instrument the entire binary offline and generate the

instrumented binary file that is then executed in order to collect the needed performance statistics for the optimization phase. In this mode the translation is done offline and therefore does not affect the runtime performance. However, the instrumented code is still very heavy and reaches up to a $10x$ slowdown than the original program. Tools that apply dynamic binary instrumentation such as Valgrind [12], Intel Pin [6], PEBIL [11], QEMU [3], QBDI [4], [8] apply the instrumentation in the translated code at runtime. Here the translation time affects the performance but still, the majority of the overhead comes from the heavy instrumentation stubs which can reach up to $10x$ slowdown and higher. These tools try to resolve the overhead by optimizing the code within the instrumented stubs and the wrapper code that invokes them. The Chaperone uses a different approach to reduce the overhead by applying a periodic instrumentation where the instrumented code is executed only part of the runtime. This mode is not suitable for tools that require full instrumentation at all time such as emulation. However, Chaperone is shown to be useful for performance tuning and for tracking cases of out-of-bound memory references in frequently executed code.

8 CONCLUSIONS

In this work we introduced a new approach for instrumenting binary images that adds minimal overhead to application execution. This approach uses what we called partial binary translation, wherein the control flow of the executed application is periodically diverted at runtime from the original code to the instrumented and translated code and back. The main idea is to atomically patch the image of the binary with appropriate jump instructions and without the need to stop the application. This required taking into account the size of the data cache line of the CPU generation and rely on the supported atomic *mov* instructions in x86. As a result, we were able to show between $8 - 22\%$ performance overhead with an average of 14% for instrumentation of every load and store instruction and even serve as way to tune performance while reducing power for OpenMP applications. At the same time, we noticed that the proposed approach is not suitable for every instrumentation purposes such as full emulation or cases where there is a need to instrument rarely executed code. For these cases we propose to enable in the future a persistent memory mode in Chaperone where previous invocations of the system will be logged into a file and then used in future invocations to instrument rarely executed code areas.

The Chaperone system enables the user to set the frequency of diverting between the original and the instrumented code at runtime. As a result, we believe Chaperone can be used at production level by end users and system administrators and not only during development.

REFERENCES

[1] *OpenMP Scheduling.* University of Mary Washington, http://cs.umw.edu/ finlayson/class/fall16/cpsc425/notes/12-scheduling.html.

[2] *Chapter 1.3 Execution Model.* http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, 2013.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[4] Cedric Tessier Charles Hubain. Qbdi - quarkslab dynamic binary instrumentation home page. In *https://qbdi.quarkslab.com*, September 2015.

[5] Mark Charney. Xed - x86 encoder decoder library. In *Intel software developer zone*, July 2015.

[6] Robert Muth Harish Patil Artur Klauser Geoff Lowney Steven Wallace Vijay Janapa Reddi Kim Hazelwood Chi-Keung Luk, Robert Cohn. Pin: building customized program analysis tools with dynamic instrumentationn. In *Proceedings of ACM SIGPLAN Notices*, June 2005.

[7] Vishv Malhotra Cristina Cifuentes. *Binary Translation: Static, Dynamic, Retargetable?* IEEE, 1996.

[8] Saman Amarasinghe Derek Bruening, Timothy Garnett. *An infrastructure for adaptive dynamic optimization.*

[9] Moshe Klausner Alex Warshavsky Ealan A. Henis, Gadi Haber. Fdpr - a post-link optimization tool for large subsystems. In *Proceedings of Feedback Directed Optimizations 2 Workshop*, October 1999.

[10] Chris Lattner and Vikram Adve. The llvm instruction set and compilation strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[11] Michael Laurenzano, Mustafa M. Tikir, and Allan Snavely Laura Carrington. Pebil: Efficient static binary instrumentation for linux. *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 175–183, 2010.

[12] Julian Seward Nicholas Nethercote. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, June 2007.