ELFies: Executable Region Checkpoints for Performance Analysis and Simulation

Harish Patil
Intel Corporation
USA

Alexander Isaev
Intel Corporation
Belgium

Wim Heirman
Intel Corporation
Belgium

Alen Sabu
National University of Singapore
Singapore

Ali Hajiabadi
National University of Singapore
Singapore

Trevor E. Carlson
National University of Singapore
Singapore

***Abstract*—We address the challenge faced in characterizing long-running workloads, namely how to reliably focus the detailed analysis on interesting execution regions. We present a set of tools that allows users to precisely capture any region of interest in program execution, and create a stand-alone executable, called an ELFie, from it. An ELFie starts with the same program state captured at the beginning of the region of interest and then executes natively. With ELFies, there is no fast-forwarding to the region of interest needed or the uncertainty of reaching the region. ELFies can be fed to dynamic program-analysis tools or simulators that work with regular program binaries. Our tool-chain is based on the *PinPlay* framework and requires no special hardware, operating system changes, re-compilation, or re-linking of test programs. This paper describes the design of our ELFie generation tool-chain and the application of ELFies in performance analysis and simulation of regions of interest in popular long-running single and multi-threaded benchmarks.**

***Index Terms*—record and replay, dynamic program analysis, performance monitoring, simulation region selection**

I. Introduction

Workload characterization is the process of understanding the behavior of computer systems. Characterization can be performed for the whole system or for individual programs. Typical characterization approaches include using hardware performance counters, program instrumentation with tools such as Pin [1], and performance simulation [2]–[4]. These approaches incur a varying amount of overhead depending on the degree of detail they provide. Since the detailed analysis of entire program runs can be prohibitively expensive, most approaches use sampling techniques to focus on key regions of interest [5]–[7]. Phase-based sampling techniques, such as SimPoint [5], are very effective in finding representative regions for architecture simulation. However, finding regions of interest can be time-consuming and it is desirable to be able to share such regions among researchers so the cost of generating them is amortized. One challenge is how to re-run the regions of interest once found. In addition, with native program execution, starting hardware performance counting exactly at the point of interest can be difficult. While the task may be easier with program analysis tools and simulators, given the run-to-run variation in program execution, especially for multi-threaded programs, a region of interest found in an earlier run may not always be reachable in a subsequent execution.

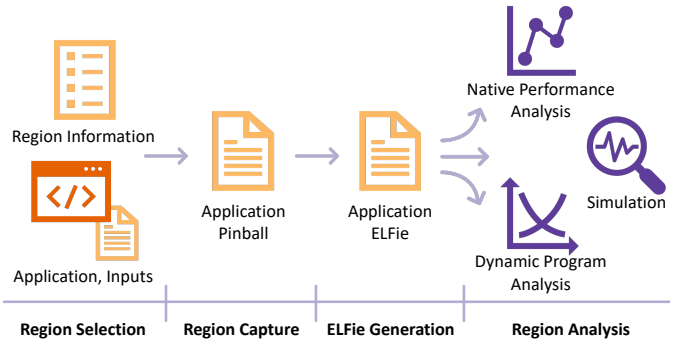

Fig. 1. ELFie: generation and use.

The PinPoints toolkit from Intel [8] automates the tedious task of profiling an x86 application, finding phases, and creating a checkpoint called a *pinball* for each representative region. These checkpoints are self-contained and can be shared among researchers easily. In fact, researchers have made SPEC CPU2006 and CPU2017 pinballs [9], [10] publicly available. However, pinballs only work with Pin-based simulators and analysis tools. There is a need for a sharing mechanism that is not dependant on Pin. We believe that having regions of interest represented as x86 binaries is one possible solution.

To accomplish this, we present a set of tools for precisely capturing interesting program regions in the form of stand-alone executables called ELFies. An ELFie is an executable in the Executable and Linkable format, ELF [11], capturing a portion of another program's execution. ELFies themselves are statically linked binaries and can be generated from executions of other statically or dynamically linked, single or multi-threaded, x86 binaries (32 or 64-bit). They can be fed to any characterization tool that accepts ELF binaries. No changes to the characterization tools are required as ELFies behave like regular program executables for all practical purposes. Further, ELFies are easier to share among researchers than the original

applications and they will start executing exactly at the point in the original application execution that was captured.

Our approach is based on the Pin-based framework for program capture and deterministic replay called PinPlay [12]. We used the publicly available version of the PinPlay toolkit [13], with the ability to generate *fat* pinballs (see Subsection II-A). The overall approach for ELFie generation is shown in Figure 1. We run the test program under the PinPlay *logger* tool providing it a specification for the region of interest, selected for instance by program phase analysis. The logger captures the specified portion of the program execution in a set of files collectively called a *pinball*. The pinball contains, among other things, a memory image (a `.text` file common to all threads), and the architectural register values at the beginning of the region (one `.reg` file per thread, which also includes register changes from system calls and signals). Next, our *pinball2elf* tool reads a pinball and generates an ELF binary by translating the pinball memory image pages to ELF sections and adding a startup code section at the entry address. The startup code creates the required number of threads, initializes each thread's architectural registers, and jumps to the beginning of the actual program code inherited from the original pinball. Other data in the pinball that is needed for constrained execution, like the details for injection of system call side-effects and the enforcement of shared memory access order of various threads, are ignored during this conversion.

Checkpoint/restore in user space (*CRIU*) [14] is a Linux tool that allows one to capture the state of a process tree to disk and to restore it later on the same or similar machine. It supports multiple usage scenarios including process migration and the debugging and analysis of applications. With ELFies, we focus on regions-of-interest from a single-process, multi-threaded application for targeted analysis. A CRIU checkpoint is a snapshot at a particular *point* in the execution with no specified end. ELFies represent a bounded *region* in the execution as captured in the corresponding pinball. ELFies are Linux binaries and hence can be run by themselves and can be used with tools that run with binaries. Since pinballs can be generated on operating systems other than Linux, one can imagine tools similar to pinball2elf that convert pinballs to other executable formats such as *Portable Executable (PE)* format on Windows and *Mach-O* on MacOS.

Table I compares ELFies, the target of this work, with pinballs. ELFies can be run natively and without any extra overhead. They can be run with any tool that otherwise accepts an ELF executable—no modifications to the tool are necessary. This flexibility allows ELFies to be used for a variety of purposes. However, since pinball2elf only uses the initial memory and register state from a pinball to create an ELFie ignoring the rest of the files required for enforcing determinism, an ELFie run is non-deterministic[1]. Also, *pinball2elf* currently does not add any debug information to ELFies hence they do not support symbolic debugging.

[1]PinPlay only guarantees that shared-memory access order in multi-threaded pinballs is repeated exactly as opposed to a guaranteed total order of instructions from various threads. Hence, we prefer the term *constrained* over *deterministic* replay.

TABLE I
PINBALL-ELFIE DIFFERENCES.

	pinballs	**ELFies**
Allow constrained replay	Yes	No
Work across OSes Windows:Linux	Yes	No
Handle all system calls	Yes	Most (stateless ones)
Allow symbolic debugging	Yes (with GDB)	No (hex-only with GDB)
Run natively	No	Yes (Linux only)
Exit gracefully	Yes	Yes (with performance counters)
Run with x86 simulators	Yes; simulator modifications needed	Yes; without simulator modifications
Run-time overhead over a native run	∼15X (ST), ∼40X (MT)	None (except start-up code overhead)

Two questions relevant for the use of ELFies are (1) How will system calls behave? and (2) How will an ELFie run end?

A. The System Call Handling Challenge

During a pinball replay, most system calls are skipped, and their register results are injected from the `.reg` files. Program memory, changed by system calls, is inserted at use time based on automatically logged values [15]. This guarantees that the results of non-repeatable system calls such as `gettimeofday()` will be the same during replay as was during logging. ELFies do not have any injection mechanism to handle system calls—they are simply re-executed natively. So, at times, some control-flow decisions based on system call results may cause ELFie execution to go on a different path compared to a pinball replay. Also, some system calls may rely on OS resources such as open file descriptors which will not be available during an ELFie run and will fail. For example, consider a file opened before a region of interest and used in the region. The region pinball replay will skip the file read and return the stored results, however an ELFie will try to repeat the file read system call and fail. Luckily, the constrained replay of the parent pinball acts as a reference for an ELFie execution in terms of the system calls it will execute and files it will access. We can use this information to guide ELFie execution in many cases. In fact, we have developed a *SYSSTATE* technique to ensure correct re-execution with file-related and other common Linux system calls. In particular, a Pin-tool re-constructs the file and heap state with replay-based analysis for the test pinball and stores it in a *sysstate* directory. pinball2elf embeds references to the *sysstate* contents in the startup code of the ELFie created from the pinball.

B. The Graceful Exit Challenge

ELFies contain only the memory (data + text) pages captured in the pinball. If the execution of an ELFie goes on a new path, it may try to access/execute a page that has not been captured, resulting in an ungraceful exit. On the other hand, while a pinball replay will always terminate after the desired number of instructions (recorded in the pinball), at times an ELFie may continue to execute far beyond the desired number of instructions—as long as it stays within the captured memory range. We support graceful exit of ELFies using hardware performance counters. Pinball2elf can program a hardware counter, one per thread, to count retired instructions and create a callback to exit each thread on reaching the expected instruction count for the region of interest captured in the incoming pinball. If an ELFie is used by some other program, such as a simulator, the graceful exit of ELFie can be ensured by that program instead.

Despite the two challenges above, we found ELFies to be quite suitable for use cases such as native performance measurement or tool-based analysis and simulation for regions of interest. For instance, verifying the quality of simulation region selection requires running both the whole-program and all selected regions using the same methodology, yet whole-program simulation is prohibitively expensive. ELFies allow us to perform the evaluation using native runs instead, thus enabling accurate representative generation of extremely long-running programs (see Section IV-A). A graceful exit can be forced by analysis tools after the desired number of instructions. The effect of system calls can be minimized by choosing the regions appropriately. If an ELFie run fails consistently before reaching the desired instruction count, starting with an alternative region of interest may help. In our experiments we used alternate region selection (i.e., the second or third best representative for a given phase/cluster that SimPoint provides), to increase coverage (the sum of the weights of correctly executing ELFies) up to 90%+ in most cases, while still maintaining high accuracy. Without any guarantee of repeat-ability or a graceful exit, other more general uses of ELFies may not be possible. In those cases, pinballs with their constrained replay guarantee may be more appropriate.

In this paper, we present details of how the pinball2elf tool works and how we are using the ELFies it generates for focused analysis of regions of interest of long-running applications. Our main contributions are as follows:

1) We describe the tool-chain (PinPlay + pinball2elf) for converting user-level checkpoints (pinballs) to executables (ELFies) as shown in Figure 1. To our knowledge, the idea of creating a stand-alone executable for an arbitrary portion of a multi-threaded x86 program execution is novel.
2) We show the use of ELFies, among other things, for validating simulation region selection techniques. Traditionally, the validation is done using simulation and hence is very slow. With ELFie-based validation, we use native hardware instead of simulators which allows for validation of really long-running programs.

Pinballs have provided a way to share regions of interest among researchers [9], [10]. We hope ELFies can play a similar role for x86-binary-based tools/simulators. We have open-sourced pinball2elf tool to facilitate that [16].

The rest of this paper is organized as follows. Section II describes the implementation of the pinball2elf tool. Section III is about typical applications of ELFies. Section IV presents some case studies showcasing advantages of using ELFies for performance analysis and simulation. Section V summarizes other work in this area and section VI concludes.

II. Implementation

This section presents an overview of the changes made to PinPlay and details of the pinball2elf tool.

A. PinPlay Changes

The motivation for creating an ELFie is that it should always start with the program state captured at the beginning of a region of interest but should then run in an un-restricted manner, without any enforcement of determinism, using any memory it requires. A pinball, on the other hand, is designed for constrained replay and hence captures only the memory pages that were actually used in the logging run. An un-restricted ELFie run could, at times, diverge from the recorded control flow, try to access an un-captured memory page, and fail. To alleviate this problem, we requested the PinPlay team to make a few changes to the PinPlay logger. First, a new switch, `-log:whole_image`, was added that records all the loaded sections, including global static data, for the test image. Second, since the logger by default creates text/data page injection records *lazily*, another switch, `-log:pages_early`, was added, which puts the pages in the initial memory image (the `.text` file). The two new switches can be combined with a single switch `-log:fat`. We call the resulting pinball a *fat* pinball. All the pinballs we generated for ELFie evaluation were fat pinballs. Depending on the number and sizes of shared libraries used by a program, a fat pinball can be much larger than a regular pinball. Finally, a new replay switch, `-replay:injection`, was added, which when set to zero, attempts replay without any system-call side-effect injection or enforcement of thread order. Such injection-less replay mimics the execution of an ELFie (while still running under Pin) and hence is useful in debugging ELFie failures.

A fat pinball for a region has all the memory used inside the region pre-loaded in the initial memory image file. This includes text/data pages from any libraries dynamically loaded inside the region and any heap pages accessed inside the region. However, heap pages that are allocated but not accessed are not included in the initial memory image.

B. The Pinball2elf Tool

Pinball2elf is a tool to create executable or object files from pinballs. The executables generated by pinball2elf are statically linked and hence are self-contained without any

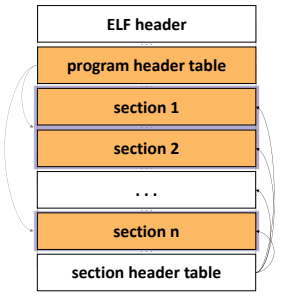

Fig. 2. Structure of an executable ELF file.

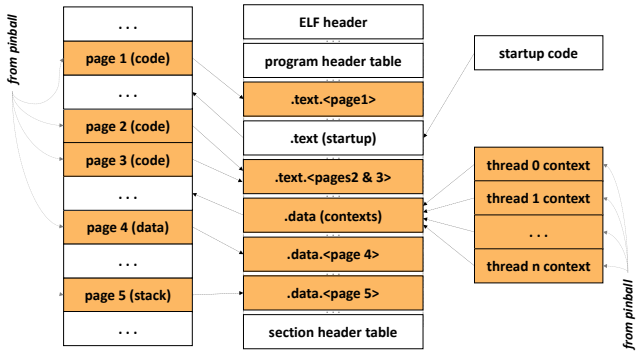

Fig. 3. Pinball to ELF mapping.

dependence on shared libraries. During execution, an ELFie has the same memory layout as the original pinball, i.e., all memory regions are mapped to the same addresses as during the pinball recording run. That makes ELFies useful for memory-characterization studies.

1) Overview of the ELF format: The Executable and Linkable Format (ELF) [11] is a standard, flexible, and extensible format for representing binary code in a system. ELF is used for object files, executables, shared libraries, core dumps, and kernel modules. An ELF file usually consists of an ELF header followed by other headers describing the file layout and of sections or segments containing data. Depending on the type, an ELF file can include a program header table and a section header table. A typical layout of object and executable ELF files is illustrated in Figure 2. The ELF header must be located at the beginning of the file and is used to locate other parts of the file. This structure defines sizes and file offsets of the section header and program header tables, specifies the virtual address of the program entry point, and stores other useful information.

The section header table is an array of section-header structures, each of which describes a section. A section header structure defines the file offset of each section, its type and size, its attributes (allocatable, executable, and writable), and its virtual address (if allocatable, i.e., if it will appear in the memory image of the process for the executable). Some sections in an ELF file have pre-defined meanings and hold program data or are used by the system loader. For example, `.text` holds executable instructions of a program, `.data` holds initialized data, and `.symtab` contains information needed to locate a program's symbolic definitions and references.

2) Pinball to ELF mapping: We chose to make ELFies statically linked because a statically-linked executable contains a set of object files and libraries which are bound at link-time with all references resolved and hence is completely self-contained. A fat pinball's `.text` file holds the memory dump of the captured program region (all pages mapped into the memory). The `.reg` files contain initial register contents of program threads at the beginning of the region. Each region from the `.text` file which consists of consecutive pages is represented with a section in the ELF file with the virtual address set to the virtual address of the start page of the region. Thread register contents are packed and written into a separate data section with the virtual address mapping to some memory range that is not used by the pinball. The resulting ELF object is then statically linked with a startup code that creates the required number of threads and initializes the thread contexts from the ELF file. The layout of such an executable is shown in Figure 3.

3) Stack collision: An ELFie generated from a fat pinball contains all the memory pages that were mapped into the memory of the traced process and includes code and data from the program binary, code and data from shared objects, and the program stack. When an ELFie is run natively, the system ELF loader first parses the ELFie file, maps its various program segments into memory, sets up the entry point, and finally initializes the stack for the ELFie process. The stack is populated in the normal way to contain command line arguments, auxiliary vectors, and arrays of pointers to environment variables of the process. However, the ELFie binary already has the stack pages from the pinball which may overlap with the address range of the new stack—which the loader is trying to reserve for the ELFie process. Even though the bottom of the stack can vary slightly from process to process, because of Linux's stack randomization feature, there is a possibility that the virtual addresses of the two stack regions, one requested by the loader and the other mapped from pinball, intersect. If such a collision happens, the loader will be able to reserve only a very small amount of the memory for the new stack which can be insufficient to pass the environment, program arguments, etc. In this case the process will be killed before any ELFie code is executed. Figure 4 illustrates the stack-collision problem.

Fortunately, the stack-collision problem can be solved. The sections in an ELFie corresponding to the stack from the parent pinball can be marked as *non-allocable*, which prevents the ELF loader from mapping them into memory during process initialization and lets the loader allocate the stack for the new process freely. At the very beginning of ELFie startup code, the pages of the newly created stack colliding with the stack from the parent pinball are unmapped, and replaced with pages

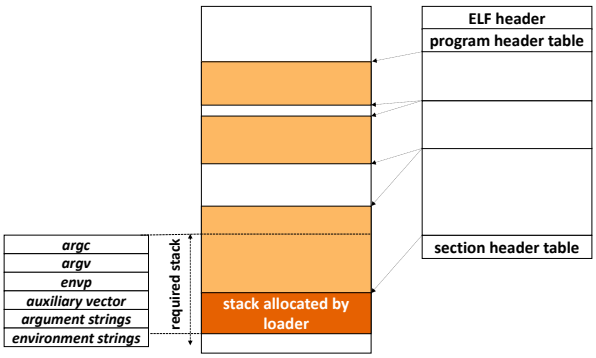

Fig. 4. A stack collision example.

```
_start:
 mov     8(%rsp),%rdi      # ELFie name
 xor     %esi,%esi         # O_RDONLY
 mov     $SYS_open,%eax
 syscall                   # open ELFie
 mov       %rax,%r15
 movabs  $PGADDR,%rbp
 movabs  $PGOFFS,%r14
 mov     $PGNUM,%r13d      # number of pages
pbs.remap_page:
 mov     (%rbp),%rdi       # page virtual address
 mov     $0x1000,%esi      # page size (4K)
 mov     $SYS_munmap,%eax
 syscall
 mov     (%rbp),%rdi       # page virtual address
 mov     $0x1000,%esi      # page size (4K)
 mov     $PROT_FLAGS,%edx
 mov     $MAP_FLAGS,%r10d  # MAP_PRIVATE|MAP_FIXED
 mov     %r15,%r8          # ELFie file descriptor
 mov     %r14,%r9          # page file offset
 mov     $SYS_mmap,%eax
 syscall
 add     $0x1000,%r14
 add     $8,%rbp
 sub     $1,%r13d
 jnz     pbs.remap_page
 mov     %r15,%rdi
 mov     $SYS_close,%eax
 syscall                   # close ELFie
```

Fig. 5. ELFie startup: remapping pages.

from ELFie segments containing the pinball stack. The code performing such remapping requires only a few system calls: `open()/close()` to open and close the ELFie being executed and `unmap()/map()` to perform remapping. The essential part of the remapping is a simple loop iterating over the array of addresses of pages to be remapped and successively unmapping and mapping corresponding pages. It is safe to unmap the stack of the new process since the startup code does not touch the stack allocated by the ELF loader. Figure 5 shows the code implementing such a procedure. Note that the remapping can be done safely for all pages from the pinball (and not just the stack pages) which is the most portable way.

4) Thread creation and initialization: The rest of the startup code creates threads and initializes their contexts.

```
pbs.init_thread:
 # restore thread context
 mov       $0x03,%eax    # xfeatures_lo_dword mask
 mov       $0x00,%edx    # xfeatures_hi_dword mask
 xrstor    (%rsp)        # restore extended state
 add       $state.fs_desc_offs,%rsp
 mov       $0x9a,%eax
 mov       $0x1,%edi
 mov       %rsp,%rsi
 mov       $state.desc_size,%edx
 syscall                 # restore fs
 . . .                   # restore gprs
 pop       %rbx
 pop       %rax
 popfq                   # restore flags
 # address of pbs.tet.t<N> is on top
 # of the stack at this point
 ret

 # thread entry table
pbs.tet.t<N>:
 pop       %rsp
 jmpq      *0x0(%rip)
pbs.tet.t<N>.rip:
 .quad     <THREAD_N_START_ADDRESS>
```

Fig. 6. ELFie startup: thread entry

Creation of threads is performed in a small loop that uses the `clone()` system call. It passes a pointer to the thread context data as the stack pointer of the thread being created and a pointer to the code loading the context (thread initialization function) as the thread function.

The context structure fully matches a thread's initial register state from the parent pinball and has two parts. The first part includes the X87/MMX, SSE, AVX, AVX-512, MPX, and other extended states. The layout of this part is the same as the FXSAVE/XSAVE area defined in the Intel Software Developers Manual [17]. This part of the context can be loaded either using the FXRSTOR or the XRSTOR instruction, depending on the CPU micro-architecture of the pinball. The second context structure part contains values of segment selectors (FS and GS bases), the flag register, and the general purpose registers (GPRs). The thread-initialization function receives these registers on the stack, and it pops and sets the flag and the GPRs. At the bottom of the thread-initialization function stack resides the pointer to a small piece of code which is called the thread entry. It sets up the stack pointer of the thread to the 'real' value and jumps to the thread's 'real' code. There are as many thread entries as there are threads in the parent pinball, one for each thread. Figure 6 shows an example of the thread entry. Note that the ELFie startup code only creates those threads that were created *before* the program region starts. New threads may still be created by the application itself during execution of the region, through the `clone` system call as normal.

5) Other pinball2elf features: The ideas described above are implemented in the pinball2elf tool. The tool can generate both ELF object files (without any startup code) and statically linked ELF executables (with startup code) from single and

multi-threaded pinballs. Additionally, pinball2elf allows users to link in extra code to be called at the beginning of each thread entry. It creates a linker script which gives users explicit control over the process of linking an ELFie object file with an object file containing user's extra code. The linker script contains the parent pinball memory layout, which is preserved in the resulting linked executable. Pinball2elf can also dump initial thread contexts in the form of assembly listing which can help users to write their own startup code. For ELFie debugging purposes, pinball2elf inserts symbols for all functions from the startup code, symbols for most of the elements of thread initial states in a format `.t<N>.<object>` (for example `.t0.rax`, `.t0.xmm` or `.t0.ext_area2`), and symbols pointing to the start of each thread.

Callback Support: Pinball2elf allows users to add calls to functions of their choice at two points early in ELFie execution:

1) `-p elfie_on_start()` after start-up but before starting application code execution,
2) `-t elfie_on_thread_start()` just before a thread jumps to user code.

These functions can be used to initialize hardware performance counters, for example, for native performance analysis of embedded application region. Another switch, `-e elfie_on_exit()`, causes a monitor thread to be created first. This thread spawns the main application thread and waits for it to exit. When the main application thread exits, a call to user-defined `elfie_on_exit()` is made. This function, for example, can be used to output final values of any hardware performance counters that were initialized on ELFie start.

Marker Support: When an ELFie is used for analysis or simulation, the startup code needs to be skipped. pinball2elf can add special 'marker' instructions at the beginning of application code using the switch `--roi-start [TYPE:]TAG` where *type* can be one of `sniper`, `ssc`, or `simics` respectively for Sniper [3], Pintools [18], and Simics [19].

Creating custom ELFies: Depending on the intended application (performance analysis or simulation), users can add the right callback routines and markers. The pinball2elf distribution has wrapper scripts with the sources for the required callback routines for common use cases. The execution flow of a typical custom ELFie is shown in Figure 7.

Debugging ELFies: The pages inside an ELFie containing application code are marked as not *loadable* hence tools, such as the GNU debugger (gdb), can not 'see' application pages inside an ELFie right away after the initial loading of an ELFie. For setting breakpoints at an application instruction, the suggested way is to first break on `elfie_on_start()` at which point all application pages are guaranteed to be in memory and then set a breakpoint at the desired application address(hex). Symbolic debugging of application code is currently not supported with ELFies although pinball2elf can be extended to add application debug information for symbolic debugging. ELFie generation scripts make sure debug information does exist for ELFie callback routines hence they can be debugged symbolically. For debugging multi-threaded ELFies with gdb, first doing a `set detach-on-fork off` followed by `break elfie_on_thread_start` and using `info inferior` and `inferior N` commands works well.

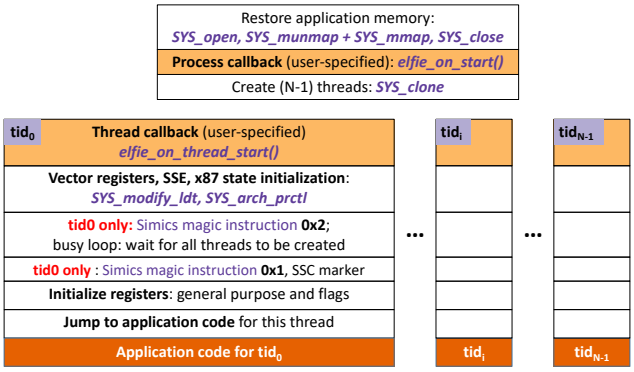

Fig. 7. ELFie execution flow.

C. ELFie Execution Challenges

As outlined earlier, there are two main challenges in ELFie execution.

1) The Graceful Exit Challenge: Figure 7 shows a typical ELFie execution flow. Once an ELFie thread starts executing application code, it can continue going, until it encounters an `exit()` or `exit_group()` system call. However, since an ELFie only represents a region of execution, it has data and text pages only to support that region. If an ELFie execution diverges from the captured execution or goes past the captured region, it might access a text or data page not present in memory and exit ungracefully. Therefore, we need a way to stop ELFie execution after the captured region ends. If a custom ELFie is being used by some dynamic analysis tool or a simulator, they can end ELFie execution explicitly using some region ending criterion (typically the instruction count recorded in the corresponding pinball). For ending a stand-alone execution of an ELFie, pinball2elf provides support for custom callback routines to program hardware performance counters for ending ELFie execution after the desired number of instructions (as recorded in the pinball) are executed (retired).

2) The System Call Handling Challenge: Unlike pinballs, ELFies do not have any injection mechanism to handle system calls—they are simply re-executed natively. Many system calls, such as `gettimeofday()` can be safely re-executed. However some system calls that rely on operating system resources, e.g., `read()` from an open file descriptor need special handling. Capturing all the operating system state in a checkpoint, as done by the CRIU infrastructure [14], can be quite a daunting task. Luckily, since ELFies only need to execute the captured region correctly, we know which system calls occur in the region (from pinball analysis) and need to handle them correctly. Based on our experience with mostly user-level programs (which are a good match for Pin-based

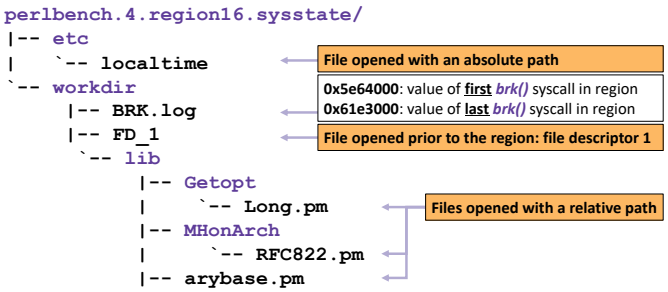

Fig. 8. Example `pinball_sysstate` output.

analysis anyway), we found two classes of system calls that need handling: (1) file-related system calls such as `open()`, `lseek()` and (2) heap memory handling system call `brk`. We have developed a Pintool, *pinball_sysstate* that looks at system calls in a pinball and extracts file-related state and puts it in a `pinball.sysstate` directory. Figure 8 shows an example output from the `pinball_sysstate` tool. Files that are opened inside the pinball region get a proxy version created with the right name and it is populated solely based on the relevant `read()` system calls in the region. Files opened prior to the region, and hence only referred via a file descriptor, have a proxy file with a dummy name *FD_n*, where *n* is the relevant file descriptor. The pinball2elf toolkit provides a generic `elfie_on_start()` callback that pre-opens any *FD_n* files in the sysstate directory and assigns them the right file-descriptor using the `dup()` system call. An ELFie is supposed to be executed in the `sysstate/workdir` directory created by the `pinball_sysstate` tool. Files that were opened using an absolute path-name during the pinball region, are copied to their rightful location (alternatively, the Linux command `chroot`, with the `sysstate` directory as the special *root*, can be used to execute the ELFie). The same Pintool, `pinball_sysstate`, also outputs the return values of the first and the last `brk()` system call in the pinball region in a file called `BRK.log`. A custom `elfie_on_start()` callback uses these values to set the memory layout for the ELFie process using the `prctl()` system call.

III. Applications

In this section we describe the use of ELFies for dynamic analysis, performance analysis, and simulation of regions of interest they represent.

A. Dynamic Analysis with Pin

Pin [1] is a popular dynamic instrumentation framework that works with x86 binaries and hence can handle ELFies. However, an ELFie executes some startup code before it jumps to the actual embedded code from the captured region of interest. Therefore, we need a way to skip Pin-based analysis of ELFie startup code. That can be done in multiple ways. Pinball2elf can add special *marker* instructions (`cpuid` or a special `nop`) and the Pintool doing the analysis can start the actual analysis only after reaching the special marker. Pinball2elf can be used to add a special callback function in an ELFie to be called before the application code executes and then Pin-based analyis can begin on seeing this function. Graceful exit of analysis can be achieved using a Pin-based instruction counting mechanism (based on the instruction count of the corresponding pinball) or using the hardware performance counter based exit mentioned in Section I.

B. Native Performance Analysis

ELFie captures a region of interest from an application execution and provides a precise way to focus performance analysis on that region. While Linux utilities like *perf stat* will work with ELFies, they need a way to avoid measuring the startup code and handling exit gracefully. Pinball2elf features for adding process-wide and per-thread callback functions are useful in this context. Pinball2elf provides a library, *libperfle.a*, with an API for initializing hardware performance counters. Per-process and per-thread callback functions can be used to program desired hardware performance counters and also to exit each thread gracefully on executing its expected (as listed in the pinball) instruction count. A callback on process exit can also be inserted using pinball2elf which causes a monitor thread to be created that watches the (graceful) exit of the application process and then can output the final values of various performance counters programmed.

C. Workload Simulation

While a Pin-based simulator, such as Sniper [3], can be modified to work with replay of pinballs, it does require extra work. With ELFies, which are x86 Linux binaries, simulations can be performed for the region of interest without having to modify the simulator. There is still a requirement to be able to skip the startup code during simulation which can be done using special *marker* instructions or special instruction/call-back function addresses as described in Section II-B5.

We have tested ELFies with three different x86 simulators:

1) *Pin-based multi-core simulator:* Sniper [2], [3] is a Pin-based x86 multi-core simulator which works with x86 Linux binaries. It has been modified to include the PinPlay library [13] and hence can simulate pinballs as well. We use Sniper to evaluate the performance of both a set of multi-threaded pinballs and the corresponding ELFies.
2) *SDE and Simics-based x86 simulator:* CoreSim is an Intel-internal cycle-accurate x86 many-core simulator. This simulator supports SDE [20] as a front-end (which itself is based on Pin, but adds emulation of future instructions) as well as Simics [19] to enable full-system simulation. Given that CoreSim is an execution-driven simulator, ELFies can run on CoreSim natively just like any Linux executable. We added Simics magic instructions to the ELFies so we can enable the performance model only after the ELFie startup code has been executed.
3) *Binary-driven x86 simulator:* gem5 [4] is an open source computer architecture simulator supporting multi-

ple ISAs, including x86. gem5 can simulate a complete system and an OS in Full-System (FS) mode, or user space only programs where system services are provided directly by the simulator in Syscall Emulation (SE) mode. Often, gem5 is used with *atomic CPU* and *fast-mem* options to collect Basic Block Vectors (BBVs) as input to a sampling methodology such as SimPoint [5]. However, generating pinballs and ELFies is much faster, which is useful for studies that involve frequent re-compilation of binaries.

IV. Case Studies

In this section, we present some case studies showing the application of ELFies to performance analysis and simulation. Although the focus mostly is architecture simulation, many other dynamic analyses can benefit from the ability to focus on regions of interest via ELFies.

A. Validating Simulation Region Selection with ELFies

Simulation region selection is an active research area. To measure the quality of simulation region selection, the standard practice is to report an error in projection based on simulation of just the selected regions. The error is computed by comparing some metric, such as cycles per instruction (CPI) obtained two ways: (1) with simulation of the entire program (gives the *true* value) and (2) with simulation of just the selected region, and by using simulation region *weights* to compute the estimate for the entire execution (gives the *predicted* value). The problem with this approach is that (1) requires simulation of the entire program which can take a really long time and hence prompts the need for simulation region selection in the first place. To make the validation practical, researchers either use a reduced length benchmark run (e.g. *train* instead of *ref* for SPEC2006/2017) or use a faster, but crippled, version of their simulator. ELFies provide quite an elegant way out of this situation by enabling hardware counter based metric computation for selected regions. Interesting metrics can now be collected by running the entire program and ELFies for selected regions to get *true* and *predicted* values. Using real hardware instead of a simulator, simulation region selection for really long-running programs can be quickly validated.

1) Validating train SPEC2017 Simulation Regions: The objectives of this case-study were (1) to compare ELFie-based and tranditional, simulation-based, simulation validation techniques; and (2) to show the use of ELFies in the tuning of simulation region selection.

We applied the PinPoints [8] simulation region selection methodology to *train* runs of SPEC CPU2017 [21] *rate*, *int* subset. All the benchmarks in the suite were compiled using the GCC (version 8) with optimizations enabled (-O2) and built for the 64-bit instruction-set architecture. We used a *slicesize* (region length) parameter of 200 million instructions, a warmup region of 800 million instructions and *maxK* (maximum number of phases/regions) value of 50. The dynamic instruction count of the benchmarks used was in the range 1.3–452 billion.

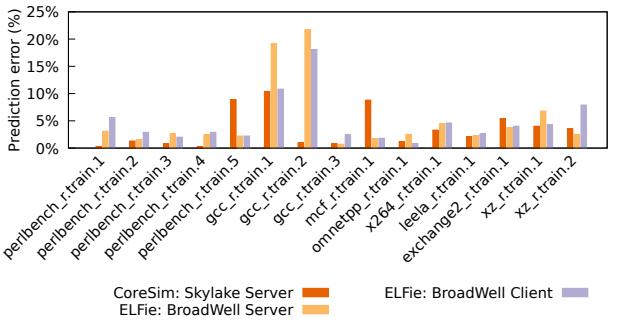

Fig. 9. Prediction errors: Simulation vs. ELFie-based.

TABLE II
Warm-up tuning for Prediction Error reduction.

Warm-up	800 M		1.2 B	
	BDW server	BDW client	BDW server	BDW client
gcc_r.train.1	19.2%	10.8%	2.3%	1.8%
gcc_r.train.2	21.7%	18.1%	2.8%	2.5%

We did validation of these PinPoints using the traditional approach first, where both the region and whole-program cycles-per-instructions (CPI) were computed with detailed simulation using CoreSim. Even though we used *train* inputs to keep the whole-program simulation time reasonable, it still took a few weeks for the longest simulation to finish. With ELFie-based validation, real hardware was used instead of simulation for getting whole-program and region CPI values. We did ten trials for each measurement and took the average. Still, the entire process finished within one hour which is a drastic reduction over the few weeks of turnaround time with the traditional approach. We use a common definition of prediction error: *((whole_program_CPI) – (region_predicted_CPI)) / (whole_program_CPI)*. How did the prediction errors compare with the two approaches? Figure 9 shows the prediction errors with simulation-based approach and two instances of ELFie-based validation. While the errors do not match exactly, they follow similar trends.

Figure 9 shows some high errors, especially for *gcc* which is notoriously hard to represent and requires some PinPoints tuning to bring down the prediction error. In this case, increasing the size of warm-up region from 800 million instructions to 1.2 billion instructions brought down the prediction error as shown in Table II.

2) Validating ref SPEC2017 Simulation Regions: The objective of this case-study was to validate simulation region selection for long-running workloads. We applied the PinPoints methodology to *ref* runs of SPEC CPU2017 [21] *rate int* and *fp*, again compiled using GCC 8 with optimizations enabled (-O2), a slicesize of 200 million instructions, 800 million instructions warmup and a maxK value of 50. Some basic statistics for our benchmarks are shown in Table III. With

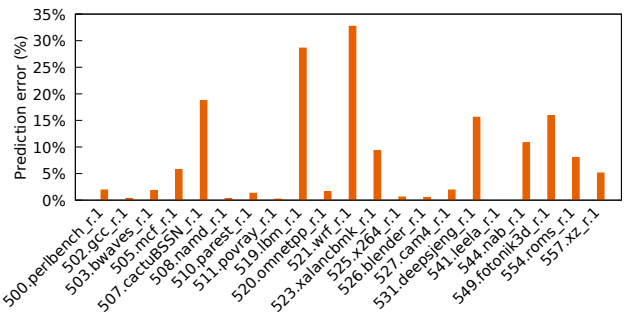

Fig. 10. SPEC CPU2017 *ref* PinPoints prediction errors.

TABLE III
SPEC CPU2017 *ref* INPUT BENCHMARK STATISTICS.

program.input	global-icount	# PinPoints
500.perlbench_r.1	1,183.2 B	14
502.gcc_r.1	205.2 B	20
503.bwaves_r.1	1,027.1 B	32
505.mcf_r.1	966.5 B	32
507.cactuBSSN_r.1	1529.7 B	36
508.namd_r.1	2648.3 B	38
510.parest_r.1	4025.9 B	42
511.povray_r.1	3521.6 B	29
519.lbm_r.1	1595.3 B	20
520.omnetpp_r.1	1088.2 B	3
521.wrf_r.1	3881.8 B	36
523.xalancbmk_r.1	1294.5 B	31
525.x264_r.1	580.5 B	29
526.blender_r.1	1791.0 B	17
527.cam4_r.1	2820.0 B	30
531.deepsjeng_r.1	1914.2 B	22
541.leela_r.1	2382.2 B	28
544.nab_r.1	2197.8 B	20
549.fotonik3d_r.1	2222.4 B	36
554.roms_r.1	3183.0 B	38
557.xz_r.1	413.3 B	26

rather large instruction counts (trillions in some cases), traditional simulator-based validation would have taken months to finish hence was not even attempted. ELFie-based validation, with ten trials each, on the other hand, finished in a few hours. CPI prediction errors on an Intel Broadwell server are shown in Figure 10. The high error cases can be tuned in a way similar to what was done in Section IV-A1 earlier for gcc with *train* inputs. While such tuning was not the objective of this case study, the relatively high projection errors obtained by an untuned set of PinPoints clearly illustrates its need—yet tuning of the SimPoint parameters over multiple iterations would not be possible without the hardware-based methodology enabled by ELFies.

B. Deterministic vs. Execution-Driven Multi-Threaded Simulation with Sniper

The objectives of this case study were to (1) show the use of multi-threaded ELFies for simulation and (2) compare the simulation of an ELFie and its parent pinball.

For multi-threaded pinball and ELFie simulation with Sniper, we specified end of simulation as a *(PC,count)* pair where PC was the address of a specific instruction at the end of the code region outside any spin-loops or synchronization code and count was its execution count (globally, across all threads) determined using a separate profiling run.

We present our results comparing the simulation performance of arbitrary multi-threaded regions of SPEC CPU2017 applications (the OpenMP subset from the *speed* version) collected as pinballs and the corresponding ELFies. For this case study, we use a configuration that mimics an Intel Gainestown out-of-order 8-core processor.

For our experiments with Sniper, we use the *speed* version of the SPECCPU2017 benchmark suite with *train* inputs running with eight threads. The benchmarks are compiled using Intel C++ Compiler with -O2 optimization enabled and built for the 64-bit instruction-set architecture. We collect fixed-length regions from the execution of each application in the benchmark suite as pinballs. These can be replayed or simulated later, following the recorded memory access patterns and synchronization among the threads. We keep the size of the chosen multi-threaded region close to 2.4 billion instructions (aggregate for eight threads). We use *active* wait policy in OpenMP for the threads, which means that the threads use CPU cycles while waiting.

The thread ordering during replay of multi-threaded pinballs is pre-determined at various synchronization and memory access points. This means that the simulation is constrained and can introduce an artificial stall among certain threads, that yields inaccurate simulation results [22]. However, simulating a multi-threaded ELFie is totally un-restricted, and therefore, the simulation results are more realistic. Figure 11 shows the comparison of Sniper simulation results using ELFies and pinballs. We observe that the instruction counts of pinball simulations and the corresponding recorded instruction counts of the pinballs closely match. However, the instruction counts of similar ELFie simulations are much higher because of the presence of spin-loops, and the threads behaving in a non-deterministic way, unlike pinball replay. The runtime prediction of both sets do not match which is expected as the simulation for pinballs is constrained and that for ELFies is not. Among the applications presented here, 657.xz_s.1 is single-threaded, therefore the instruction count remains the same for un-constrained simulation using ELFies as well as for constrained simulation using pinballs.

C. Application-Level vs. Full System Simulation with CoreSim

The objectives of this case study were (1) to show the use of ELFies with a system-level, Simics-based, simulator, and (2) to compare user-level (SDE-based) and system-level (Simics-based) simulation for the same set of ELFies.

Conventional wisdom is that for compute-bound applications, instructions executed by the operating system have little effect on performance. Many popular academic simulators, including Sniper and several others, are built on Pin as it provides an easy framework for user-space x86 functional

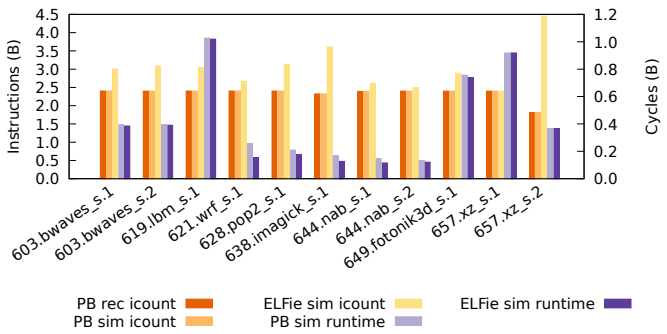

Fig. 11. Sniper results using *multi-threaded* ELFies and pinballs (PB).

TABLE IV
CORESIM SIMULATION RESULTS FOR X264. MPKI UNITS DENOTE MISSES PER 1,000 INSTRUCTIONS.

Metric	User-space	Full-system	Difference
Instructions (B)	10.008	10.172	+1.6%
Runtime (ms)	1.337	1.408	+5.2%
IPC	3.117	3.011	-3.4%
Branch (mpki)	1.41	1.41	-0.4%
TLB (mpki)	0.008	0.010	+29.2%
I-cache (mpki)	0.580	0.940	+62.0%
D-cache (mpki)	1.263	1.508	+19.4%
LLC (mpki)	0.553	0.687	+24.2%
Footprint (MB)	79.1	115.0	+45.4%

simulation. But how big is the error that this class of simulators make by ignoring OS interference?

Using ELFies and CoreSim, we can now make a direct comparison of user-space only vs. full-system simulation, with a realistic workload and input set size. We simulate an identical ELFie (a 10 B instruction single-region SimPoint of 525.x264 from the SPECCPU2017 rate/train suite) on CoreSim's detailed Intel Skylake model, once with SDE as the front-end (simulating user-space instructions only) and once with Simics (providing full-system simulation).

Results of relevant low-level architectural statistics are shown in Table IV. The number of simulated instructions in user-space only simulation equals the expected length of the ELFie (just over 10 B instructions). In full-system mode, the number of instructions executed in ring3 (userspace) was equal, but there were an additional 165M instructions executed by the kernel in ring0. These extra 1.6% instructions increased the simulated runtime by 5.2%, in part due to the additional pressure on TLBs and caches. The total data footprint accessed during the run was 45.4% larger in the full-system simulation; the relatively few OS instructions had a disproportionate effect on prefetcher activity and memory bandwidth pressure, especially for applications that have a small data footprint.

D. Binary-Driven Simulation with gem5

The objective of this case-study was to show the use of ELFies with a popular x86 simulator, gem5, that is *not* Pin-based while being driven by x86 binaries.

TABLE V
GEM5 SIMULATION: SPEC CPU2006 ELFIES.

Application	Total slices	Representative (single) slice number	IPC NHM	IPC HSW
400.perlbench	1070	576	0.426	0.434
401.bzip2	178	68	0.385	0.385
403.gcc	143	43	0.289	0.291
410.bwaves	2118	231	0.233	0.417
416.gamess	371	314	0.590	0.631
429.mcf	322	2	0.116	0.121
433.milc	912	198	0.160	0.388
434.zeusmp	1688	1448	0.261	0.456
435.gromacs	2009	44	0.550	0.997
436.cactusADM	1425	1181	0.232	0.555
444.namd	2053	1	0.495	0.496
445.gobmk	210	76	0.401	0.407
450.soplex	330	124	0.164	0.177
453.povray	966	465	0.431	0.471
454.calculix	4368	3679	0.350	0.399
462.libquantum	1386	1013	0.154	0.219
464.h264ref	464	164	0.353	0.409
465.tonto	2891	2265	0.540	0.583
471.omnetpp	643	29	0.255	0.264

Table V shows the IPC of 19 applications from SPEC CPU2006 simulated with gem5 for two processor configurations (Intel Nehalem-like and Haswell-like processors) to study the impact of increasing the size of critical resources (like register file, ROB, load/store queues, etc.). We used GCC to compile the programs and SE mode of gem5 to simulate the ELFies. We used SDE [20] for profiling as gem5 only supports the SSE and SSE2 extensions for the x86 ISA; we also use the SDE `-pentium` core option to limit the ISA extensions available. The size of each region is 1 B instructions and we use SimPoint to find the most representative region of the code. Total slices (column 2) denotes the number of 1 B instruction regions in entire execution of the program. The representative slice number (column 3) is the slice selected by SimPoint [5]. The IPC values (column 4) are those reported by the simulator. These results help to demonstrate the flexibility of ELFies.

V. RELATED WORK

Ringenberg and Mudge [23] present a methodology for converting program execution regions into Intrinsically Checkpointed Assembly Code (ITCY), which can be simulated as a binary application. Based on a functional simulator, static assembly instructions from representative regions are extracted and extra sections added to handle memory initialization, system call emulation, and exit. Application code gets relocated to a new address causing several complications. The resulting binary provides repeatable execution, much like the PinPlay replayer does with one important difference: the simulator sees the extra book-keeping code, possibly perturbing results. Their technique works for single-threaded Alpha workloads. ELFies move away from determinism provided by pinballs and ITCY, supports x86 binaries and multi-threaded workloads.

Checkpoint/restore in user space (*CRIU*) [14] is a very robust and popular check-pointing technique on Linux and was

described in Section I. DMTCP [24] is a user-level checkpointing technique for distributed, multi-threaded checkpointing. A dynamic library is injected in each user process started under their check-point command with a checkpointing manager thread spawned. User-space state is saved, data in the network is drained to process memory, and kernel state (including open file descriptors) is probed and stored in the check-point. The restart under DMTCP has significant complexity around restoring the kernel state. DMTCP does not support statically-linked program binaries. DMTCP checkpoints are not stand-alone executables like ELFies and are, in general, larger.

VI. Summary

Analyzing entire runs of large programs can be time consuming. Analyzing only interesting portions from long-running executions poses the challenge of precisely reaching the regions for multiple analyses. We address this challenge by presenting a set of tools that capture regions of interest from program executions into stand-alone executables called ELFies. ELFies always start with the exact same program state captured at the beginning of the regions of interest, avoiding the time and the uncertainty in reaching those regions. While ELFies are generated using a Pin-based tool-chain, they can be run natively or with any analysis tool, Pin-based or not, that executes program binaries. Also, they can be easily shared among researchers for reproducing analysis results and comparing analyses with the exact same input binary.

As part of our work, we created thousands of ELFies for regions of interest from popular single and multi-threaded benchmarks. We showcase the use of ELFies for simulation, native performance analysis and for validating simulation region selection. We believe that ELFies can be useful in many scenarios and have made our tools publicly available [16].

Acknowledgements

This work has benefited tremendously from the support, contribution, and suggestions of many individuals. We sincerely thank all of them. In particular, we acknowledge Robert Cohn, Karthik Sankaranarayanan, Ady Tal, Moshe Klausner, Alexey Klimkin, Dan Baum, Olivier Serres, Carl Beckmann, and Christian Karl. We also thank the anonymous reviewers and artifact evaluators for their suggestions. This work was partially funded by a Startup Grant from the National University of Singapore and a grant from the Intel Corporation.

References

[1] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2005. doi:10.1145/1064978.1065034 pp. 190–200.

[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *Transactions on Architecture and Code Optimization (TACO)*, pp. 28:1–28:25, Aug. 2014. doi:10.1145/2629677

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011. doi:10.1145/2063384.2063454 pp. 52:1–52:12.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. doi:10.1145/2024716.2024718

[5] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002. doi:10.1145/605397.605403 pp. 45–57.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2013. doi:10.1109/ISPASS.2013.6557141 pp. 2–12.

[7] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Jun. 2014. doi:10.1109/ISPASS.2014.6844456 pp. 2–12.

[8] "Pinpoints from Intel," https://software.intel.com/content/www/us/en/develop/articles/pin-a-binary-instrumentation-tool-pinpoints.html.

[9] "SPEC CPU2006 and Simulation," https://www.spec.org/cpu2006/research/simpoint.html.

[10] "SPEC CPU2017 and Simulation," https://www.spec.org/cpu2017/research/simpoint.html.

[11] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," http://refspecs.linuxbase.org/elf/elf.pdf, May 1995.

[12] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *International Symposium on Code Generation and Optimization (CGO)*, Apr. 2010. doi:10.1145/1772954.1772958 pp. 2–11.

[13] Intel, "Program Record/Replay Toolkit," http://www.pinplay.org.

[14] "Checkpoint/Restore in Userspace," http://criu.org.

[15] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, "Automatic logging of operating system effects to guide application-level architecture simulation," in *Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Jun. 2006. doi:10.1145/1140277.1140303 pp. 216–227.

[16] "Pinball2elf (Intel)," http://github.com/intel/pinball2elf, http://pinelfie.org.

[17] Intel, "Intel® 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2016.

[18] "Pintool Regions," https://software.intel.com/en-us/articles/pintool-regions.

[19] D. Aarno and J. Engblom, *Software and System Development Using Virtual Platforms: Full-System Simulation with Wind River Simics*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN 978-0-12-800725-9

[20] "Intel Software Development Emulator," http://www.intel.com/software/sde.

[21] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *International Conference on Performance Engineering (ICPE)*, Apr. 2018. doi:10.1145/3185768.3185771 pp. 41–42.

[22] C. Pereira, H. Patil, and B. Calder, "Reproducible simulation of multi-threaded workloads for architecture design exploration," in *IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2008. doi:10.1109/IISWC.2008.4636102 pp. 173–182.

[23] J. Ringenberg and T. N. Mudge, "SuiteSpecks and SuiteSpots: A methodology for the automatic conversion of benchmarking programs into intrinsically checkpointed assembly code," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009. doi:10.1109/ISPASS.2009.4919654 pp. 227–237.

[24] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *International Symposium on Parallel Distributed Processing (IPDPS)*, May 2009. doi:10.1109/IPDPS.2009.5161063 pp. 1–12.