# Hijacker: Efficient Static Software Instrumentation with Applications in High Performance Computing

POSTER PAPER

Alessandro Pellegrini
Dipartimento di Ingegneria Informatica, Automatica, Gestionale
Sapienza, University of Rome
pellegrini@dis.uniroma1.it

***Abstract*—Static Binary Instrumentation is a technique that allows compile-time program manipulation. In particular, by relying on ad-hoc tools, the end user is able to alter the program's execution flow without affecting its overall semantic.**

**This technique has been effectively used, e.g., to support code profiling, performance analysis, error detection, attack detection, or behavior monitoring. Nevertheless, efficiently relying on static instrumentation for producing executables which can be deployed without affecting the overall performance of the application still presents technical and methodological issues.**

**In this paper, we present Hijacker, an open-source customizable static binary instrumentation tool which is able to alter a program's execution flow according to some user-specified rules, limiting the execution overhead due to the code snippets inserted in the original program, thus enabling for the exploitation in high performance computing. The tool is highly modular and works on an internal representation of the program which allows to perform complex instrumentation tasks efficiently, and can be additionally extended to support different instruction sets and executable formats without any need to modify the instrumentation engine.**

**We additionally present an experimental assessment of the overhead induced by the injected code in real HPC applications.**

## I. Introduction

Binary Instrumentation is a technique which modifies a binary program by inserting additional instructions or by changing existing ones at compile time (*static instrumentation*) or at runtime (*dynamic instrumentation*), in order to observe or modify the execution's flow, without altering the overall program's semantic. This technique has been successfully exploited in several fields, such as behavior monitoring [1], performance analysis [2], attack detection [3], or to alter code for supporting transactional memories [4]. Intrumentation tools like Pin [5], Dyninst [6], Valgrind [7], and DynamoRIO [8] have been extensively used in order to analyze various applications and observe their behavior.

In static instrumentation, the executable is analyzed at a certain stage during its compiling process (i.e. before or after the final linking stage), while in dynamic instrumentation some logic is inserted into the application so that, when the process is launched, the control is taken by a runtime module which alters the code during the actual program's execution. While the latter could be regarded as a more powerful technique, due to the fact that at runtime more information is available to the runtime disassembling module (e.g. any conditional branch can be successfully evaluated), the former technique allows for the creation of more efficient executables, due to the fact that only the strictly-necessary instrumentation code is inserted in the application binary. Considering that we explicitly address High Performance Computing scenarios, Hijacker necessarily falls in the former category, although, as it will be discussed later, few additional runtime tasks are performed.

Static binary instrumentation has nevertheless some disadvantages. In fact, the instrumentation process cannot target third-party libraries, especially shared libraries. In fact, if a static instrumentation tool were to target such libraries, the resulting instrumented library would affect every executable in the system which is relying on it. Considering that instrumentation could significantly affect the behavior of the code (depending on the user needs), this approach would be non-viable. On the other hand, Hijacker offers a set of tools to redirect specific (third-party libraries) functions calls to user-defined stubs, or to efficiently wrap specific ones, thus giving the user the freedom to wisely use third-party code under controlled execution flow.

The process of instrumenting an executable poses two challenges: on the one hand, since instrumentation works on machine-level code, this process is intrinsically instruction-set-dependent. On the other hand, in order for the user to correctly modify the application's flow (without altering its semantic), she has to manually provide the tool with the additional code to be injected, which can be a non-trivial task, since it may depend on the actual compiler, architecture and calling conventions specified by the current ABI. In order to hide this complexity away, we have specifically designed Hijacker in order to provide three main features: i) the instrumentation process is *rule-driven*, i.e. the operations performed by Hijacker are specified via an xml file, which instructs the tool on the specific tasks to be performed in the process; ii) the most common tasks can be performed *transparently*, since Hijacker comes bundled with a set of *instumentation features* (e.g. target memory address reconstruction) which can be inserted into the original binary; iii) the process of instrumenting the code is instruction-set and executable independent, i.e. Hijacker performs its tasks on an internal binary representation, decoupling the specific details of the underlying architectures and therefore allowing the tool to instrument the same original high-level code compiled for different architectures.

The remainder of this paper is structured as follows. In Section II related work is presented. Section III describes the basic design and implementation of Hijacker. Examples of applications in HPC are provided in Section IV. Finally, in Section V we present the experimental assessment of an application instrumented using Hijacker.

## II. Related Work

There are several works in literature and several tools which address the problem of instrumentation. The earliest implementations of binary instrumentation toolkits are ATOM [9] and EEL [10]. They both instrument code at compile time, avoiding as mush as possible runtime overhead. ATOM is targeted at alpha machines only, while EEL tries to hide the complexities of the underlying instruction sets providing abstract C++ interfaces for altering the code. On the contrary, our tool drives the instrumentation process by relying on rules provided in a xml file.

BIRD [11] is a binary rewriting platform for Windows/x86 only. This tool basically relies on the insertion of a 5-byte branch instruction in order to give control to instrumentation code, or relies on interrupts when 5 bytes are not available. This technique is similar to the one presented by PEBIL [12], although the latter tool is targeted at Linux boxes and uses function relocation allowing the tool to rely on the 5-byte branch at any instrumentation point. The main difference from our proposal is that we completely rebuild the final executable, and thus there are no limits in the amount of code which can be inserted in-place (i.e. without relying on calls to other portions of the executable)

Dyninst [6] is a tool for static and dynamic instrumentation. It can either create a modified version of the binary at compile time, or can operate at runtime. The most notable feature of this tool is the ability to perform liveness analysis on registers' values and on flag register's bits.

Some tools like Pin [5] rely on just-in-time instrumentation. In particular, Pin can be seen as a middleware which places itself under the original application, and at applications interrupt points it instruments the upcoming parts of the original code. Just-in-time instrumentation is used as well by other tools like DynamoRIO [8] and Valgrind [7]. The latter tool offers a very large set of functionalities, ranging from memory-management errors detection to cache utilization analysis. Nevertheless, while incredibly useful, techniques used by this tools produce a heavyweight overhead, making it not suitable for deployed HPC applications.

## III. Design and Implementation

In the compiling process, Hijacker lays just before the final linking stage. In fact, we have explicitly decided to work on a relocatable representation of the executable because we can rely on the additional linking metadata in order to perform our application analysis and build our internal binary representation.

Hijacker's architecture is divided in a front-end module—which provides several compatibility layers with different executable formats and assembly languages, and is able to parse xml rule files—and a back-end module—which performs the actual instrumentation operations on an intermediate (machine-independent) representation of the executable. Hijacker is open source[1], and is available as one of the tools released by the HPDCS research group[2]. The overall architecture is depicted in Figure 1.

[1]http://www.dis.uniroma1.it/∼pellegrini/?p=hijacker

[2]http://www.dis.uniroma1.it/∼hpdcs

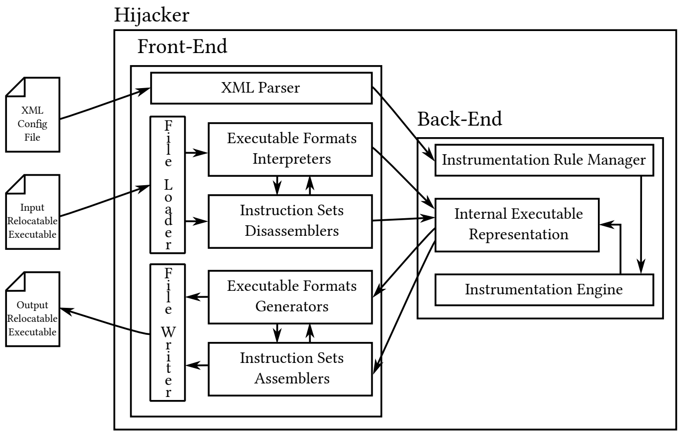
Hijacker
Front-End
XML Config File
Input Relocatable Executable
Output Relocatable Executable
XML Parser
File Loader
File Writer
Executable Formats Interpreters
Instruction Sets Disassemblers
Executable Formats Generators
Instruction Sets Assemblers
Back-End
Instrumentation Rule Manager
Internal Executable Representation
Instrumentation Engine

Fig. 1. General Hijacker's Architecture

### A. Rule Specification

As mentioned before, Hijacker is a rule-based instrumentation tool. To support the instrumentation process and to leverage the user from many technical details, we allow her to instruct Hijacker via a simple xml file, an example of which is provided in Figure 2. As it can be seen, the user can instruct Hijacker to perform several actions on the executable on a per-function basis or on a global-basis.

In particular, the configuration file gives the freedom to insert manually-written portions of code (by relying on the `<Inject>` tag). The code should be written in the target machine's assembly language, or in a higher language for which a compiler is available on the machine, and is automatically compiled by Hijacker. Instructions to be altered are specified using the `<Instruction>` tag, which supports several attributes: `instruction`, specifies either a single (target-architecture) assembly instruction, or a (machine-independent) family of instructions, as it will be clearly discussed in Section III-B; `injectBefore`, `injectAfter`, and `replace` specify either an assembly code file or a specific instruction to be placed (respectively) before, after or in place of the related instruction. The `<AddCall>` tag allows the user to insert specific calls to original or injected functions in the code. Several attributes are allowed: `where` determines whether the call is placed before or after the target instruction; `function` specifies which function must be called; `arguments` specifies which arguments should be passed to the callee (as it will be explained in Section III-D); `convention` determines if the arguments are passed either by stack or by registers (respecting the target architecture's calling convention). The user can specify which operations should be performed on a specific function by enclosing the relevant tags in the `<Function>` tag, where the function name is specified in the `name` attribute. We note that if `<AddCall>` is used within a `<Function>` tag, the meaning of the `where` attribute specifies whether the function call is performed after the function specified in `<Function>` is called (i.e., the call is injected in the function's code) or before any call to it in the executable.

When Hijacker is launched, the front-end's XML parser module loads the configuration file and instructs the back-end's instrumentation rules manager about the operations which will be performed during the instrumentation process.

```
<HijackerRules>
  <Inject file="memorycopy.c"/>
  <Function name="foo">
    <Instruction instruction="I_MEMWR|I_MEMRD" injectBefore="memcount.S">
     <AddCall where="before" function="monitor" arguments="target" convention="stack"/>
    </Instruction>
  </Function>
  <Instruction instruction="I_JUMP|I_CONDITIONAL" injectAfter="jumpcount.S">
    <AddCall where="before" function="monitor" arguments="register" convention="stdcall"/>
  </Instruction>
  <Instruction instruction="movs" replace="nop">
    <AddCall where="after" function="memorycopy" arguments="target" convention="registers"/>
  </Instruction>
</HijackerRules>
```

Fig. 2. Example configuration file

## B. *Application Analysis and Internal Binary Representation*

In order to perform the instrumentation tasks, the original (relocatable) executable must be processed first. The front-end's file loader module performs a sequence of tests on the executable in order to determine which executable format is used to represent the program, and triggers the corresponding executable format interpreter (among the ones registered at Hijacker's compile time) in order to start loading the program. The first step undertaken by the executable interpreter is to check which assembly language is used to represent instructions in the executable, and this information is reported back to the file loader manager, which searches among the available disassembler engines for one able to interpret the code. If a suitable disassembler is found, this information is reported to the executable interpreter.

This modular approach allows any combination of assembly languages and executable formats for the representation of a program, and allows for high extendibility of the tool, decoupling the process of adding supports for new/additional formats and languages from the instrumentation process itself. At the time of this writing, Hijacker is bundled with an ELF format interpreter, an x86 disassembler and an x86_64 disassembler, while a PE interpreter and an ARM disassembler are under development.

The executable interpreter then starts analyzing the program and builds an intermediate representation of it which we refer to as *program map*. This program map is structured in sections, whose type can be code, data, or raw. The latter section describes any type of section which is not involved in the instrumentation process, and is therefore straight recreated in the altered program. The data section keeps track of global data used by the program, in terms of name of the variables (if any), their size in byte (if available), and their initial value.

The code section actually contains an intermediate representation of the instructions. In particular, the executable interpreter gives control to the suitable disassembler in order to start a linear scan of the assembly code. Each assembly instruction gets stored into a data structure that keeps the original bytes of the instruction, along with several attributes describing the instruction itself. Each data structure keeps a set of pointers to target adjacent instructions in the function, data (if any), and/or other instructions (if any, in case of branches or function calls). Depending on the actual assembly language used by the program (and on the compiler which generated the code), this process can present more or less difficulties. As an example, some compilers emit data within the code in order to support efficient execution of, e.g., indirect banches deriving from the compilation of switch cases. The disassembler module is able to communicate with the executable interpreter if, during the instrumentation process, some data segments are discovered within the instructions[3]. In this case, the executable interpreter adds to the data section the additional data structures, enforcing the logical separation between data and code used to build the program map. Of course, this action slightly changes the "shape" of the altered program which will be produced as output of the instrumentation process, but does not change its operational flow.

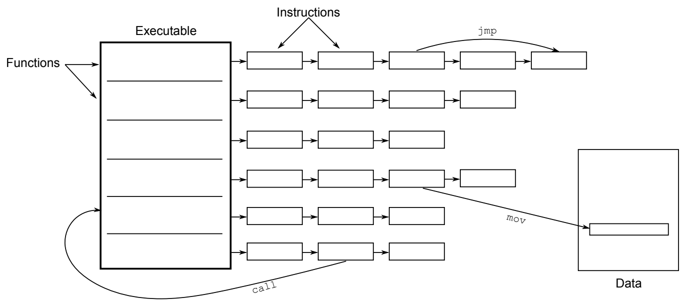

Fig. 3. Hijackers Internal Representation of Executables

During the linear scan of the assembly code, Hijacker's disassemblers cross-check the information retrieved from symbol and relocation tables from the original relocatable program. In this way, the disassembler is able to decode instructions and organize their internal represenation in functions, preserving the connection among instructions and the data, in order to produce an intermediate representation of code as depicted in Figure 3. We note that the connections between instruction-instruction (due to branches in the code), instruction-function (due to function calls), and instruction-data (due to data movement) are realized in the intermediate representation as memory pointers, rather than offsets as most assembly languages do. Considering that functions are represented as linked lists of instructions (the elements' order in such lists is their appearance in the code, which does not necessarily correspond to the program's execution flow), inserting or removing any instruction at any point of code does not alter the linking between objects in the intermediate representation.

Whenever an instruction is interpreted by the disassembler,

[3]The methodologies used by disassembler modules for discovering such portions of data are different, depending on the architecture and its ABI, and discussing them is out of the scope of this paper. Nevertheless, to give the reader an idea, they mostly rely on decompilation error-retry algorithms, coupled with address/register-value evaluation.

it gets marked using special flag values which describe the actual *family* of the instruction. The available flags and their meanings are:

I_MEMRD: The instruction reads from memory
I_MEMWR: The instruction writes to memory
I_CTRL: The instruction performs checks on data
I_JUMP: The instruction alters the execution flow
I_CALL: The instruction calls a different function
I_RET: The instruction returns from a callee
I_CONDITIONAL: The instruction is executed only if a condition is met
I_STRING: The instruction operates on large amount of data
I_ALU: The instruction does some logical/arithmetic operation
I_FPU: The instruction does some floating-point operation
I_STACK: The instruction works on stack
I_INDIRECT: The instruction behavior might depend on some runtime value

or any or'ed combination of them. For example, an instruction marked as `I_MEMWR|I_MEMRD` reads from and writes to memory, while an instruction marked as `I_JUMP|I_CONDITIONAL` is an indirect branch. These combinations of flags can be used in the `instruction` attribute described in Section III-A to specify instrumentation rules for groups (families) of instructions which perform a certain action.

## C. Code and Data Instrumentation

Once the program map is completely built, the execution control is given to Hijacker's back-end, in particular to the instrumentation rules manager. This module starts applying the rules parsed from the xml configuration file, by triggering (for each rule) the corresponding operation in the instrumentation engine.

The instrumentation engine operates on the internal intermediate representation of the executable. Whenever some rule requires a modification in a particular function, the corresponding entry is found in the functions array and the instructions's data structures are linearly scanned in order to identify the ones which must be instrumented. If the rule involves adding some instructions before or after the target one, some new nodes are simply inserted in the list of instructions. For the generation of machine-level code, the instrumentation engine asks the instruction-set assembler to produce the machine-level representation of the instruction. If the insertion of code entails the compilation of some assembly file, the default compiler installed on the host machine (`cc`) is invoked automatically. When Hijacker is launched, a custom compiler can be specified, which will override this setting.

As hinted before, whenever some rule is applied, the connections between instructions, functions and data are preserved since they are realized as memory pointers between structures. This is true even in the most complex cases: if instructions are referred from the data section (e.g. in the aforementioned branch table case) then the approach described in Section III-B had notified executable interpreter about the presence of data segments within the code. If the disassembler module was able, at runtime, to discover the presence of references to instructions, then they are replaced with memory pointers targeting the destination instructions. This approach has been shown to cover most of the code generated by standard compilers. Nevertheless, there are some cases which involve the generation of instruction addresses completely at runtime. If such a case is found, then Hijacker is not currently able to correctly instrument the executable, although we are working on making the technique discussed in [13] more efficient and suitable for HPC.

## D. Bundled Instrumentation Features

During the instrumentation process, disassemblers populate the data structures used for representing instructions in the intermediate representation with all the information which can be gathered from their binary representation during the process. This information can be stored in the executable and therefore used at runtime by, e.g., user-injected functions. This is exactly the goal of the `arguments` attribute described in Section III-A, and the one of the *bundled instrumentation features*: provide the end user with some cached disassembly information in order to allow some sort of dynamic monitoring without having to rely on any kind of dynamic instrumentation.

At the time of this writing, Hijacker provides the end user with two bundled instrumentation features, namely *target address reconstruction* and *indirect register detection*. The former is a feature which allows, whenever an `I_MEMRD` or `I_MEMWR` instruction is being instrumented, to insert some additional code which evaluates by software the destination address of such instruction, and the size of the access. This information can be passed as argument of any (user-injected or not) function in the binary.

The branch register instrumentation feature is similar in spirit to target address reconstruction. This feature allows any function in the code to be called passing two arguments: an integer code representing the register which is used to execute an indirect memory acccess (either for reading/writing or for altering the control flow with an indirect branch), and its actual runtime value. This can be particularly useful to trace at runtime the actual execution patterns of an application, when this information cannot be detected at compile time.

## E. Binary Multiversioning

Hijacker can make multiple (differently-instrumented) versions of the same executable coexist in the same image, in case more `<Executable>` tags in the xml configuration file are found. This facility has been successfully exploited in [13] to create differently-instrumented versions of the same original application-level program which coexist and the execution flow is passed from one version to the other.

In order to support such a scenario, Hijacker allows to specify a new entry point for the program. This facility can be used to insert, e.g., some logic in the instrumented application which dinamycally selects one of the two versions of the application binary depending on the current execution dynamics. This approach can be regarded as a means for efficiently creating executables which, according to the autonomic paradigm [14], are able to efficiently react to changes in the execution dynamics by, e.g., selecting a different operating mode. The different operating modes could be realized as differently-instrumented versions of the same executable, leveraging the programmer from implementing slightly different versions of the same program, a process which is inherently long and error-prone.

From a technological point of view, whenever two or more versions of the executable must be created, the rule manager asks the instrumentation engine to create multiple copies of the program map as the first action. Each copy will have its internal symbols renamed adding a progressive number, so that if the user wants to inject code which calls either instrumented version, she will be able to do so consistently. The data section is not duplicated, so to allow a consistent sharing of data among the versions. Each program map version will be then instrumented applying the related rules.

### *F. Binary Recreation*

The last step in the instrumentation process is the recreation of a (relocatable) executable which can be later passed to the linker to complete the compiling process. When the rules manager has finished applying modifications to the program map(s), the control is returned to Hijacker's front-end.

The proper executable formats generator is thus triggered in order to recreate a new executable on disk. This process entails repeatedly accessing the program map(s) in order to build all the data structures which are required by the format itself. As it can be clearly seen, this process is highly format-dependent, and the description of the operations involved would be in the scope of specifically-targeted papers. Nevertheless, to give the reader an idea, they are similar in what actual compilers do whenever they are generating a program from a set of source files.

### *G. Third-party libraries*

The possibility to rely on third-party libraries depends on the actual behavior of used functions. In fact, as an example, if the user is instrumenting the executable to track memory updates and then relies, e.g., on standard `memset` library function, then memory updates performed by `memset` would not be tracked.

As mentioned before, there are several solutions to this problem, each one having different drawbacks. In order to provide efficient solutions to this issue, we have explicitly avoided to rely on costly runtime analysis approaches, while on the contrary we have given the user the possibility to wrap third-party library calls, by relying on a specific `<Library>` tag. In this way, the user can specify which are the external library calls which she wants to wrap, and therefore any call in the executable to them will be actually redirected to user-specified functions, in a way similar to what standard linkers allow to do during the compilation process.

## IV. Applications in HPC

In this Section, we briefly highlight some application fields which can benefit by tools like Hijacker. Code instrumentation can be efficiently used in order to support memory access tracing procedures, and therefore collect information which can be used to create incremental snapshots of programs memory maps for, e.g., efficiently enforce failure recovery, or to support speculative execution. This is the case, e.g., of [15], where memory writes are instrumented for creating incremental snapshots of application's memory map in the context of optimistic simulation.

Another relevant field is that of autonomic computing [16], [14], [17], i.e. computing systems which are able to actively react to changes in the execution dynamics due to internal or external solicitations. This is the case, e.g., of [13] or [18], where the coexistence of differently instrumented versions of the same executable is exploited in order to dinamically reselect the best suited execution mode depending on the actual execution dynamics.

Of course, this tool can be used as well in the context of code profiling, in order to detect which portions of an applications are the actual bottlenecks of the execution. In fact, the user can just maintain one single version of the sources, and write rules to inject efficient profiling routines to track the execution performance of single functions, class of functions or even single code snippets.

The problem of finding bugs in an HPC program is non-trivial, especially when in complex systems it is not possibile to reproduce the problem in the development/testing phase. On the contrary, if the production version of the software is lightweightly instrumented, it is possible to trace execution and store light metadata which allow, by analyzing a core dump, to step back between instructions (restoring previous snapshots of the memory map) and find where the actual bug is. This technique, referred to as Post-Mortem Debugging [19], can be supported by instrumentation tools like the one hereby described.

## V. Experimental Results

In order to evaluate the instrumentation overhead induced by the proposed architecture, we have conducted experiments on a family of configurations of *Personal Communications Service* (PCS), parallely run on top of ROOT-Sim [20], an optimistic parallel discrete event simulator. PCS is a GSM wireless communication systems simulation model, where channels are modeled in high fidelity via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding cell. Also, the power regulation model has been implemented according to the results in [21]. Accurate descriptions of this model can be found in [22]. However, for the reader's convenience we report below some details related to the measures which we have taken.

Upon the start of a call destined to a mobile device currently hosted by a given wireless cell, a call setup record is instantiated via dynamically-allocated data structures, which gets linked to a list of already active records within that same cell. Each record gets released when the corresponding call ends or is handed-off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call-setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call-setup to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations). The climatic model accounts for variations of the climatic conditions (e.g. the current wind speed) with a minimum time granularity of ten seconds.

We have performed a set of experiments where each cell sustains the same workload of incoming calls, whose inter-arrival time is exponentially distributed, and whose average duration is set to 2 minutes. The call interarrival frequency to each cell has been varied in the interval between 1 and 6.25 calls per simulation time unit, thus providing increasing values of the channel utilization factor (in between 12% and 75%), and hence increasing values of the expected length

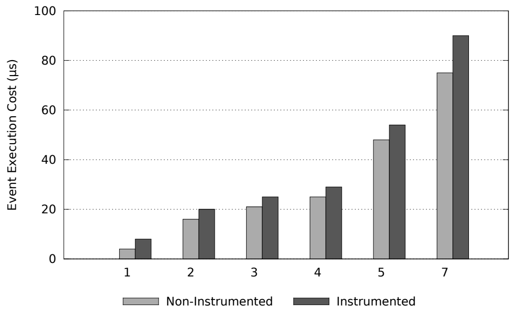

Fig. 4. Overheads associated with different workloads

of the aforementioned list of in-use records. The residence time of an active device within a cell has a mean value of 5 min and follows the exponential distribution. This has the effect of performing an increasing number of memory updates whenever the climatic model starts scanning the allocated channels for recomputing the optimal power allocation values. For the above scenario, we have run experiments with 1024 wireless cells, modeled as hexagons covering a square region, each one managing 1000 wireless channels. These have been evenly distributed across 32 kernel instances running on a 64-bit NUMA machine, namely an HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache.

In Figure 4 we present the results for the various PCS configurations (i.e. interarrival frequencies) in two cases: the first represents the execution of PCS instrumented using Hijacker, where every memory-write access (i.e. `I_MEMWR` instructions) have been instrumented by placing before each of them a call to a hand-written module which relies on the target address reconstruction bundled instrumentation feature for creating a memory access map; the second represents an execution of the original benchmark, with no instrumentation. The instrumented scenario is non trivial, since it additionally requires the execution of a runtime module to compute the actual memory addresses. Figure 4 shows average execution time of PCS events in both cases. As it can be clearly seen, the overhead added by the modifications of the code by Hijacker are negligible, while by relying on the instrumented code the user is able to reduce actual management costs by using incremental logging facilities like the ones described in [15].

## VI. Conclusion and Future Work

In this work we have presented Hijacker, a rule-based static binary instrumentation tool, which is able to modify at compile time the execution flow of an executable without altering its overall semantics. We have shown implications by the usage of this tool in the context of HPC applications, and we have provided an assessment of the overhead induced by the injection of code in a deployed executable.

Future work entails augmenting the set of bundled instrumentation features in order to increase the level of transparency provided to the user, the development of more front-end disassemblers and format interpreters, and an overall assessment of the produced instrumented binaries in more complex/differentiated scenarios.

## References

[1] W. Drewry and T. Ormandy, "Flayer: exposing application internals," in *Proceedings of the first USENIX workshop on Offensive Technologies*, ser. WOOT. USENIX Association, 2007, pp. 1:1–1:9.

[2] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal on High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[3] J. Newsome, D. Brumley, and D. Song, "Vulnerability-specific execution filtering for exploit prevention on commodity software," in *Proceedings of the 13th Symposium on Network and Distributed System Security*, ser. NDSS, 2005.

[4] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A dynamic binary-rewriting approach to software transactional memory," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT. IEEE Computer Society, 2007, pp. 365–375.

[5] Pin, http://www.pintool.org/.

[6] DynInst, http://www.dyninst.org/.

[7] Valgrind, http://valgrind.org/.

[8] DynamoRIO, http://www.dynamorio.org/.

[9] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Languages and Design Implementation*. ACM, 1994, pp. 196–205.

[10] EEL, http://pages.cs.wisc.edu/∼larus/eel.html.

[11] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chiueh, "Bird: Binary interpretation using runtime disassembly," in *International Symposium on Code Generation and Optimization*, ser. CGO, Mar. 2006.

[12] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *IEEE International Symposium on Performance Analysis of Systems Software*, ser. ISPASS, Mar. 2010, pp. 175–183.

[13] R. Vitali, A. Pellegrini, and F. Quaglia, "Autonomic log/restore for advanced optimistic simulation systems," in *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS. IEEE Computer Society, 2010, pp. 319–327.

[14] P. Horn, "Autonomic computing: IBMs perspective on the state of information technology," vol. 15, pp. 1–39, 2001.

[15] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS. IEEE Computer Society, 2009, pp. 45–53.

[16] S. Hassan, D. Al-Jumeily, and A. J. Hussain, "Autonomic computing paradigm to support system's development," in *Proceedings of the 2nd International Conference on Developments in eSystems Engineering*, ser. DESE. IEEE Computer Society, Dec. 2009, pp. 273–278.

[17] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41 – 50, Jan. 2003.

[18] A. Pellegrini, R. Vitali, and F. Quaglia, "An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools. SIGSIM, 2011.

[19] D. Pacheco, "Postmortem debugging in dynamic environments," *Communications of the ACM*, vol. 54, no. 12, pp. 44–51, Dec. 2011.

[20] HPDCS Research Group, "ROOT-Sim: The ROme OpTimistic Simulator - v 1.0," http://www.dis.uniroma1.it/∼hpdcs/ROOT-Sim/, Oct. 2012.

[21] S. Kandukuri and S. Boyd, "Optimal power control in interference-limited fading wireless channels with outage-probability specifications," *IEEE Transactions on Wireless Communications*, vol. 1, no. 1, pp. 46–55, 2002.

[22] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards symmetric multi-threaded optimistic simulation kernels," in *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS. IEEE Computer Society, Aug. 2012, pp. 211–220.