Language-Agnostic Optimization and Parallelization for Interpreted Languages

Michelle Mills Strout, Saumya Debray, Kate Isaacs, Barbara Kreaseck, Julio Cárdenas-Rodríguez, Bonnie Hurwitz, Kat Volk, Sam Badger, Jesse Bartels, Ian Bertolacci, Sabin Devkota, Anthony Encinas, Ben Gaska, Brandon Neth, Theo Sackos, Jon Stephens, Sarah Willer, and Babak Yadegari

University of Arizona,
Department of Computer Science
Tucson, Arizona,
mstrout@cs.arizona.edu

Abstract. Scientists are increasingly turning to interpreted languages, such as Python, Java, R, Matlab, and Perl, to implement their data analysis algorithms. While such languages permit rapid software development, their implementations often run into performance issues that slow down the scientific process. Source-level approaches for parallelization are problematic for two reasons: first, many of the language features common to these languages can be challenging for the kinds of analyses needed for parallelization; and second, even where such analysis is possible, a language-specific approach implies that each language would need its own parallelizing compiler and/or constructs, resulting in significant duplication of effort.

The Science Up To Par project is investigating a radically different approach to this problem: automatic parallelization at the machine code level using trace information. The key to accomplishing this will be the static and dynamic analysis of executables and the reconstitution of such executables into parallel executables. The key insight is that with trace information it should be possible optimize out the interpreter and other dynamic features in a language-agnostic manner and create parallelized executables for multicore architectures. If successful, this can enable scientists to continue to develop in programming environments that most conveniently support their scientific exploration without paying the performance overheads currently associated with many such environments.

1 Introduction

Scientific communities, such as medical imaging, the life sciences, and planetary sciences, rely extensively on computer software to process and analyze the wealth of data they and others are generating. In recent years, interpreted languages such as Python, Perl, and R have come to dominate data analysis software development in many areas of science: for example, most of the bioinformatics software developed in the last five years was implemented in Python, JavaScript, or Perl [7]. Such languages have been referred to as productivity

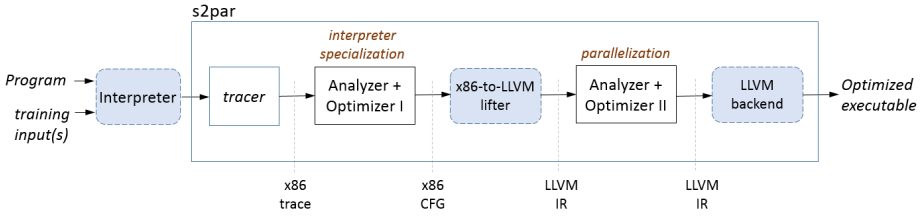


Fig. 1. s2par tool. Solid boxes are modules being developed as part of this project; dashed shaded boxes represent third-party software.

languages [2]. The high-level abstractions supported by such languages enable rapid prototyping that, together with the re-use of contributed code from the scientific community, has led to productivity gains in the development of data analysis and simulation programs.

Unfortunately, some of the features that make these languages productive, e.g., dynamic typing, dynamic error checking, not requiring programmers to specify the parallelization strategy, and being interpreted, incur significant runtime overheads and lead to execution times that are orders of magnitude more than programming languages such as Fortran, C/C++, and parallel programming languages. Scientists can therefore either work within the constraints of inefficient software, which can limit the problem sizes they can address; or rewrite their software in a different programming language, where they would also have to port their colleagues' algorithms to reuse sub-routines and/or compare results. Neither alternative is very appealing because of the iterative nature of data analysis algorithm development that involves evolving the algorithms based on feedback from evaluating such algorithms on large datasets. The Science Up To Par project, presented in this paper, aims to provide the advantages of current alternatives while mitigating their disadvantages.

Our goal is to bring multicore parallelism to scientists while still allowing them to use programming environments that most conveniently support their scientific exploration. *We are developing a language-agnostic, trace-guided optimization tool that operates directly on the productivity-language software written by scientists. This tool will combine dynamic instrumentation and analysis with aggressive optimization and parallelization to create specialized and parallelized executables for use with large datasets based on example runs with small representative datasets.* Figure 1 illustrates the Science Up To Par optimization tool (`s2par`) and the dynamic analysis toolchain that `s2par` is built on.

Scientists will extend their development cycle with a step where they let the Science Up To Par optimization tool trace the processing of small example datasets. The optimization tool will provide a specialized, optimized, and parallelized executable based on the traces. Scientists will then be able to analyze their larger datasets with the provided executable. Our usage goal is for the tool to "just work" as illustrated in the following example:

```
s2par --profile <profile_name>  python myprog.py <parameters>
s2par --optimize <profile_name> -o newexec
./newexec <parameters for larger datasets>
```

Thus, scientists will be able to continue using the programming languages that they are most productive in while still being able to leverage multicore resources to analyze large datasets with scientifically useful turnaround times.

To achieve these goals, we need to solve the following technical problems:

- Given an execution trace (a sequence of machine instructions), how can we separate out the control-flow and data-flow logic of the interpreted program (the *interpretee*) from those of the interpreter?
- How representative are traces of small inputs in scientific codes?
- How can we detect parallel and/or reduction loops in the recovered control flow graphs?
- How can we implement the parallel loops without assuming an underlying memory model, (i.e., without assuming arrays are being used)?
- How can we efficiently catch control-flow that did not occur in the traced input execution but does for larger datasets?

The remainder of this paper overviews the progress we have made in solving these problems.

2 Control-Flow and Data-Flow Separation

The machine-level instruction sequence observed in an execution trace reflects control flows and data flows resulting from a combination of the program logic of the interpreter and the interpreted program. This intermingling of the logic of these two programs can hamper parallelization. For example, branch instructions in the interpreter's dispatch code can result in spurious control dependencies, while data movement to and from the interpreter's expression evaluation stack can result in spurious data dependencies. To permit effective parallelization, therefore, we have to separate out the program logic of the interpreted program from that of the interpreter. This involves a number of nontrivial challenges, for example:

- Translating from the input program to the interpreter's internal representation (IR) of that code involves the interpreter's front-end (possibly including the compiler that generates the IR), whose logic can be complex and difficult to untangle.
- Different interpreters may use different IRs, e.g., a linear array of byte code instructions, as in Python and Java, or a tree representation, as in Perl and some implementations of Ruby.
- The dispatch mechanism may be different, e.g., byte code as in Python and Java, direct-threading as in Ruby.
- Some of the interpreter code may be created dynamically at interpreter startup time, as in the Hotspot template-based interpreter for Java [6].

– Depending on the optimizations performed by the interpreter front end, the dispatch code may be replicated, resulting in multiple different dispatch instructions in the executed code (e.g., as in optimized CPython).
– An interpreter that supports multi-threading (or simulates it, as with the `thread` library in CPython) may have multiple virtual instruction pointers (vips), making it necessary to untangle the code corresponding to the different vips.

Many of these issues arise from the diversity of design choices available for implementing interpreters, and they mean that a language-agnostic system such as that proposed here cannot make *a priori* assumptions about any particular design choice. For example, we cannot assume that the IR is a byte-code, or that it occupies a contiguous region of memory. Coming up with effective ways to identify and reason about interpreters and interpreted programs under weak assumptions is a major research thrust of this research.

2.1 Control-flow Separation

Control-flow separation refers to the process of untangling and separating the control flow logic of the interpreted program from that of the interpreter. Furthermore, in an interpreted execution of a program, control dependencies in the input program are mapped to data dependencies through the interpreter's virtual instruction pointer (vip) [14]. For example, a conditional branch in the input program is implemented by updating the value of the vip appropriately, thereby inducing a data dependence through that variable. These data dependencies have to be identified, and the corresponding control dependencies reconstructed, when separating out the control flow logic of the interpreter from that of the input program (see Figure 2).

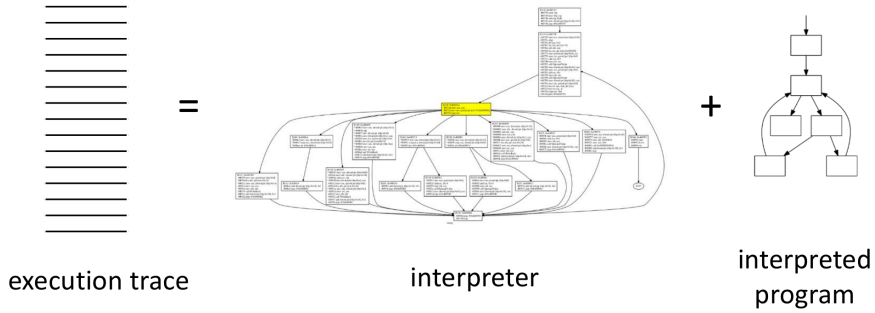


Fig. 2. Illustrating the process of deriving the interpreters control flow graph from a trace and then specializing then deriving the control flow graph of the program being interpreted.

We are decomposing the interpreter specialization problem into a collection of smaller and simpler problems.

1. Given the file from which the input program is read (specified, for example, as a command-line argument), identify the memory regions corresponding to the IR of the program being interpreted. An example of such an IR is the byte code for the input program.
2. Given the set of locations comprising the input program's IR, identify the control transfers corresponding to the dispatch instruction(s) of the interpreter.
3. Given the set of dispatch instructions, identify the machine instructions corresponding to the handler for each byte code instruction and thereby reconstruct the control flow graph of the input program.
4. Given the control flow graph of the input program, identify and optimize out inefficiencies due to interpretation.

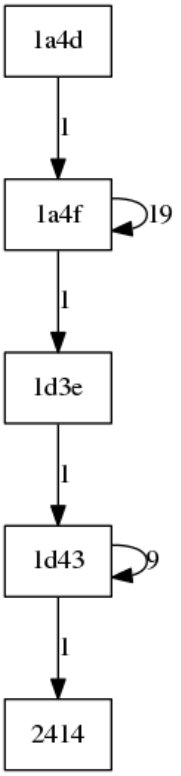


Fig. 3. Example recovered CFG

We propose to use *dynamic taint analysis* [10] (augmented to deal with implicit flows through control dependencies) to follow the flow of values through the computation: e.g., from the input program through the front end to the IR; and from the IR to the dispatch code. To obtain accurate results, it will be essential to minimize imprecision arising from over-tainting; we propose to apply ideas from our earlier work on bit-precise architecture-aware taint analysis [12, 13] to address this issue.

To explore the viability of these ideas, we have experimented with a simple prototype tool for analyzing interpreter traces for a variety of different languages, including Java, Perl, Python, and Ruby. These experiments have helped identify, and clarify our understanding of, many of the research challenges identified above. This prototype does not address the issues that arise from the interactions between the interpreter's code and data structures (e.g., the interpreter's expression stack) as well as interactions with other components of the runtime system (e.g., the garbage collector). Nevertheless, we have been able to make progress on the third research subproblem mentioned above: namely, given a set of dispatch instructions, reconstruct the control flow graph of the input program.

We recover the control flow graph using a method broadly analogous to that of Sharif *et al.* [11], though significantly different in details. Like Sharif et al, we employ a multi-label taint analysis to identify the dispatch of an interpreter; unlike that work, however, we do not make any assumptions about the interpreter or the interpreted IR (e.g., Sharif *et al.* assume a bytecode interpreter where the executed bytecode is laid out as a contiguous array of memory locations—assumptions that do not hold for AST interpreters as for Perl and direct-threaded interpreters as for Ruby). The generality of our approach, while important for applicability to a wide variety of interpreters, can sometimes result in an over-approximation of the bytecode executed. Additionally, we extend past

Sharif's work by using the identified bytecode to construct a CFG of the input source file, allowing us to determine not only what x86 instructions are related to a particular bytecode instruction, but also what x86 instructions execute a particular instance of a bytecode instruction. With this information, we believe the interpreter can be optimized for a particular input program using techniques similar to those employed by trace based JIT compilers [1].

Experiments using the above approach on a few small programs have been encouraging. Figure 3 shows the control flow graph of a histogram loop written in Python, recovered from the dynamic trace using our method. Each node represents a basic block of byte codes, each bytecode is composed of multiple x86 instructions, and the label on the node represents the address of the first x86 instruction in the basic block. The edge labels represent dynamic trip count. Our method correctly retrieves two loops, one to generate the histogram and another to write it out and reconstitutes them into a working executable.

2.2 Data-flow separation

Data-flow separation refers to the process of separating the data-flow logic of the interpreted program from that of the interpreter and the runtime system. The issue arises because computations of data values in the interpreter involve data movement into and out of a set of locations used for expression evaluation (e.g., an expression stack, as in the Java Virtual Machine and CPython interpreter; or virtual registers, as in the Dalvik virtual machine and the SPIM interpreter for MIPS assembly code). The reads and writes involving these locations can then induce spurious dependencies between instructions. Such dependencies can also arise from data movements in the runtime system, e.g., due to garbage collection or just-in-time compilation.

We plan to apply compiler optimization techniques to effect data-flow separation. For example, using an SSA representation may allow us to identify and separate out distinct uses of expression evaluation locations in the interpreter, such as the expression stack, without having to presuppose any particular mechanism for expression evaluation. There may also be complexities arising from architectural features, e.g., the stack of floating point registers on x86 and x86-64 processors.

3 Small Datasets Appear Representative

A potential drawback of optimization based on dynamic analysis is that of code coverage: the only code paths observed in dynamic analyses are those executed with the profiling inputs. This can be problematic if the "real" datasets exercise code paths that deviate from those observed on profiling runs. Avoiding correctness problems resulting from such deviations requires adding additional runtime checks into the code, which then incur some runtime overhead.

As an initial check that the dynamic analyses performed on small input data are representative enough to be applied to larger-scale target inputs without

loss of correctness, we examined coverage between the training and reference inputs of twelve SPECfp-2006 benchmarks.[1] We used our binary-level dynamic analysis toolset to determine, for each benchmark program tested, the fraction of the machine code executed on the reference inputs that was also executed on the training inputs.

Our experiments indicate that, on average, 96.5% of the code executed on the reference inputs is also executed on the training inputs, with ten of the programs having >99% coverage. Only one, `calculix`, had a significant difference (69%) in coverage of the scientific features. This suggests that while smaller training inputs may not always provide complete coverage of the code executed on the reference inputs, in most cases the difference will likely be small. To guide scientists in choosing a set of small representative inputs, we plan to automatically pre-check inputs and provide feedback when coverage tools exists for their language. Examples of such tools include `Figleaf` for Python, `simplecov` for Ruby, and `Devel::Cover` for Perl.

When coverage is incomplete, we must insert appropriate and efficient runtime checks to ensure correctness should a given input attempt to access a non-traced feature of the software. Interpreter specialization will impose strong assumptions regarding facets of the program such as control flow and data types. We will identify the locations of these assumptions, encode those assumptions as checks, and insert the checks into the control flow where they dominate the assumption. For example, the Python interpreter includes many conditional checks involving type information. We can assume types do not change while data analysis is being performed, assume the largest version of the datatype (e.g., double vs. float) used during training runs on representative inputs should be used throughout, and therefore remove extraneous conditional checks.

We also plan an in-depth analysis of these scripts to determine the characteristics of the aliasing used in their data structures, with a focus on multiply-referenced values in the same data structure, amount of indirection, and the differences between small and large inputs with respect to these measures. Multiply referenced values could affect the correctness of our parallelization. This analysis will guide our strategy in handling these cases.

4 Parallelism Exists

Data analysis scripts contain significant parallelism. In current and previous work [4, 3], we have been collaborating with scientists who write data analysis codes in Python, Perl, Matlab, and Julia. These data analysis codes typically have a single bottleneck loop. The bottleneck is often a reduction of some kind: adding items to a list, set, or matrix or performing some calculation and maintaining summary information.

[1] We had problems building and running the remaining five benchmarks in the SPECfp-2006 suite. Some of these problems may have been due to non-standard-conformant code in the benchmarks.

As part of an in depth analysis of scientific data analysis codes, done in conjunction with a graduate level course, we ported Matlab and Perl scientific data analysis codes to new parallel programming models, leading to significant speedups (60x for a medical imaging analysis) [3, 5]. Although some of this speedup was due to porting a snap-shot of the program to a compiled programming language, through this process we also discovered significant parallelism in the computations causing performance bottlenecks (in [5] over 6× speedup on 8 cores for an orbital analysis code in Python and in [3] over 7× speedup for a medical imaging analysis code in Matlab).

Finding the parallelism in a trace that includes the interpreter code as its interpreting is more challenging than finding parallelism of a compiled program. Oh et al. [9, 8] showed that if an interpreter is specialized for a specific input program, it is possible to find pipeline and speculative loop-level parallelism at the LLVM IR level. They found that performing profiling at the LLVM level significantly reduced the speculation overhead thus leading to decent parallel scaling with some Lua and Perl programs.

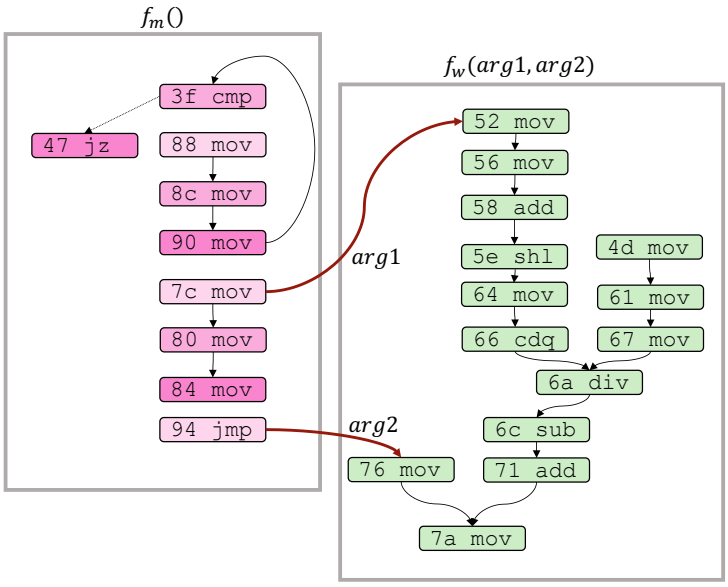
5 Implementing the Parallelism

We plan to implement loop-level parallelism by breaking the LLVM IR instructions from time-consuming loops into two sets: the instructions that perform the (parallelizable) work and the instructions that determine the next iteration, including the loop completion. A master thread will execute the iterator code and then spawn off tasks to a task pool implementation.

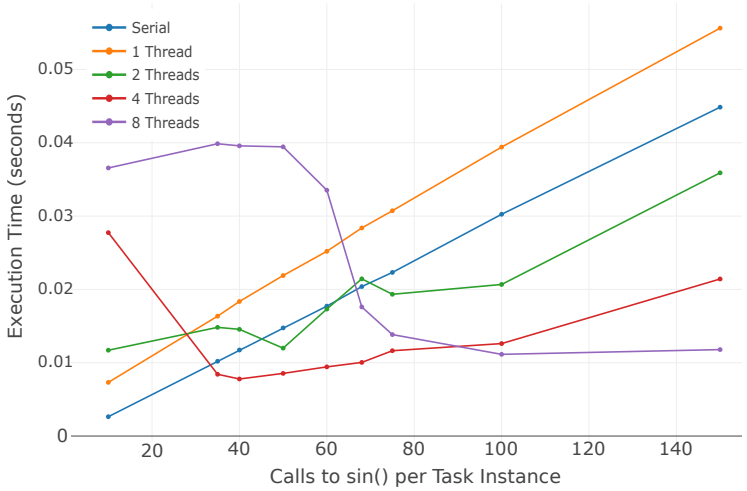
The proposed work includes plans to raise the interprocedural control flow graph of x86 instructions into annotated LLVM and then analyze for parallelism. In our initial experiments, we determined how to find parallelism in the full x86 traces using a back tainting analysis (e.g., equivalent to backward slicing. The example code was a C++ loop traversing an input linked list, performing some busy work computation in the form of a loop of `sin()` calls, and writing the sum of those `sin()` call results into a node of the output linked list.

Figure 4(a) illustrates the split of x86 instructions for one loop iteration into a master thread that deals with the linked list traversal that needs to be serialized and the task instructions (minus the `sin()` loop due to space considerations) for tasks that can be computationally overlapped. We have also started experimenting with finding the parallelism in the traces. The algorithm for finding the split involves identifying loop-exit branches and upwards-exposed reads per iteration and placing the instructions that influence those into the master thread. The leftover instructions can be encapsulated into a worker task function.

To implement the found parallelism, we use POSIX threads (pthreads). Using a hand-implemented version of the example loop at the C level, we see promising results. The goal of these tests were to determine the size of work tasks required to see a performance increase over serial execution. To see execution speeds on par with the serial execution, the work function must execute 8,000 instructions



(a) The x86 instructions of one loop iteration are split into those to be serialized by the master thread (pink) and those that can be parallelized into tasks (green). Parameters needed by the tasks are also determined (red lines).



(b) Execution time scaling with number of threads and amount of discovered parallel work. As the work per iteration crosses the equivalent of 35 calls to sin(), our method run on four threads executes faster than the original C serial code.

Fig. 4. Proof of concept identification of serial master thread and parallel worker instructions from an x86 trace of a C program and performance of an initial pthreads implementation.

with two worker threads, 5,000 with four, and 11,000 with eight. This corresponds to 50, 40, and 70 calls to sin(), respectively. Figure 4(b) summarizes these results.

Another major issue that will need to be addressed is the detection of reduction dependences in the loop and implementing their parallelism. Their detection and implementation will be somewhat intertwined, because it will be more complex than a read, add, and update operation on a register. Some of the values experiencing a reduction might be values in a dictionary. Therefore the reduction operation will have to be detected within a trace through memory loads and stores as well as operations on registers. We plan to leverage the existing research on source-level detection of reductions and other commutative operations and extend that to the LLVM IR.

One possible approach for handling reductions is that once the loads and stores involved have been detected, the implementation will be built by putting off the loads and stores to the shared memory accesses. Each thread could be given its own map that maps memory addresses to value and address pairs at runtime. At end of executing all tasks in a loop, all thread maps would be reduced into shared memory.

With reduction and loop parallelism detected, the next step will be experimenting with implementation approaches that leverage that parallelism while amortizing overhead. The most problematic overhead will probably be the serial bottleneck of the master thread. Providing each thread its own address space and then mapping results back into shared spaces once large-grained tasks are complete might help break this bottleneck, but will introduce memory copying overhead. Experimentation and modeling of the various tradeoffs will be needed.

6 Conclusion

This research aims to develop software analysis, optimization, and parallelization techniques to obtain significant performance improvements in research software developed by scientists (including several collaborators from a diversity of scientific disciplines). Our goal is to do this in a way that (a) is language-agnostic and transparent to the scientists, so that they can continue to work in the programming language of their choice and (b) leverages state-of-the-art compiler technology to effectively utilize multicore parallelism. The impact of this research will be two-fold: first and foremost, it will benefit a wide variety of scientists—most immediately medical imaging analysis and life sciences at the University of Arizona but also elsewhere—by boosting the speed of their research software with little to no additional effort on their part; second, it will benefit computer science research by developing new techniques for software performance optimization and thereby giving rise to additional new and exciting research problems.

References

1. C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation,*

Compilation, Optimization of Object-Oriented Languages and Programming Systems, pages 18–25. ACM, 2009.
2. B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.
3. F. Danford, E. Welch, J. Cárdenas-Ródriguez, and M. M. Strout. Analyzing parallel programming models for magnetic resonance imaging. In *The 29th International Workship on Languages and Compilers for Parallel Computing (LCPC)*, 2016.
4. B. J. Gaska. Parforpy: Loop parallelism in python. Master's thesis, University of Arizona, 2017.
5. B. J. Gaska, N. Jothi, M. S. Mohammadi, K. Volk, and M. M. Strout. Handling nested parallelism, load imbalance, and early termination in an orbital analysis code. Technical Report arXiv:1707.09668, University of Arizona, 2017.
6. T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.
7. P. Lindenbaum. Programming language use distribution from recent programs/articles, Apr. 2017. `https://www.biostars.org/p/251002/`.
8. T. Oh, S. R. Beard, N. P. Johnson, S. Popovych, and D. I. August. A generalized framework for automatic scripting language parallelization. In *To appear in the Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
9. T. Oh, H. Kim, N. P. Johnson, J. W. Lee, and D. I. August. Practical automatic loop specialization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 419–430, New York, NY, USA, 2013. ACM.
10. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
11. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109. IEEE, 2009.
12. B. Yadegari and S. Debray. Bit-level taint analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2014.
13. B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proc. 22nd ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.
14. B. Yadegari and S. Debray. Control dependencies in interpretive systems. In *Proc. 17th International Conference on Runtime Verification (RV 2017)*, Sept. 2017.