View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by IST Austria: PubRep (Institute of Science and Technology)

Differential Monitoring - Technical Report*

Fabian Muehlboeck[0000−0003−1548−0177]
and Thomas A. Henzinger[0000−0002−2985−7724]

IST Austria, 3400 Klosterneuburg, Austria {fabian.muehlboeck, tah}@ist.ac.at

Abstract. We argue that the time is ripe to investigate differential monitoring, in which the specification of a program's behavior is implicitly given by a second program implementing the same informal specification. Similar ideas have been proposed before, and are currently implemented in restricted form for testing and specialized run-time analyses, aspects of which we combine. We discuss the challenges of implementing differential monitoring as a general-purpose, black-box run-time monitoring framework, and present promising results of a preliminary implementation, showing low monitoring overheads for diverse programs.

Keywords: Run-time verification · Software engineering · Implicit specification

1 Introduction

Run-time verification has a major advantage on static verification: it is easier to decide whether one particular run of a program conforms to a specification than reasoning about all possible runs. While some run-time verification frameworks are based on similar techniques as static approaches [3, 16], run-time verification also allows us to focus on end-to-end properties of the program, by checking the correctness of the response of a program to some input while ignoring its inner workings. Such a black-box approach is especially appealing if the program source is unavailable, or untrusted. However, for long-running and stateful programs, which transform input streams into output streams, the complete specification of the program's end-to-end behavior may itself become complicated and can amount to essentially writing the program a second time, often in a more cumbersome language that is also slower to execute.

Differential monitoring is the idea of running different versions of the same program in parallel, duplicating any external inputs and merging any outputs after checking them for equivalence. In this way, each program acts as an end-to-end specification for the other. On a system with enough idle hardware resources, this represents a natural method for improving software quality and security through redundancy and over-engineering.

The underlying idea is not new — it dates back to the 1960s [8, 9] under the name of n-version programming. The closest current incarnation of this concept is called n-version execution [4, 26, 40, 11], where the system calls of

* Supported in part by Austrian Science Fund (FWF) grant Z211-N23 (Wittgenstein Award).

the executed programs need to match (almost) exactly. Hence, the differences between the programs must be minimal and typically are variations on possible memory layouts to catch (often security-related) memory over/underflow errors, or updated versions of the original program that should largely behave the same.

In contrast, differential testing [32, 15, 24] exploits *true* diversity of implementations to find bugs with respect to a common specification, e.g. for compilers [41] or SQL-databases [39]. However, differential testing mainly applies to finite (and not necessarily parallel) runs in a controlled environment.

We argue that the time is ripe to explore the idea of running and monitoring truly diverse versions of the same program in parallel: the two versions could be written by independent development teams, in different languages, implementing different algorithms, against a common input-output interface. In this way, run-time monitoring can increase the trust in the correctness of programs and program updates without looking into the internals of the different implementations: if both independent implementations yield the same results, our confidence increases that the results are correct. On the other hand, if the monitor discovers a run-time discrepancy, a warning can be issued. This set-up presupposes, of course, that the duplication and monitoring can be done with little overhead. This can be the case, for example, if there are available hardware resources such as unused cores (/nodes) on a processor (cluster), or if the gain in confidence is worth the extra hardware, such as in safety-critical applications (where redundancy has long been a dominant paradigm) or in finance.

In comparison to traditional run-time monitoring, no formal specification of the program is needed in order to monitor it: on any given input, the expected outputs of one implementation are generated by the second implementation, and vice versa. The main overhead of differential monitoring comes from code comparing these outputs. Given sufficient operating system support, much of this work can be done when the program is paused anyway, such as during file operations. Preliminary benchmarks on a modified Linux kernel to monitor such file operations show very low overheads from monitoring and merging the outputs, even if the two implementations are written in different languages.

The main challenge for differential monitoring is that one implementation may overconstrain the expected behavior of the other implementation, mainly due to acceptable non-determinism and differences in timing, but also due to acceptable differences in system calls. Therefore, a differential monitor may also need a specification of how the output streams of two equivalent implementations may differ for the same input stream, and how the monitor can check and enforce such an equivalence run-time, for example, by delaying an implementation to let the other implementation catch up. The complexity of the monitor is proportional to the amount of acceptable differences in a program's behaviors. The specification and monitoring/enforcement of equivalence relations on input/output traces is an important area for future work; for now, we describe a relatively simple version of our vision, and how to extend it in the future. The main goal of the present paper is to demonstrate that for a practical definition of behavioral equivalence —essentially, trace equality where individual

inputs/outputs can be delayed by the monitor— the equivalence monitoring can be performed on real systems with a very modest overhead.

In summary, differential monitoring is a low-cost, black-box, on-line, and end-to-end run-time verification method requiring redundant hardware but no or little formal specification. These properties make it ideal for scenarios where one seeks to gain confidence in or improve the quality of continuously running software by using otherwise unused or easily obtainable hardware resources.

The rest of the paper is organized as follows:

- In Section 2, we review the current state of the art in related fields, and discuss how differential monitoring builds upon and extends it.
- In Section 3, we discuss the logical setup of differential monitoring and its main challenges.
- Finally, in Section 4, we present the results of preliminary experiments on differentially monitored programs written in different languages.

2 Background and Related Work

***n*-Version Programming/Execution** Running several versions of the same program in parallel to improve software reliability dates back to the 1970s [8, 9, 14, 20, 30, 18]. Chen and Avizienis [8, 9] rely on the cooperation of the various versions of the program: part of the process of n-version programming is to specify interesting kinds of data, and points of synchronization where each version explicitly presents that data to a coordinator process, who then judges which versions have produced correct data (via some voting mechanism, for example) and which need to either be aborted or otherwise corrected. Once this coordination step is complete, the (correct) versions can resume their work.

Modern works on n-version execution [4, 12, 6, 26, 40, 11, 33] follow this model in the sense that system calls and their arguments are the synchronization points and interesting data, respectively. Thus, correct versions of the same program generate the same sequence of system calls with the same arguments, including not only outputs, but also any form of reads: only one process actually reads; the results are shared with the others. This naturally side-steps the main challenges of differential monitoring we discuss in Section 3. However, it requires the different versions to be very closely related, to a point where it is implausible for the versions to be developed independently, or in different programming languages.

Though limited in this way, n-version execution can be used to guard against memory-related safety and security problems by varying memory layouts of data structures, including the stack, between versions [4, 12, 26, 40]. Another scenario in which two programs are related sufficiently closely are program updates: n-version execution can be used to have an oracle for regression testing, and also to update running programs in the middle of processing requests [6, 33].

While the core idea of differential monitoring is the same as that of n-version programming, the technical and theoretical environment is vastly different today, and our proposed blueprint and the challenges we discuss in Section 3 reflect this. Like n-version execution, we focus on the interactions of programs with the

environment rather than arbitrary program state in order to both provide a less intrusive interface and exploit modern hardware/operating system architecture, but unlike n-version execution, we seek to recover the idea of truly diverse implementations.

Differential Testing Differential Testing [32, 24, 15] is a well-known technique to test programs for which multiple versions exist. A large number of automatically generated test inputs are fed to $n > 1$ supposedly equivalent programs. Any differences detected in their output indicate possible bugs that need to be investigated. This technique has been fruitfully applied to finding bugs in Javascript debuggers [31], C compilers [41], and SQL databases [39, 37, 36].

DiffStream [27] is a framework supporting differential testing of stream outputs, which is closely related to our implementation of differential monitoring. They key technical difference is that differential monitoring does not only track and compare a set of (potentially unbounded) streams, but also needs to help programs stay in sync (see Section 3). For system calls and other events, the atomicity of stream elements can itself be in need of specification, as one system call may be equivalent to a sequence of several other ones. Finally, DiffStream ignores the question of what to output for equivalent but unequal streams.

Knight and Leveson [28, 29] took issue with the claim that independently produced programs contain independent errors. Their experiments showed that faults exhibited by programs written independently by different programmers to the same specification are not completely independent. As a result, n-version execution has dropped high-level correctness claims, instead focusing on targeted variations (which are thus not independent of each other) of a program, and thus finding errors related to those variations. On the other hand, differential testing shows that a large variety of bugs can be found (and eliminated) by simply comparing the outputs of different but supposedly equivalent programs.

Run-Time Verification/Monitoring Run-time verification (RV) is the general area of monitoring and possibly enforcing that a given program satisfies some properties, typically related in some way to the program's overall correctness [25, 2]. In RV, a program generates a trace of interesting events, and a specification of the program's behavior allows us to build a monitor that checks such a trace of interesting events for whether it (possibly or definitely) conforms to the specification. A considerable body of work exists on various specification languages based on linear temporal logic and similar logics [34, 7, 13, 1, 5, 23], and there are specification languages specifically for properties of streams [38], but these languages are interpreted over individual traces, rather than tuples of traces produced by supposedly equivalent programs. Especially in the area of security, languages like Hyper-LTL [10] are used on sets of traces (or, oftentimes, pairs of traces). However, similar to n-version execution, hyperlogics are usually interpreted over sets (or pairs) of traces that are generated by multiple executions of a single (often reactive or otherwise nondeterministic) program.

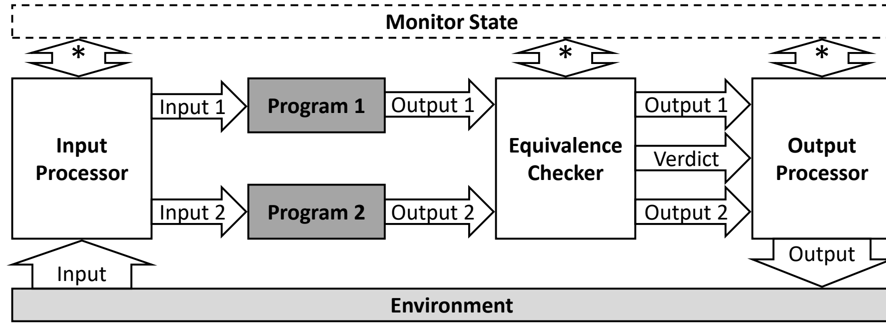

Fig. 1. The Logical Parts of Differential Monitoring

In process algebra, there has been much work on trace equivalence and other equivalence relations for comparing individual behaviors [21, 22, 19], but comparatively little attention to the online monitoring and enforcement of these equivalences. The definition of distinguishability in DiffStream [27] echoes some of our concerns. Interestingly, enforcement [17] has been a concern of n-version programming [8, 9], which implements it by voting among the different versions.

3 Challenges

The goal of differential monitoring is to provide a low-cost, black-box, end-to-end run-time verification method, where the low cost relates to both the effort required in terms of specification and any run-time overhead caused by monitoring. There are two key challenges here: first, that an executable program may over-specify the desired behavior of the other program; second, how much "enforcement" a differential monitor may perform, say, by delaying or reordering inputs to the monitored programs, and by "merging" outputs of the monitored program (e.g., interpolation of different outputs, or voting for $n \geq 3$ programs). These challenges are two sides of the same coin, with the monitor trying to ensure that the monitored programs run independently as if they were running alone, yet are kept sufficiently in sync to produce equivalent results. A more advanced differential monitor may adjust scheduling decisions by the operating system, or try to synchronize the effects of some otherwise nondeterministic system calls.

Figure 1 shows the logical parts of a differential-monitoring setup: typically, a program would receive its inputs from and send its outputs to some environment, including the rest of the system it is running on as well as any network or other devices it has access to. The differential monitor inserts itself into this relationship on both ends, and additionally does this for two programs at once. The *input processor* is the part of the monitor that handles any input the monitored programs receive. By default, it simply duplicates any inputs it receives and forwards them to both programs. The *equivalence checker* receives the outputs of both programs and checks them for equivalence, which by default simply

means equality. Finally, the *output processor* produces the output that is finally sent to the environment. If equivalence is defined as equality, its default behavior is to send the output of one of the programs to the environment as long as the equivalence checker's verdict is positive, and some error message aborting the programs when the verdict becomes negative. However, for more complex notions of input processing, output equivalence, output merging, and error handling can be specified. All three components may communicate with each other via some notion of monitor state.

Each program may expect to see the effects of its output in the environment. Thus the monitor may have to slow down the inputs and/or outputs of the faster program to let the slower program catch up. In general, a differential monitor should prevent either program from being confronted with an environment state it does not expect, keeping up the illusion that it is running alone. Beyond using additional memory to buffer input or output elements as in DiffStream [27], differential monitors may need additional power and resources to ensure that the different programs' interactions with their environment do not get too out-of-sync. Kallas et al. [27] already recognized that some parts of the output (for example, timings and random numbers) may have to be relaxed or ignored for equivalence checking, though deciding on the "merged" output may be harder.

In differential monitoring, the program specification is replaced by the monitor specification, which ought to be simpler. All three components of a differential monitor have to be accompanied by a specification defining exactly when and how to defer, transform, deem equivalent, and merge any inputs or outputs. The precise form and power of such specifications, and of the monitors implementing them (e.g., their memory needs), will be an interesting area of research, as will be the automatic synthesis of differential monitors from formal specifications.

4 Experimental Results

In this section, we report on experiments evaluating the feasibility of differential monitoring and the overheads it causes in practice based on a simple framework.

Experimental Setup To test the basic overheads of running two programs side-by-side, duplicating inputs and comparing outputs, we modified a current version of the Linux kernel to support an additional system call that activates monitoring for a pair of processes and any of their children. In particular, we watch the basic file operations of these processes. When a file is opened, we determine whether its operations need to be monitored. For example, regular files opened in a read-only fashion can be ignored and any further interactions of the programs with them incur no overhead. On the other hand, non-seekable files (pipes, the user's terminal, etc.) need the monitor to provide the same data to both processes. Finally, files opened for writing are monitored to ensure that both programs write the same data to them.

In terms of our model from Section 3, the input processor duplicates all inputs (mostly reads from the standard input) by buffering the result of the

faster reader to also provide it to the second process. The equivalence checker checks the bytes written out for equality; neither monitored writes nor reads need to match exactly in terms of how many bytes are read/written in a single system call — the monitor will proceed as far as possible. The input processor holds up the faster program by not returning from the write system call until all bytes a process wanted to write have been matched and sent to the environment. If the output of the two programs does not match, the monitor aborts both programs—this is of course the most extreme measure that could be taken, but suffices for our goal of measuring overheads in the case where both programs are correct. We did test that our monitor indeed stops programs that do not produce equivalent output, and does so before actually printing that output.

Our benchmarks often write a large number of small output lines in rapid succession. For this case, an optional, experimental optimization allows the faster program to continue execution until it gets too far ahead, buffering writes in the meantime. This is valid when programs do not need to see the effects of their outputs on the environment immediately and expect the writes to always succeed, as is the case for our test programs, or, for example, web servers. In general, a monitor specification would specify in what cases this optimization can be applied. By default, our benchmarks run without this optimization.

This simple prototype of course does not capture the full complexity of the specifications eventually needed to run more complex programs side-by-side, but it lets us explore the overheads of what we believe are the most common cases in differential monitoring. To this end, we wrote several small benchmark programs in C, Java, and Python, sometimes using different algorithms between the programs, and ran them in various pairings and alone to compare the slowdowns caused by our monitor. Here, we present these benchmarks and the results of our measurements. As the relevant metric we compare, for each pairing of programs, the wall-clock time of running the pairing against the wall-clock time of the slower program (which is a natural lower bound for the pairing). All benchmarks were run on a minimal Gentoo installation using our modified kernel on an Intel(R) Core(TM) i5-4690K processor with 16GiB RAM and a mid-level SSD.

Main Benchmark: Primes Consider verifying programs that should answer queries about whether a given number n is prime or what the nth prime number is. Any black-box attempt to verify the outputs of such a program invariably needs to do a similar amount of work as the original program.

For this benchmark, we picked two algorithms to determine primality: the Sieve of Eratosthenes and the Baillie-PSW [35] primality test. We implemented the former in C and the latter in Java. The programs have two modes. In interactive mode, they accept a stream of queries for either the n-th prime number or whether a given number n is prime, and produce a corresponding answer. In non-interactive mode, they simply enumerate all the primes up to a certain index, in our case up to the 10 000th prime. For that mode, we also included another Java ("Java-E") and a Python implementation of the Sieve of Eratosthenes.

		Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Interactive	Sml Qs	C	C	3.11s	3.11s	3.13s	0.56%	3.13s	0.69%
		C	Java	3.11s	3.82s	3.87s	1.31%	3.80s	-0.33%
		Java	Java	3.82s	3.82s	3.84s	0.69%	3.92s	2.74%
	Lrg Qs	C	C	1.83s	1.83s	1.87s	1.94%	1.84s	0.35%
		C	Java	1.83s	2.32s	2.36s	1.79%	2.33s	0.63%
		Java	Java	2.32s	2.32s	2.48s	7.17%	2.33s	0.56%
		Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Non-Interactive		C	C	0.40s	0.40s	0.43s	8.06%	0.42s	4.53%
		C	Java	0.40s	0.66s	0.69s	4.80%	0.67s	2.74%
		C	Java-E	0.40s	0.28s	0.56s	39.80%	0.47s	17.63%
		C	Python	0.40s	4.22s	4.25s	0.75%	4.32s	2.35%
		Java	Java	0.66s	0.66s	0.77s	17.75%	0.69s	5.10%
		Java	Java-E	0.66s	0.28s	0.77s	16.68%	0.68s	3.66%
		Java	Python	0.66s	4.22s	4.42s	4.72%	4.33s	2.70%
		Java-E	Java-E	0.28s	0.28s	0.38s	33.33%	0.31s	7.94%
		Java-E	Python	0.28s	4.22s	4.32s	2.31%	4.35s	3.20%
		Python	Python	4.22s	4.22s	4.49s	6.48%	4.33s	2.57%

Fig. 2. Benchmark Results for Primes

Figure 2 shows the average running times (in seconds) of 20 runs for each language on its own (WT-1/WT-2) and in a paired monitored setting (WT), and the corresponding overhead. For the interactive mode, in one run we generated 300 queries with $n < 4\,000$, and in the other, we generated 10 000 such queries with $n < 500$, trading off internal computation time vs. interaction with the system. As we see, the overhead is negligible for the fewer requests where both programs spend more time simply computing the response, while it is still relatively low for programs where our monitoring code is invoked more often. The overheads for the non-interactive version are significantly higher — they are writing significantly more lines than the interactive version in less time; the write-buffering optimization mentioned above (results shown in the "-O" columns) drastically improves our results. Overall, Java seems to suffer the most from being run side-by side with another program; however, this is also true for just running the Java program twice at the same time without monitoring. We believe the cause to be the optimization behavior of the JVM, which spawns around 12 threads for these single threaded applications. In so far as the negative overhead of the C/Java pairing is not a measuring artifact, it is likely for similar reasons as the negative overheads for the Echo benchmark discussed below.

Sort For this benchmark, we implemented Insertion-Sort in C, Merge-Sort in Java, and Quicksort in Python. In interactive mode, they accept three sorts of commands: one to add a number to a currently maintained list, one to print the list in sorted order, and one to clear the maintained list. The input we generate for interactive mode sorts the list on roughly every 10th command, and

	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Interactive	C	C	1.47s	1.47s	1.56s	6.54%	1.49s	1.77%
	C	Java	1.47s	1.52s	2.62s	72.63%	1.57s	3.19%
	C	Python	1.47s	1.51s	2.05s	35.78%	1.55s	2.85%
	Java	Java	1.52s	1.52s	2.77s	81.91%	1.60s	5.13%
	Java	Python	1.52s	1.51s	3.81s	150.46%	1.64s	8.19%
	Python	Python	1.51s	1.51s	2.28s	50.81%	1.55s	2.55%
	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Non-Interactive	C	C	5.55s	5.55s	5.84s	5.23%	5.74s	3.52%
	C	Java	5.55s	0.52s	6.22s	12.07%	5.84s	5.24%
	C	Python	5.55s	0.54s	5.88s	5.96%	5.67s	2.30%
	Java	Java	0.52s	0.52s	1.11s	114.45%	0.75s	44.70%
	Java	Python	0.52s	0.54s	1.26s	132.56%	0.70s	29.61%
	Python	Python	0.54s	0.54s	0.84s	54.80%	0.64s	17.34%

Fig. 3. Benchmark Results for Sort

clears it after roughly every 6 of those, which makes for relatively short lists, but still a high output-to-input ratio. In non-interactive mode, the programs read in a list of 100 000 numbers from a file, sort it, and print the result. As an interesting variation, we give each program a different permutation of the same list — as this does not affect the results of sorting those lists, the programs still produce the same out. In this way, this benchmark simulates different ways in which programs may keep private data. The net result of this setup is that the programs do a batch of reading first (except for the C program, whose insertion sort is running while reading the list), followed by a large burst of writes (this is also true for the C program). As we can see in Figure 3, the high rates of writes for both modes can cause quite extreme overheads, which again can be brought down significantly with our write-buffering optimization.

Further Benchmarks We only briefly describe our other two benchmark programs here. More details on them can be found in Appendix A.

Echo Echo was intended to be a worst-case benchmark for our framework: the programs written in C, Java, and Python simply read text from the standard input line by line and write it back to the standard output, thus maximally using both our input-splitting and output-comparing facilities. Overheads for Echo reached 67.50% for two Java programs (10.26% for two C programs), which fell to 10.05% (−1.24% for C-C) using our writer-buffering optimization. The same optimization made all other pairings produce negative overhead, as it turned out that the programs were now parallelizing the reading IO operations.

Mod-Squares Mod-squares was designed to simulate single-threaded computational activity that is not parallelizable and works in constant memory, thus eliminating any sort of resource constraints other than the extra computation

and coordination caused by the monitor. At its core, it simply squares a (hard-coded constant) number some number of times, always modulo some other number. The highest overhead, again in the Java-Java pairing, was 34.42% (the C-C pairing had 4.20%), which the write-buffering optimization reduced to 7.45% (or 0.22% for C-C).

Discussion Our benchmarks tested a general framework to monitor the IO operations of programs written in different programming languages. Previous work would have been unable to do so, as works in multi-version execution [40, 26] depend on the programs making the exact same system calls, which would already be violated by the Java and Python virtual machines' startup activities, while DiffStream [27] works on a different level of abstraction and does not consider having to delay outputs. The overheads we saw for our main benchmark are relatively low, and naturally somewhat related to the ratio of work a program does to how often it interacts with its environment and thus the monitor. The other benchmarks we ran consider various worst-case scenarios with extremely heavy interaction with the monitor; overheads in these benchmarks go up to 150% in extreme cases. These extreme overheads go down to 45% with our write-buffering optimization, showing that there is much room for optimizations both in our basic implementation and in exploiting monitoring specifications to allow for more efficient processing of those particular cases.

5 Conclusion

Differential Monitoring has the potential to be a comparatively light-weight runtime-verification method that is able to check programs' end-to-end behavior in an efficient way, simply through redundancy and overengineering. Similar efforts have both a long history and recent activity, and the ubiquitousness of multi-core hardware suggests that the approach can be applied in many scenarios without too much of a performance penalty. For complicated programs, the lack of formal specification is not absolute, but turned upside down: a differential monitor may need a specification of two programs *potential differences*, which should be comparatively small in any case. The precise formalisms for such a specification will draw heavily on existing work on runtime monitoring but pose some interesting challenges on their own, including for their eventual implementation. However, we believe that these challenges can be overcome, thereby significantly adding to the toolbox that runtime verification offers its users to increase the quality of software.

Acknowledgements

The authors would like to thank Borzoo Bonakdarpour, Derek Dreyer, Adrian Francalanza, Owolabi Legunsen, Matthew Milano, Manuel Rigger, Cesar Sanchez, and the members of the IST Verification Seminar for their helpful comments and insights on various stages of this work, as well as the reviewers of RV'21 for their helpful suggestions on the actual paper.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: FM 2012. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9
2. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5
3. Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20**(4) (Sep 2011). https://doi.org/10.1145/2000799.2000800
4. Berger, E.D., Zorn, B.G.: DieHard: Probabilistic Memory Safety for Unsafe Languages. In: PLDI 2006. p. 158–168. Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1133981.1134000
5. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Time-triggered runtime verification. Formal Methods in System Design **43**(1), 29–60 (2013). https://doi.org/10.1007/s10703-012-0182-0
6. Cadar, C., Hosek, P.: Multi-version software updates. In: HotSWUp 2012. pp. 36–40 (2012). https://doi.org/10.1109/HotSWUp.2012.6226615
7. Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: TACAS 2009. Lecture Notes in Computer Science, vol. 5505, pp. 246–261. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_23
8. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: FTCS 1978. vol. 1, pp. 3–9 (1978)
9. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: FTCS 1995, 'Highlights from Twenty-Five Years'. p. 113ff (1995). https://doi.org/10.1109/FTCSH.1995.532621
10. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal Logics for Hyperproperties. In: POST 2014. pp. 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
11. Coppens, B., Sutter, B.D., Volckaert, S.: Multi-variant execution environments. In: Larsen, P., Sadeghi, A. (eds.) The Continuing Arms Race: Code-Reuse Attacks and Defenses, pp. 211–258. ACM / Morgan & Claypool (2018). https://doi.org/10.1145/3129743.3129752
12. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-Variant Systems: A Secretless Framework for Security through Diversity. In: USENIX-SS 2006. USENIX Association, USA (2006), https://www.usenix.org/conference/15th-usenix-security-symposium/n-variant-systems-secretless-framework-security-through
13. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Transactions on Computational Logic **10**(3), 16:1–16:30 (2009). https://doi.org/10.1145/1507244.1507246
14. Elmendorf, W.: Fault-Tolerant Programming. In: FTCS 1972. pp. 79–83 (1972)
15. Evans, R.B., Savoia, A.: Differential Testing: A New Approach to Change Detection. In: ESEC-FSE companion 2007. pp. 549–552. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1295014.1295038
16. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer **14**(3), 349–382 (2012). https://doi.org/10.1007/s10009-011-0196-8

17. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime Failure Prevention and Reaction. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 103–134. Springer (2018). https://doi.org/10.1007/978-3-319-75632-5_4
18. Fischler, M.A. et al.: Distinct Software: An Approach to Reliable Computing. In: 2nd USA-Japan Computer Conference. pp. 1–7 (1975)
19. Fokkink, W.J.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series, Springer (2000). https://doi.org/10.1007/978-3-662-04293-9
20. Girard, E., Rault, J.: A Programming Technique for Software Reliability. In: IEEE Symposium on Computer Software Reliability. pp. 44–50 (1973)
21. van Glabbeek, R.J.: The Linear Time-Branching Time Spectrum (Extended Abstract). In: CONCUR 1990. Lecture Notes in Computer Science, vol. 458, pp. 278–297. Springer (1990). https://doi.org/10.1007/BFb0039066
22. van Glabbeek, R.J.: The Linear Time - Branching Time Spectrum II. In: CONCUR 1993. Lecture Notes in Computer Science, vol. 715, pp. 66–81. Springer (1993). https://doi.org/10.1007/3-540-57208-2_6
23. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime Verification Based on Register Automata. In: TACAS 2013. Lecture Notes in Computer Science, vol. 7795, pp. 260–276. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_19
24. Groce, A., Holzmann, G., Joshi, R.: Randomized Differential Testing as a Prelude to Formal Verification. In: ICSE 2007. p. 621–631. IEEE Computer Society, USA (2007). https://doi.org/10.1109/ICSE.2007.68
25. Havelund, K., Reger, G., Rosu, G.: Runtime Verification Past Experiences and Future Projections. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 532–562. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_25
26. Hosek, P., Cadar, C.: VARAN the Unbelievable: An Efficient N-Version Execution Framework. In: ASPLOS 2015. p. 339–353. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2694344.2694390
27. Kallas, K., Niksic, F., Stanford, C., Alur, R.: DiffStream: Differential Output Testing for Stream Processing Programs. PACMPL **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428221
28. Knight, J.C., Leveson, N.G.: An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. IEEE Transactions on Software Engineering **12**(1), 96–109 (Jan 1986). https://doi.org/10.1109/TSE.1986.6312924
29. Knight, J.C., Leveson, N.G.: A Reply to the Criticisms of the Knight & Leveson Experiment. ACM SIGSOFT Software Engingeering Notes **15**(1), 24–35 (Jan 1990). https://doi.org/10.1145/382294.382710
30. Kopetz, H.: Software Redundancy in Real Time Systems. In: IFIP Congress 1974. pp. 182–186. North-Holland (1974)
31. Lehmann, D., Pradel, M.: Feedback-Directed Differential Testing of Interactive Debuggers. In: ESEC/FSE 2018. pp. 610–620. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236037
32. McKeeman, W.M.: Differential Testing for Software. Digital Technical Journal **10**(1), 100–107 (1998), http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

33. Pina, L., Andronidis, A., Hicks, M., Cadar, C.: MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In: ASPLOS 2019. pp. 573–585. ACM (2019). https://doi.org/10.1145/3297858.3304063
34. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: FM 2006. Lecture Notes in Computer Science, vol. 4085, pp. 573–586. Springer (2006). https://doi.org/10.1007/11813040_38
35. Pomerance, C., Selfridge, J.L., Wagstaff, S.S.: The pseudoprimes to $25 \cdot 10^9$. Mathematics of Computation **35**(151), 1003–1026 (1980)
36. Rigger, M., Su, Z.: Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In: ESEC/FSE 2020. pp. 1140–1152. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409710
37. Rigger, M., Su, Z.: Finding Bugs in Database Systems via Query Partitioning. PACMPL **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428279
38. Sánchez, C.: Online and Offline Stream Runtime Verification of Synchronous Systems. In: RV 2018. Lecture Notes in Computer Science, vol. 11237, pp. 138–163. Springer (2018). https://doi.org/10.1007/978-3-030-03769-7_9
39. Slutz, D.R.: Massive Stochastic Testing of SQL. In: VLDB 1998. pp. 618–622. Morgan Kaufmann (1998), http://www.vldb.org/conf/1998/p618.pdf
40. Volckaert, S., Sutter, B.D., Baets, T.D., Bosschere, K.D.: GHUMVEE: Efficient, Effective, and Flexible Replication. In: FPS 2012. Lecture Notes in Computer Science, vol. 7743, pp. 261–277. Springer (2012). https://doi.org/10.1007/978-3-642-37119-6_17
41. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and Understanding Bugs in C Compilers. In: PLDI 2011. pp. 283–294. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993532

	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Interactive	C	C	0.64s	0.64s	0.71s	10.26%	0.64s	-1.24%
	C	Java	0.64s	0.64s	0.94s	46.19%	0.64s	-0.31%
	C	Python	0.64s	0.64s	0.86s	33.51%	0.62s	-3.34%
	Java	Java	0.64s	0.64s	1.07s	67.50%	0.70s	10.05%
	Java	Python	0.64s	0.64s	1.05s	64.15%	0.62s	-3.90%
	Python	Python	0.64s	0.64s	0.96s	49.34%	0.61s	-5.69%

Fig. 4. Benchmark Results for Echo

	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
I-active	C	C	0.68s	0.68s	0.71s	4.20%	0.70s	2.87%
	C	Java	0.68s	0.76s	0.93s	21.07%	0.81s	6.28%
	Java	Java	0.76s	0.76s	1.03s	34.42%	0.82s	7.72%
N-I-active	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
	C	C	2.68s	2.68s	2.73s	1.60%	2.69s	0.22%
	C	Java	2.68s	2.63s	3.04s	13.10%	2.78s	3.39%
	Java	Java	2.63s	2.63s	3.01s	14.57%	2.82s	7.45%

Fig. 5. Benchmark Results for Modsquares

A Detailed Benchmark Results

A.1 Additional Benchmarks

Echo Echo was intended to be a worst-case benchmark for our framework: the programs written in C, Java, and Python simply read from the standard input line by line and write it back to the standard output, thus maximally using both our input-splitting and output-comparing facilities.

Figure 4 shows the results for running echo on 100 000 lines of inputs drawn from a selection of a few random lines of text. By default, the overheads are indeed somewhat heavy. However, our write-buffering optimization eliminates almost all of them (modulo the Java anomaly we already mentioned). In fact, the monitored instances of echo run slightly faster than the single unmonitored programs. The reason for that lies in the design of the benchmark: only one process actually reads from the input, while the other gets a buffered result. Using the write-buffering optimization, a process can run ahead and already read the next input for the second process. As they will likely switch places some times, this amounts to them parallelizing the (not cost-free) read operation, effectively reading the input faster than a single-threaded program would.

Mod-Squares Mod-squares was designed to simulate single-threaded computational activity that is not parallelizable and works in constant memory, thus eliminating any sort of resource constraints other than the extra computation and coordination caused by the monitor. At its core, it simply squares a (hard-

	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Interactive	C	C	1.47s	1.47s	1.56s	6.54%	1.49s	1.77%
	C	Java	1.47s	1.52s	2.62s	72.63%	1.57s	3.19%
	C	Python	1.47s	1.51s	2.05s	35.78%	1.55s	2.85%
	Java	Java	1.52s	1.52s	2.77s	81.91%	1.60s	5.13%
	Java	Python	1.52s	1.51s	3.81s	150.46%	1.64s	8.19%
	Python	Python	1.51s	1.51s	2.28s	50.81%	1.55s	2.55%
	Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Non-Interactive	C	C	5.55s	5.55s	5.84s	5.23%	5.74s	3.52%
	C	Java	5.55s	0.52s	6.22s	12.07%	5.84s	5.24%
	C	Python	5.55s	0.54s	5.88s	5.96%	5.67s	2.30%
	Java	Java	0.52s	0.52s	1.11s	114.45%	0.75s	44.70%
	Java	Python	0.52s	0.54s	1.26s	132.56%	0.70s	29.61%
	Python	Python	0.54s	0.54s	0.84s	54.80%	0.64s	17.34%

Fig. 6. Benchmark Results for Sort

coded constant) number some number of times, always modulo some other number. In interactive mode, it takes queries consisting of a pair of numbers, the first being the number of squaring operations and the second a modulus to use — it then returns the result of the repeated squaring operation. We generate 10 000 queries, each for some random number of up to 10 000 squarings. In non-interactive mode, it enumerates the first 20 000 squarings of our constant, similar to the non-interactive mode of `primes`.

Figure 5 shows the results for both modes. The short running times of the interactive mode for 10 000 queries means that each query was handled within a rather short time, hence the program more heavily interacted with the monitor. However, as we see, the write-buffering optimization reduces those overheads significantly. The overheads are less severe for the non-interactive mode, where programs even without monitoring take about four times as long to emit twice the output of the interactive mode. Even so, the write-buffering optimization reduces the overheads to quite low levels.

Sort For this benchmark, we implemented Insertion-Sort in C, Merge-Sort in Java, and Quicksort in Python. In interactive mode, they accept three sorts of commands: one to add a number to a currently maintained list, one to print the list in sorted order, and one to clear the maintained list. The input we generate for interactive mode sorts the list on roughly every 10th command, and clears it after roughly every 6 of those, which makes for relatively short lists, but still a high output-to-input ratio. In non-interactive mode, the programs read in a list of 100 000 numbers from a file, sort it, and print the result. As an interesting variation, we give each program a different permutation of the same list — as this does not affect the results of sorting those lists, the programs still produce the same out. In this way, this benchmark simulates different ways in which programs may keep private data. The net result of this setup is that the

programs do a batch of reading first (except for the C program, whose insertion sort is running while reading the list), followed by a large burst of writes (this is also true for the C program). As we can see in Figure 6, the high rates of writes for both modes can cause quite extreme overheads, which again can be brought down significantly with our write-buffering optimization.

		Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Interactive	Sml Qs	C	C	3.11s	3.11s	3.13s	0.56%	3.13s	0.69%
Interactive	Sml Qs	C	Java	3.11s	3.82s	3.87s	1.31%	3.80s	-0.33%
Interactive	Sml Qs	Java	Java	3.82s	3.82s	3.84s	0.69%	3.92s	2.74%
Interactive	Lrg Qs	C	C	1.83s	1.83s	1.87s	1.94%	1.84s	0.35%
Interactive	Lrg Qs	C	Java	1.83s	2.32s	2.36s	1.79%	2.33s	0.63%
Interactive	Lrg Qs	Java	Java	2.32s	2.32s	2.48s	7.17%	2.33s	0.56%
		Program 1	Program 2	WT-1	WT-2	WT	Overhead	WT-O	Overhead-O
Non-Interactive		C	C	0.40s	0.40s	0.43s	8.06%	0.42s	4.53%
Non-Interactive		C	Java	0.40s	0.66s	0.69s	4.80%	0.67s	2.74%
Non-Interactive		C	Java-E	0.40s	0.28s	0.56s	39.80%	0.47s	17.63%
Non-Interactive		C	Python	0.40s	4.22s	4.25s	0.75%	4.32s	2.35%
Non-Interactive		Java	Java	0.66s	0.66s	0.77s	17.75%	0.69s	5.10%
Non-Interactive		Java	Java-E	0.66s	0.28s	0.77s	16.68%	0.68s	3.66%
Non-Interactive		Java	Python	0.66s	4.22s	4.42s	4.72%	4.33s	2.70%
Non-Interactive		Java-E	Java-E	0.28s	0.28s	0.38s	33.33%	0.31s	7.94%
Non-Interactive		Java-E	Python	0.28s	4.22s	4.32s	2.31%	4.35s	3.20%
Non-Interactive		Python	Python	4.22s	4.22s	4.49s	6.48%	4.33s	2.57%

Fig. 7. Full Benchmark Results for Primes

Primes For this benchmark, we picked two algorithms to determine primality: the Sieve of Eratosthenes and the Baillie-PSW [35] primality test. We implemented the former in C and the latter in Java. The programs have two modes. In interactive mode, they accept a stream of queries for either the n-th prime number or whether a given number n is prime, and produce a corresponding answer. In non-interactive mode, they simply enumerate all the primes up to a certain index, in our case up to the 10 000th prime. For that mode, we also included another Java ("Java-E") and a Python implementation of the Sieve of Eratosthenes.

Figure 7 shows the average running times (in seconds) of 20 runs for each language on its own (WT-1/WT-2) and in a paired monitored setting (WT), and the corresponding overhead. For the interactive mode, in one run we generated 300 queries with $n < 4\,000$, and in the other, we generated 10 000 such queries with $n < 500$, trading off internal computation time vs. interaction with the system. As we see, the overhead is negligible for the fewer requests where both programs spend more time simply computing the response, while it is still relatively low for programs where our monitoring code is invoked more often.

The overheads for the non-interactive version are significantly higher — they are writing significantly more lines than the interactive version in less time, so we also ran it under the write-buffering optimization mentioned above (results shown in the “-O” columns), which drastically improves our results. Overall, Java seems to suffer the most from being run side-by side with another program; however this is also true for just running the Java program twice at the same time without monitoring. We believe the cause to be the optimization behavior of the JVM, which spawns around 12 threads for these single threaded applications. In so far as the negative overhead of the C/Java pairing is not a measuring artifact, it is likely for similar reasons as the negative overheads for the `echo` benchmark.