On Benchmarking for Concurrent Runtime Monitoring*

Luca Aceto[2,3], Duncan Paul Attard[1,2], Adrian Francalanza[1], and Anna Ingólfsdóttir[2]

[1] University of Malta, Malta {duncan.attard.01,adrian.francalanza}@um.edu.mt
[2] Reykjavík University, Iceland {luca,duncanpa17,annai}@ru.is
[3] Gran Sasso Science Institute, L'Aquila, Italy {luca.aceto}@gssi.it

Abstract. We present a synthetic benchmarking framework targeting the systematic evaluation of Runtime Verification (RV) tools for concurrent systems. Our framework can emulate various load profiles via configuration. It also provides a multi-faceted measurement that allows for a more comprehensive assessment of the runtime overheads induced. The framework can generate significant loads to reveal edge case behaviour that may only emerge when a monitored system is pushed to its limit. Our framework is evaluated in two ways. First, we conduct sanity checks to assess the precision of the measurement mechanisms used, the repeatability of the results obtained, and the veracity of the behaviour emulated by the synthetic benchmark. We then showcase the utility of the features offered by our benchmarking tool via a RV case study.

Keywords: Concurrent Runtime Verification · Synthetic benchmarking · Software performance evaluation

1 Introduction

Large-scale software design has shifted from the classic monolithic architecture to one where applications are structured in terms of *independent components* that execute asynchronously to one another [14]. Runtime monitoring [21] is a lightweight *post-deployment* technique that is used to complement other conventional methods such as testing [36], to assess the *functional* (*e.g.* correctness) and *non-functional* (*e.g.* quality-of-service) aspects of concurrent software. This analysis technique relies on instrumenting the system with monitors. Inevitably, monitoring introduces *runtime overhead* that should be kept to a minimum [6]. Although worst-case complexity bounds for monitor-induced overheads can be calculated via standard methods (*e.g.* see [32,11,1,24]), *benchmarking* is, by far, the preferred method for assessing these overheads [6,23]. One reason for this

* Supported by the IRF project "TheoFoMon" (No:163406-051), project BehAPI, funded by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant (No:778233), the RU Research Fund, ENDEAVOUR Scholarship Scheme - Group B - National Funds, and the MIUR project PRIN 2017FTXR7S IT MATTERS.

is that benchmarks tend to be more *representative* of the overheads observed in practice [27,12]. Benchmarks also provide a *common* platform for measuring workloads, making it possible to *compare* different RV tool implementations, or rerun specific experiments to *reproduce* and *confirm* existing results.

The utility of a benchmarking tool typically rests on two aspects: *(i)* the *coverage* of scenarios of interest, and, *(ii)* the quality of *runtime metrics* collected by the benchmark harness. To represent the scenarios of interest, benchmarking tools generally employ *a suite of third-party off-the-shelf (OTS) programs* (*e.g.* [45,8,44]). OTS software is appealing because it is readily usable and inherently provides realistic scenarios; benchmarks typically rely on a range of OTS programs to broaden the coverage of real-world scenarios (*e.g.* DaCapo [8] uses eleven open-source applications). However, the use of OTS programs as benchmarks poses challenges. By design, OTS software does *not* provide hooks that allow harnesses to easily and accurately collect the runtime metrics of interest. When the OTS programs are treated as black-boxes, such benchmarks become harder to control, which impacts their ability to produce repeatable results. Such benchmarks are also limited when inducing specific edge cases; this aspect is critical when assessing the safety of software such as runtime monitors, considered to be part of the trusted computing base. A second alternative for benchmarking is that of custom-built *synthetic programs* (*e.g.* [29]). These tend to be less popular because of the perceived drawbacks associated with developing such programs from scratch, and the lack of 'real-world' behaviours intrinsic to benchmarks based on OTS programs. However, synthetic benchmarks offer benefits that offset these drawbacks. For one, specialised hooks can be built into the synthetic set-up to collect a broader range of runtime metrics. For two, synthetic benchmarks can also be parametrised to emulate variations on the same core benchmark behaviour; this is usually harder to achieve using OTS programs that implement specific use-cases.

Established benchmarking tools such as SPECjvm2008 [45], DaCapo [8], ScalaBench [44] and Savina [29] are often cited in RV literature; some examples include [37,16,15,41,35,10]. With the exception of [35], these RV tools assess the runtime overhead by measuring only the *execution slowdown*, *i.e.,* the difference in running time between the system fitted with and without monitors. Recently, the International RV competition (CRV) [5] has advocated for other metrics, such as *memory consumption*, to give a more qualitative view of runtime overhead. This is not surprising, since the program running time and its memory consumption are generally at odds with one another, *e.g.* caching speeds up the program at the expense of larger memory footprints. In the case of concurrency, RV set-ups for component systems benefit from other facets of runtime behaviour. For instance, measuring the *response time* captures the overhead between communicating components. In practical terms, this reflects the *perceived responsiveness* from an end-user standpoint (*e.g.* interactive apps) [39,46]; more generally, it describes the *service degradation* that must be accounted for to ensure adequate quality-of-service [12,31]. Arguably, benchmarking tools such as those mentioned above should provide even more. It is often the case that RV

set-ups for component systems need to scale in response to dynamic changes, and the capacity for a benchmark to emulate *high loads* is vital. These loads are known to assume distinct *profiles* (*e.g.* spikes or uniform rates), something that is hard to administer with the benchmarks mentioned above.

The state-of-the-art in benchmarking for the RV of component systems suffers from another issue. Existing benchmarks—built for validating other tools—are *repurposed for use* in RV, and these do not cater for concurrent scenarios where RV is realistically employed. SPECjvm2008, DaCapo, and ScalaBench lack workloads that leverage the JVM concurrency primitives [40], whereas [9] shows that the Savina microbenchmarks are essentially sequential, and that the remaining programs in the suite are sufficiently simple to be regarded as microbenchmarks themselves. The CRV suite includes benchmarks contributed by various participants: most of them target *monolithic* software with limited concurrency and the possibility of scaling up to high loads is therefore, severely curbed.

We propose a framework that targets the asynchronous message-passing paradigm. It generates synthetic system models that follow the *master-slave* architecture [46]. Master-slave designs are used pervasively in distributed computing areas such as DNS, IoT, and Big Data applications that scale massively; on a local concurrency level, this architecture underlies software such as thread pools and web servers [46,25]. Our concrete contributions are:

- The development of a *configurable benchmark* and harness that emulates various master-slave models under commonly-observed load profiles, Sec. 2.
- A validation demonstrating that our synthetic benchmarking framework can be engineered to approximate the *realistic behaviour* of web server traffic with high degrees of precision and repeatability, Sec. 3.1.
- A case study that shows how the load shapes and *parametrisability* of the benchmark can produce edge cases that can be measured via a number of metrics to assess RV tools in a comprehensive manner, Sec. 3.2.

2 Design and Implementation

Our benchmarking set-up can emulate a range of system models and subject them to various types of loads. We focus on *master-slave* system models, where one central process, called the *master*, creates and allocates tasks to *slave* processes [46]. Slaves work concurrently on tasks, relaying the result to the master when ready; the master then combines these results to yield the final output.

2.1 Approach

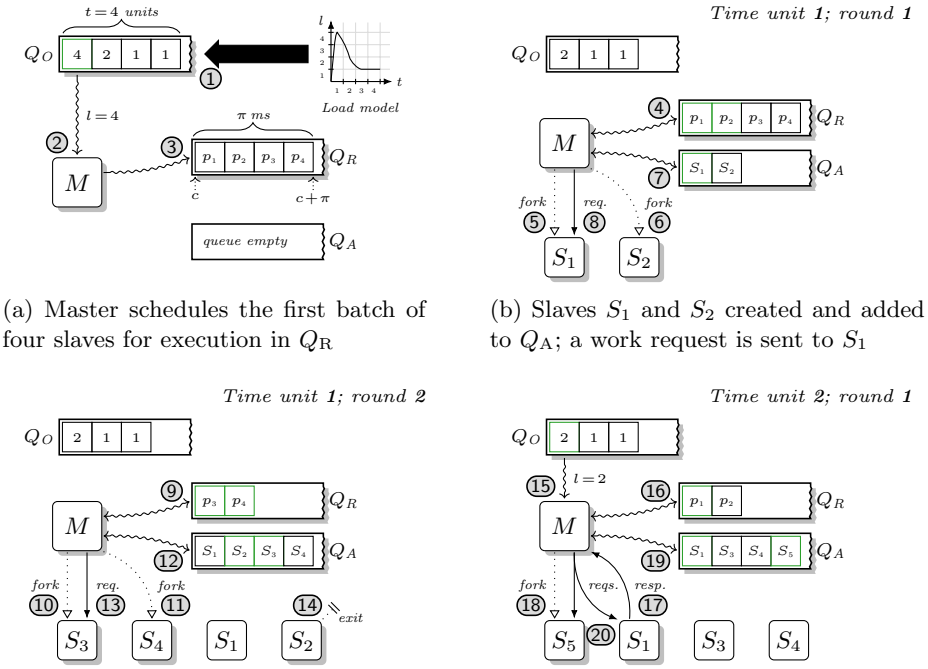
Our design focusses on concurrent applications that execute on a single node. It however adheres to three criteria that facilitate the extension of our tool to a distributed setting. Specifically, components: *(i)* share neither a common clock, *(ii)* nor memory, and *(iii)* communicate via asynchronous messages. Our current set-up assumes that communication is reliable and components do not fail-stop or exhibit Byzantine failures; this extension will be added in future releases.

Generating load. System load is induced by the master process when it creates slave processes and allocates *tasks*. The total number of slaves in one run can be set via the parameter n. Tasks are allocated to slave processes by the master, and consist of one or more *work requests* that a slave receives and echoes back. A slave terminates its execution when all of its allocated work requests have been processed *and* acknowledged by the master. The number of work requests that *can* be batched in a task is controlled via the parameter w; the *actual* batch size per slave is then drawn randomly from a normal distribution with mean, $\mu = w$ and standard deviation $\sigma = \mu \times 0.02$. This induces a degree of variability in the amount of work requests exchanged between master and slaves. The master and slaves communicate *asynchronously*: an allocated work request is delivered to a slave process' incoming work queue where it is eventually handled. Work responses issued by a slave are queued and processed similarly by the master.

Load models. We consider *three load profiles* (see fig. 4) that establish how the creation of slaves is distributed along the load timeline t. The timeline is modelled as a sequence of *discrete logical time units* representing instants at which a new set of slaves is created by the master. *Steady* loads replicate executions where a system operates under stable conditions. These are modelled on a homogeneous Poisson distribution with *rate* λ, specifying the mean number of slaves that are created at each time instant along the load timeline with duration $t = \lceil n/\lambda \rceil$. *Pulse* loads emulate settings where a system experiences gradually increasing load peaks. The Pulse load shape is parametrised by t and the *spread*, s, that controls how slowly or sharply the system load increases as it approaches its maximum peak, halfway along t. Pulses are modelled on a normal distribution with $\mu = t/2$ and $\sigma = s$. *Burst* loads capture scenarios where a system is stressed due to load spikes; these are based on a log-normal distribution with $\mu = \ln(m^2/\sqrt{p^2+m^2})$, $\sigma = \sqrt{ln(1+p^2/m^2)}$, where $m = t/2$ and p is the *pinch* controlling the concentration of the initial load burst.

Wall-clock time. A load model generated for a logical timeline t needs to be put into effect by the master process when the system starts running. The master *cannot* simply create the slave processes that are set to execute in a particular time unit *all at once*, since this naïve strategy risks saturating the system, deceivingly increasing the load. Concretely, the system may become overloaded not because the mean request rate is high, but because the created slaves send their initial requests at one go. We address this issue by introducing the notion of *concrete timing* that maps a discrete time unit in t to a real time *period*, π. The parameter π is specified in milliseconds (ms), and defaults to 1000 ms.

Slave scheduling. The master process employs a scheduling scheme to distribute the creation of slaves uniformly across the time period π. It makes use of three queues: the *Order* queue, *Ready* queue, and *Await* queue, denoted by Q_{O}, Q_{R}, and Q_{A} respectively in fig. 1. Q_{O} is initially populated with the load profile, step ① in fig. 1a. It consists of an array with t elements (each corresponding to a discrete time instant in t), where the value l of every element indicates the number of slaves to be created at that instant. Slaves are scheduled and created in *rounds*, as follows. The master picks the first element from Q_{O} to



(a) Master schedules the first batch of four slaves for execution in Q_{R}

(b) Slaves S_1 and S_2 created and added to Q_{A}; a work request is sent to S_1

(c) Slaves S_3 and S_4 created and added to Q_{A}; slave S_2 completes execution

(d) Q_{R} becomes empty; master schedules next batch of two slaves

Fig. 1: Master M scheduling slave processes S_i and allocating work requests

compute the upcoming schedule, step ②, that starts at the current time, c, and finishes at $c+\pi$. A series of l time points, $p_1, p_2, \ldots, p_l$, in the schedule period π are *cumulatively* calculated by drawing the next p_i from a normal distribution with $\mu = \lceil \pi/l \rceil$ and $\sigma = \mu \times 0.1$. Each time point stipulates a moment in *wall-clock* time when a new slave is to be created; this set of time points is *monotonic*, and constitutes the Ready queue, Q_{R}, step ③. The master checks Q_{R}, step ④ in fig. 1b, and creates the slaves whose time point p_i is smaller than or equal to the current wall-clock time[4], steps ⑤ and ⑥. Newly-created slaves are removed from Q_{R} and appended to the Await queue Q_{A}, step ⑦, ready to receive work requests from the master, step ⑧. Q_{A} is traversed by the master at this stage so that work requests can be allocated to existing slaves. The master continues processing queue Q_{R} in subsequent rounds, creating slaves, issuing work requests, and updating Q_{R} and Q_{A} accordingly as shown in steps ⑨–⑬ in fig. 1c. At any point, the master can receive responses, *e.g.* step ⑰ in fig. 1d; these are *buffered* inside the masters' incoming work queue and handled once the scheduling and work allocation actions are complete. A fresh batch of slaves

[4] We assume that the platform scheduling the master and slave processes is *fair*.

from Q_{O} is scheduled by the master whenever Q_{R} becomes empty (fig. 1d), and the whole procedure is repeated. Scheduling stops when all the entries in Q_{O} are processed. The master then transitions to *work-only* mode, where it continues allocating work requests and handling incoming responses from slaves.

Reactiveness and task allocation. Systems generally respond to load with differing rates, due to the computational complexity of the task at hand, IO, or slowdown when the system itself becomes gradually loaded. We simulate this phenomenon using the parameters $\Pr(send)$ and $\Pr(recv)$. The master *interleaves* the sending and receiving of work requests to distribute tasks uniformly among the various slaves: $\Pr(send)$ and $\Pr(recv)$ bias this behaviour. Specifically, $\Pr(send)$ controls the probability that a work request is sent by the master to a slave, whereas $\Pr(recv)$ determines the probability that a work response received by the master is processed. Sending and receiving is *turn-based* and modelled on a Bernoulli trial. The master picks a slave S_i from Q_{A} and sends *at least* one work request when $X \leq \Pr(send)$; X is drawn from a uniform distribution on the interval $[0,1]$. Further requests to the *same* slave are allocated following the same procedure, steps ⑧, ⑬ and ⑳ in fig. 1, and the entry for S_i in Q_{A} is updated accordingly with the number of work requests remaining. When $X > \Pr(send)$, the slave misses its turn, and the next slave in Q_{A} is picked. The master also checks its incoming work queue to determine whether a response can be processed. A response is taken out from its incoming work queue when $X \leq \Pr(recv)$, and the attempt is repeated for the next response in the queue until $X > \Pr(recv)$. Slaves are instructed to terminate by the master once all of their work responses have been acknowledged (*e.g.* step ⑭). Due to the load imbalance that can occur when the master becomes overloaded with work responses sent by slaves, the *dequeuing* procedure is repeated $|Q_{\mathrm{A}}|$ times. This encourages an even load distribution in the system as the number of slaves *fluctuates* at runtime.

2.2 Realisability

We implemented our benchmarking tool using Erlang [13]. Erlang adopts the actor model [2], a message-passing paradigm where the units of decomposition are *actors*: concurrent units of decomposition that do not share mutable memory with other actors. Instead, they interact via *asynchronous messaging*, and change their internal state based on messages they receive. Each actor owns a (received) message queue, called a *mailbox*, where messages can consumed by the recipient at any stage, possibly *out-of-order*. Besides sending and receiving messages, an actor can also *fork* other actors. Actors can be uniquely addressed via a dynamically-assigned *identifier*, called the PID. Erlang implements actors as *lightweight* processes to enable massively-scalable architectures that span multiple machines. The terms *actor* and *process* are used interchangeably henceforth.

Implementation. We map the master and slave processes from sec. 2.1 to Erlang actors. The incoming work request queues for these processes coincide with actor mailboxes. We abstract the task computation, and model work requests as Erlang messages. Slaves do not emulate delay, but respond instantly to work requests;

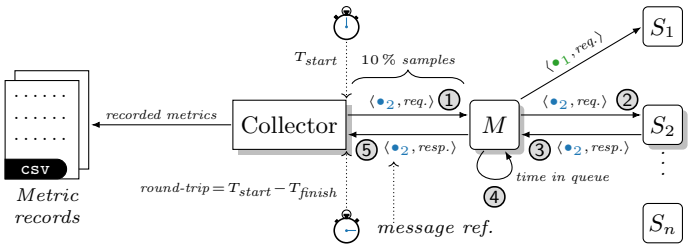


Fig. 2: Collector tracking the round-trip time for work requests and responses

delay in the system can be induced using parameters Pr(*send*) and Pr(*recv*). To maximise efficiency, the Order, Ready and Await queues used by our scheduling scheme are maintained *locally* within the master. The master process keeps track of other details, such as the total number of work requests sent and received, to determine when the system should stop executing. Our implementation extends the parameters in sec. 2.1 with a *seed* parameter, r, used to fix the Erlang pseudorandom number generator to output consistent number sequences.

2.3 Measurement Collection

Our set-up collects these metrics: *(i)* mean *scheduler utilisation*, as a percentage of the total available capacity, *(ii)* mean *memory consumption*, measured in GB, *(iii)* mean *response time (RT)*, measured in milliseconds (ms), and, *(iv)* mean *execution duration*, measured in seconds (s). The definition of runtime overhead we use encompasses all four metrics. Measurement taking largely depends on the platform on which the benchmark executes: one often leverages *platform-specific* optimised functionality in order to attain high levels of efficiency. Our implementation relies on the functionality provided by the Erlang ecosystem.

Sampling. We collect measurements centrally using a special process, called the *Collector*, that samples the runtime to obtain periodic snapshots of the environment (see fig. 2). Sampling is often necessary to induce low overhead in the system, especially in scenarios where the system components are sensitive to latency. Our sampling frequency is set to 500 ms: this figure was determined empirically, whereby the measurements gathered are neither too coarse, nor excessively fine-grained such that sampling affects the runtime. Every sampling snapshot combines the four metrics mentioned above and formats them as records that are written *asynchronously* to disk to minimise IO delays.

Performance metrics. Memory and scheduler readings using built-in functions offered by the Erlang Virtual Machine (EVM). We sample the scheduler rather than CPU utilisation at the OS-level, since the EVM keeps scheduler threads momentarily spinning to remain reactive; both would inflate the metric reading[5].

[5] This EVM feature can be turned off, but we opted for the default settings and to measure the scheduler utilisation metric inside the EVM instead.

The overall system responsiveness is captured by the RT metric. The collector exposes a hook that the master uses to obtain *unique timestamped references*, step ① in fig. 2; these opaque values are embedded in every work request message the master issues to slaves. Each reference enables the collector to track the time taken for one message to travel from the master to a slave and back, in addition to the time it spends waiting in the master's mailbox until dequeued, *i.e.,* the *round-trip* in steps ②–④. To efficiently compute the RT, the collector samples 10 % of the total number of messages exchanged between the master and slaves, step ⑤, and calculates the mean using Welford's [47] online algorithm.

3 Evaluation

We evaluate the synthetic benchmarking tool developed in Sec. 2 in a number of ways. Sec. 3.1 discusses sanity checks for its measurement collection mechanisms and assesses the repeatability of the obtained results from the synthetic system executions. Importantly, it provides evidence that the benchmarking tool is expressive enough to cover a number of execution profiles, that are also shown to emulate realistic scenarios. Sec. 3.2 demonstrates the utility of the features offered by the tool for the purposes of assessing RV tools.

Experiment set-up. We define an *experiment* to consist of ten benchmarks, each performed by running the system set-up with incremental loads. Our experiments were conducted on an Intel Core i7 M620 64-bit machine with 8GB of memory, running Ubuntu 18.04 and Erlang/OTP 22.2.1.

3.1 Benchmark Expressivity and Veracity

The benchmark parameters from sec. 2.1 can be configured to model a range of master-slave scenarios. However, not all of these configurations are meaningful in practice. For example, setting $\Pr(send)=0$ does not enable the master to allocate work requests to slaves; with $\Pr(send)=1$, this allocation is enacted sequentially, defeating the purpose of a concurrent master-slave set-up. We therefore establish a set of parameter values that model experiment set-ups whose behaviour *approximates* that of systems typically found in practice. Experiments in this section are conducted with $n = 500\text{k}$ slaves and $w = 100$ work requests per slave. This generates $\approx n \times w \times 2$ (work requests and responses) $= 100\text{M}$ messages exchanges between the master and slave processes. We initially fix $\Pr(send)=\Pr(recv)=0.9$, and choose a Steady (*i.e.,* Poisson process) load model with $\lambda = 5\text{k}$; this model is selected since it features in popular application load testing tools such as Tsung [38], Gatling [18] and JMeter [20]. The total loading time is set to $t=100\text{s}$.

Measurement precision. A series of trials were conducted to select the appropriate sampling window size for the RT. This step is crucial because it directly affects the capability of the benchmark to scale in terms of its number of slave processes and work requests. Our RT sampling algorithm was validated by taking various sampling window sizes over numerous runs for different load models

of ≈ 1M slaves. The results were compared to the *actual* mean calculated on *all* work request and response messages exchanged between master and slaves. Values close to 10 % yielded the best outcomes ($\approx \pm 1.4\%$ drift from the expected RT). Smaller window sizes produced excessive drift; larger sizes induced noticeably higher system loads. We also cross-checked the precision of our scheduler utilisation sampling method our benchmark against readings obtained via the Erlang Observer tool to confirm that these coincide [13].

Experiment repeatability. Data variability affects the *measurement repeatability*. It also plays a role when determining the number of repeated readings, i, required before the data measured is deemed *sufficiently representative*. Choosing the lowest i is crucial when experiment runs are time consuming. The *coefficient of variation* (CV), *i.e.,* the ratio of the standard deviation to the mean, $\text{CV} = \frac{\sigma}{\bar{x}} \times 100$, is often used to establish the value of i empirically, as follows. Initially, the CV for one batch of experiments for some number of repetitions i is calculated. The result is then compared to the CV for the next batch of repetitions $i' = i + b$, where b is the step size. When the difference between successive CV metrics i' and i is sufficiently small (for some percentage ϵ), the value of i is chosen, otherwise the procedure described is repeated with i'. Crucially, the CV condition must hold for *all variables* measured in the experiment before i can be fixed. For the results presented next, the CV values were calculated manually; we plan to implement the mechanism that determines the CV automatically later on.

Data variability. The data variability between experiments can be reduced by seeding the Erlang pseudorandom number generator (parameter r in sec. 2.2) with a constant value. This, in turn, tends to require fewer repeated runs before the metrics of interest—scheduler utilisation, memory consumption, RT, and execution duration—converge to an acceptable CV. We conduct experiments set with three, six and nine repetitions. For the majority of cases, the CV for the data variables considered is *lower* when a constant seed is used, by comparison to its unseeded counterpart (refer to fig. 9 in app. A). In fact, very low CV values for the scheduler utilisation, memory consumption, RT, and execution duration, 0.17 %, 0.15 %, 0.52 % and 0.47 % respectively, were obtained with three repeated runs. Consequently, we set the number of repetitions to *three* for all experiment runs in the sequel. Note that fixing the seed still allows the system to exhibit a modicum of variability: this stems from the inherent *interleaved execution* of components resulting from process scheduling.

System reactivity. The reactivity of the master-slave system correlates with the idle time of each slave which, in turn, affects the capacity of the system to *absorb* overheads. This can skew the results obtained when assessing overheads, and it was imperative for our benchmarking tool to provide methods to control this aspect. Parameters $\Pr(send)$ and $\Pr(recv)$ regulate the speed with which the system reacts to load. We study how these parameters affect the overall performance of system models set up with $\Pr(send) = \Pr(recv) \in \{0.1, 0.5, 0.9\}$. The results are shown in fig. 3, where each metric (*e.g.* memory consumption) is plotted against the total number of slaves. At $\Pr(send) = \Pr(recv) = 0.1$, the system has the lowest RT out of the three configurations, as indicated by the

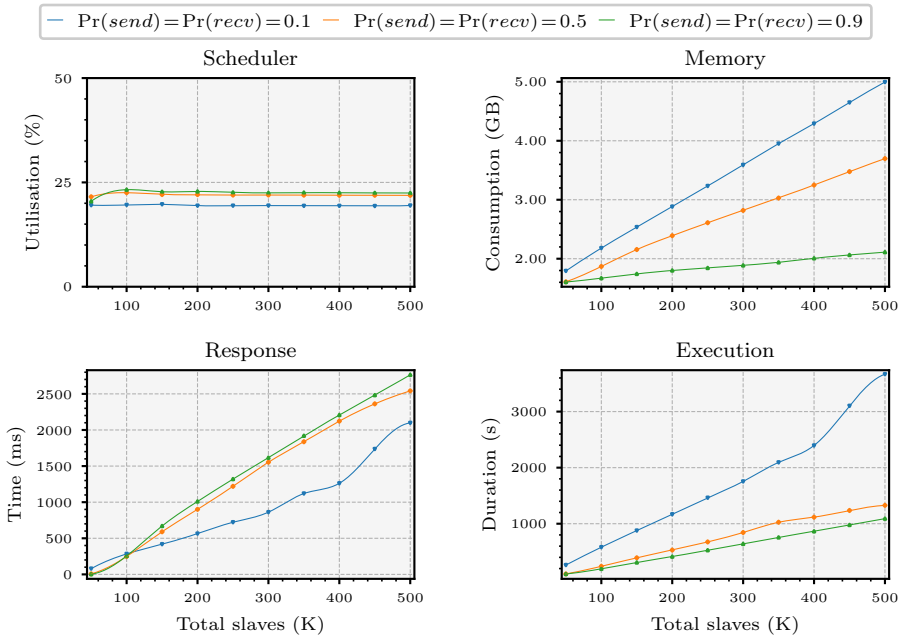


Fig. 3: Performance benchmarks of system models for Pr(*send*) and Pr(*recv*)

gentle linear increase of the respective plot. One may expect the RT to be *lower* for the system models configured with probability values of 0.5 and 0.9. However, we recall that at $\Pr(send) = 0.1$, work requests are allocated infrequently by the master, such that slaves are *often idle*, and can *readily* respond to (low numbers of) incoming work requests. The slow rate with which the master allocates work requests prolongs the overall execution duration, when compared to that of the system for $\Pr(send) = \Pr(recv) \in \{0.5, 0.9\}$. Meanwhile, the effect of idling can be gleaned from the relatively low scheduler utilisation. Idling also increases memory consumption, since slaves created by the master typically remain alive for extended periods. By contrast, the system model plots with $\Pr(send) = \Pr(recv) \in \{0.5, 0.9\}$ exhibit markedly lower gradients in the memory consumption and execution duration charts, and slightly steeper slopes in the RT chart. This indicates that values between 0.5 and 0.9 generate system models that: *(i)* consume reasonable amounts of memory, *(ii)* execute in respectable amounts of time, and, *(iii)* maintain tolerable RT. Since the master-slave architecture is typically employed in settings where high throughput is demanded, choosing values that are less than 0.5 goes against this notion. In what follows, we opt for $\Pr(send) = \Pr(recv) = 0.9$.

Load profiles. When judging the scalability of a monitoring set-up, one should consider its behaviour under varying forms of stress loads. Our tool is expressive enough to generate the load profiles introduced in sec. 2.1 (see fig. 4). These make it possible to mock specific system scenarios to test different implementation

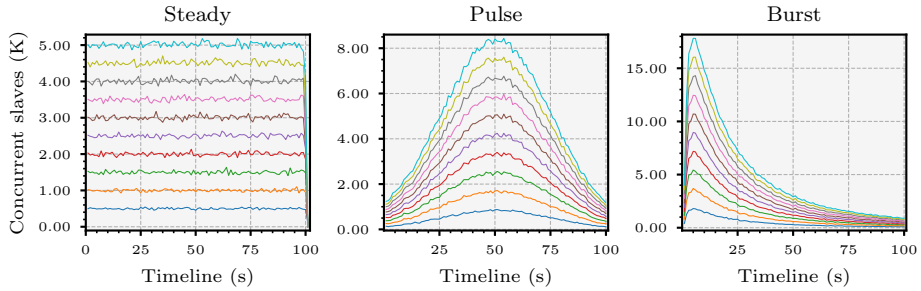


Fig. 4: Steady, Pulse and Burst load distributions of 500 k slaves in 100 s

aspects. For example, a test that subjects the system to load surges could uncover buffer overflows in certain monitor implementations that arise when the length of the request queue exceeds some pre-set length.

Emulation veracity. Our benchmarks can be configured to closely model *realistic* web server traffic where the request intervals observed at the server are known to follow a Poisson process [28,34,30]. The probability distribution of the RT of web application requests is generally right-skewed, and approximates log-normal [28,17] or Erlang distributions [30]. We conduct three experiments using *Steady loads* fixed with $n = 10$k; $\Pr(send) = \Pr(recv)$ are varied through 0.1, 0.5 and 0.9 to establish whether the RT in our system set-ups resembles the aforementioned probability distributions. Our results, summarised in fig. 5, were obtained by estimating the parameters for a set of candidate probability distributions (*e.g.* normal, log-normal, gamma, *etc.*) using Maximum Likelihood Estimation [42] on the RT obtained from *each* experiment. We then performed goodness-of-fit tests on these parametrised distributions, selecting the most appropriate RT fit for each of the three experiments using the Kolmogorov-Smirnov test. The fitted distributions in fig. 5 indicate that the RT of our system models follows the findings reported in [28,17,30]. Our benchmarking tool thus gives us the best of both worlds: the realism of benchmarks based on OTS programs coupled with the control that is characteristic of synthetic benchmarks.

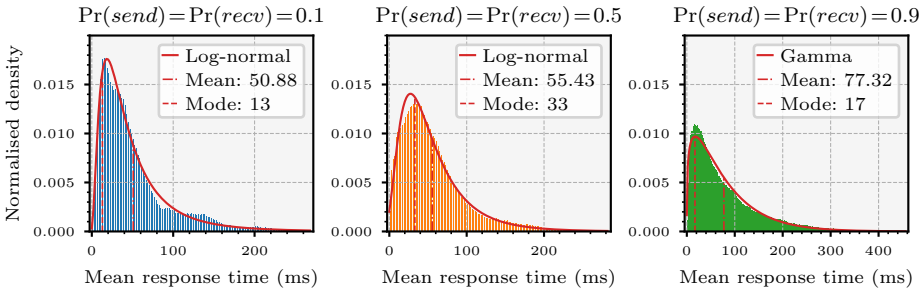


Fig. 5: Fitted probability distributions on RT for Steady loads for $n = 10$k

3.2 Case Study

We demonstrate how our benchmarking tool can be used to assess the runtime overhead more comprehensively via a concurrent RV case study. By controlling the benchmark parameters and subjecting the system to various workloads, we are able to induce certain overhead behaviour.

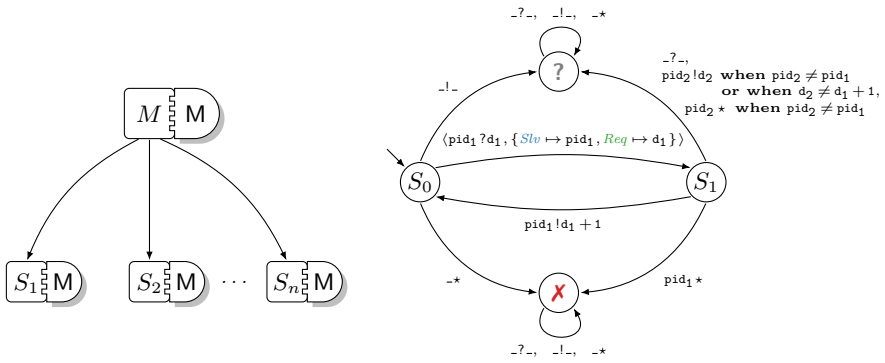
The RV tool. The tool used for our case study [3,4] synthesises *automata-like* monitors from sHML [22] specifications. It instruments them into the target system via *code injection* by manipulating the program abstract syntax tree (see app. B). sHML is a *fragment* of the Hennesy-Milner logic with recursion [33] that can express all regular safety properties [22]. Our tool augments it to handle pattern matching and data dependencies for three kinds of event patterns, namely *send* and *receive* actions, denoted by ! and ? respectively, and process *crash*, denoted by $\star$. This logic suffices to specify properties of both the master and slave processes, resulting in the set-up depicted in fig. 6a. For instance, the recursive property (ExSpec) below describes an invariant of the master-slave communication protocol (from the slave's perspective), stating that 'a slave process processing integer successor requests should not crash':

$$\mathsf{max}X.\Big([\backslash \mathit{Slv}\,\star]\mathsf{ff} \wedge [\backslash \mathit{Slv}\,?\backslash \mathit{Req}]\big([\mathit{Slv}\,\star]\mathsf{ff} \wedge [\mathit{Slv}\,!(\mathit{Req}+1)]X\big)\Big) \qquad \text{(ExSpec)}$$

The key construct in sHML is the modal formula $[p]\varphi$, stating that *whenever* a satisfying system produces an event e matching pattern p, its continuation *must* satisfy φ. More concretely, (ExSpec) is an *invariant*, denoted by recursion binder $\mathsf{max}X$, specifying that a slave cannot crash immediately, via the subformula $[\backslash \mathit{Slv}\,\star]\mathsf{ff}$. In addition, whenever it receives a request carrying a payload Req, denoted by $[\backslash \mathit{Slv}\,?\backslash \mathit{Req}]\ldots$, this same slave still cannot crash, $[\mathit{Slv}\,\star]\mathsf{ff}$, and if this slave answers the request correctly by carrying payload $\mathit{Req}+1$, the property recurses, $[\mathit{Slv}\,!(\mathit{Req}+1)]X$. Action patterns employ two types of value variables: binders, $\backslash x$, that are pattern-matched to concrete values learnt at runtime, and variable instances, x, that are bound by the respective binders and instantiated to concrete data via pattern matching at runtime. This induces the usual notion of free and bound value variables; we assume closed terms. For example, when checking (ExSpec) against the trace event `pid?42`, the analysis would unfold the recursion and pattern match the event against $\backslash \mathit{Slv}\,?\backslash \mathit{Req}$, substituting the variables Slv and Req with `pid` and `42` respectively, leaving the residual formula:

$$[\mathtt{pid}\star]\mathsf{ff} \wedge [\mathtt{pid}!(\mathtt{42}+1)]\mathsf{max}X.\begin{pmatrix}[\backslash \mathit{Slv}\,\star]\mathsf{ff} \wedge \\ [\backslash \mathit{Slv}\,?\backslash \mathit{Req}]\big([\mathit{Slv}\,\star]\mathsf{ff} \wedge [\mathit{Slv}\,!(\mathit{Req}+1)]X\big)\end{pmatrix}$$

Specifications produce *embedded* monitor code that executes in the same process space of a system component, inducing the lowest possible amount of runtime overhead. This enables us to scale our benchmarks to considerably high loads. Our case study experiments focus on correctness properties that are *parametric* w.r.t. to system components [4,16,41,37]; with this approach, monitors need not interact with one another and can reach verdicts independently. Verdicts



(a) Monitor inlining (b) Synthesised monitor from (ExSpec)

Fig. 6: Synthesised monitors instrumented with master and slave processes

are communicated by monitors to a central monitor that records the expected number of verdicts in order to determine when the experiment can be stopped. The set of properties used in our benchmarks translate to monitors that loop continually to exert the maximum level of runtime overhead possible. Fig. 6b shows the monitor synthesised from (ExSpec), consisting of states S_0, S_1, the rejection state ✗ and inconclusive state ?. The rejection state corresponds to a *violation* of the property, *i.e.,* ff, whereas the *inconclusive* state is reached when the trace events analysed do not contain enough information to enable the monitor to transition to any other state. In fig. 6b, the rejection and inconclusive states are depicted as sinks, modelling the irrevocability of verdicts [21,22].

The modality $[\backslash \mathit{Slv}?\backslash \mathit{Req}]$ in (ExSpec) corresponds to the transition between S_0 and S_1 in fig. 6b. The monitor follows this transition when it analyses the trace event $\texttt{pid}_1\texttt{?d}_1$ exhibited by the slave with PID $\texttt{pid}_1$ when it receives work chunk payload $\texttt{d}_1$ from the master; as a side effect, the transition binds the variable *Slv* to $\texttt{pid}_1$ and *Req* to $\texttt{d}_1$ in state S_1. From S_1, the monitor transitions to S_0 only when the event $\texttt{pid}_1\texttt{!d}_2$ is analysed, where $\texttt{d}_2 = \texttt{d}_1 + 1$ and $\texttt{pid}_1$ is the slave PID *previously* assigned to *Slv*. From S_0 and S_1, the rejection state ✗ can be reached when a crash event is analysed. In the case of S_0, the transition to ✗ is followed for *any* crash event _⋆ (_ denotes the *anonymous* variable). By contrast, the monitor reaches ✗ from S_1 *only* when the slave with PID $\texttt{pid}_1$ crashes, otherwise it transitions to the inconclusive state ?. Other transitions from S_0 and S_1 leading to ? follow a similar reasoning.

Case study set-up. Our benchmarks are set with $n = 20\text{k}$ for *moderate* loads and $n = 500\text{k}$ for *high* loads; $\Pr(\mathit{send}) = \Pr(\mathit{recv})$ is fixed at 0.9 as in sec. 3.1. These configurations generate $\approx n \times w \times 2$ (work requests and responses) $= 4\text{M}$ and 100M messages respectively, producing 8M and 200M analysable trace events per run. We seed the pseudorandom number generator with a constant value, and perform three experiment repetitions for the load profiles of fig. 4. A loading

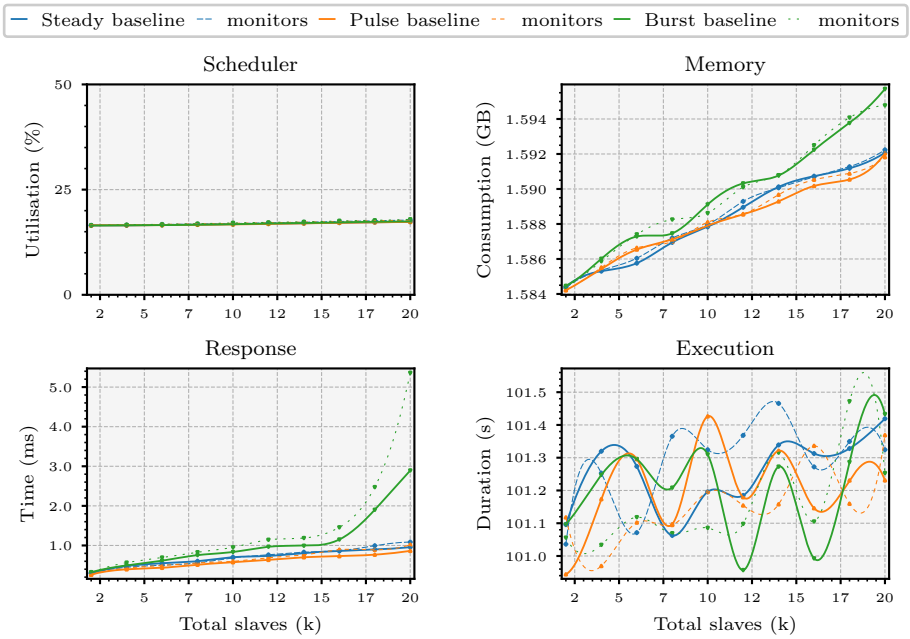


Fig. 7: Mean runtime overhead for master and slave processes (20 k slaves)

time of $t=100$s is used. The results are summarised in figs. 7 and 8. Each of the four chart in these figures plots the particular performance metric (*e.g.* memory consumption) for the system without monitors, *i.e.,* the *baseline*, together with the overhead induced by runtime monitors for our three load profiles (fig. 4).

Moderate loads. The loads considered in the first benchmark (n=20k) are comparable to those used by the state-of-the-art to evaluate component-based runtime monitoring, *e.g.* [43,4,7,19,37] (ours are slightly higher). None of the benchmarks used in these works employ different load profiles: they either model load on a Poisson process, or fail to specify the kind of load used. Fig. 7 shows the plots for the system set with $n = 20$k. The execution duration chart (bottom right) shows that, regardless of the load profile used, the running time of each experiment is comparable to the baseline (within 500 ms). We deem these results to be inconclusive. Despite the broad benchmark coverage provided by the three load profiles, this trend is again mirrored in the scheduler utilisation plot (top left), where both the baseline and monitored system induce a constant load of $\approx$17.5%. The load profiles however induce *different* overheads for the RT (bottom left), and, to a lesser degree, the memory consumption plots (top right). Specifically, when the system is subjected to a Burst load, it exhibits a surge in the RT for both the baseline and monitored system, at ≈ 16k. While this does not occur in the memory consumption plot, we note that the aforementioned Burst plots exhibit a larger, albeit linear, rate of increase in memory when compared to the Steady and Pulse plots. The latter plots once again display analogous trends,

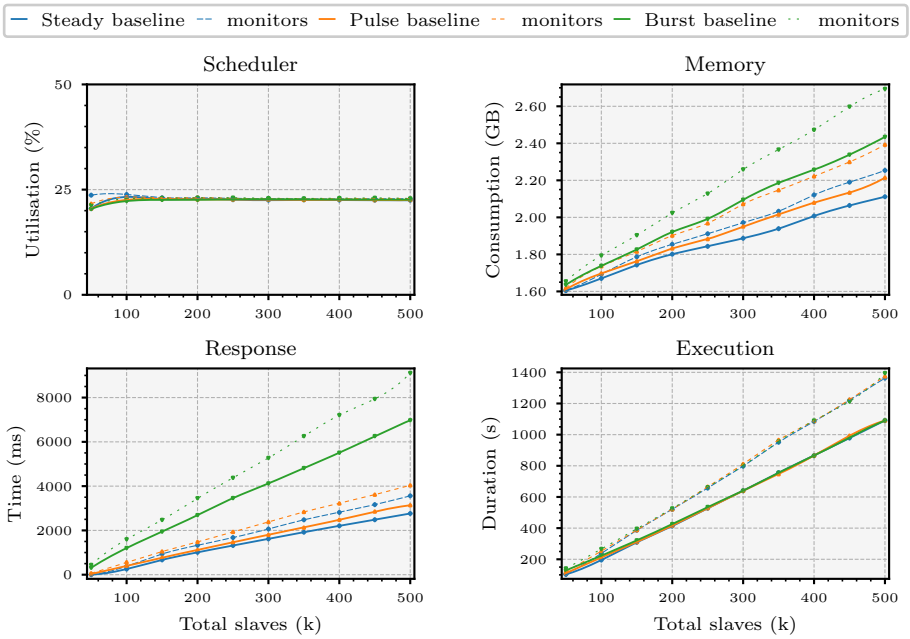


Fig. 8: Mean runtime overhead for master and slave processes (500 k slaves)

suggesting that Steady and Pulse loads have similar memory requirements, and exhibit comparable responsiveness under the respectable load of 20k slaves. Importantly, our data plots do not allow us to confidently extrapolate our results. The RT metric Burst plots raise the question of whether the trend observed remains consistent when the number of slave processes exceeds 20k.

High loads. We increase the number of slaves to $n = 500$k, to determine whether the benchmark set-up can *adequately scale* to reveal how the monitored system performs under stress. The RT chart in fig. 8 shows that for Burst loads, it *grows linearly* in the number of slaves. This confirms our earlier assertion that moderate loads provide scant empirical evidence to extrapolate to general conclusions. However, for memory consumption, Burst plots exhibit similar trends to the ones in fig. 7. Subjecting the system to high loads renders discernible the *discrepancy* between the RT and memory consumption gradients for the Steady and Pulse plots; these appeared to be similar for moderate loads (fig. 7). Perhaps deceivingly, the execution duration plots of fig. 8 induce virtually identical overhead to those in fig. 7, notwithstanding the *distinct* load profiles. But this *erroneous* observation is clearly refuted by the RT and memory consumption plots. This highlights the importance of gathering *multi-faceted* performance metrics to better assist in the interpretation of runtime overhead results.

We extend the argument for multi-faceted views to the scheduler utilisation metric in fig. 8 that reveals a subtle aspect of our concurrent set-up. Specifically, the charts show that while the the execution duration, RT and memory con-

sumption plots grow in the number of slave processes, scheduler utilisation stabilises at ≈22.7%. This is partly caused by the master-slave design that becomes susceptible to bottlenecks when the master is overloaded with requests [46]. In addition, the preemptive scheduling of the EVM [13] ensures that the master *shares* the computational resources of the same machine with the rest of the slaves. We conjecture that, in a distributed set-up where the master resides on a *dedicated* node, the overall system throughput may be further pushed. Fig. 8 also attests to the utility of having a benchmarking framework that scales considerably well to increase the chances of detecting likely trends. For instance, the evidence gathered in fig. 7 could have misled one to assert that the RV tool under inspection scales poorly under Burst loads of moderate size.

4 Conclusion

We presented a benchmarking framework that targets RV tools for concurrent settings. Our set-up emulates various system models via configurable parameters, and can scale considerably to reveal behaviour that only emerges when software is pushed to its limit. We show that, despite its synthetic nature, our chosen master-slave architecture faithfully approximates the behaviour observed in realistic web server traffic. The benchmark harness gathers different metrics, providing a multi-faceted view of runtime overhead that, to wit, other state-of-the-art tools do not currently offer. Our experiments demonstrate that these metrics benefit the interpretation of results, providing assistance when forming conclusions that may otherwise, lack generality or in certain cases, be erroneously drawn. Our tool source code can be found at `https://github.com/duncanatt/detecter`.

Future work. We plan to transition to a distributed set-up where master and slaves reside on different nodes, using the network simulator Kollaps [26], that can be used to induce unreliable network conditions such as latency and packet loss. Our tool can also be extended to support peer-to-peer architectures.

Related work. There are other less popular benchmarks that are specific to the JVM besides those mentioned in sec. 1. Renaissance [40] employs workloads that leverage the concurrency primitives exposed by the JVM, targeting the performance of compiler optimisations similar to DaCapo and ScalaBench. These benchmarks collect metrics that measure software quality and complexity, as opposed to metrics that gauge runtime overhead. The CRV benchmark suite [5] was developed to standardise the evaluation of RV tools. It mainly focusses on RV for monolithic programs that run on the JVM and programs written in C. We are not aware of any RV-centric benchmarks for concurrent systems such as ours that target the JVM or EVM platforms. In [34], the authors propose a queueing model to analyse web server traffic, and develop a benchmarking tool to validate it. Their model coincides with our master-slave architecture, and considers loads based on a Poisson process. A study of message-passing communication on parallel computers conducted in [28] uses systems loaded with different numbers of processes; this is similar to our approach. Importantly, we were able to confirm the findings presented in both [34] and [28] (see sec. 3.1).

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Kjartansson, S.Ö.: Determinizing Monitors for HML with Recursion. JLAMP **111**, 100515 (2020)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A Foundation for Actor Computation. JFP **7**(1), 1–72 (1997)
3. Attard, D.P., Francalanza, A.: A Monitoring Tool for a Branching-Time Logic. In: RV. LNCS, vol. 10012, pp. 473–481 (2016)
4. Attard, D.P., Francalanza, A.: Trace Partitioning and Local Monitoring for Asynchronous Components. In: SEFM. LNCS, vol. 10469, pp. 219–235 (2017)
5. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First International Competition on Runtime Verification: Rules, Benchmarks, Tools, and Final Results of CRV 2014. Int. J. Softw. Tools Technol. Transf. **21**(1), 31–70 (2019)
6. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Lectures on RV, LNCS, vol. 10457, pp. 1–33 (2018)
7. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime Verification with Minimal Intrusion through Parallelism. FMSD **46**(3), 317–348 (2015)
8. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: OOPSLA. pp. 169–190 (2006)
9. Blessing, S., Fernandez-Reyes, K., Yang, A.M., Drossopoulou, S., Wrigstad, T.: Run, Actor, Run: Towards Cross-Actor Language Benchmarking. In: AGERE!@SPLASH. pp. 41–50 (2019)
10. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. J. Log. Comput. **20**(3), 707–723 (2010)
11. Bonakdarpour, B., Finkbeiner, B.: The Complexity of Monitoring Hyperproperties. In: CSF. pp. 162–174 (2018)
12. Buyya, R., Broberg, J., Goscinski, A.M.: Cloud Computing: Principles and Paradigms (2011)
13. Cesarini, F., Thompson, S.: Erlang Programming: A Concurrent Approach to Software Development (2009)
14. Chappell, D.: Enterprise Service Bus: Theory in Practice (2004)
15. Chen, F., Rosu, G.: Mop: An Efficient and Generic Runtime Verification Framework. In: OOPSLA. pp. 569–588 (2007)
16. Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: TACAS. LNCS, vol. 5505, pp. 246–261 (2009)
17. Ciemiewicz, D.M.: What Do You mean? - Revisiting Statistics for Web Response Time Measurements. In: CMG. pp. 385–396 (2001)
18. Corp., G.: Gatling (2020), https://gatling.io
19. El-Hokayem, A., Falcone, Y.: Monitoring Decentralized Specifications. In: ISSTA. pp. 125–135 (2017)
20. Foundtation, A.S.: Jmeter (2019), https://jmeter.apache.org
21. Francalanza, A.: A Theory of Monitors (Extended Abstract). In: FoSSaCS. LNCS, vol. 9634, pp. 145–161 (2016)
22. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the Hennessy-Milner Logic with Recursion. FMSD **51**(1), 87–116 (2017)

23. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems. In: Lectures on RV, LNCS, vol. 10457, pp. 176–210 (2018)
24. Francalanza, A., Xuereb, J.: On Implementing Symbolic Controllability. In: COORDINATION. LNCS, vol. 12134, pp. 350–369 (2020)
25. Ghosh, S.: Distributed Systems: An Algorithmic Approach (2014)
26. Gouveia, P., Neves, J., Segarra, C., Liechti, L., Issa, S., Schiavoni, V., Matos, M.: Kollaps: Decentralized and Dynamic Topology Emulation. In: EuroSys. pp. 23:1–23:16 (2020)
27. Gray, J.: The Benchmark Handbook for Database and Transaction Processing Systems (1993)
28. Grove, D.A., Coddington, P.D.: Analytical Models of Probability Distributions for MPI Point-to-Point Communication Times on Distributed Memory Parallel Computers. In: ICA3PP. LNCS, vol. 3719, pp. 406–415 (2005)
29. Imam, S.M., Sarkar, V.: Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In: AGERE!@SPLASH. pp. 67–80 (2014)
30. Kayser, B.: What is the expected distribution of website response times? (2017), https://blog.newrelic.com/engineering/expected-distributions-website-response-times
31. Kshemkalyani, A.D.: Distributed Computing: Principles, Algorithms, and Systems (2011)
32. Kuhtz, L., Finkbeiner, B.: LTL Path Checking is Efficiently Parallelizable. In: ICALP (2). LNCS, vol. 5556, pp. 235–246 (2009)
33. Larsen, K.G.: Proof systems for satisfiability in hennessy-milner logic with recursion. Theor. Comput. Sci. **72**(2&3), 265–288 (1990)
34. Liu, Z., Niclausse, N., Jalpa-Villanueva, C.: Traffic Model and Performance Evaluation of Web Servers. Perform. Evaluation **46**(2-3), 77–100 (2001)
35. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An Overview of the MOP Runtime Verification Framework. STTT **14**(3), 249–289 (2012)
36. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing (2011)
37. Neykova, R., Yoshida, N.: Let it Recover: Multiparty Protocol-Induced Recovery. In: CC. pp. 98–108 (2017)
38. Niclausse, N.: Tsung (2017), http://tsung.erlang-projects.org
39. Nielsen, J.: Usability Engineering (1993)
40. Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., Würthinger, T., Binder, W.: Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In: PLDI. pp. 31–47 (2019)
41. Reger, G., Cruz, H.C., Rydeheard, D.E.: MarQ: Monitoring at Runtime with QEA. In: TACAS. LNCS, vol. 9035, pp. 596–610 (2015)
42. Rossi, R.J.: Mathematical Statistics: An Introduction to Likelihood Based Inference (2018)
43. Scheffel, T., Schmitz, M.: Three-Valued Asynchronous Distributed Runtime Verification. In: MEMOCODE. pp. 52–61 (2014)
44. Sewe, A., Mezini, M., Sarimbekov, A., Binder, W.: DaCapo con Scala: design and analysis of a Scala benchmark suite for the JVM. In: OOPSLA. pp. 657–676 (2011)
45. SPEC: Specjvm2008 (2008), https://www.spec.org/jvm2008
46. Tarkoma, S.: Overlay Networks: Toward Information Networking (2010)
47. Welford, B.P.: Note on a Method for Calculating Corrected Sums of Squares and Products. Technometrics **4**(3), 419–420 (1962)

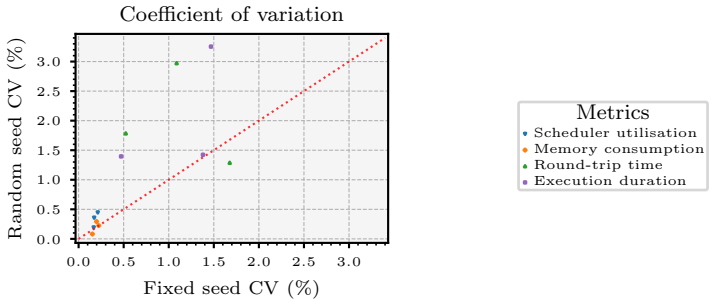


Fig. 9: CV for unfixed and fixed randomisation seeds for 3,6,9 repetitions

A Model System Parameters

Further to the model system parameters discussed in sec. 3.1, the following supporting empirical measurements were also taken.

Data variability. Fig. 9 shows the relationship between different CV metrics obtained for the system when executed with unfixed ($y-$axis) and fixed ($x-$axis) pseudorandom number generator seeds. The experiments were performed for three, six and nine repetitions, using the parameters fixed in sec. 3. For our four chosen performance metrics, using a constant seed tends to induce less variability in the experiments, *i.e.,* a low CV, as indicated in fig. 9, where only two points lie *below* the identity line $y=x$.

Load profiles. The load profiles presented in sec. 2.1 induce different performance overhead on the system. Fig. 4 shows the plots for the performance metrics considered in our experiments under the loads in fig. 4, where $n=500$k. As anticipated, every load shape yields different results for the metrics considered, all of which grow linearly in the size of the total number of slaves; scheduler utilisation does not follow this trend, and plateaus at around 22 % in all three cases. The Steady and Pulse plots coincide in the case of RT and share similar execution duration gradients, but then exhibit a slight degree of divergence in the growth rates of memory consumption. Steady loads yield the lowest and most consistent overhead in all the performance metrics considered. This is attributable to the regularity of the homogeneous Poisson process on which Steady loads are modelled. By contrast, Bursts induce the highest levels of overheads, where the growth rate factors for RT and memory consumption relative to Steady loads are $\approx$2.8 and $\approx$1.9 respectively. This load-inducing behaviour did not emerge for the execution duration, where the plot for Burst is analogous to the one produced the Steady load.

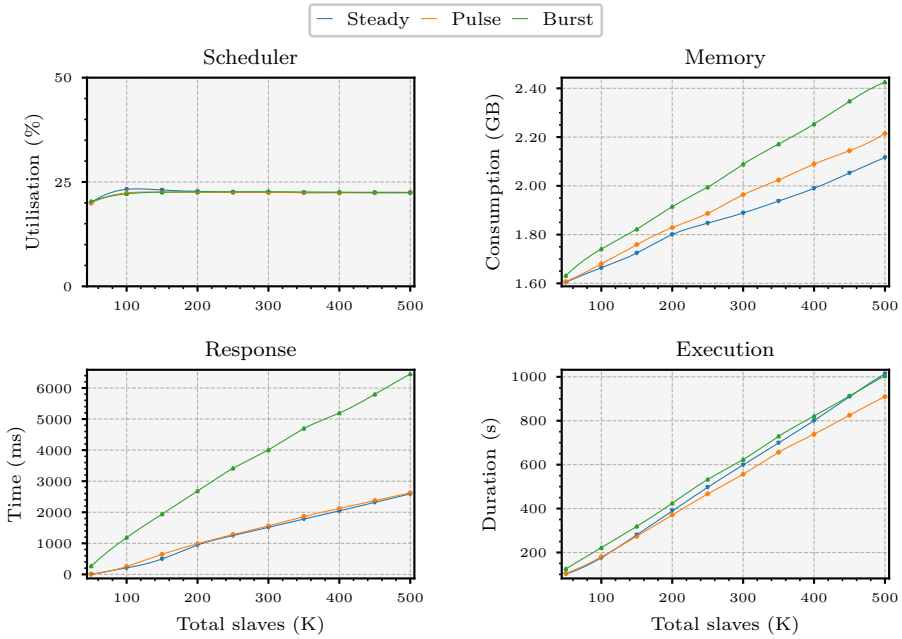


Fig. 10: Performance benchmarks of system models under the loads in fig. 4

B Case Study

Monitor instrumentation. Our RV tool instruments monitors into the target system via *code injection* by manipulating the program abstract syntax tree (AST). Fig. 11 outlines how this process is carried out. In step ①, the Erlang source code of the system is parsed into the corresponding AST, step ②. The Erlang compilation process contains a *parse transform phase* [13], step ③, that provides a hook to enable the AST to be post-processed. Our custom-built weaver leverages this mechanism in step ④, to embed into the program AST the AST of the synthesised monitor, step ⑤. The weaver performs two types of transformations:

(i) *Monitor bootstrapping.* The function encoding the monitor is stored in the process dictionary (a process-local key-value store) of the monitored process.
(ii) *Instrumentation points.* The program AST is instrumented with calls at the points of interest: these calls constitute the extraction, and analysis of system trace events by the monitor code.

The instrumented calls in transformation *(ii)* retrieve the monitor function stored in the process dictionary in transformation *(i)*, and apply it to the system trace event. The function application returns the monitor continuation function that is used to replace the current monitor in the process dictionary. Our two-step weaving procedure produces the instrumented program AST in step ⑥, that can be subsequently compiled by the Erlang compiler into the system binary.

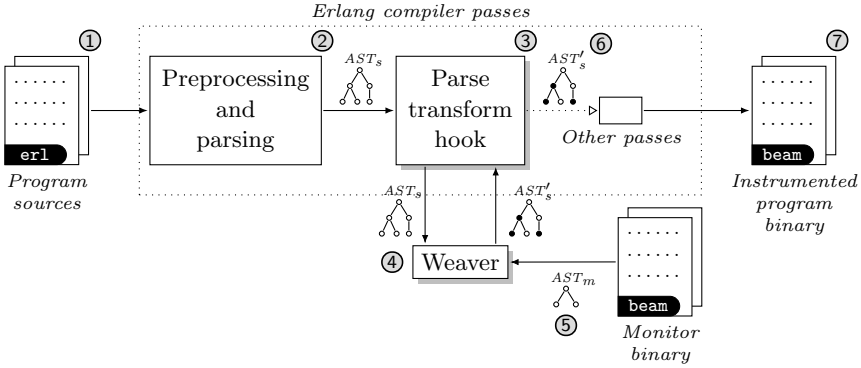


Fig. 11: Instrumentation pipeline for inline monitors