Push it to the Limit*

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. When assessing the prototyping of software tools, one often resorts to off-the-shelf systems to acquire empirical evidence. This approach, while valid, has two principal shortcomings. First, third-party systems do not, by design, provide hooks that permit the acquisition of experiment data, complicating most experiment set-ups. Second, they make it hard to control certain experiment parameters. Thus, repeatability becomes difficult to achieve. We present a framework that can be used in lieu of the aforementioned systems to set up and drive experiments in a systematic fashion. Our framework can emulate various system models via configuration and, crucially, is able to generate significant loads to reveal behaviour that may emerge when software is pushed to its limit. We discuss the challenges faced when developing this framework together with the preventive steps taken to measure experiment data while inducing minimal perturbations. We also show how our framework can be instantiated with parameters that approximate particular realistic behaviour, using it to conduct a study for runtime monitoring.

Keywords: Parametrisable driver systems · Synthetic load models

1 Introduction

Large-scale software design has recently shifted from the classic monolithic architecture to one where applications are structured as *independent components* [11]. Component-based approaches spurred the growth of cloud computing, where state-of-the-art commercial platforms offer various configuration options to suit different deployment needs [27]. Serverless computing [8] is one such execution model, where application code segments are run on the cloud and billed dynamically based on resource usage (*e.g.* AWS Lambda, Azure Functions), decreasing the costs associated with server set up and administration. Internally,

* This research was supported by the Icelandic Research Fund project "TheoFoMon: Theoretical Foundations for Monitorability" (No:163406-051), project BehAPI, funded by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant (No:778233), the Reykjavík University Research Fund, ENDEAVOUR Scholarship Scheme - Group B - National Funds, and the Italian MIUR project PRIN 2017FTXR7S IT MATTERS "Methods and Tools for Trustworthy Smart Systems".

components consist of *distinct* logical parts that interact together to fulfil some functional goal. This logic is often developed in *layers*, starting from the core component functionality, and adding ancillary behaviour (*e.g.* logging or packet filtering) on top of existing code in *incremental stages*. Such functionality often consists of *boiler-plate* code that can be generalised and offered as a *supporting library* (service, or tool) to software developers, promoting modularity and reuse. Various third-party industry-strength application servers (*e.g.* WildFly, Payara), frameworks (*e.g.* Spring, Google Guice) and libraries that cater for customary use cases (*e.g.* database transaction management, runtime monitoring, code profiling and tracing, *etc.*) are instantiations of this principle.

One crucial step in developing these libraries is to assess their intended behaviour in terms of their *functional* and *non-functional* aspects. Testing [24] is a common method used to quickly check whether software meets its functional requirements. It is a non-exhaustive approach, and can be easily integrated into the development life-cycle to provide software writers with immediate feedback during the *pre-deployment* phase [19]. By contrast, evaluating the non-functional requirements of software is much harder to accomplish and is not as clear-cut as testing, since this focusses on judging the *operation of software*, rather than its range of specific capabilities. Amongst the many non-functional requirements, runtime overhead and resource consumption are key attributes that affect the operation of software, *e.g.* a micro-service that exposes correct functionality is unusable if it is susceptible to time-outs. Analysing and interpreting non-functional attributes in a *systematic manner*, as practised in testing, carries numerous benefits. It can enable developers to properly estimate the potential increase in resource usage when a library or feature is incorporated into an existing system; it may also offer clues that help identify and correct possible defects (*e.g.* performance bottlenecks) that could otherwise remain undetected. This analysis can also provide evidence regarding the *overall stability* of the completed software.

However, assessing non-functional requirements is challenging since it needs to be measured post-deployment when the software *actually runs*. This makes the collective effects of certain software features, such as combining different libraries or replicating components, *hard to predict*. For instance, a slow logging library could cause a database transaction library that depends on it to fail, because the latter assumes some predetermined timeout. In a component-based setting, slight overheads deemed benign in one component can quickly add up when this is massively replicated: this behaviour may never emerge under normal operating conditions, but surfaces only when the system is subjected to *substantial loads*.

By and large, the analysis of runtime overhead is typically conducted using a *driver* system that acts as a *harness* into which the library under study can be integrated and run. The choice of driver system depends on the evaluation sought. One may employ a third-party off-the-shelf (OTS) system as a driver when the *applicability* of the library on real-world systems needs to be understood, (*e.g.* [3,34,7,12,5]), or opt for a custom-built synthetic set-up that can be parametrised to emulate different settings, thus enabling more *comprehensive* studies (*e.g.* [14,30,16,4]). An appealing feature that makes an OTS system a *de*

facto candidate to use as a driver is the fact that such an artefact is ready-to-use, minimising upfront development costs. At the same time, however, this impinges on the evaluation one wishes to perform, since the set-up is at the mercy of the functionality the chosen OTS system offers.

This paper argues that custom synthetic driver systems offer a host of qualitative benefits that offset the assumed drawbacks associated with developing driver systems from scratch. For example, the scalability of software in distributed settings is difficult to assess, and researchers often resort to driver systems or simulators for this purpose [19]. This is attested by the plethora of third-party load testing tools such as Tsung [26] or Gatling [15], and simulators like Packet Tracer [31] or NetSim [33]. Notwithstanding the capability of some of these tools to generate substantial loads, their effectiveness nevertheless *depends* on the architecture of driver system they interface with. A driver comprised of an OTS system could, for instance, *restrain the potential* of load testing tools simply because it fails to scale: as a result, the software library under investigation is never pushed to its proper limit. Driver systems can be engineered to circumvent these limitations, and to streamline the evaluation of certain aspects of the software being assessed. In this context, our contributions are:

- The development of a configurable driver framework that emulates various distributed system models, implemented in Erlang (Sec. 3);
- The assessment of different system models generated by the framework via a series of controlled experiments to identify a set of configuration parameter values that approximate realistic system behaviour (Sec. 4);
- The application of these models to carry out a systematic assessment of the runtime overheads at high loads for a prototype monitoring tool (Sec. 4).

2 The Case for Synthetic Driver Systems

Adapting a third-party OTS system as a driver poses numerous challenges. Often, the chosen system is treated as a *black box*. However, a decent level of *internal knowledge* about the OTS system and its characteristics is typically required if one is to interpret and draw meaningful conclusions from the analysis being made. This would mostly limit the selection of OTS systems to ones whose source code is easily accessible. OTS systems also make it challenging to *precisely control* the specific system parameters or variables required to drive an experiment. This tends to impact the analysis' capability of obtaining *repeatable results* in cases where empirical metrics for *sensitive* non-functional attributes need to be collected. While these issues can be partially addressed by modifying the OTS system, this requires considerable development effort that risks introducing subtle bugs (*e.g.* unintended overheads), undermining the experiment itself. An OTS system is bound to only cover *highly-specific* use cases, and the runtime data generated cannot be used to generalise the results concluded. To alleviate this problem, one could build a *suite* of OTS systems to broaden the scope of the testing scenarios, but this endeavour is tedious and time-consuming; the

Java-based DaCapo [6] benchmark adopts this strategy, using a suite of eleven open-source applications to drive its analysis.

These reasons make OTS systems hard to use as drivers when these are included in tight software development cycles. Synthetic driver systems can complement the OTS set-up described above. This view, of course, assumes that the driver system abstracts some specific architectural details, but *adequately approximates* the characteristic behaviour of the real systems one wishes to model. The major drawback of synthetic drivers is the initial effort that must be invested in their development. While this seems daunting, we hold that synthetic drivers offer a number of advantages that counterbalance their development cost. We show how *generic driver systems* can be engineered to make it possible to control specific evaluation parameters and collect runtime metrics efficiently. To this end, we design the driver as a *parametrisable harness* that can be configured to produce system models which cover a varied range of scenarios; these include *corner cases* that might not be possible to emulate via other means. This set-up makes it viable to conduct quick *sanity checks* or to extract exploratory findings on the software functionality being evaluated. For instance, getting a broad estimate of resource consumption is useful to gauge the cost of deploying the functionality on a serverless platform. This can be done without resorting to fully-featured system set-ups or deployments. Being able to effect high-level tests is indispensable when the software under evaluation is developed incrementally, where it may undergo several redesigns before it is finally released.

3 Design and Implementation

Our driver system set-up can emulate a range of distributed system models and subject them to various types of loads. Its implementation addresses two aspects that we deem central in this setting: *(i)* the facility to create different system models solely through configuration, and *(ii)* the capacity to generate very high loads efficiently in order to push the model under investigation to its limits, and reveal behaviour of interest. We focus on system models that have a *master-slave* architecture, where one central process called the *master*, creates and allocates tasks to *slave* processes [32]. Slaves work independently on the task allocated, and relay back the result to the master when ready; the master then combines the different results to produce the final output. Master-slave architectures are widely employed in areas such as DNS and IoT, and can also be found in local concurrency settings like thread pools and web servers [32,17].

3.1 Approach

To render our ensuing empirical evaluation of sec. 4 more manageable, we scope our implementation to a *local* concurrency setting. This still retains the crucial aspects of a distributed set-up, namely that 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

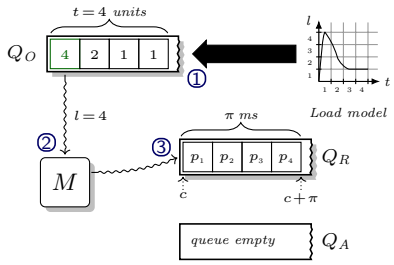
not fail-stop or exhibit Byzantine failures; this extension will be added in the future release of the tool.

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.

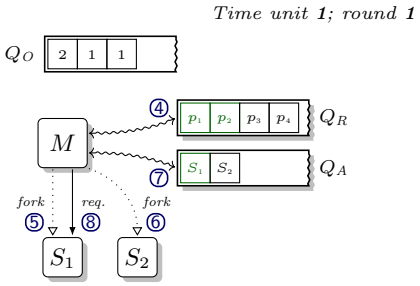
Load models. Our system considers *three load shapes* (see fig. 4) that establish how the creation of slave processes is distributed along the load timeline t. The timeline is modelled as a sequence of *discrete logical time units* that represent instants at which a new set of slaves is created by the 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 $t = \lceil n/\lambda \rceil$. *Pulses* 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 instant 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. We remark that in realistic scenarios, a given system may possibly experience combinations of these types of loads during its lifetime.

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 simultaneously. 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.

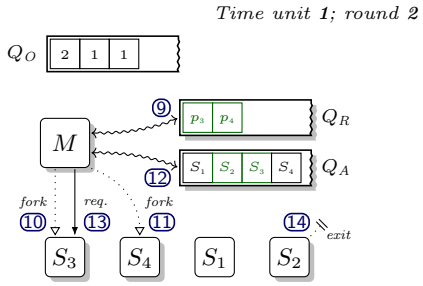
Slave scheduling. The master process employs a scheduling scheme to distribute the creation of slaves uniformly across the time period π. It makes use of three



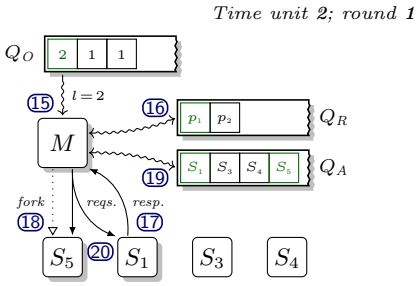
(a) Master schedules the first batch of four slaves for execution in Q_{R}



(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

queues: the *Order* queue, *Ready* queue, and *Await* queue, which we denote by Q_{O}, Q_{R}, and Q_{A} respectively, shown in fig. 1. Q_{O} is initially populated with the load model (*e.g.* burst), 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 that is to be created at that instant. Slaves are then scheduled and created in *rounds*, as follows. The master picks the first element from Q_{O} to 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 is *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 by the master; 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.

[4] We assume that the platform scheduling the master and slave processes is *fair* and does not starve the master. Starving the master would derange our scheduling scheme since it uses wall-clock time to determine when new slaves get created.

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 can be handled once the scheduling and work allocation actions are complete. A fresh batch of slaves from Q_{O} is scheduled by the master whenever Q_{R} becomes empty (fig. 1d), and the whole procedure is repeated as described. Scheduling stops when all the entries in Q_{O} have been dequeued. 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. This occurs due to a number of factors, such as 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 slave processes: $\Pr(send)$ and $\Pr(recv)$ bias this behaviour. Concretely, $\Pr(send)$ 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 condition, steps ⑧, ⑬ and ⑳ in fig. 1, and the entry for S_i in Q_{A} is updated accordingly. 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 work 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 work queue until $X > \Pr(recv)$. Slaves are instructed to terminate by the master process 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.

3.2 Realisability

We implemented our proposed driver system using Erlang [2,10]. Erlang adopts the actor model of computation [1], a message-passing paradigm where programs are structured in terms of *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 received. Each actor owns a message queue, called a *mailbox*, where messages can be deposited

by other actors, and consumed by the recipient at any stage. Messages from the mailbox can be taken *out-of-order*, and processed atomically by actors. Besides sending and receiving messages, an actor can also *fork* other actors to execute independently in their own private process space. Actors can be uniquely addressed via an *identifier* that is assigned to them upon forking. Erlang implements actors as *lightweight* processes that are efficiently forked and terminated, to enable massively-scalable system architectures that can span multiple machines. The terms *actor* and *process* are used interchangeably henceforth.

Implementation. We map the master and slave processes from sec. 3.1 to Erlang actors. The respective incoming work request queues for these processes conveniently coincide with actor mailboxes. This facilitates our implementation, since no dedicated *listener process* is required to accept incoming messages (for non-blocking communication) when the master is busy scheduling slaves and allocating work tasks. We abstract the computation involved in tasks, and simply model work requests as Erlang messages. Slaves are programmed not to emulate delay, but to respond instantly to work requests; delay in the system can be induced using parameters Pr(*send*) and Pr(*recv*) mentioned above. To maximise efficiency, the Order, Ready and Await queues used by our scheduling scheme are maintained *locally* within the master, rather than as independent processes without. The master process keeps track of other details, such as the total number of work requests sent and received, to determine the point at which the driver system should stop executing. Our implementation extends the parameters mentioned in sec. 3.1 with a *seed* parameter, r, to make it possible to fix the Erlang pseudorandom number generator to output consistent number sequences.

3.3 Measurement Collection

Our set-up collects three performance metrics: *(i)* scheduler utilisation, as a percentage of the total available capacity, *(ii)* memory consumption, measured in GB, and, *(iii)* mean response time (MRT), measured in milliseconds (ms). Measurement taking greatly depends on the platform on which the driver system executes: for instance, one often leverages *platform-specific* optimised functionality in order to attain high levels of efficiency. We here describe our implementation that relies on 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 driver system and its 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 [29]. Our sampling frequency is set to 500 ms[5]: this figure was determined empirically, whereby measurements gathered are neither too coarse, nor excessively fine-grained such that sampling affects the runtime. Every sampling snapshot combines the three metrics mentioned above and formats them as records that are written *asynchronously* to disk to reduce IO delays.

[5] Incidentally, this agrees also with the default sampling interval used by the Erlang Observer, a graphical tool for observing the characteristics of Erlang systems [10].

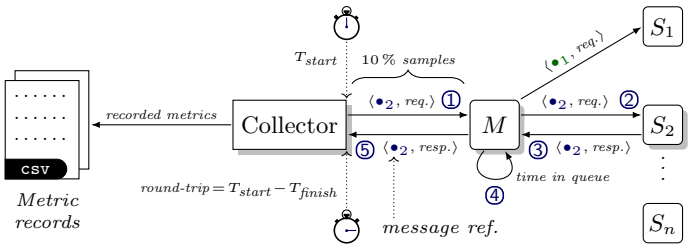


Fig. 2: Collector tracking the round-trip time for work requests and responses

Performance metrics. Memory and scheduler readings within our sampling window are obtained 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; this would inflate the metric reading[6]. Our notion of overall system responsiveness is captured by the MRT 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 MRT, the collector samples 10 % of the total number of messages exchanged between the master and slaves, step ⑤, and calculates the arithmetic mean using Welford's [35,21] online algorithm.

Validation. A series of trials were conducted to select an appropriate sampling window size for our MRT measurements. We found this step to be crucial, since it directly affects the capability of the driver system to scale in terms of its number of slave processes and work requests. Our MRT algorithm was validated by taking various sampling window sizes over numerous runs set up with different load models of ≈1 M slaves. The results were compared to the actual arithmetic mean calculated on *all* work request and response messages exchanged between master and slaves. Values close to 10 % yielded the best outcomes (≈ ±1.4 % drift from the expected MRT). Smaller window sizes produced excessive drift, while larger window sizes induced noticeably higher system loads. We also cross-checked the scheduler utilisation sampling procedure which we implemented from scratch against readings obtained via the Erlang Observer tool.

4 Evaluation

The parameters described in sec. 3 can be configured to model a range of driver systems. Not all of these configurations make sense in practice, however. For ex-

[6] 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.

ample, setting $\Pr(send) = 0$ rarely enables the master to allocate work requests to slaves, whereas with $\Pr(send) = 1$, this allocation is done sequentially, defeating the purpose of a master-slave set-up. Similarly, fixing the number of slaves, to $n = 1$ hardly emulates useful behaviour. The aim of this section is twofold. We first establish a set of parameter values that model experiment set-ups whose behaviour *approximates* that of systems typically found in practice, sec. 4.1. In particular, we limit our study to instantiations of the master-slave architecture that model *web server traffic*. We then use this set-up to evaluate a runtime monitoring tool prototype at *high loads*, sec. 4.2.

Experiment set-up. We define an *experiment* to consist of ten benchmarks, each performed by running the system set-up with incremental loads, starting at $n = 50\,\text{k}$ and progressing to $n = 500\,\text{k}$ in steps of $50\,\text{k}$. All 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.

Experiment repeatability. Data variability affects the *reproducibility* of measurements collected quantitatively. It also plays a role when one wants to determine the number of repeated readings, i, that need to be taken before the data measured is deemed sufficiently *representative*. Choosing the lowest i is crucial when single experiment runs are time consuming, as this expedites measurement taking. The *coefficient of variation* (CV), *i.e.,* the ratio of the standard deviation to the mean, $\text{CV} = \frac{\sigma}{\overline{x}} \times 100$, is often used to establish the value of i empirically, as follows. Initially, the CV for one batch of experiments repeated for some i is calculated, and the result is compared to the CV for the next batch with $i' = i + b$, where b is the step size. When the difference between successive CV metrics is sufficiently small (for some percentage ϵ), the value of i is chosen, otherwise the procedure is repeated for the next i. Crucially, this condition must hold for *all* variables measured in the experiment before i can be fixed.

4.1 Choosing the System Model Parameters

Experiments for this section are set up at $n = 500\,\text{k}$ slaves and $w = 100$ work requests per slave, generating $\approx n \times w \times 2 = 100\,\text{M}$ messages each run. 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 load testing tools such as Tsung [26] and Gatling [15]. The total loading time is set to $t = 100\,\text{s}$.

Data variability. We show that the data variability between experiments can be reduced by seeding the pseudorandom number generator (parameter r in sec. 3.2) with a constant value. This, in turn, tends to require less repeated runs before the metrics of interest—scheduler utilisation, memory consumption, and MRT—converge to an acceptable CV. We conduct benchmarks on experiments set with three, six and nine repetitions. For the majority of cases, the CV for the three data variables considered is *lower* when a constant seed is set, as opposed to its unseeded counterpart (refer to fig. 7 in app. A). In fact, very low CV values

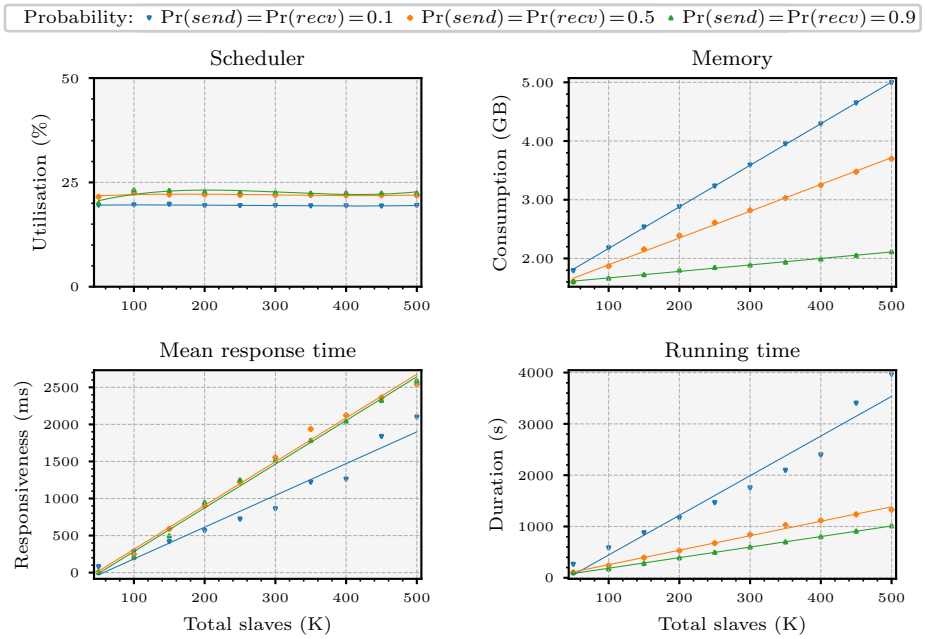


Fig. 3: Performance benchmarks of system models for Pr(*send*) and Pr(*recv*)

for the scheduler utilisation, memory consumption, and MRT of 0.17 %, 0.15 %, 0.52 % respectively were obtained when the experiment was performed with three repeated runs. Consequently, we fix the number of repetitions to *three* for all experiment runs discussed in the sequel. Fixing the seed still allows the system to exhibit a modicum of variability: this stems from the inherent interleaved execution of asynchronous components as a result of process scheduling.

System reactivity. Pr(*send*) and Pr(*recv*) control the speed with which the system reacts to load. We study how these parameters affect the overall performance of system models set up with $\Pr(send) = \Pr(recv) = 0.1, 0.5, 0.9$. The results are shown in fig. 3, where each performance metric under consideration (*e.g.* memory consumption) is plotted against the total number of slaves; the full experiment running time is also included. At $\Pr(send) = \Pr(recv) = 0.1$, the system has the lowest MRT out of the three configurations, as indicated by the gentle linear increase of the respective plot. One may expect the MRT to be *lower* for the system models configured with probability values of 0.5 and 0.9. However, we recall that with $\Pr(send) = 0.1$, work requests are allocated infrequently by the master, such that slave processes remain *idle often*, and in a state where they can *readily* respond to (low numbers of) incoming work requests. The slow rate with which the master allocates work requests prolongs the overall running time, when compared to that of the system for $\Pr(send) = \Pr(recv) = 0.5, 0.9$; meanwhile, the effect of idling can be gleaned from the relatively low scheduler util-

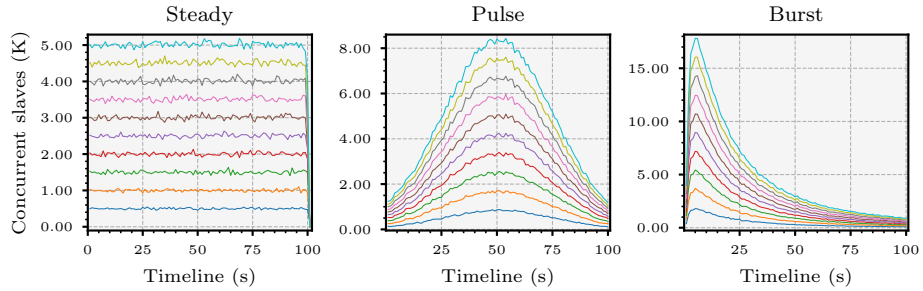


Fig. 4: Steady, pulse and burst load distributions of 500 k slaves in 100 s

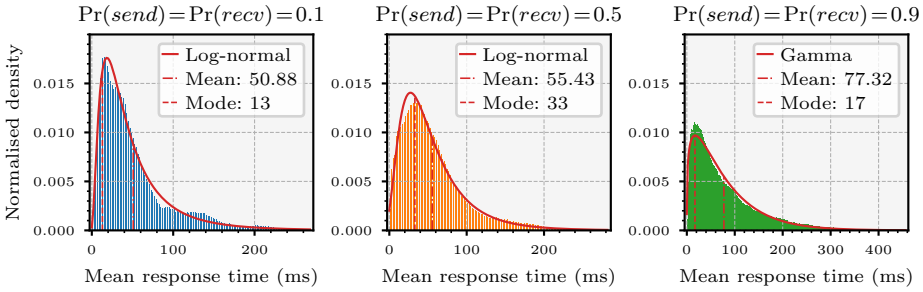


Fig. 5: Fitted probability distributions on MRT for steady loads for $n = 10\,\text{k}$

isation. Idling also increases the consumption of memory, since slave processes created by the master will typically remain alive for extended periods. By contrast, the plots of the system models with $\Pr(send) = \Pr(recv) = 0.5, 0.9$ exhibit markedly lower gradients in the memory consumption and running time charts, and slightly steeper and analogous slopes in the MRT chart. This indicates that probability values between 0.5 and 0.9 yield system models that: *(i)* consume reasonable amounts of memory, *(ii)* are able to execute in respectable amounts of time, and, *(iii)* maintain tolerable MRTs. Since the master-slave architecture is typically employed in settings where high throughput is demanded, choosing values that are less than 0.5 contradicts this notion for the reasons mentioned above. In what follows, we opt for $\Pr(send) = \Pr(recv) = 0.9$ to account for the delays in processing that are bound to arise in a practical master-slave set-up.

Load shapes. The load shapes presented in sec. 3.1 are designed to induce various performance overheads on the system model (see fig. 4). These make it possible to mock specific system scenarios to test different implementation aspects. For example, a test that subjects the system to load surges could uncover buffer overflows that arise when the length of the request queue exceeds some pre-set length. We observed relatively close levels of overhead for steady and pulse loads, whereas bursts yielded the highest overhead. Refer to app. A for the full details.

MRT distribution. Our driver system 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 [18,22,20]. The probability distribution of the MRT of web application requests is generally right-skewed, and can be approximated to a log-normal [18,13] or an Erlang (a special case of a gamma) distribution [20]. We conduct three experiments using *steady loads* fixed with $n = 10\,\text{k}$; $\Pr(send) = \Pr(recv)$ are varied through 0.1, 0.5 and 0.9 to establish whether the MRT in our system set-ups resembles the aforementioned probability distributions. Our results, summarised in fig. 5, were obtained as follows. The parameters for a set of candidate probability distributions (*e.g.* normal, log-normal, gamma, *etc.*) were estimated using Maximum Likelihood Estimation (MLE) [28] on the MRT obtained from *each* experiment. We then performed goodness-of-fit tests on these parametrised distributions, selecting the most appropriate MRT fit for each of the three experiments. Our goodness-of-fit measure was derived using the Kolmogorov-Smirnov test. The fitted distributions in fig. 5 indicate that the MRT of our chosen system models follows the findings reported in [18,13,20], which show that web response times follow log-normal or Erlang distributions.

4.2 Case Study

We employ our driver system to study the behaviour of a generic runtime monitoring tool prototype that measures the mean idle time (MIT) on system components. As discussed in sec. 1, this runtime monitoring tool can be regarded as a software layer that *extends the core system* with new functionality. Our aim is to understand the performance overheads induced by this tool prototype, as well as the MIT sustained by the system when the set-up is subjected to gradually increasing loads. To this end, the driver system is configured with $n = 20\,\text{k}$ for low loads, $n = 50\,\text{k}$ for moderate loads, and $n = 500\,\text{k}$ for high loads; $\Pr(send) = \Pr(recv)$ is fixed at 0.9, as established in sec. 4.1. We seed the pseudo-random number generator with a constant value, and perform three repetitions of the same experiment under the load shapes shown in fig. 4. A loading time of $t = 100\,\text{s}$ is used in each case.

Runtime monitoring tool. The runtime monitoring tool considered instruments the target system via *code injection* by manipulating the program abstract syntax tree. Embedded code instructions in the form of a monitor analyse the execution of the instrumented component to keep track of the time (measured in ms) the component does not spend handling messages. Monitors execute in the *same process space* of the components to induce the lowest possible amounts of runtime overheads. Each monitor collects the MIT *locally* at the instrumented system component, *independent* of other monitors, and relays the metric to a central coordinating monitor. This, in turn, aggregates the various results using an extended version of Welford's online algorithm that accounts for unequal weights [36]. This is crucial, since each monitor may calculate its local MIT over a *different number* of messages than other monitors. The *global* MIT is produced by the coordinating monitor once the system completes its execution.

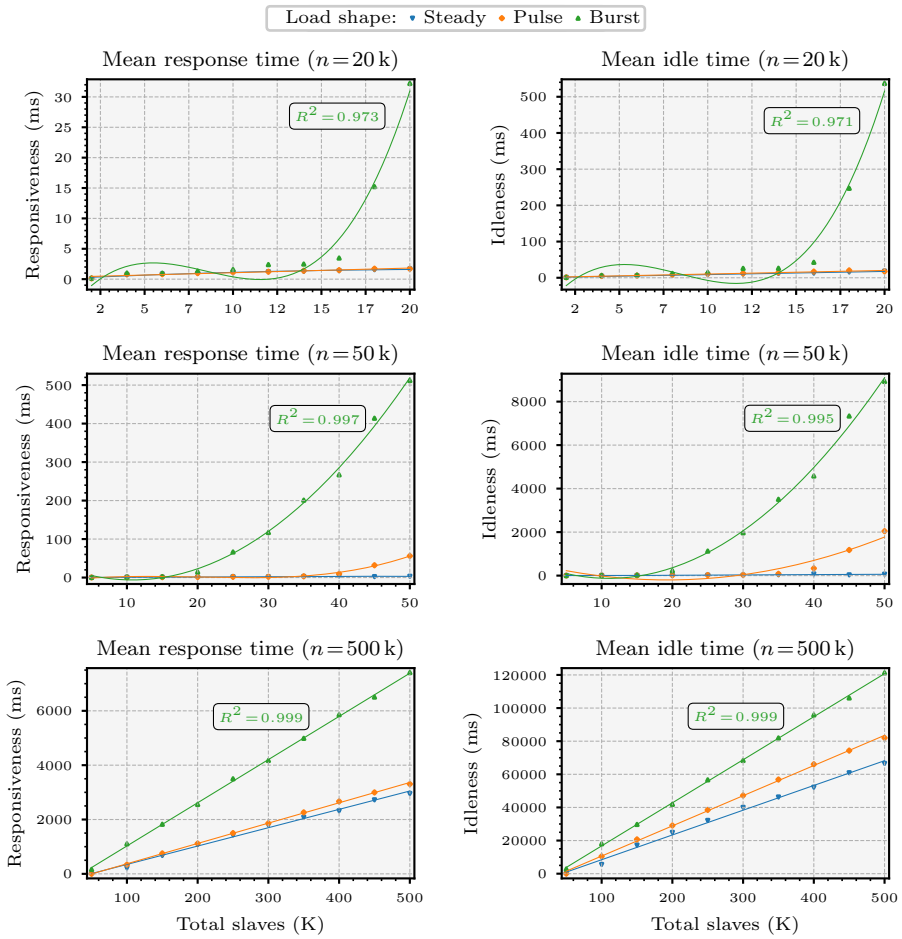


Fig. 6: MRT and MIT for system with $n=20, 50, 500\,\mathrm{k}$ under the loads in fig. 4

Discussion. The results of our experiments in fig. 6 show that certain software behaviour *emerges only* when the system is subjected to high loads. Each plot is fitted with polynomials for a minimum R^2 value of 0.9. Rather than including the performance metrics discussed in sec. 4.1, we focus on the MRT, as this is typically a manifestation of the others, *e.g.* memory consumption. Refer to fig. 8 in app. A. The MRT is also relevant in practice since it is synonymous with the *usability* of the system. Fig. 6 demonstrates that the monitored system tends to exhibit relatively similar behaviour when subjected to steady and pulse loads that are low or high ($n = 20\,\mathrm{k}$ and $n = 500\,\mathrm{k}$). With a moderate load ($n = 50\,\mathrm{k}$), both MRT and MIT plots for the pulse load start to diverge from that of the steady load at the $35\,\mathrm{k}$ and $30\,\mathrm{k}$ marks respectively. The absence of these divergences at low and high loads suggests that the plots in question

actually exhibit linear behaviour. This pattern is especially evident in the burst load plots: these follow a cubic trend when the load is low ($n = 20\,\text{k}$), and a quadratic trend when the load is moderate ($n = 50\,\text{k}$). Once again, the same plots exhibit linear gradients when at a high load ($n = 500\,\text{k}$). Such occurrences greatly indicate that for our experiment set-up, the MRT and MIT of the monitored system has a linear growth pattern for *all* load shapes: crucially, this behaviour emerges only when the total number of slaves is high. The empirical evidence for low to moderate loads may lead us to *wrongly* assert that the tool induces far greater overheads, and that it will become mostly idle as the number of slaves increases. Inverse scenarios may also arise: a different tool under investigation could be declared efficient, only to discover that it then fails to scale or perform as anticipated the moment high loads are attained.

5 Conclusion

We presented a framework that can be used to systematically evaluate the non-functional attributes. Our set-up can emulate various system models and induce different loads at high levels to reveal behaviour that may emerge when software is pushed to its limit. We validate a set of particular parameter values, and show how these allow us to generate system models that approximate certain realistic behaviour; we use these to evaluate a prototype runtime monitoring tool.

Future work. We plan to cater for scenarios where failure arises due to unreliable communication or process crashes by injecting failures based on configurable probability values, as done in tools such as Chaos Monkey [25]. The task model used by the master and slaves can be enhanced to emulate more realistic scenarios such as embarrassingly parallel work loads. We also intend to transition to a distributed set-up where the master and slaves reside on different nodes, and extend our implementation to one that supports peer-to-peer architectures.

Related work. The authors in [9] conduct in-vivo testing of Android apps for open systems within highly-scalable environments. In contrast to our framework, their testing approach has to account for the added complications of deploying apps on various devices; we take an abstract view of the distributed set-up model instead. We also focus on non-functional aspects of software, whereas [9] considers the functional ones only. The independent and ongoing line of work on Angainor [23] concentrates on setting up reproducible experiments for distributed systems. They also insist on systematic evaluation, attaining repeatability via the use of configuration to fix the experiment parameters.

In [22], 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 load based on a Poisson process. A study of message-passing communication on parallel computers is conducted in [18] that uses systems loaded with different numbers of processes, similar to our approach. We were able to confirm the findings presented in both [22] and [18] in our results (see sec. 4.1).

References

1. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A Foundation for Actor Computation. JFP **7**(1), 1–72 (1997)
2. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
3. Attard, D.P., Francalanza, A.: Trace Partitioning and Local Monitoring for Asynchronous Components. In: SEFM. LNCS, vol. 10469, pp. 219–235. Springer (2017)
4. Bauer, A., Falcone, Y.: Decentralised LTL Monitoring. FMSD **48**(1-2), 46–93 (2016)
5. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Runtime Verification with Minimal Intrusion through Parallelism. FMSD **46**(3), 317–348 (2015)
6. 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. ACM (2006)
7. Cassar, I., Francalanza, A., Aceto, L., Ingólfsdóttir, A.: eAOP: An Aspect Oriented Programming Framework for Erlang. In: Erlang Workshop. pp. 20–30. ACM (2017)
8. Castro, P.C., Ishakian, V., Muthusamy, V., Slominski, A.: The Rise of Serverless Computing. Commun. ACM **62**(12), 44–54 (2019)
9. Ceccato, M., Gazzola, L., Kifetew, F.M., Mariani, L., Orrù, M., Tonella, P.: Toward In-Vivo Testing of Mobile Applications. In: ISSRE Workshops. pp. 137–143. IEEE (2019)
10. Cesarini, F., Thompson, S.: Erlang Programming: A Concurrent Approach to Software Development. O'Reilly Media (2009)
11. Chappell, D.: Enterprise Service Bus: Theory in Practice. O'Reilly Media (2004)
12. Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: TACAS. LNCS, vol. 5505, pp. 246–261. Springer (2009)
13. Ciemiewicz, D.M.: What Do You mean? - Revisiting Statistics for Web Response Time Measurements. In: CMG. pp. 385–396. Computer Measurement Group (2001)
14. Colombo, C., Pace, G.J., Schneider, G.: Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In: FMICS. LNCS, vol. 5596, pp. 135–149. Springer (2008)
15. Corp., G.: Gatling (2020), https://gatling.io
16. El-Hokayem, A., Falcone, Y.: Monitoring Decentralized Specifications. In: ISSTA. pp. 125–135. ACM (2017)
17. Ghosh, S.: Distributed Systems: An Algorithmic Approach. Chapman and Hall/CRC (2014)
18. 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. Springer (2005)
19. Harman, M., O'Hearn, P.W.: From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In: SCAM. pp. 1–23. IEEE Computer Society (2018)
20. Kayser, B.: What is the expected distribution of website response times? (2017), https://blog.newrelic.com/engineering/expected-distributions-website-response-times
21. Knuth, D.E.: Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley Professional (1997)

22. Liu, Z., Niclausse, N., Jalpa-Villanueva, C.: Traffic Model and Performance Evaluation of Web Servers. Perform. Evaluation **46**(2-3), 77–100 (2001)
23. Matos, M.: Towards Reproducible Evaluation of Large-Scale Distributed Systems. In: ApPLIEDPODC. pp. 5–7. ACM (2018)
24. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley (2011)
25. Netflix: Chaos monkey (2020), https://github.com/Netflix/chaosmonkey
26. Niclausse, N.: Tsung (2017), http://tsung.erlang-projects.org
27. Rafaels, R.J.: Cloud Computing: From Beginning to End. CreateSpace Independent Publishing Platform (2015)
28. Rossi, R.J.: Mathematical Statistics: An Introduction to Likelihood Based Inference. Wiley (2018)
29. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google, Inc. (2010)
30. Swords, C., Sabry, A., Tobin-Hochstadt, S.: Expressing Contract Monitors as Patterns of Communication. In: ICFP. pp. 387–399. ACM (2015)
31. Systems, C.: Packet tracer (2020), https://www.netacad.com/courses/packet-tracer/introduction-packet-tracer
32. Tarkoma, S.: Overlay Networks: Toward Information Networking. Auerbach Publications (2010)
33. TetCos: Netsim (2020), https://www.tetcos.com/netsim-acad.html
34. Vilk, J., Berger, E.D.: BLeak: Automatically Debugging Memory Leaks in Web Applications. In: PLDI. pp. 15–29. ACM (2018)
35. Welford, B.P.: Note on a Method for Calculating Corrected Sums of Squares and Products. Technometrics **4**(3), 419–420 (1962). https://doi.org/10.1080/00401706.1962.10490022
36. West, D.H.D.: Updating Mean and Variance Estimates: An Improved Method. CACM **22**(9), 532–535 (1979)

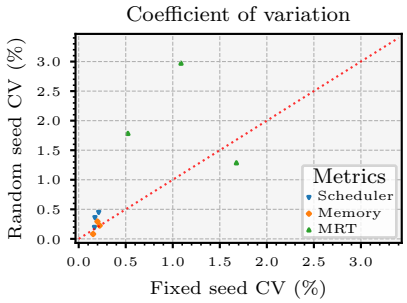


Fig. 7: CV for unfixed and fixed randomisation seeds for $3, 6, 9$ repetitions

A Supporting Evaluation

Further to the model system parameters discussed in sec. 4.1, the following supporting empirical measurements were also taken.

Data variability. Fig. 7 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. 4. For our three chosen performance metrics, using a constant seed tends to induce less variability in the experiments, *i.e.,* a low CV, as indicated in fig. 7 where only two points lie below the identity line $y = x$.

Load shapes. The load shapes presented in sec. 3.1 induce different performance overheads on the system. Fig. 4 shows the plots for the performance metrics considered in our experiments under the loads in fig. 4, where $n = 500\,\text{k}$. 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 MRT, and share similar running time 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 MRT and memory consumption relative to steady loads are ≈ 2.8 and ≈ 1.9 respectively. This load-inducing behaviour did not emerge for running times, and the plot for burst is analogous to the one produced the steady load.

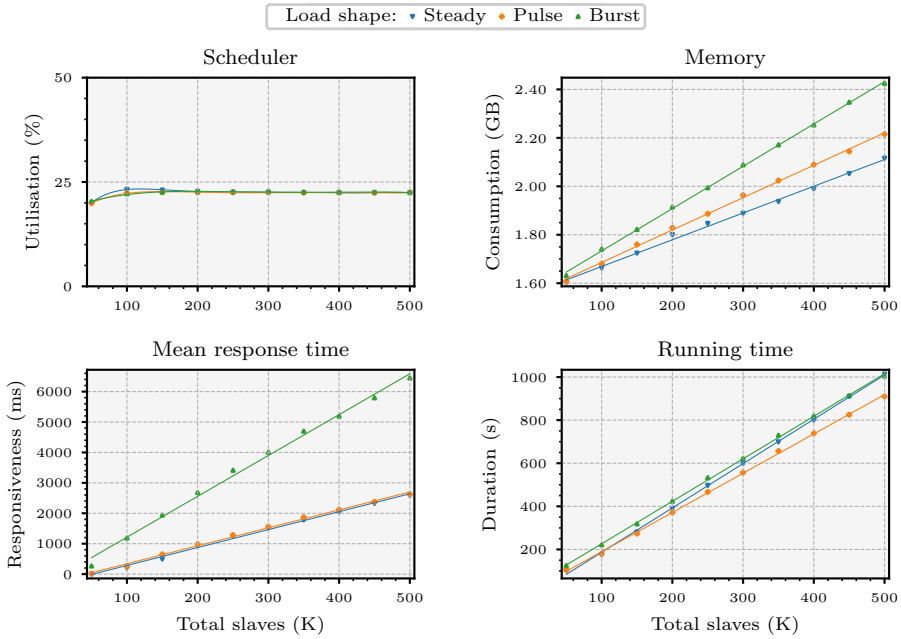


Fig. 8: Performance benchmarks of system models under the loads in fig. 4