Accelerated Runtime Verification of LTL Specifications with Counting Semantics

Ramy Medhat[2], Borzoo Bonakdarpour[1], Sebastian Fischmeister[2], and Yogi Joshi[2]

[1] McMaster University, Canada,
[2] University of Waterloo, Canada

Abstract. This paper presents a novel and efficient parallel algorithm for runtime verification of an extension of LTL that allows for nested quantifiers subject to numerical constraints. Such constraints are useful in evaluating thresholds (e.g., expected uptime of a web server). Our algorithm uses the *MapReduce* programming model to split a program trace into variable-based clusters at run time. Each cluster is then mapped to its respective monitor instances, verified, and reduced collectively on a multi-core CPU or the GPU. Our experiments on real-world case studies show that the algorithm imposes negligible monitoring overhead.

1 Introduction

Runtime verification (RV) is an automated specification-based technique, where a *monitor* evaluates the correctness of a set of logical properties on a particular execution. RV complements exhaustive approaches such as model checking and theorem proving and under-approximated methods such as testing. RV can be particularly helpful in scenarios, where one needs to monitor parametric requirements on types of execution entities (e.g., processes and threads), user- and kernel-level events and objects (e.g., locks, files, sockets), web services (e.g., requests and responses), and network traffic. For example, the requirement 'every open file should eventually be closed' specifies a rule for causal and temporal order of opening and closing individual objects which generalizes to *all* files. Such properties can become even more complex by incorporating numerical constraints such as thresholds, floors, ceilings. However, to our knowledge existing RV frameworks fall short in expressing counting semantics.

In this paper, we extend the 4-valued semantics of LTL (denoted RV-LTL in this paper) [6] by adding counting semantics with numerical constraints and propose an efficient parallel algorithm for their verification at run time. Inspired by the work in [15], the syntax of our language (denoted $\text{LTL}_4\text{−C}$) extends LTL syntax by the addition of *counting quantifiers*. That is, we introduce two quantifiers: the *instance counting quantifier* ($\mathbb{E}$) which allows expressing properties that reason about the number of satisfied or violated instances, and the *percentage counting quantifier* ($\mathbb{A}$) which allows reasoning about the percentage of satisfied or violated instances out of all instances in a trace. These quantifiers are subscripted with numerical constraints to express the conditions used to evaluate the count. For example, the following LTL$_4$−C formula:

$$\mathbb{A}_{\geq 0.95}\, s : \mathsf{socket}(s) \cdot (\mathbf{G}\,\mathsf{receive}\,(s) \Rightarrow \mathbf{F}\,\mathsf{respond}\,(s))$$

intends to express the property that 'at least 95% of TCP/UDP sockets must eventually respond to a received request'. For a web admin, ideally the number of dropped requests is zero, however in reality requests will be dropped sometimes [19]. Thus it is beneficial for a monitor to keep track of the percentage of dropped requests and fire an alert once a certain threshold is exceeded.

The first contribution of the paper is extending RV-LTL by redefining presumably true/false within the context of counting semantics. Consider the example demonstrated above, where it is required that a web server drops less than 5% of the requests. For such a property, counting semantics justify the need for presumably true/false in a similar fashion to incomplete executions in RV-LTL. For instance, if only 4% of the requests have been dropped so far, that does not mean that the property is permanently satisfied. There could be more requests that arrive in the future and are dropped, increasing the percentage of dropped requests beyond 5%. A verdict of true is incorrect, since the property can be violated in the future. In LTL$_4$−C, this property is presumably satisfied, with a potential to be violated by more requests. For a web admin, this verdict indicates that the system is currently healthy.

The second contribution of this paper is a divide-and-conquer-based online monitor generation technique for LTL$_4$−C specifications. Our technique first synthesizes an RV-LTL monitor for the inner LTL formula of the given LTL$_4$−C formula at pre-compile time using the technique in [6]. Then, based upon the values of variables observed at run time, submonitors are generated and merged to compute the current truth value of a property for the current program trace.

Our third contribution is a monitoring algorithm that implements the above approach for verification of LTL$_4$−C properties at run time. The monitoring algorithm evaluates properties in parallel, utilizing multicore CPUs or GPUs and maximizing the throughput of the monitor. The algorithm utilizes the popular *MapReduce* programming model to (1) spawn submonitors that aim at evaluating subformulas using partial quantifier elimination, and (2) merge partial evaluations to compute the current truth value of properties.

Our parallel algorithm for verification of LTL$_4$−C properties is fully implemented on multi-core CPU and GPU technologies using our own simple implementation of the MapReduce programming model. We report experimental results by conducting three real-world independent case studies. The first case study is a monitor for HTTP requests and responses on an Apache Web Server. The second case study is a monitor for upload chunk size based on a dataset for profiling DropBox traffic. The third case study monitors a network proxy cache to reduce the bandwidth usage of online video services, based on a YouTube request dataset. We present performance results comparing single-core CPU, multi-core CPU, and GPU implementations. Our results show that our GPU-based implementation provides an average speed up of 6.3x when compared to single-core

CPU, and 1.75x when compared to multi-core CPU. The CPU utilization of the GPU-based implementation is negligible compared to multi-core CPU, freeing up the system to perform more computation. Thus, the GPU-based implementation manages to provide competitive speedup while maintaining a low CPU utilization, which are two goals that the CPU cannot achieve at the same time. Put it another way, the GPU-based implementation incurs minimal monitoring costs while maintaining a high throughput.

2 LTL with counting semantics

Let IP be a finite set of interpreted predicates, and let $\Sigma = 2^{IP}$ be the power set of IP. We call each element of Σ an *event*.

Definition 1 (Trace). *A* trace $w = w_0 w_1 \cdots$ *is a finite or infinite sequence of events where each event consists of interpreted predicates; i.e,* $w_i \in \Sigma$*, for all* $i \geq 0$. ■

We denote the set of all infinite traces by Σ^ω and the set of all finite traces by Σ^*.

2.1 Syntax of LTL4-C

LTL$_4$−C extends RV-LTL [6] (also known as RV-LTL) with two counting quantifiers: the instance counting quantifier ($\mathbb{E}$) and the percentage counting quantifier ($\mathbb{A}$). The semantics of these quantifiers are introduced in subsection 2.4. The syntax of LTL$_4$−C is defined as follows:

Definition 2 (LTL$_4$−C Syntax). LTL$_4$−C *formulas are defined using the following grammar:*

$$\varphi ::= \mathbb{A}_{\sim k}\, x : p(x) \cdot \varphi \;\mid\; \mathbb{E}_{\sim l}\, x : p(x) \cdot \varphi \;\mid\; \psi$$
$$\psi ::= \top \;\mid\; a \;\mid\; p\,(x_1 \cdots x_n) \;\mid\; \neg\psi \;\mid \psi_1 \wedge \psi_2 \;\mid\; \mathbf{X}\,\psi \;\mid\; \psi_1\,\mathbf{U}\,\psi_2$$

where $\mathbb{A}$ *is the percentage counting quantifier,* $\mathbb{E}$ *is the instance counting quantifier,* x*,* $x_1 \cdots x_n$ *are variables with finite domains* $\mathcal{D}, \mathcal{D}_1, \cdots \mathcal{D}_n$*,* $\sim \in \{<, \leq, >, \geq, =\}$*,* $k : \mathbb{R} \in [0,1]$*,* $l \in \mathbb{Z}^+$*, a is an atomic proposition,* $\mathbf{X}$ *is the next, and* $\mathbf{U}$ *is the until temporal operators.* ■

If we omit the numerical constraint in $\mathbb{A}_{\sim k}$ (respectively, $\mathbb{E}_{\sim l}$), we mean $\mathbb{A}_{=1}$ (respectively, $\mathbb{E}_{\geq 1}$). The syntax of LTL$_4$−C forces constructing formulas, where a string of counting quantifiers is followed by a quantifier-free formula. We emphasize that $\mathbb{A}$ and $\mathbb{E}$ do not necessarily resemble standard first order quantifiers $\forall$ and $\exists$. In fact, $\neg\mathbb{A}$ and $\mathbb{E}$ are not equivalent.

Consider the LTL$_4$−C property $\varphi = \mathbb{A}x : p(x) \cdot \psi$, where the domain of x is $\mathcal{D}$. This property denotes that for any possible valuation of the variable x ($[x := v]$), if $p(v)$ holds, then ψ should hold. If $p(v)$ does not hold, then $p(v) \cdot \psi$

evaluates to true. This effectively means that the quantifier $\mathbb{A}x$ is in fact applied only over the sub-domain $\{v \in \mathcal{D} \mid p(v)\} \subseteq \mathcal{D}$.

To give an intuition, consider the scenarios where file management anomalies can cause serious problems at run time (e.g., in NASA's Spirit Rover on Mars in 2004). For example, the following LTL$_4$−C property expresses "at least half of the files that a process has previously opened must be closed":

$$\varphi = \mathbb{A}_{\geq 50\%}\, f : \mathsf{inevent}(f) \cdot \mathbf{G}(\mathsf{opened}(f)\, \mathbf{U}\, \mathsf{close}(f)) \tag{1}$$

where inevent is the p predicate of the quantifier, denoting that the concrete file appeared in an event in the trace.

2.2 Truth Values of LTL4-C

The objective of LTL$_4$−C is to verify the correctness of quantified properties at run time with respect to finite program traces. Such verification attempts to produce a sound verdict regardless of future continuations.

Similar to RV-LTL, we incorporate four truth values to define the semantics of LTL$_4$−C: $\mathbb{B}_4 = \{\top, \bot, \top_p, \bot_p\}$; *true*, *false*, *presumably true*, and *presumably false*, respectively. The values in $\mathbb{B}_4$ form a lattice ordered as follows: $\bot < \bot_p < \top_p < \top$. Given a finite trace u and an LTL$_4$−C property φ, the informal description of evaluation of u with respect to φ is as follows:

– **True** ($\top$) denotes that any infinite extension of u satisfies φ. For example, $\varphi_1 = \mathbb{E}_{\geq 1}t : \mathsf{thread}(t) \cdot \mathsf{log}(t)$ is a property that checks a process has at least one log thread. If one log thread is found in the trace, the property is permanently satisfied.
– **False** ($\bot$) denotes that any infinite extension of u violates φ. For example, $\varphi_2 = \mathbb{E}_{=1}t : \mathsf{thread}(t) \cdot \mathsf{log}(t)$ is a property that checks a process has *exactly* one log thread. If more than one log thread is found in the trace, the property is permanently violated.
– **Presumably true** ($\top_p$) extends the definition of *presumably true* in RV-LTL [6], where $\top_p$ denotes that u satisfies the inner LTL property and the counting quantifier constraint in φ, if the program terminates after execution of u. An example is

$$\varphi_3 = \mathbb{E}_{\geq 1}t : \mathsf{thread}(t) \cdot \mathsf{log}(t) \wedge \mathbf{G}(\mathsf{event}(t) \Rightarrow \mathbf{F}\mathsf{write}(t))$$

which evaluates to $\top_p$ if there is only one log thread that has received an event and has written it, but can still potentially receive another event and never write it, thus violating the property.
– **Presumably false** ($\bot_p$) extends the definition of *presumably false* in RV-LTL [6], which denotes that u presumably violates the quantifier constraint in φ. For example, Property φ_3 evaluates to $\bot_p$ if there is one log thread that has received an event and not yet written it. A future extension of the trace can potentially contain a write event, thus transforming the valuation of the property to $\top_p$.

2.3 Valuation in LTL$_4$−C

An LTL$_4$−C property essentially defines a set of traces, where each trace is a sequence of events (i.e., sets of predicates). We define the semantics of LTL$_4$−C with respect to finite traces and present a method of utilizing these semantics for runtime verification. In the context of runtime verification, the objective is to ensure that a trace is in the set of traces that the property defines.

To introduce the semantics of LTL$_4$−C, we examine counting quantifiers further. Since the syntax of LTL$_4$−C allows nesting of counting quantifiers, a canonical form of properties is $\varphi = \mathbb{Q}_\varphi\ \psi$ where ψ is an LTL property and $\mathbb{Q}_\varphi$ is a sequence of counting quantifiers $\mathbb{Q}_\varphi = \mathcal{Q}_0\mathcal{Q}_1 \cdots \mathcal{Q}_{n-1}$ such that each $\mathcal{Q}_i = \langle q_i, \sim_i, c_i, x_i, p_i\rangle$, $0 \leq i \leq n-1$, is a tuple encapsulating the counting quantifier information. That is, $q_i \in \{\mathbb{A}, \mathbb{E}\}$, $\sim_i \in \{<, \leq, >, \geq, =\}$, c_i is the constraint constant, x_i is the bound variable, and p_i is the predicate within the quantifier (see Definition 2).

Variable Valuation We define a vector D_φ with respect to a property φ as $D_\varphi = \langle d_0, d_1, \cdots, d_{n-1}\rangle$ where $n = |\mathbb{Q}_\varphi|$ and d_i, $0 \leq i \leq n-1$, is a value for variable x_i. We denote the first m components of the vector D_φ (i.e., $\langle d_0, d_1, \cdots, d_{m-1}\rangle$) by $D_\varphi|^m$. We refer to D_φ as a *value vector* and to $D_\varphi|^m$ as a *partial value vector*.

A *property instance* $\hat{\varphi}(D_\varphi|^m)$ is obtained by replacing every occurrence of the variables $x_0 \cdots x_{m-1}$ in φ with the values $d_0 \cdots d_{m-1}$, respectively. Thus, $\hat{\varphi}(D_\varphi|^m)$ is free of quantifiers of index less than m, yet remains quantified over variables $x_m \cdots x_{n-1}$. $\hat{\varphi}(D_\varphi)$ denotes replacing all quantified variables with values in D_φ, resulting in an unquantified LTL property. For instance, for the following property $\varphi = \mathbb{A}_{>c_1}\, x : p_x(x) \cdot (\mathbb{A}_{<c_2}\, y : p_y(y) \cdot \mathbf{G}\, q(x, y))$ and value vector $D_\varphi = \langle 1, 2\rangle$ (i.e., the vector of values for variables x and y, respectively), $\hat{\varphi}(D_\varphi)$ will be $\hat{\varphi}(\langle 1, 2\rangle) = p_x(1) \cdot (p_y(2) \cdot \mathbf{G}\, q(1, 2)) = \mathbf{G}\, q(1, 2)$.

We now define the set $\mathbb{D}_{\varphi,u}$ as the set of all value vectors with respect to a property $\varphi = \mathbb{Q}_\varphi\ \psi$ and a trace $u = u_0u_1 \cdots$:

$$\mathbb{D}_{\varphi,u} = \{D_\varphi \mid \exists j \geq 0 : \forall i \in [0, |\mathbb{Q}_\varphi|) : p_i(d_i) \in u_j\} \tag{2}$$

Valuation of Property Instances As per the definition of $\mathbb{D}_{\varphi,u}$, every value vector $D_\varphi = \langle d_0 \cdots d_{n-1}\rangle$ in $\mathbb{D}_{\varphi,u}$ contains values for which the predicates $p_i(d_i)$ hold in some trace event u_j. For simplicity, we denote this as a value vector *in* a trace event u_j. These value vectors can possibly be in multiple and interleaved events in the trace. Thus, we define a trace $u^{D_\varphi} = u_0^{D_\varphi}u_1^{D_\varphi} \cdots$ as a subsequence of the trace u such that the value vector D_φ is in every event:

$$\forall\, j \geq 0 : \forall\, i \in [0, n-1] : p_i(d_i) \in u_j^{D_\varphi}$$

2.4 Semantics of LTL4-C

Definition 3 (LTL$_4$−C **Satisfaction Relation).** *Given an* LTL$_4$−C *property $\varphi = \mathcal{Q}\, \psi$ where $\mathcal{Q}$ is a quantifier (either $\mathbb{A}$ or $\mathbb{E}$), and ψ is an* LTL$_4$−C *formula.*

Also, given an infinite trace w, we define the satisfaction relation $w \models_4 \varphi$ as follows:

$$\begin{aligned}
w \models_4 \psi \quad & \textit{iff } w \models \psi \textit{ and } \psi \textit{ is an } \textsc{Ltl} \textit{ property} \\
w \models_4 \mathbb{E}_{\sim c} x : p_x(x) \cdot \psi \quad & \textit{iff } \exists \mathbb{D}'_{\varphi,w} \subset \mathbb{D}_{\varphi,w} \textit{ s.t. } \forall D_\varphi \in \mathbb{D}'_{\varphi,w} : w^{D_\varphi} \models \hat{\varphi}(D_\varphi) \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,w} : w^{D_\varphi} \not\models \hat{\varphi}(D_\varphi) \wedge |\mathbb{D}'_{\varphi,w}| \sim c \\
w \models_4 \mathbb{A}_{\sim c} x : p_x(x) \cdot \psi \quad & \textit{iff } \exists \mathbb{D}'_{\varphi,w} \subset \mathbb{D}_{\varphi,w} \textit{ s.t. } \forall D_\varphi \in \mathbb{D}'_{\varphi,w} : w^{D_\varphi} \models \hat{\varphi}(D_\varphi) \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,w} : w^{D_\varphi} \not\models \hat{\varphi}(D_\varphi) \wedge |\mathbb{D}'_{\varphi,w}|/|\mathbb{D}_{\varphi,w}| \sim c
\end{aligned}$$

where $\mathbb{D}_{\varphi,w}$ is the finite set of all value vectors in the infinite trace w. $\hat{\varphi}(D_\varphi)$ is an Ltl *property and $\models$ is the satisfaction relation as defined in* Ltl *semantics.*

Definition 4 (Ltl$_4$−C Semantics for finite prefixes). *Given an* Ltl$_4$−C *property $\varphi = \mathcal{Q}_{\sim c}\,\psi$ where $\mathcal{Q}$ is a quantifier and ψ is an* Ltl$_4$−C *formula. Also, given a finite prefix u of a trace,* Ltl$_4$−C *semantics are defined as follows:*

$$[u \models_4 \varphi] = \begin{cases}
[u \models_{\text{RV-}\textsc{Ltl}} \varphi] & \textit{iff } \varphi \textit{ is an } \textsc{Ltl} \textit{ property} \\
\top & \textit{iff } \exists \mathbb{D}'_{\varphi,u} \subset \mathbb{D}_{\varphi,u} \textit{ s.t.} \\
& \quad \forall D_\varphi \in \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] = \top \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \neq \top \wedge \\
& \quad |\mathbb{D}'_{\varphi,u}| \sim c \textit{ if } \mathcal{Q} = \mathbb{E} \textit{ else } |\mathbb{D}'_{\varphi,u}|/|\mathbb{D}_{\varphi,u}| \sim c \wedge \\
& \quad \forall v \in \Sigma^\omega : uv \models_4 \varphi \\
\bot & \textit{iff } \exists \mathbb{D}'_{\varphi,u} \subset \mathbb{D}_{\varphi,u} \textit{ s.t.} \\
& \quad \forall D_\varphi \in \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \neq \bot \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] = \bot \wedge \\
& \quad |\mathbb{D}'_{\varphi,u}| \not\sim c \textit{ if } \mathcal{Q} = \mathbb{E} \textit{ else } |\mathbb{D}'_{\varphi,u}|/|\mathbb{D}_{\varphi,u}| \not\sim c \wedge \\
& \quad \forall v \in \Sigma^\omega : uv \not\models_4 \varphi \\
\top_p & \textit{iff } \exists \mathbb{D}'_{\varphi,u} \subset \mathbb{D}_{\varphi,u} \textit{ s.t.} \\
& \quad \forall D_\varphi \in \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \in \{\top, \top_p\} \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \notin \{\top, \top_p\} \wedge \\
& \quad |\mathbb{D}'_{\varphi,u}| \sim c \textit{ if } \mathcal{Q} = \mathbb{E} \textit{ else } |\mathbb{D}'_{\varphi,u}|/|\mathbb{D}_{\varphi,u}| \sim c \wedge \\
& \quad \exists v \in \Sigma^\omega : uv \not\models_4 \varphi \\
\bot_p & \textit{iff } \exists \mathbb{D}'_{\varphi,u} \subset \mathbb{D}_{\varphi,u} \textit{ s.t.} \\
& \quad \forall D_\varphi \in \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \in \{\top, \top_p\} \wedge \\
& \quad \forall D_\varphi \notin \mathbb{D}'_{\varphi,u} : [u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)] \notin \{\top, \top_p\} \wedge \\
& \quad |\mathbb{D}'_{\varphi,u}| \not\sim c \textit{ if } \mathcal{Q} = \mathbb{E} \textit{ else } |\mathbb{D}'_{\varphi,u}|/|\mathbb{D}_{\varphi,u}| \not\sim c \wedge \\
& \quad \exists v \in \Sigma^\omega : uv \models_4 \varphi
\end{cases} \qquad \blacksquare$$

The semantics are defined for five cases:

– If φ is an Ltl property, then we use the four-valued semantics in RV-Ltl.

- Let $\mathbb{D}'_{\varphi,u}$ be a subset that contains all values of the quantified variable that satisfy the inner property. If the cardinality of this subset satisfies the numerical constraint on the quantifier, and no infinite extension of the trace can violate it, the valuation is $\top$.
- Now, let $\mathbb{D}'_{\varphi,u}$ contain all values for which the inner property is not $\bot$, i.e. it could be $\top$, $\top_p$, or $\bot_p$. If the cardinality of this subset violates the numerical constraint on the quantifier, and no infinite extension of the trace can satisfy it, the valuation is $\bot$.
- $\top_p$ is similar to $\top$, except that $\mathbb{D}'_{\varphi,u}$ can include values with which the inner property evaluates to $\top_p$, and there exists an extension to the trace prefix that can violate the quantifier constraint.
- $\bot_p$ is the opposite of $\top_p$, where $\mathbb{D}'_{\varphi,u}$ violates the quantifier constraint, and there exists an extension to the trace prefix that can satisfy the constraint.

Note that LTL$_4$−C semantics are defined recursively from the outermost quantifier. The recursion can be observed in $[u^{D_\varphi} \models_4 \hat{\varphi}(D_\varphi)]$ where D_φ is a value vector $\langle d \rangle$ for the quantified variable in $\mathcal{Q}$, and $\hat{\varphi}(D_\varphi)$ is property φ without quantifier $\mathcal{Q}$. Hence, the semantics recurse with one less quantifier at each step until there are no counting quantifiers and φ is an LTL property, at which case we use RV-LTL semantics. Also note that for a finite prefix of a trace, the semantics of LTL$_4$−C is decidable since the quantification is over a finite set of objects that exist in the trace.

3 Divide-and-Conquer-based Monitoring of LTL4-C

In this section, we describe our technique inspired by divide-and-conquer for evaluating LTL$_4$−C properties at run time. This approach forms the basis of our parallel verification algorithm in Section 4.

Unlike runtime verification of propositional RV-LTL properties, where the structure of a monitor is determined solely based on the property itself, a monitor for an LTL$_4$−C needs to evolve at run time, since the valuation of quantified variables change over time. More specifically, the monitor $\mathcal{M}_\varphi$ for an LTL$_4$−C property $\varphi = \mathbb{Q}_\varphi \psi$ relies on instantiating a *submonitor* for each property instance $\hat{\varphi}$ obtained at run time. We incorporate two type of submonitors: (1) RV-LTL *submonitors* evaluate the inner LTL property ψ. An RV-LTL submonitor instance is denoted as $\mathcal{M}^*_{D_\varphi}$, where D_φ is the unique value vector that binds all quantified variables in the property, leaving only a simple LTL property to be monitored. (2) The second time of submonitors is *quantifier submonitors*, described in Subsection 3.1. In Subsection 3.2, we explain the conditions under which a submonitor is instantiated at run time. Finally, in Subsection 3.3, we elaborate on how submonitors evaluate an LTL$_4$−C property.

3.1 Quantifier Submonitors

Given a finite trace u and an LTL$_4$−C property $\varphi = \mathbb{Q}_\varphi \psi$, a *quantifier submonitor* ($\mathcal{M}^{\mathcal{Q}}$) is a monitor responsible for determining the valuation of a property instance $\hat{\varphi}(D_\varphi|^i)$ with respect to a trace subsequence $u^{D_\varphi|^i}$, if $i < |\mathbb{Q}_\varphi|$.

Definition 5 (Quantifier Submonitor). *Let $\varphi = \mathbb{Q}_\varphi \psi$ be an* LTL$_4$−C *property and $\hat{\varphi}(D_\varphi|^i)$ be a property instance, with $i \in [0, |\mathbb{Q}_\varphi| - 1]$. The* quantifier submonitor *for $\hat{\varphi}(D_\varphi|^i)$ is the tuple $\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^i} = \langle \mathcal{Q}_i, \mathbb{M}_{D_\varphi|^i}, \mathcal{F} \rangle$, where*

- *$\mathcal{Q}_i$ encapsulates the quantifier information (see Subsection 2.4)*
- *$\mathbb{M}_{D_\varphi|^i}$ is the set of child submonitors (submonitors of child property instances) defined as follows:*

$$\mathbb{M}_{D_\varphi|^i} = \begin{cases} \{\mathcal{M}^*_{D'_\varphi} \mid D'_\varphi|^i = D_\varphi|^i\} & \text{if } i = |\mathbb{Q}_\varphi| - 1 \\ \{\mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^{i+1}} \mid D'_\varphi|^i = D_\varphi|^i\} & \text{if } i < |\mathbb{Q}_\varphi| - 1 \end{cases}$$

- *$\mathcal{F}$ is a function that applies the quantifier constraint $\mathcal{Q}_i$ on the truth values of all the child submonitors $\mathbb{M}_{D_\varphi|^i}$.*

Thus, if $i = |\mathbb{Q}_\varphi| - 1$, all child submonitors are RV-LTL *submonitors. Otherwise, they are quantifier submonitors of the respective child property instances.* ■

3.2 Instantiating Submonitors

Let an LTL$_4$−C monitor $\mathcal{M}_\varphi$ for property φ evaluate the property with respect to a finite trace $u = u_0 u_1 \cdots$. Let $D_\varphi = \langle d_0, d_1, \cdots \rangle$ be a value vector and u_0 the first trace event such that $\forall d_i : p_i(d_i) \in u_0$. In this case, the LTL$_4$−C monitor instantiates submonitors for every property instance resulting from that value vector. A value vector of length $|\mathbb{Q}_\varphi|$ results in $|\mathbb{Q}_\varphi|+1$ property instances: one for each quantifier in addition to an RV-LTL inner property. Figure 1 demonstrates the tree structure of submonitors graphically.

3.3 Evaluating LTL4-C Properties

Once the LTL$_4$−C monitor instantiates its submonitors, every submonitor is responsible for updating its truth value. The truth value of an RV-LTL submonitor ($\mathcal{M}^*$) is determined by its automaton. Quantifier submonitors update their truth value by applying the quantifier constraint on their child submonitors and producing a valuation based on LTL$_4$−C semantics.

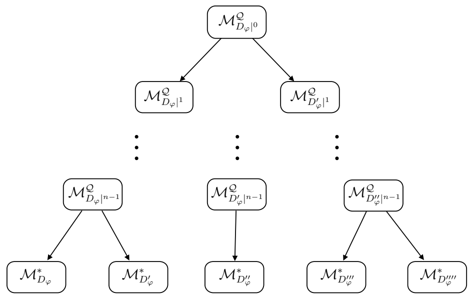

$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^0}$
$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^1}$
$\mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^1}$
$\mathcal{M}^{\mathcal{Q}}_{D_\varphi|^{n-1}}$
$\mathcal{M}^{\mathcal{Q}}_{D'_\varphi|^{n-1}}$
$\mathcal{M}^{\mathcal{Q}}_{D''_\varphi|^{n-1}}$
$\mathcal{M}^*_{D_\varphi}$
$\mathcal{M}^*_{D'_\varphi}$
$\mathcal{M}^*_{D''_\varphi}$
$\mathcal{M}^*_{D'''_\varphi}$
$\mathcal{M}^*_{D''''_\varphi}$

Fig. 1: Tree structure of an LTL$_4$−C monitor.

4 Parallel RV Algorithm

The main challenge in designing a runtime monitor is to ensure that its behavior does not intervene with functional and extra-functional (e.g., timing constraints) behavior of the program under scrutiny. This section presents a parallel algorithm for verification of $\textsc{Ltl}_4{-}\mathrm{C}$ properties. Our idea is that such a parallel algorithm enables us to offload the monitoring tasks into a different computing unit (e.g., the GPU). The algorithm utilizes the *MapReduce* programming model to spawn and merge submonitors to determine the final verdict. It is important to note that the algorithm supports both online and offline monitoring. We generalize the input to the algorithm as a trace, which could be the entire program trace in the case of offline monitoring, or an event or a buffered sequence of events in the case of online monitoring.

This section is organized as follows: Subsection 4.1 describes how valuations are extracted from a trace in run time, and Subsection 4.2 describes the steps of the algorithm in detail.

4.1 Valuation Extraction

Valuation extraction refers to obtaining a valuation of quantified variables from the trace. As described in $\textsc{Ltl}_4{-}\mathrm{C}$ semantics, the predicate $p_i(x_i)$ identifies the subset of the domain of x_i over which the quantifier is applied: namely the subset that exists in the trace. From a theoretical perspective, we check whether the predicate is a member of some trace event, which is a set of predicates. From an implementation perspective, the trace event is a key-value structure, where the key is for instance a string identifying the quantified variable, and the value is the concrete value of the quantified variable in that trace event.

4.2 Algorithm Steps

Algorithm 1 presents the pseudocode of the parallel monitoring algorithm. Given an $\textsc{Ltl}_4{-}\mathrm{C}$ property $\varphi = \mathbb{Q}_\varphi\, \psi$, the input to the algorithm is the RV-$\textsc{Ltl}$ monitor $\mathcal{M}^*$ of RV-$\textsc{Ltl}$ property ψ, a finite trace u, the set of quantifiers $\mathbb{Q}_\varphi$, and the vector of keys K used to extract valuations. Note that the algorithm supports both online and offline runtime verification. Offline mode is straightforward since the algorithm receives a finite trace that it can evaluate.

In the case of online mode, the algorithm maintains data structures that represent the tree structure shown in Figure 1, and repeated invocation of the monitor updates these data structures incrementally. Thus, an online monitor will receive batches of events in run time and process them, building the tree of monitors with every invocation of the monitor. These invocations can be periodic or event based, and the batches can be of any length.

The entry point to the algorithm is at Line 5 which is invoked when the monitor receives a trace to process. This can be the entire trace in offline monitoring, or a buffered segment of the trace in online monitoring. The algorithm returns a truth value of the property at Line 8. Subsections 4.2 – 4.2 describe the functional calls between Lines 5 – 8.

The MapReduce operations are visible in functions *SortTrace* and *ApplyQuantifiers*, which perform a *map* ($\rightrightarrows$) in Lines 10 and 40 respectively. *ApplyQuantifiers* also performs a reduction ($\rightarrowtail$) in Line 41.

Trace Sorting As shown in Algorithm 1, the first step in the algorithm is to sort the input trace u (Line 5). The function *SortTrace* performs this functionality as follows:

1. The function performs a parallel map of every trace event to the value vector that it holds (Line 10).
2. The mapped trace is sorted in parallel using the quantifier variable as a key (Line 11).
3. The sorted trace is then compacted based on valuations, and the function returns a map μ where keys are value vectors and values are the ranges of where these value vectors exist in trace u (Line 12). A range contains the start and end index. This essentially defines the subsequences u^{D_φ} for each property instance $\hat{\varphi}(D_\varphi)$ (refer to Subsection 2.4).

Monitor Spawning Monitor spawning is the second step of the algorithm (Line 6). The function *SpawnMonitors* receives a map μ and searches the cached collection of previously encountered value vectors $\mathbb{D}$ for duplicates. If a value vector in μ is new, it creates submonitors and inserts them in the tree of submonitors T (Line 18). The function *AddToTree* attempts to generate $|\mathbb{Q}_\varphi|-1$ quantifier submonitors $\mathcal{M}^{\mathcal{Q}}$ (Line 22) ensuring there are no duplicate monitors in the tree (Line 23).

Algorithm 1 LTL_4−C Monitor

```
 1: INPUT: An RV-LTL monitor $\mathcal{M}^*$ of LTL property $\psi$, a finite trace $u$, a set of quantifiers $\mathbb{Q}_\varphi$, and a vector of keys $K$ to extract valuations of quantified variables.
 2: declare $T = \{\mathcal{M}^{\mathcal{Q}}_{D|0}\}$    ▷ Tree of quantifier submonitors
 3: declare $\mathbb{D} = \{\}$,    ▷ Value vector set
 4: declare $\mathbb{M}^* = \{\}$    ▷ RV-LTL submonitor set
 5: $\mu \leftarrow$ SORTTRACE($u$)    ▷ The entry point
 6: SPAWNMONITORS($\mu$)
 7: DISTRIBUTE($u$,$\mu$)
 8: return APPLYQUANTIFIERS($|\mathbb{Q}_\varphi - 1|$)

 9: function SORTTRACE($u$)    ▷ Trace sorting and compaction
10:     $u_i \rightrightarrows u'_i$ :=VALUEVEC($u_i, K$)    ▷ ‖ map to value vectors
11:     PARALLELSORT($u'$,$K$)
12:     $\mu\langle D, r\rangle \leftarrow$ PARALLELCOMPACT($u'$)
13:     return $\mu$

14: function SPAWNMONITORS($\mu$)    ▷ Monitor spawning
15:     for $D \in \mu$ do in parallel
16:         if $D \notin \mathbb{D}$ then
17:             ADD($\mathbb{D}$,$D$)
18:             $t \leftarrow$ ADDTOTREE($D$)
19:             $t$.addMonitor(CREATEMONITOR($D$))

20: function ADDTOTREE($D$)
21:     $t$ = $T$.root
22:     for $i \in [1, |\mathbb{Q}_\varphi| - 1]$ do
23:         if $\mathcal{M}^{\mathcal{Q}}_{D|^i} \notin t$.children then
24:             $t$.addchild($\mathcal{M}^{\mathcal{Q}}_{D|^i}$)
25:         $t \leftarrow t$.children$\left[\mathcal{M}^{\mathcal{Q}}_{D|^i}\right]$
26:     return $t$

27: function CREATEMONITOR($D$)    ▷ Monitor creation
28:     $\mathcal{M}^*_D \leftarrow$ LAUNCHMONITORTHREAD($D$)
29:     $\mathcal{M}^*_D.D \leftarrow D$
30:     ADD($\mathbb{M}^*$,$\mathcal{M}^*_D$)
31:     return $\mathcal{M}^*_D$

32: function DISTRIBUTE($u$,$\mu$)    ▷ Distribute trace to monitors
33:     for $\mathcal{M}^*_D \in \mathbb{M}^*$ do in parallel
34:         PROCESSBUFFER($\mathcal{M}^*_D$,$u$,$\mu[\mathcal{M}^*_D.D]$)

35: function PROCESSBUFFER($\mathcal{M}^*_D$,$u$,$r$)    ▷ Process trace
36:     filter include $u \rightrightarrows u' := u[r.\text{start}, r.\text{end}]$    ▷ ‖ filter
37:     $\mathcal{M}^*_D.b \leftarrow$UPDATEMONITOR($\mathcal{M}^*_D$, $u'$)

38: function APPLYQUANTIFIERS($i$)    ▷ Apply quantifiers
39:     for $t \in T$.nodesAtDepth($i$) do in parallel
40:         $t$.children $\rightrightarrows \{s := [v, v', \cdots]\}$    ▷ ‖ map
41:         $s \rightarrowtail t.v$    ▷ ‖ reduction to truth vector
42:         $t.b \leftarrow$ VALUATION($t$)    ▷ $\text{LTL}_4$−C semantics
43:     if $i = 0$ then
44:         return $t.b$
45:     return APPLYQUANTIFIERS($i - 1$)
```

After all quantifier submonitors are created, *SpawnMonitors* creates an RV-Ltl submonitor $\mathcal{M}^*$ and adds it as a child to the leaf quantifier submonitor in the tree representing the value vector (Line 19). This resembles the structure in Figure 1.

Checking whether submonitors do not already exist and the creation of new submonitors is performed in parallel for all value vectors in trace u. This is because the trace has been sorted and grouped by unique value vectors in the previous step. Thus, each subtree of monitors that corresponds to a unique value vector is created in parallel, and connected to its parent via locks.

Distributing the Trace The next step in the algorithm is to distribute the sorted trace to all RV-Ltl submonitors (Line 7). The function *Distribute* instructs every RV-Ltl submonitor to process its respective trace by passing the full trace and the range of its respective subsequence, which is provided by the map μ (Line 34). The RV-Ltl monitor updates its state according to the trace subsequence and stores its truth value b.

Applying Quantifiers Applying quantifiers is a recursive process, beginning with the leaf quantifier submonitors and proceeding upwards towards the root of the tree (Line 8). Function *ApplyQuantifiers* operates in the following steps:

1. The function retrieves all quantifier submonitors at the i^{th} level in the tree T (Line 39).
2. In parallel, for each quantifier submonitor, all child submonitor truth values are reduced into a single truth value of that quantifier submonitor (Lines 40-42). This step *reduces* all child truth vectors into a single vector and then applies $\text{Ltl}_4{-}\text{C}$ semantics to determine the truth value of the current submonitor, essentially applying function $\mathcal{F}$ on the truth values of all child submonitors.
3. The function proceeds recursively calling itself on submonitors that are one level higher. It terminates when the root of the tree is reached, where the truth value is the final verdict of the property with respect to the trace.

5 Implementation and Experimental Results

We have implemented Algorithm 1 for two computing technologies: Multi-core CPUs and GPUs. We applied three optimizations in our GPU-based implementation: (1) we use *CUDA Thrust API* to implement parallel sort, (2) we use *Zero-Copy Memory* which parallelizes data transfer with kernel operation without caching, and (3) we enforced alignment, which enables coalesced read of trace events into monitor instances. In order to intercept systems calls, we have integrated our algorithm with the Linux `strace` application, which logs all system calls made by a process, including the parameters passed, the return value, the time the call was made, etc. Notice that using `strace` has the benefit of eliminating static analysis for instrumentation.

5.1 Case studies

We have conducted three case studies, the first demonstrates using our implementation for online monitoring, while case study 2 and 3 are monitored offline. Following is a detailed description of the case studies:

1. **Ensuring every request on a socket is responded to.** This case study monitors the responsiveness of a web server. Web servers under heavy load may experience some timeouts, which results in requests that are not responded to. This is a factor contributing to the uptime of the server, along with other factors like power failure, or system failure. Thus, we monitor that at least 95% of requests are indeed responded to:

$$\mathbb{A}_{\geq 0.95}\, s : \mathsf{socket}(s) \cdot (\mathbf{G}\, \mathsf{receive}\,(s) \Rightarrow \mathbf{F}\, \mathsf{respond}\,(s))$$

   We use the Apache Benchmarking tool to create varying loads on the Apache web server, and monitor the property *online*.
2. **Ensuring fairness in utilization of personal cloud storage services.** This case study is based on the work in [12], which discusses how profiling DropBox traffic can identify the bottlenecks and improve the performance. Among the issues detected during this analysis, is a user repeatedly uploading chunks of maximum size to DropBox servers. Thus, it is beneficial for a runtime verification system to ensure that the majority of chunks are not of maximum size, ensuring fairness of service use. The corresponding $\text{LTL}_4\text{–C}$ property is as follows:

$$\mathbb{A}u : \mathsf{user}(u) \cdot \mathbb{A}_{<0.5}\, c : \mathsf{chunk}(c) \cdot \mathsf{ismaxchunksize}\,(u, c)$$

3. **Ensuring proxy cache is functioning correctly.** This experiment is based on a study that shows the effectiveness of utilizing proxy cache in decreasing
   YouTube videos requests in a large university campus [20]. Thus, we monitor that no video is requested externally more than once:

$$\mathbb{A}v : \mathsf{vid}(v) \cdot \mathbb{E}_{\leq 1}\, r : \mathsf{req}(r, v) \cdot \mathsf{external}(r, v)$$

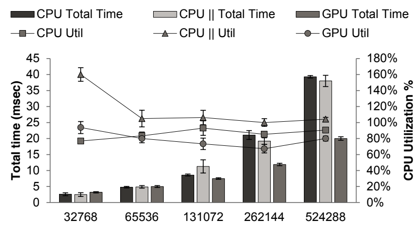
Notice that the formulas in the above case studies utilize counting semantics to express properties that cannot be expressed in standard LTL. Moreover, evaluation of these properties can never result in permanent satisfaction nor permanent violation (Case Study 1 and 2 only). Thus, the use of four-valued logic allows the monitor to produce a meaningful verdict for these properties.

5.2 Experimental Setup

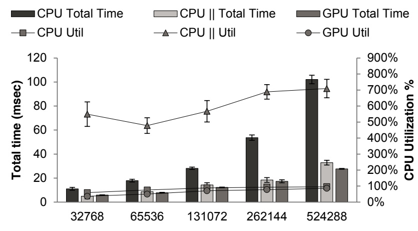
Experiment Platform. The experiments machine comprises of a 12-core Intel Xeon E5-1650 CPU, an Nvidia Tesla K20c GPU, and 32GB of RAM, running Ubuntu 12.04.

Experimental Factors. We experiment with three implementations: single CPU, parallel CPU, and a GPU based implementation. We also experiment with multiple trace sizes to demonstrate scalability.
Experimental Metrics. We measure the total execution time and the monitor's CPU utilization. This is to demonstrate the impact of monitoring on overall CPU utilization. We perform 20 replicates of each experiment and present error bars of a 95% confidence interval.

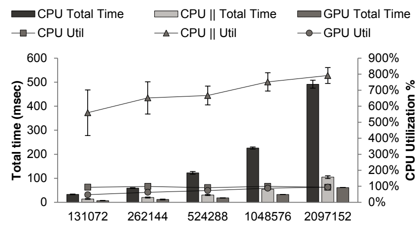
5.3 Results



(a) Case Study 1



(b) Case Study 2



(c) Case Study 3

Fig. 2: Experimental results.

First of all, the case studies have been evaluated with real-world datasets, except in Case Study 1, where we use the Apache benchmark tool to generate traces. We have validated that the monitor works correctly by ensuring that verdicts of presumably true/false are produced appropriately for Case Studies 1 and 2, and verdicts of presumably true / permanently false are produced appropriately for Case Study 3.

The performance results of Case Study 1 are shown in Figure 2a. As seen in the figure, the GPU implementation scales efficiently with increasing trace size, resulting in the lowest monitoring time of all three implementations. The GPU versus single core CPU speedup ranges from 0.8 to 1.6, increasing with the increasing trace size. When compared to parallel CPU (CPU ||), the speedup ranges from 0.78 to 1.59. This indicates that parallel CPU outperforms GPU for smaller traces (32768), yet does not scale as well as GPU in this case study. This is attributed to the low number of individual objects in the trace, making parallelism less impactful.

CPU utilization results in Figure 2a show a common trend with the increase of trace size. When the trace size is small, parallel implementations incur high CPU utilization as opposed to a single core implementation, which could be attributed to the overhead of parallelization relative to the small trace size. On the other hand, GPU shows a stable utilization percentage, with a 78% average utilization. The single core CPU implementation shows a similar trend, yet slightly elevated average utilization (average 86%). The parallel CPU implementation imposes a higher CPU utilization (average 115%), since more cores are being used to process the trace. This result indicates that shipping the monitoring workload to GPU consistently provides more time for CPU to execute other processes including the monitored process. The results of Case Study 2 and Case Study 3 in Figures 2b and 2c respectively demonstrate a more prominent advantage of using GPU in terms of speedup. The number of individual objects in these traces are large, making parallelism highly effective. For Case Study 2, the speedup of the GPU implementation over single core CPU ranges from 1.8 to 3.6, and 0.83 to 1.18 over parallel CPU. The average CPU utilization of GPU, single core CPU, and parallel CPU is 64%, 82%, and 598% respectively. For Case Study 3, speedup is more significant, with 6.3 average speedup of GPU over single core CPU, and 1.75 over parallel CPU. The average CPU utilization of GPU, single core CPU, and parallel CPU is 73%, 95%, and 680% respectively. Thus, the parallel CPU implementation is showing large speedup similar to the GPU implementation, yet also results in a commensurate CPU utilization percentage, since most cores of the system are fully utilized.

Although the parallel CPU implementation provides reasonable speedup and the single-core CPU implementation imposes low CPU overhead, the GPU implementation manages to achieve both simultaneously.

6 Related Work

Runtime verification of parametric properties has been studied by Rosu et al [13, 16]. In this line of work, it is possible to build a runtime monitor parameterized by objects in a Java program. The work by Chen and Rosu [9] presents a method of monitoring parametric properties in which a trace is divided into slices, such that each monitor operates on its slice. This resembles our method of identifying trace subsequences and how they are processed by submonitors. However, parametric monitoring does not provide a formalization of applying existential and numerically constrained quantifiers over objects.

Bauer et al. [5] present a formalization of a variant of first order logic combined with LTL. The work by Leucker et al. presents a generic approach for monitoring modulo theories [11]. This work provides a more expressive specification language. Our work enforces a canonical syntax which is not required in [11], resulting in more expressiveness. However, the monitoring solution provided requires SMT solving at run time. $\textsc{Ltl}_4\text{−C}$ extends RV-$\textsc{Ltl}$ by redefining $\top_p$ and $\bot_p$ to support quantifiers and their numerical constraints. This four-valued semantics provides a more accurate assessment of the satisfaction of the property based on finite traces as opposed to the three-valued semantics in [11].

The work in [14] presents an extension to LTL that allows counting events associated with the *Until* operator. In this work, it is possible to apply a numerical constraint on the number of events satisfying subformulas. This differs from $\textsc{Ltl}_4\text{−C}$, where numerical constraints are applied on quantified objects, allowing us to reason about the number or percentage of objects that satisfy a property. The work in [2] allows a limited form of quantification over values of a variable, yet does not support a higher level of quantification on the entire LTL property parameterized by the quantified variable, as is possible in LTL_4−C. The work in [18] presents a property specification language that allows quantification, and separates propositional evaluation from quantifier evaluation, similar to LTL_4−C. However, LTL_4−C supports LTL operators and quantification with numerical constraints.

The work in [1] presents a method of using MapReduce to evaluate LTL properties. The algorithm is capable of processing arbitrary fragments of the trace in parallel. The work in [3] presents a MapReduce method for offline verification of LTL properties with first-order quantifiers. Our approach supports both offline and online monitoring by extending RV-LTL's four valued semantics, which are capable of reasoning about the satisfaction of a partial trace. This is unclear in [3], since there is no evidence of supporting online monitoring.

The work in [10] presents a specification language for defining properties on input streams. The work in [4] presents an extension to metric first order temporal logic that allows aggregate operations.

Finally, the work in [7, 8] presents two parallel algorithms for verification of propositional LTL specifications at run time. These algorithms are implemented in the tool RiTHM [17]. This paper enhances the framework in [7, 8, 17] by introducing a significantly more expressive formal specification language along with a parallel runtime verification system.

7 Conclusion

In this paper, we proposed a specification language (LTL_4−C) for runtime verification of properties of types of objects in software and networked systems. Our language is an extension of LTL that adds counting semantics with numerical constraints. The four truth values of the semantics of LTL_4−C allows system designers to obtain informative verdicts about the status of system properties at run time. We also introduced an efficient and effective parallel algorithm with two implementations on multi-core CPU and GPU technologies. The results of our experiments on real-world case studies show that runtime monitoring using GPU provides us with the best throughput and CPU utilization, resulting in minimal intervention in the normal operation of the system under inspection.

For future work, we are planning to design a framework for monitoring LTL_4−C properties in distributed systems and cloud services. Another direction is to extend LTL_4−C such that it allows non-canonical strings of quantifiers. Finally, we are currently integrating LTL_4−C in our tool RiTHM [17].

8 Acknowledgement

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grants 430575-2012 and 463324-2014.

References

1. B. Barre, M. Klein, M. Soucy-Boivin, P.-A. Ollivier, and S. Hallé. Mapreduce for parallel trace validation of ltl properties. In *Runtime Verification*, pages 184–198. Springer, 2012.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 264. IEEE, 2004.
3. D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring. In *Runtime Verification*, pages 31–47. Springer, 2014.
4. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
5. A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *Runtime Verification*, pages 59–75. Springer, 2013.
6. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
7. S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1025–1036. IEEE, 2013.
8. S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Runtime verification with minimal intrusion through parallelism. *Formal Methods in System Design*, 46(3):317–348, 2015.
9. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261. Springer, 2009.
10. B. d'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174. IEEE, 2005.
11. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–356. Springer, 2014.
12. I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.
13. D. Jin, P. O. Meredith, C. Lee, and G. Rosu. Javamop: Efficient parametric runtime monitoring framework. In *34th International Conference on Software Engineering (ICSE)*, pages 1427–1430, June 2012.
14. F. Laroussinie, A. Meyer, and E. Petonnet. Counting ltl. In *Proceedings of the 2010 17th International Symposium on Temporal Representation and Reasoning*, TIME '10, pages 51–58. IEEE, 2010.
15. L. Libkin. *Elements of finite model theory.* Springer, 2004.
16. P. Meredith and G. Rosu. Efficient parametric runtime verification with deterministic string rewriting. In *International Conference on Automated Software Engineering (ASE)*, pages 70–80. IEEE, 2013.
17. S. Navabpour, Y. Joshi, W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for c programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 603–606, 2013.

18. O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144(4):91–108, 2006.
19. M. Williams. Scaling Web Applications with NGINX, Part II: Caching and Monitoring. `https://www.nginx.com/blog/`, 2015. [Online; accessed 27-May-2016].
20. M. Zink, K. Suh, Y. Gu, and J. Kurose. Watch global, cache local: Youtube network traffic at a campus network: measurements and implications. In *Electronic Imaging 2008*, pages 681805–681805. International Society for Optics and Photonics, 2008.