GPU-based Runtime Verification

Shay Berkovich
Dept. of Elec. and Comp. Eng.
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: sberkovi@uwaterloo.ca

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: borzoo@cs.uwaterloo.ca

Sebastian Fischmeister
Dept. of Elec. and Comp. Eng.
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: sfischme@uwaterloo.ca

***Abstract*—*Runtime verification* is a monitoring technique to gain assurance about well-being of a program at run time. Most existing approaches use sequential monitors; i.e., when the state of the program with respect to an event of interest changes, the monitor interrupts the program execution, evaluates a set of logical properties, and finally resumes the program execution. In this paper, we propose a GPU-based method for design and implementation of monitors that enjoy two levels of parallelism: the monitor (1) works along with the program in parallel, and (2) evaluates a set of properties in a parallel fashion as well. Our parallel monitoring algorithms effectively exploit the many-core platform available in the GPU. In addition to parallel processing, our approach benefits from a true separation of monitoring and functional concerns, as it isolates the monitor in the GPU. Our method is fully implemented and experimental results show significant reduction in monitoring overhead, monitoring interference, and power consumption due to leveraging the GPU technology.**

***Keywords*-Runtime monitoring; parallel algorithms; temporal logic; formal methods**

I. Introduction

In computing systems, *correctness* refers to the assertion that a system satisfies its specification. *Runtime verification* [1], [2], [3] refers to a technique where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness temporal property. Runtime verification complements exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing and debugging.

The main challenge in augmenting a system with runtime verification is dealing with its runtime *overhead*. This overhead often introduces two types of defects to the system under scrutiny: (1) unpredictable execution, and (2) possible bursts of monitoring intervention in program execution. Several techniques have been introduced in the literature for reducing and controlling runtime overhead. Examples include:

- improved instrumentation (e.g., using aspect-oriented programming [4], [5]),
- combining static and dynamic analysis techniques (e.g., using typeset analysis [6] and PID controllers [7]),
- efficient monitor generation and management (e.g., in the case of parametric monitors [8]), and
- schedulable monitoring (e.g., using time-triggered monitors [9]).

Each of the above approaches remedies the overhead issue to some extent and in specific contexts. However, there has been little work on reducing and containing the overhead of runtime verification through isolating the monitor in a different processing unit.

With this motivation in mind, in this paper, we propose a technique that permits the separation of the functional from monitoring concerns into different computing units; i.e., the program under inspection and its monitor run in parallel on different hardware processing units. Our formal language for monitoring properties is the linear temporal logic (LTL) and, in particular its 3-valued semantics (LTL_3) [10]. We utilize the algorithm introduced in [10] for synthesizing a monitor as a deterministic finite state machine from an LTL_3 formula.

Although the idea of parallelizing the execution of a monitor seems counterintuitive, we introduce two algorithms that take a program trace as input and evaluate the trace in a parallel fashion as follows. The algorithms divide each trace into a set of sub-traces (multiple data) and then perform identical evaluation functions on each sub-trace on different processing units (single instruction). In other words, the algorithms employ Single Instruction Multiple Data (SIMD) parallelism and, hence, one can leverage a graphics processing unit (GPU) for parallel evaluation of LTL properties. The resulting monitoring architecture, where the program runs on a CPU while simultaneously the monitor runs on a GPU is shown in Figure 1. The main challenge in implementing such algorithms is to ensure that the causal order of occurrence of events in the program trace is respected when evaluating a property in a parallel fashion. To this end, we formalize a notion of LTL_3 property *history* that encodes the causal order of events for parallelization. Intuitively, a property history is the maximum sequence of a monitor state changes to decide a verification verdict. We show that this length may influence the parallelization of verification of a program trace.

In Figure 1, the program under scrutiny stores a finite program trace, which is the sequence of events that need to

This is an extended version of the paper that appeared in IPDPS'13.

be examined. The motivation behind building this trace is to accumulate a load of events so the verification of the properties with respect to the events can be parallelized. The size of the trace and the decision on when to provide the monitor with the trace are given as input parameters by the system designer. Examples of contributing factors in specifying these parameters include tolerable latency for detecting violations of the specification and monitor scheduling constraints. A *host* process receives the program trace in the shared memory and distributes chunks of this trace among a set of monitoring worker threads running on the GPU. The worker threads are capable of monitoring one or more properties simultaneously. Once one of the properties is violated or satisfied, the monitor will report this event back to the program. In case of a violation, there needs to be a recovery or steering action to lead the program back to its normal behavior. The feedback arrow in Figure 1 and its consequences are out of the scope of this work.

One can observe that this architecture leverages two levels of parallelism: (1) between the program running on the CPU and the monitor running on the GPU, and (2) among the monitoring worker threads running on the GPU cores. Moreover, notice that the main overhead incurred is the cost of the data transfer between the host process (on the CPU) and the monitoring threads on the GPU. Given the current trend in merging the address space of CPU and GPU (e.g., in the AMD Fusion family of APUs), we expect this cost to decrease.

Our objective in designing such an architecture is twofold: (1) reducing the overhead through parallel processing, and (2) decreasing the monitoring interference during the program execution. While the former is the objective of most research activities in the area of runtime verification, the latter is highly desirable in the design, development, and engineering of software applications especially in the domain of safety-critical systems. For example, isolating the monitor in a separate device facilitates certification of embedded software, as regulators often require evidence of independence between the safety and control systems (e.g., in the DO-178c standard).

Although our algorithms are designed for SIMD architectures, our theoretical results and algorithms can be implemented in both GPU-based and multi-core programming environments, such as POSIX threads, MPI, OpenCL, and CUDA with minimal portability efforts. Our approach is fully implemented using the OpenCL language[1] and, thus, it is not limited to GPU platforms and can be used in multi-core platforms that support OpenCL as well.

We present the results of a set of experiments to (1) compare the monitoring overhead on the CPU and GPU, and (2) study the throughput and scalability of our algorithms for parallel verification of different properties. We also

[1]http://www.khronos.org/opencl/

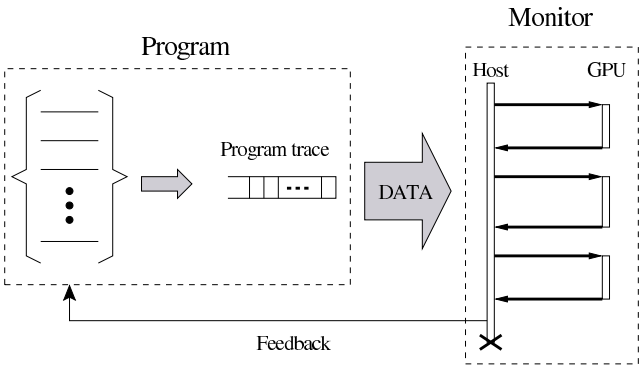

Figure 1. GPU-based Monitor Architecture

describe a case study based on monitoring the embedded controller of a flying unmanned aerial vehicle (UAV). We analyze the power consumption and the CPU utilization of GPU-based monitoring. Our conclusion is that GPU-based runtime verification is an effective approach to significantly reduce the runtime overhead and to isolate functional from monitoring concerns. Moreover, we observe that GPU-based monitoring leads to significantly lower power consumption when compared to CPU-based monitoring. This is crucial considering power constraints presented on some embedded and safety-critical systems.

Organization: Section II presents the preliminary concepts. Our notion of monitoring history is described in Section III, while Section IV is dedicated to our algorithms for parallel verification of LTL_3 properties at run time. We analyze experimental results in Section V. Section VI discusses related work. Finally, in Section VII, we make concluding remarks and identify future work.

II. Preliminaries

A. Linear Temporal Logic (LTL)

Linear temporal logic (LTL) is a popular formalism for specifying properties of (concurrent) programs. The set of well-formed linear temporal logic formulas is constructed from a set of atomic propositions, the standard Boolean operators, and temporal operators. Precisely, let AP be a finite set of *atomic propositions* (e.g., $x \geq 2$, where x is an integer) and $\Sigma = 2^{AP}$ be a finite *alphabet*. A letter a in Σ is interpreted as assigning truth values to the elements of AP; i.e., elements in a are assigned true $\top$ and elements not in a are assigned false $\bot$ (e.g., $(x \geq 2) \wedge (y = 5)$). A *word* is a finite or infinite sequence of letters $w = a_0 a_1 a_2 \ldots$, where $a_i \in \Sigma$ for all $i \geq 0$. We denote the set of all finite words over Σ by Σ^* and the set of all infinite words by Σ^ω. For a finite word u and a word w, we write $u \cdot w$ to denote their *concatenation*.

*Definition 1 (*LTL *Syntax):* LTL formulas are defined inductively as follows:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in \Sigma$, and, $\bigcirc$ (next) and $\mathbf{U}$ (until) are temporal operators. ■

*Definition 2 (*LTL *Semantics):* Let $w = a_0 a_1 \ldots$ be an infinite word in Σ^ω, i be a non-negative integer, and $\models$ denote the *satisfaction* relation. Semantics of LTL is defined inductively as follows:

$w, i \models \top$
$w, i \models p$ iff $p \in a_i$
$w, i \models \neg\varphi$ iff $w, i \not\models \varphi$
$w, i \models \varphi_1 \vee \varphi_2$ iff $w, i \models \varphi_1 \ \vee \ w, i \models \varphi_2$
$w, i \models \bigcirc\varphi$ iff $w, i+1 \models \varphi$
$w, i \models \varphi_1 \,\mathbf{U}\, \varphi_2$ iff
$\exists k \geq i : w, k \models \varphi_2 \ \wedge \ \forall j : i \leq j \leq k : w, j \models \varphi_1$.
In addition, $w \models \varphi$ holds iff $w, 0 \models \varphi$ holds. ■

Notice that an LTL formula φ defines a set of words (i.e., a *language* or a *property*) that satisfies the semantics of that formula. We denote this language by $L(\varphi)$. For simplicity, we introduce abbreviation temporal operators: $\Diamond\varphi$ (*eventually* φ) denotes $\top \,\mathbf{U}\, \varphi$, and $\Box\varphi$ (*always* φ) denotes $\neg\Diamond\neg\varphi$. For instance, formula $\Box(p \Rightarrow \Diamond q)$ means that 'it is always the case that if proposition p holds, then eventually proposition q holds'. One application of this formula is in reasoning about non-starvation in mutual exclusion algorithms: 'if a process requests entering critical section, then it eventually is granted to do so'.

In order to reason about correctness of programs with respect to an LTL property, we describe a program in terms of its state space and transitions. A *program* is a tuple $P = \langle S, T \rangle$, where S is the non-empty *state space* and T is a set of *transitions*. A transition is of the form (s_0, s_1), where $s_0, s_1 \in S$. A state of a program is normally obtained by valuation of its variables and transitions are program instructions. In this context, an atomic proposition in a program is a Boolean predicate over S (i.e., a subset of S).

We define a *trace* of a program $P = \langle S, T \rangle$ as a finite or infinite sequence of subsets of atomic propositions obtained by valuation of program variables (i.e., program states). Thus, a program trace can be defined as $\sigma = s_0 s_1 s_2 \ldots$, such that $s_i \in S$ and each $(s_i, s_{i+1}) \in T$, for all $i \geq 0$. A program trace σ *satisfies* an LTL property φ (denoted $\sigma \models \varphi$) iff $\sigma \in L(\varphi)$. If σ does not satisfy φ, we say that σ *violates* φ. A program P satisfies an LTL property φ (denoted $P \models \varphi$) iff for each trace σ of P, $\sigma \models \varphi$ holds.

B. 3-Valued LTL

Implementing runtime verification boils down to the following problem: given the current program *finite* trace $\sigma = s_0 s_1 s_2 \ldots s_n$, whether or not σ belongs to a set of words defined by some property φ. This problem is more complex than it looks, because LTL semantics is defined over infinite traces and a running program can only deliver a finite trace at a verification point. For example, given a finite trace $\sigma = s_0 s_1 \ldots s_n$, it may be impossible for a *monitor* to decide weather the property $\Diamond p$ is satisfied.

To formalize satisfaction of LTL properties at run time, in [10], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values '$\top$', '$\bot$', and '?' (denoted LTL_3). The latter value expresses the fact that it is not possible to decide on the satisfaction of a property, given the current program finite trace.

Definition 3 (LTL_3 *semantics):* Let $u \in \Sigma^*$ be a finite word. The truth value of an LTL_3 formula φ with respect to u, denoted by $[u \models \varphi]$, is defined as follows:

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall w \in \Sigma^\omega : u \cdot w \models \varphi, \\ \bot & \text{if } \forall w \in \Sigma^\omega : u \cdot w \not\models \varphi, \\ ? & \text{otherwise.} \end{cases}$$ ■

Note that the syntax $[u \models \varphi]$ for LTL_3 semantics is defined over finite words as opposed to $u \models \varphi$ for LTL semantics, which is defined over infinite words. For example, given a finite program trace $\sigma = s_0 s_1 \cdots s_n$, property $\Diamond p$ holds iff $s_i \models p$, for some i, $0 \leq i \leq n$ (i.e., σ is a good prefix). Otherwise, the property evaluates to ?.

Definition 4 (Good and Bad Prefixes): Given a language $L \subseteq \Sigma^\omega$ of infinite words over Σ, we call a finite word $u \in \Sigma^*$

- a *good prefix* for L, if $\forall w \in \Sigma^\omega : u \cdot w \in L$
- a *bad prefix* for L, if $\forall w \in \Sigma^\omega : u \cdot w \notin L$
- an *ugly* prefix otherwise. ■

In order to declare a verification verdict by a monitor more efficiently, it is advantageous to recognize good and bad prefixes as early as possible.

Definition 5 (Minimal Good/Bad Prefixes): A bad (good) prefix u for language $L \subseteq \Sigma^\omega$ is called *minimal* if each strict prefix of u is not a bad (good) prefix. ■

For example, for property $\varphi \equiv (p \,\mathbf{U}\, q)$, the prefix $(p \wedge \neg q)^*$ is an ugly prefix, $(p \wedge \neg q)^* \cdot (\neg p \wedge \neg q)$ is a minimal bad prefix, and $(p \wedge \neg q)^* \cdot (\neg p \wedge q)$ is a minimal good prefix.

Implementing runtime verification for an LTL_3 property involves synthesizing a monitor that realizes the property. In [10], the authors introduce a stepwise method that takes an LTL_3 property φ as input and generates a deterministic finite state machine (FSM) $\mathcal{M}^\varphi$ as output. Intuitively, simulating a finite word u on this FSM reaches a state that illustrates the valuation of $[u \models \varphi]$.

Definition 6 (Monitor): Let φ be an LTL_3 formula over alphabet Σ. The *monitor* $\mathcal{M}^\varphi$ of φ is the unique FSM $(\Sigma, Q, q_0, \delta, \lambda)$, where Q is a set of states, q_0 is the initial state, δ is the transition relation, and λ is a function that maps each state in Q to a value in $\{\top, \bot, ?\}$, such that:

$$[u \models \varphi] = \lambda(\delta(q_0, u)).$$ ■

Examples of monitors appear in Figures 2, 3, and 4. We use the term a *conclusive state* to refer to monitor states

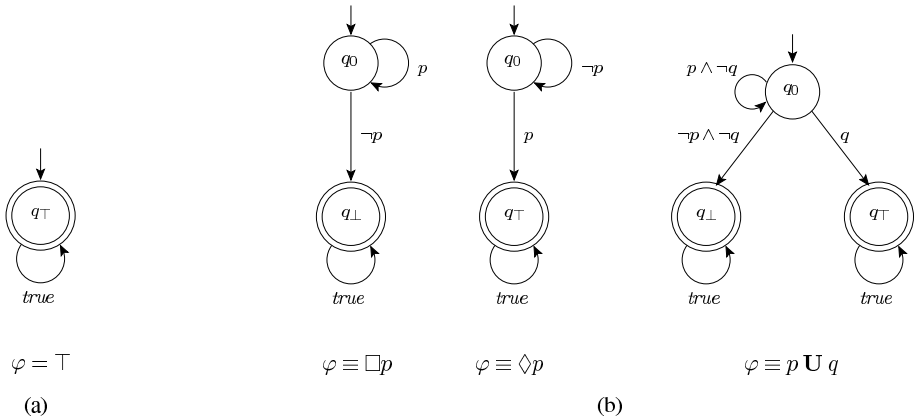

Figure 2. Monitors for properties, where (a) $||\mathbb{H}^\varphi|| = 0$, and (b) $||\mathbb{H}^\varphi|| = 1$.

$q_\top$ and $q_\bot$; i.e., states where $\lambda(q) = \top$ and $\lambda(q) = \bot$, respectively. Other states are called an *inconclusive state*. A monitor $\mathcal{M}^\varphi$ is constructed in a way that it recognizes minimal good and bad prefixes of $L(\varphi)$. Hence, if $\mathcal{M}^\varphi$ reaches a conclusive state, it stays in this *trap* state. We note that the construction algorithm in [10] synthesizes a minimized automaton for an LTL$_3$ property.

Definition 7 (Monitorable Property): An LTL$_3$ property φ is *monitorable* if $L(\varphi)$ has no ugly prefixes. We denote the set of all monitorable LTL$_3$ properties by LTL$_3^{mon}$. ■

In other words, a property is *monitorable* if for every finite word, there still exists a (possibly) infinite continuation that will determine whether the property is violated or satisfied.

III. LTL$_3$ Monitoring History

The construction of a monitor for an LTL$_3$ property as described in Section II, allows evaluation of a prefix with respect to the property by observing state-by-state changes in the program under scrutiny. In other words, the monitor processes each change of state in the program individually. Such state-by-state monitoring is inherently sequential. The core of our idea to leverage parallel processing in runtime verification of a property is to buffer finite program traces and then somehow assign one or more sub-trace to a different monitoring processing units. We assume that the length of the program trace is given as an input parameter by the system designer. This length may depend on factors such as: (1) hardware constraints (e.g., memory limitations), (2) tolerable detection latency (i.e., the time elapsed since a change of state until a property violation is detected), or (3) the sampling period in time-triggered runtime verification [9] (e.g., for scheduling purposes in real-time systems).

Although our idea seems simple and intuitive, its implementation may become quite complex. This is due to the fact that truthfulness of some LTL$_3$ properties is sensitive to the causal order of state changes. For example, monitoring property $\varphi \equiv (p \Rightarrow \Diamond q)$ has to take the order of occurrence of atomic propositions p and q into account. This leads us to our notion of *history* carried by an LTL$_3$ formula. This history can be seen as a measure of "parallelizability" of the property. We first informally motivate the idea and then present formal definitions.

One can observe that each state of $\mathcal{M}^\varphi$ for a property φ in LTL$_3^{mon}$ represents a different logical step in evaluation of φ. Thus, the structure of $\mathcal{M}^\varphi$ characterizes the temporal complexity of φ. Running a finite program trace on $\mathcal{M}^\varphi$ results in obtaining a sequence of states of $\mathcal{M}^\varphi$. This sequence encodes a history (denoted $\mathbb{H}^\varphi$) of state changes in the program under inspection and consequently in the monitor. In the context of monitoring, we are only concerned with the longest minimal good or bad prefixes in $L(\varphi)$. Thus, the length of the history for property φ (denoted $||\mathbb{H}^\varphi||$) is the number of steps that the monitor needs at most to evaluate φ. For example, for the trivial property $\varphi \equiv \top$, we have $||\mathbb{H}^\varphi|| = 0$, since φ is evaluated in 0 steps (see Figure 2(a)). Figure 2(b), shows three properties and their corresponding monitors, where $||\mathbb{H}^\varphi|| = 1$. Figure 3 demonstrates the monitor of property $\varphi_1 \equiv p \wedge (q \;\mathbf{U}\; r)$, where $||\mathbb{H}^{\varphi_1}|| = 2$. This is because the length of the longest path from the initial state q_0 to a conclusive state is 2 and the monitor has no cycles.

Definition 8 (History): Let φ be a property in LTL$_3^{mon}$ and $w \in \Sigma^\omega$ be an infinite word. The *history* of φ with respect to w is the sequence of states $\mathbb{H}_w^\varphi = q_0 q_1 \ldots$ of $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, such that $q_i \in Q$ and $q_{i+1} = \delta(q_i, w_i)$, for all $i \geq 0$. ■

Definition 9 (History Length): Let φ be a property in LTL$_3^{mon}$ and $w \in \Sigma^\omega$ be an infinite word. The *history length* of φ with respect to w (denoted $||\mathbb{H}_w^\varphi||$) is the number of state transitions in history $\mathbb{H}_w^\varphi = q_0 q_1 q_2 \ldots$, such that $q_i \neq q_{i+1}$, for all $i \geq 0$. The history length of a property is then: $||\mathbb{H}^\varphi|| = \max\{||\mathbb{H}_w^\varphi|| \mid w \in \Sigma^\omega \wedge w \text{ has a minimal good/bad prefix}\}$. ■

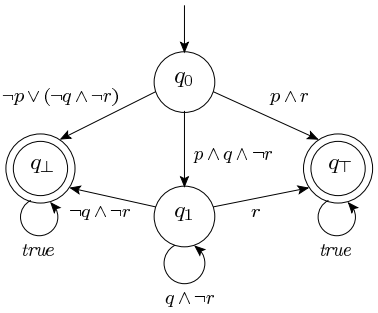

Figure 3. Monitor for property $\varphi_1 \equiv p \wedge (q \,\mathbf{U}\, r)$ with $||\mathbb{H}^{\varphi_1}|| = 2$.

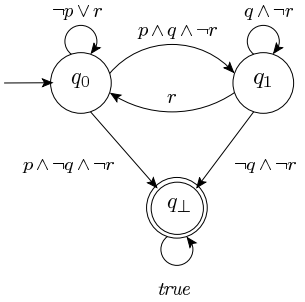

Figure 4. Monitor for property $\varphi_2 \equiv \Box(p \Rightarrow (q \,\mathbf{U}\, r))$ with $||\mathbb{H}^{\varphi_2}|| = \infty$.

We clarify a special case, where a monitor contains a cycle reachable from its initial state and a conclusive state is reachable from the cycle. In this case, according to Definition 9, the history length of the associated property is infinity. For example, Figure 4 illustrates such a monitor. Obtaining length of infinity is due to the existence of cycle $q_0 - q_1 - q_0$.

Theorem 1: Let φ be a property in LTL_3^{mon}. $\mathcal{M}^\varphi$ is cyclic iff $||\mathbb{H}^\varphi|| = \infty$.

Proof: We distinguish two cases:

- ($\Rightarrow$) If a cycle exists in $\mathcal{M}^\varphi$, then it does not involve a conclusive state. This is because any conclusive state is a trap. Thus, given a cycle $\bar{q} = q_0 - q_1 - q_2 \ldots q_k - q_0$ of inconclusive states, one can obtain an infinite word $w \in \Sigma^\omega$, such that the corresponding history $\mathbb{H}^\varphi_w$ will run infinitely on $\bar{q}$ and has infinite number of state changes. Therefore, $||\mathbb{H}^\varphi|| = ||\mathbb{H}^\varphi_w|| = \infty$.
- ($\Leftarrow$) If the length of a property history is ∞, then it has to be the case that some states in the history are revisited. This implication is trivial, because the number of states of a monitor is finite. Hence, the monitor must contain a cycle.

■

In general, the structure of a monitor depends on the structure of its LTL_3 formula (i.e., the number of temporal and Boolean operators as well as nesting depth of operators). For instance, the temporal operator $\bigcirc$ increments the history length of an LTL_3 formula by 1. If a formula does not include the $\mathbf{U}$ temporal operator, then the structure of its monitor will be a linear sequence of states and will not contain any loops. However, the reverse direction does not hold. For instance, properties $\varphi \equiv p \vee (q \,\mathbf{U}\, r)$ and $\psi \equiv p \,\mathbf{U}\, (q \,\mathbf{U}\, r)$ have both history of length 2.

Note that a finite history length for a property φ (i.e., $||\mathbb{H}^\varphi|| = n$, where $n < \infty$) does not necessarily imply that any history will have a finite number of states before reaching a conclusive state. This is because self-loops in $\mathcal{M}^\varphi$ may exist. Finiteness of $||\mathbb{H}^\varphi||$ simply means that monitoring φ takes at most n state changes in $\mathcal{M}^\varphi$ to reach a verdict.

IV. Parallel Algorithms for Evaluation of LTL_3 Properties

In Subsection IV-A, we analyze sequential evaluation of LTL_3^{mon} properties. Then, we introduce our parallel algorithm for evaluation of properties with finite history length in Subsection IV-B. Subsection IV-C proposes a more general (but less efficient) algorithm for properties with infinite or finite history length. Both algorithms exhibit SIMD parallelism. The parallelism comes from the fact that the data trace is processed in parallel (multiple data) by different processing units using the same monitoring kernel code. However, the algorithms logic ensures the right verification order, so that the final result is consistent with sequential execution.

A. Sequential Evaluation

Let $P = \langle S, T \rangle$ be a program and $\sigma = s_0 s_1 s_2 \ldots$ be a trace of P. Also, let φ be a property in LTL_3^{mon} and $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be its monitor, which is intended to inspect program P. One can build a sequential implementation of $\mathcal{M}^\varphi$ by employing a sequence of conditional statements as follows. By every change of program state s_i, $i \geq 0$, monitor $\mathcal{M}^\varphi$ calculates $q_{i+1} = \delta(q_i, u_i)$, where u_i is a mapping of program state s_i onto an alphabet Σ. The output of each algorithm step is $\lambda(q_{n+1})$. Thus, the time required to evaluate n program states can be described by the following equation:

$$T_{seq} = n.(t_{calc} + t_{branch}) \tag{1}$$

where t_{branch} is the time spent in a conditional statement to compute the next state of $\mathcal{M}^\varphi$ and t_{calc} is the (proposition to alphabet) translation time. For instance, consider the LTL_3^{mon} property $\varphi_1 \equiv p \wedge (q \,\mathbf{U}\, r)$ in Figure 3, where $p \equiv (\log(x) \leq 1.0)$, $q \equiv (\sin(y) \leq 1)$, and $r \equiv (\tan(z) \geq 3.0)$. Here, x, y, and z are program variables and p, q, and r are atomic propositions over the program states. Hence, t_{calc} involves calculation of $\log(x)$, $\sin(y)$, and $\tan(z)$.

Note that t_{calc} and consequently T_{seq}, increase linearly with computational complexity of (proposition to alphabet) translation. Now, if we are to evaluate multiple LTL_3^{mon} properties (say m), the algorithm complexity becomes $\mathcal{O}(m.n)$ and this linear increase appears in all evaluations:

$$T_{seq} = n.(\sum_{j=1}^{m}(t^j_{calc} + t^j_{branch})) \quad (2)$$

Our parallel algorithms described in this section tackle this linear increase by distributing the calculation load across multiple processing units.

B. Parallel Algorithm 1

Our first algorithm takes a program trace and processes it in a parallel fashion by assigning different program states to different GPU cores. To handle the causal order between the state transitions, the algorithm starts a new iteration every time a state transition occurs. Consequently, in each new iteration, a different current monitor state will be given. Intuitively, the number of iterations required to process the whole program trace, is bounded by the history length of the monitored property.

We now describe the algorithm in detail. The algorithm takes a finite program trace $\sigma = s_0 s_1 \dots s_n$, a monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, and a *current* state $q_{current} \in Q$ (possibly q_0) of the monitor as input and returns resulting state q_{result} and the monitoring verdict $\lambda(q_{result})$ as output (see Algorithm 1). The algorithm works as follows:

1) *(Initialization)* The algorithm initializes tuple $\mathcal{I}$ to $\langle n+1, q_{current}\rangle$ (line 2). This tuple is used to keep the index of the left-most program state in trace σ that causes a state transition in the monitor and the resulting state of this transition. The tuple may return either the key (program state index) as in line 7 or the value (resulting monitor state) as in line 15. $StartIndex$ points to the first program state to be processed in the current iteration (0 in the beginning).
2) *(Parallel computation of next monitor state)* In lines 4-11, the algorithm computes the contents of tuple $\mathcal{I}$ in parallel. Tuple $\mathcal{I}$ is set to $\langle i, q_{result}\rangle$ only if program state s_i results in enabling a transition from current monitor state $q_{current}$ and i is strictly less than the previous value of $key(\mathcal{I})$. This way, at the end of this phase, tuple $\mathcal{I}$ will contain the left-most transition occurred in the monitor. Observe that lines 7-9 are protected as a critical section and, hence, different processing units are synchronized. Notice that this synchronization is implemented at the GPU work-group level within the same computation unit. Thus, there is no global synchronization and parallelism among work groups is ensured. Moreover, observe that the most costly task in the algorithm (i.e., predicate evaluation in Line 5) is not in the critical section and executes in parallel with other such evaluations. This also means that the variable q_{result} is not shared between different threads and each thread has its own local copy of q_{result}. All other variables are shared.

Algorithm 1 For finite-history LTL_3^{mon} properties

Input: A monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, a state $q_{current} \in Q$, and a finite program trace $\sigma = s_0 s_1 s_2 \dots s_n$.
Output: A state $q_{result} \in Q$ and $\lambda(q_{result})$.

```
    /* Initialization */
 1: StartIndex ← 0
 2: I ← ⟨n + 1, q_current⟩
 3: Let m be a mutex

    /* Parallel computation of next monitor state given the current state */
 4: for all (s_i, StartIndex ≤ i ≤ n) in parallel do
 5:     q_result ← δ(q_current, s_i)
 6:     lock(m)
 7:     if (q_current ≠ q_result ∧ i < key(I)) then
 8:         I ← ⟨i, q_result⟩
 9:     end if
10:     unlock(m)
11: end for

    /* Obtaining the result */
12: if key(I) = n + 1 then
13:     return q_result, ?
14: else
15:     q_result ← value(I)
16:     if λ(q_result) ≠ ? then
17:         return q_result, λ(q_result)
18:     end if
19:     q_current ← q_result
20:     StartIndex ← StartIndex + key(I) + 1
21:     goto line 2
22: end if
```

3) *(Obtaining the result)* The third phase of the algorithm computes the final state of the monitor. If key of $\mathcal{I}$ is set to the initial value $n+1$, then no transition of the monitor gets enabled by program trace σ. In this case, the algorithm terminates and returns $(q_{result}, ?)$ as output (Line 13). Otherwise, if a change of state in the monitor occurs and results in a conclusive state, then the algorithm returns this state and the monitoring verdict (line 17). Transition to an inconclusive state yields update of both $q_{current}$ and $StartIndex$ and a new iteration (lines 19-21).

For illustration, consider property $\varphi_1 \equiv p \wedge (q \mathbf{U} r)$ and its monitor in Figure 3. Let the current state of the monitor be $q_{current} = q_0$ and the input program trace be $\sigma = (p \wedge q \wedge \neg r) \cdot (\neg p \wedge q \wedge \neg r) \cdot (p \wedge q \wedge \neg r) \cdot (p \wedge \neg q \wedge \neg r)$. By applying this trace, first, the monitor makes a transition to state q_1 after meeting $p \wedge q \wedge \neg r$. Although the next two program states (i.e., $(\neg p \wedge q \wedge \neg r) \cdot (p \wedge q \wedge \neg r)$) of the input trace also cause program state transitions, the algorithm considers only the left-most transition, thus the first iteration records transition to q_1 and the second iteration starts from s_1. Finally, during the second iteration only the last program state $p \wedge \neg q \wedge \neg r$ enables the monitor transition from q_1 to $q_\perp$.

In the context of this example, it is straightforward to observe that algorithm performs 2 iterations, which is precisely $||\mathbb{H}^{p \wedge (q \mathbf{U} r)}||$. In general, the number of algorithm iterations is equal to the number of state transitions in the underlying

monitor automaton. Thus, the complexity of the algorithm for verifying m properties is $\mathcal{O}(\max\{||\mathbb{H}^{\varphi_i}|| \mid 1 \leq i \leq m\})$. Following Theorem 1, the monitor constructed from a finite-history property does not contain a cycle. In other words, the number of monitor state transitions during (and, hence, before reaching a conclusive state) is bounded by $||\mathbb{H}^{\varphi}||$. Since there can be a maximum of $||\mathbb{H}^{\varphi}||$ algorithm iterations, this number may be less if an input trace imposes a shorter path from $q_{current}$ to a conclusive state of the monitor. In case of the infinite history length, there are two conditions that combined together may cause excessive iterations and lead to performance degradation of the algorithm: (1) existence of a self-loop in $\mathcal{M}^{\varphi}$, and (2) sequence of the program states that results in constant stuttering in the monitor automaton. The Algorithm presented in Subsection IV-C addresses these two conditions.

The execution time of the algorithm on one program trace can be described by the following formula:

$$T_1 = t_{mt_1} + E(n).\max\{\sum_{j=1}^{m}(t^i_{calc} + t^{i,j}_{branch}) \mid 1 \leq i \leq n\} \quad (3)$$

where m is the number of properties and t_{mt_1} is the memory transfer time to the processing unit (which is negligible in APUs). $E(n)$ is the expected number of the monitor state transitions (and algorithm iterations consequently) per n data items.

C. Parallel Algorithm 2

The second algorithm is an adaptation of the algorithm in [11] for parallel execution of deterministic finite state machines. This algorithm does not tackle the issues with causal order of state changes directly. Instead, it calculates possible state transitions for every monitor state (except the conclusive states, since they are traps), regardless of the actual current state of the monitor. Then, in a sequential phase the algorithm aggregates all the results to compute one and only one currenloop traversingt state. This sequential run reflects the transitions in the underlying monitor that are identical to those caused by sequential processing. Consequently, this algorithm does not depend on the history length of a property.

Now, we describe the algorithm in more detail. Similar to Algorithm 1, it takes a finite program trace $\sigma = s_0 s_1 \dots s_n$, a monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, and a *current* state $q_{current} \in Q$ (possibly q_0) of the monitor as input and returns a resulting state q_{result} and the monitoring verdict $\lambda(q_{result})$ as output. For parallelization, we leverage a well-known exhaustive execution method, where we eliminate the input interdependencies by considering all possible outputs. The algorithm works as follows:

1) *(Initialization)* First, the algorithm computes all inconclusive states of the monitor. We do not consider conclusive states, since they always act as a trap. In order to eliminate input interdependencies, we maintain a lookup matrix $\mathcal{A}$ for storing intermediate results, for each state of the monitor and a program state from trace σ (see Figure 5).
2) *(Parallel exhaustive state computation)* In lines 3-7, the algorithm computes the columns of matrix $\mathcal{A}$ in parallel. Each element of $\mathcal{A}$ is calculated using a monitor state and a program state. Note that a program state identifies the set of atomic propositions and, hence, the letters in Σ that hold in that state. Although calculation of step $i+1$ depends on the output of step i, by calculating and storing q_{i+1} for all possible q_i, this interdependency can be eliminated (i.e., for every s_i and q_j, we calculate element $\mathcal{A}_{q_j,s_i} = \delta(s_i, q_j)$).
3) *(Actual state computation)* The third phase of the algorithm consists of a sequential pass of length at most n over the columns of matrix $\mathcal{A}$. Initially, the resulting state is $q_{current}$. The output of step i is the content of $\mathcal{A}_{q_{result},s_i}$. Notice that in this step, the value of q_{result} changes only if a transition to a different state of the monitor is enabled. This way, the algorithm starts from state $q_{current}$ and terminates in at most n steps. The value of q_{result} reflects every change of the state of the monitor when the algorithm executes lines 8-14. In Figure 5, for example, this step-by-step sequential evaluation jumps from $q_{current} = q_0$ to $q_{result} = q_{|Q_?|}$ after the second step and finally concludes in $q_{result} = q_1$. At any point, if q_{result} happens to be a conclusive state, then the algorithm terminates and returns the verification verdict of the

Algorithm 2 For infinite-history LTL_3^{mon} properties

Input: A monitor $\mathcal{M}^{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, a state $q_{current} \in Q$, and a finite program trace $\sigma = s_0 s_1 s_2 \dots s_n$.
Output: A state $q_{result} \in Q$ and $\lambda(q_{result})$.

```
   /* Initialization */
1: Q? = {q ∈ Q | λ(q) =?}
2: Let A be a |Q?| × n matrix

   /* Parallel exhaustive computation of monitor states */
3: for all (s_i, 0 ≤ i ≤ n) in parallel do
4:     for all q_j ∈ Q? do
5:         A_{q_j,s_i} ← δ(q_j, s_i)
6:     end for
7: end for

   /* Sequential computation of actual monitor state */
8: q_result ← q_current
9: for all (0 ≤ i ≤ n) sequentially do
10:    q_result = A_{q_result,s_i}
11:    if λ(q_result) ≠ ? then
12:        return q_result, λ(q_result)
13:    end if
14: end for

15: return q_result, ?
```

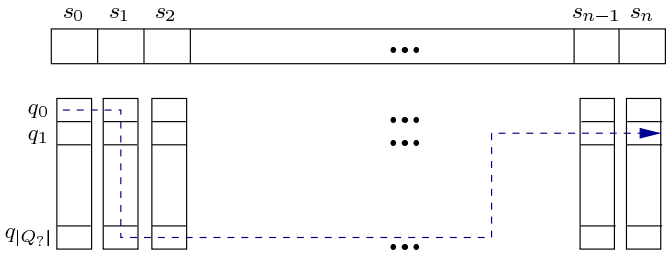

Figure 5. Eliminating interdependencies for parallel execution

monitor (lines 11-13).

Finally, if the monitor does not reach a conclusive state, then the algorithm returns the reached state by trace σ and value ?.

Unlike the first algorithm, the performance of this algorithm does not depend on the structure of the monitor, as all possible state transitions are absorbed by the final sequential pass. On the other hand, the extra calculations undertaken for every inconclusive state, as well as memory transfer of the results of those calculations back to the host process on the CPU, add to the execution time. The complexity of the algorithm now depends on the size of the input and the number of inconclusive states in the underlying monitor: $\mathcal{O}(n.\sum_{i=1}^{m}|Q_?|_i)$. More precisely, the total execution time of the algorithm can be described by the following equation:

$$T_1 = t_{mt_1} + t_{mt_2} + t_{seq} + \max\{\sum_{j=1}^{m}(t_{calc}^{i,j} + \sum_{k=1}^{|Q_?|_j} t_{branch}^{i,j,k}) \mid 1 \leq i \leq n\} \quad (4)$$

where m is the number of properties, t_{seq} is the time spent in the sequential phase (lines 8 to 14), and t_{mt_1} and t_{mt_2} are memory transfer times to and from processing unit respectively. Again, if a parallel evaluation takes place on a CPU or the program trace resides in the shared memory (e.g., in APUs), then t_{mt_1} and t_{mt_2} are negligible.

V. Implementation and Experiments

This section is organized as follows. Subsection V-A presents the implementation issues. Subsection V-B analyzes the throughput and scalability of GPU-based monitors. Subsection V-C presents a case study for measuring the runtime overhead and power consumption of our approach. We do not compare our approach with the other methods, as current monitoring frameworks use sequential monitors, whereas our approach incorporates the GPU technology.

A. Implementation

The monitor code is specific to each property under evaluation. Thus, in our implementation, for each LTL_3^{mon} property in the given specification, we automatically generate the monitor OpenCL *kernel* code by mapping the program state variables onto the predicate alphabet. The kernel generator uses the implementation of the monitor building algorithm by incorporating the tool presented in [10]. A specific command-line parameter controls which of the three algorithms (1, 2, or sequential) will be the core of the kernel code.

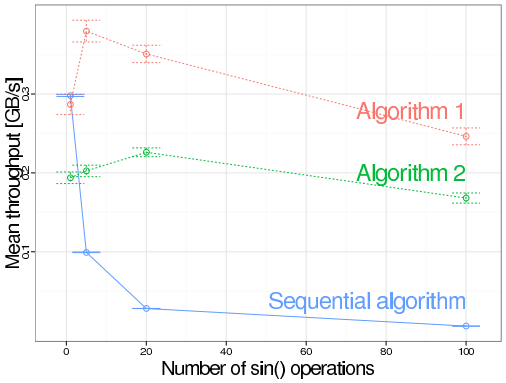

Figure 6. Throughput analysis - effect of computational load.

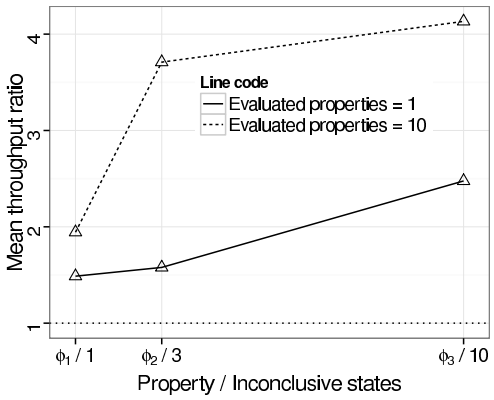

Figure 7. Throughput analysis - comparison of the algorithms.

Most of our implementation issues were related to tuning the generated kernel code, as the performance of GPU-based execution is highly sensitive to changes in the OpenCL kernel code. In particular, we took the following into account:

- Heterogeneous memory hierarchy promotes local synchronization. This became especially obvious in using the function `atomic_min()` for critical section protection when evaluating the left-most state transition in Algorithm 1. If `atomic_min()` is applied to the global state transition index, it slows down Algorithm 1 by approximately 30%. Thus, we reduced the problem domain to the local memory. After every work-group of 256 threads evaluated its left-most state transition, one thread passes over the results and picks the first-

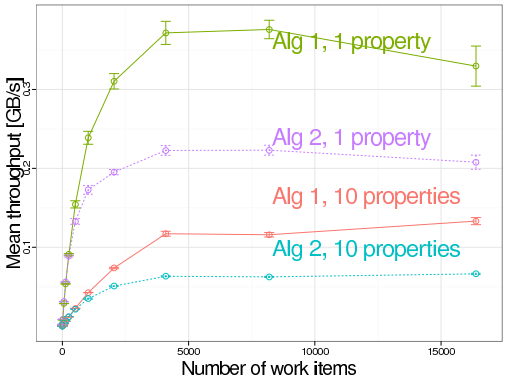

Figure 8. Throughput analysis - throughput vs. the number of work items.

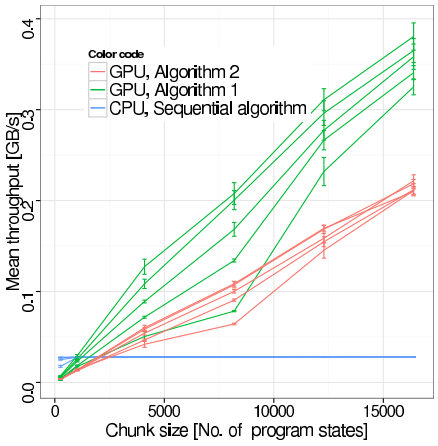

Figure 9. Throughput analysis - effect of the buffer and data sizes.

in-order transition. This way, we also ensure deadlock freedom.

- Another implementation challenge is related to scheduling. When a worker thread is blocked or is waiting for completion of an I/O operation, it may automatically switch to the next work-item to continue its work. We leverage this feature by assigning several work-items to the same thread (depending on the number of items in the work buffer). Obviously, this will only work if the number of program states in the verification buffer is higher than the number of worker threads available.
- In AMD APUs, memory transfer between CPU and GPU tends to be faster, as they reside on the same die. The specific location of the memory allocation is controlled by a set of memory flags when calling the allocation function `clCreateBuffer()`. Specifically, we observed that allowing kernels to allocate host-accessible memory and to cache the buffer contents pointed to by host pointer result in performance gain in Algorithms 1 and 2.

Unlike the parallel algorithms, the sequential algorithm runs exclusively on CPU. Thus, to allow faster CPU processing and compiler optimizations, we chose to implement this algorithm in C and not in OpenCL. In addition, as mentioned in Subsection II-B, a monitor FSM is minimized in size. These features ensure fair comparison between the sequential and the parallel algorithms.

Implementation of Algorithms 1 and 2 are incorporated in the tool RiTHM[2]. This tool takes a C program and a set of LTL properties as input and generates an instrumented C program that is verified at run time by a time-triggered monitor. The implementation of the algorithms presented in this paper are wrapped in a shared library with portable interface, which in turn, serves as a verification backend for RiTHM.

B. Throughput and Scalability Analysis

In the context of monitoring, throughput is the amount of data monitored per second. In the following experiments, the program trace is fed to the monitor directly from the main memory, thus maximizing throughput of the algorithms. All the experiments in this subsection are conducted on 32-bit Ubuntu Linux 11.10 using a 8x3.07GHz Intel Core i7 and 6GB main memory and an AMD Radeon HD5870 graphics card as GPU.

1) Effect of the Computational Load: From now on, we refer to *computational load* as the number of evaluations of all mathematical and arithmetic operations in order to evaluate atomic propositions of a property. Computational load has direct effect on the sequential monitoring time, as it is performed by the CPU. However, in our GPU-based approach, the computational load is spread over GPU cores. We hypothesize that our approach will outperform a sequential monitoring technique.

In order to validate our hypothesis, we emulate different computational loads on property $\varphi = \Box(\neg a \vee \neg b \vee \neg c \vee \neg d \vee \neg e)$, where $\|\mathbb{H}^{\varphi}\| = 1$. Notice that the number and the type of the properties are not essential, as the goal is to isolate the load factor. The loads are 1x, 5x, 20x, and 100x, where 'x' is one $\sin()$ operation on a floating point operand. We run our experiments for the parallel algorithms on 1600 cores available on the GPU and measure the throughput of three algorithms (i.e., a sequential algorithm as well as Algorithms 1 and 2 from this paper). Figure 6 shows the results of this experiment (the error bars show the 95% statistical confidence interval). As can be seen, both Algorithms 1 and 2 outperform the sequential algorithm as the computational load increases. For instance, for 100 $\sin()$ operations Algorithms 1 outperforms the sequential algorithm by a factor of 35. Thus, as Equations 3 and 4 in Section IV-B and Section IV-C suggest, there is a loose dependency between our parallel algorithms' throughput and

[2]To access the tool, please visit http://uwaterloo.ca/embedded-software-group/projects/rithm.

the computational load. This observation along with the graph in Figure 6 validates that the computational load is not a bottleneck for our algorithms. On the contrary, the sequential algorithm performance is poorer than the linear dependency on the computational load in Equation 2 in Section IV-A.

2) Performance Analysis of Algorithms 1 and 2: We hypothesize that Algorithm 1 would outperform Algorithm 2, as Algorithm 2 evaluates the next state of *all* inconclusive states in the monitor. In addition, Algorithm 1 does not transfer the calculated states back to the CPU (recall that lines 8-14 of Algorithm 2 run on the CPU). The only slowdown factor that appears in Equation 3, which does not appear in Equation 4 is $E(n)$. Given the finite number of monitor state transitions, this factor becomes negligible as the monitoring time increases.

The experimental setting consists of the following input factors: algorithm type, number of properties for monitoring, and the number of inconclusive states in the monitor of a property. This is due to the fact that the number of inconclusive states affects the amount of calculations the second algorithm should carry (see Equation 4). The size of the program trace is 16,384 states. We consider three different properties. Two of the properties are taken from the set of properties in [12] for specification patterns: $\varphi_1 \equiv \Box\neg(a \vee b \vee c \vee d \vee e)$ and $\varphi_2 \equiv \Box((a \wedge \Diamond b) \Rightarrow ((\neg c)\,\mathbf{U}\,b))$, where the number of inconclusive states are 1 and 3, respectively. The monitors of properties in [12] are relatively small (the largest monitor has five states). Thus, our third property is the following (rather unrealistic) formula whose monitor has 10 inconclusive states:

$$\varphi_3 \equiv \underbrace{\bigcirc\bigcirc\ldots\bigcirc}_{\text{10 next operators}}(a\,\mathbf{U}\,b)$$

Figure 7 confirms the hypothesis. We emphasize that the graphs in the figure represent the *ratio* of the throughputs of Algorithm 1 over Algorithm 2. The x-axis represents the number of inconclusive states in the monitored property (1 for φ_1, 3 for φ_2, and 10 for φ_3). The solid line shows the experiments for one property and the dashed line plots the experiments for a conjunction of ten identical properties. Algorithm 1 consistently performs better on all of the properties. The ratio only grows when the number of inconclusive states in the property increases.

3) Scalability: As illustrated in Figure 1, the host submits chunks of the program trace to the monitoring tasks running on the GPU. A *chunk* is the array containing a sub-trace to the GPU internal buffers. Once the GPU is available, the host process will pull the next task from the queue and will run the task on the input data following the Single Instruction Multiple Data (SIMD) pattern. The GPU also implements the internal scheduling mechanism that controls the process of assigning the data to the GPU cores. However, input array can be represented as a number of *work items*. Thus, by setting this number to one, we ensure that only one core participates in the evaluation. Consequently, by increasing the number of work items to the number of items in the input data, we control the number of cores engaged in monitoring. If the chunk size is greater than the number of cores, then at some point the GPU scheduler will have to assign several work items to the same core.

In this experiment, we fix all contributing factors as constants (including the chunk size of $16,384$) and only change the monitoring algorithm, the number of work items, and the number of properties for monitoring. Figure 8 shows that the algorithms are clearly scalable with respect to the number of cores. The error bars represent the 95% confidence interval. The graphs also show that the mean throughput increases with the number of cores engaged in monitoring. At some point, both algorithms reach the optimum, where all the cores are utilized. From that point and on, the throughput will be mostly affected by the GPU thread scheduler.

4) Effect of Data and Chunk Sizes on Throughput: The factors of this experiment are the algorithm type, the number of simultaneously-evaluated properties, and the amount of data in each run (see Figure 9). The error bars indicate the 95% confidence interval. The line color indicates the type of algorithm. The five lines for each algorithm show results for different amounts of data (i.e., 10, 20, 30, 40, and 50 data chunks). Figure 9 shows that sequential monitoring is neither influenced by the chunk size nor the amount of data. The mean throughput of the parallel monitoring shows a strong correlation with the chunk size. This is expected, since an increase of the chunk size results in a higher number of the program states processed in parallel. This, in turn, leads to the more efficient distribution of the work items among the cores. However, note that this throughput gain is limited by the amount of memory available in the GPU and the tolerable delay of the program. The minor differences between the results with different amounts of data show the independence of the mean throughput from the amount of data. This is expected behavior and increases our confidence in the results.

C. Case Study: GPU-based Runtime Monitoring of a UAV

This section describes additional benefits of using GPU-based monitoring. In our case study, the monitor checks at run time a flying unmanned aerial vehicle (UAV) autopilot software running on Beagle Board with QNX 6.5 OS. The monitor runs on an ASUS E35-M1 board containing dual-core AMD E350 APU and Ubuntu 11.10 OS. Another process running alongside with monitor is LIDAR - a grid reconstruction algorithm - that emulates the load of the autopilot[3]. The scale of the software is as follows:

[3]A video clip of the actual experiment is available at http://www.youtube.com/watch?v=Db2MifLmap0&feature=youtu.be.

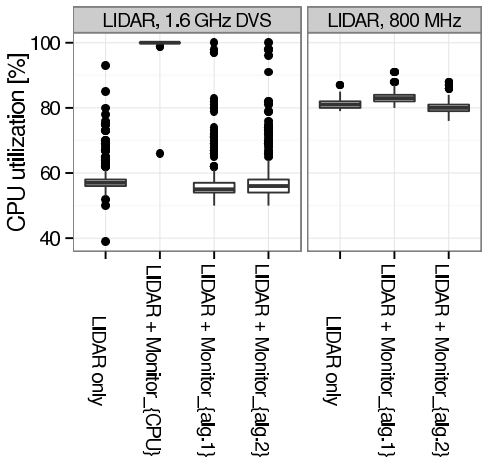

Figure 10. UAV case study - CPU utilization.

	Type	Name	Avg Watt
1	Nothing	Idle	15.56
2	No LIDAR	CPU	20.20
3		GPU (Alg 1)	15.82
4		GPU (Alg 2)	15.72
5	LIDAR, 1.6 GHz DVS	Base	27.81
6		CPU	28.80
7		GPU (Alg 1)	28.73
8		GPU (Alg 2)	28.04
9	LIDAR, 800MHz	Base	22.29
10		GPU (Alg 1)	23.00
11		GPU (Alg 2)	22.42

Figure 11. UAV case study - power consumption.

- Size of Monitor: 52K
- Size of Autopilot: 78K
- Lines of code in Monitor: 3000
- Lines of code in Autopilot: 4000

Every 10ms, the autopilot sends the program state consisting 14 float numbers to the monitor over a crossover Ethernet cable using the UDP protocol. The host thread in the monitor interpolates the data and fills up a buffer for inspection. The GPU then processes the buffer using one of the three evaluation algorithms. We verify the following five properties at runtime:

- $\varphi_1 \equiv \Box(a \wedge b \wedge c)$
  This property verifies the sanity of the IMU (Inertial Measurement Unit) sensors during the flight. IMU measures the values of pitch, yaw, roll, and their velocities. The property compares the numerical values of the angular acceleration from the sensors with calculated analytical values from the two consecutive program states. Propositions a, b, and c express that the calculations of the three dimensions are within certain ranges.

- $\varphi_2 \equiv \Box(d \Rightarrow (\bigcirc\neg d) \vee (\bigcirc\bigcirc\neg d) \vee (\bigcirc\bigcirc\bigcirc\neg d))$
  Proposition d stands for "the number of satellites is less than three". The property expresses that this undesirable proposition will not last for more than four consecutive program states.

- $\varphi_3 \equiv \Box(e)$
  This property verifies whether the autopilot translates the coordinates of latitude, longitude, and altitude correctly. Proposition e expresses the conditions on the transformation function.

- $\varphi_4 \equiv \Box(f \Rightarrow \Diamond g)$
  This property verifies that the UAV eventually lands. Proposition f stands for 'altitude higher than 500' and proposition g stands for 'altitude less than 358' - the altitude above the sea level on the university campus.

- $\varphi_5 \equiv \Box(h \wedge i \wedge j \wedge k \wedge l)$
  This property verifies the requirements of Transport Canada Agency stated in Safety Flight Operation Certificate Requirements document: the flight is permitted only within specific coordinates on the campus of the University of Waterloo.

If one of the properties is violated/satisfied, the monitor will send a UDP packet indicating the property that has converged, back to the autopilot, which notifies the ground station.

The experiment consists of two parts: first part reports the CPU utilization measurements (see Figure 10) and the second part reports the power consumption during the flight (Table 11). During the flight monitoring, we do not measure execution time. Rather, we measure CPU utilization. As Figure 10 shows, LIDAR alone consumes about 57% of CPU time when running the system with default *ondemand* policy. When this CPU scheduling policy is on, the OS switches the CPU to highest frequency upon demand and back to the lowest frequency after the task is done. While switching the frequencies, Dynamic Voltage Scaling (DVS) takes place. This explains the outliers in the GPU-based evaluation algorithms, which otherwise barely increase the regular consumption. On the contrary, running the CPU-based (sequential) monitoring, results in utilizing practically all the CPU resources. In safety critical systems with real-time constraints, this can lead to the missing cycles, which is obviously problematic. When GPU-based monitoring is on, the amount of idle CPU time indicates that the frequency can be lowered without risk of missing cycles. Second part of the graph shows the system running on the *userspace* scheduling policy with constant frequency of 800 MHz. The utilization rises to 81%, but never reaches 100%. Figure 10 also shows that the overhead of our GPU-based monitoring is negligible.

Figure 10 closely correlates with Table 11 that reports the average of the power consumption measurements over the

time. E350 APU board is given constant power supply of 15 Volts and by recording the current level, we calculate the consumed power. Rows 2-4 report the consumption of the system without background load of LIDAR, while rows 5-8 show close consumption levels with LIDAR running. Although without background load the GPU-base monitoring consumes 25% less power than CPU-based, this approach is not practical, as we aim at running the monitor and the inspected program on the same board. To deal with this issue, we set the constant CPU frequency to 800 MHz (rows 9-11 in the table correspond to the second part of Figure 10). In addition to solid 25% reduction in power consumption, this gives us predictable CPU utilization with minimum outliers.

VI. Related Work

There is a body of work that aims at reducing the runtime overhead through an additional pre-processing phase. In [6], the authors use static analysis to reduce the number of residual runtime monitors. The framework in [13] statically optimizes monitoring aspects. Although the objective of this paper is also reducing the runtime overhead, GPU-based runtime monitoring targets shipping the monitoring costs to a different processing unit. Thus, our approach is orthogonal to the existing approaches in runtime monitoring.

The main focus in the literature of runtime verification is on *event-triggered* monitors [14], where every change in the state of the system triggers the monitor for analysis. Alternatively, in *time-triggered* monitoring [9], the monitor samples the state of the program under inspection in regular time intervals. The latter approach involves an optimization problem that aims at minimizing the size of auxiliary memory needed for state reconstruction at sampling time. All these works, however, utilize sequential monitoring techniques. Parallel monitoring has been addressed in [15] to some extent by focusing on low-level memory architecture to facilitate communication between application and analysis threads. This approach is not quite relevant to the propose method in this paper.

The concept of separation of a monitor from the monitored program is also considered in [16], [17], [18]. However, to the best of our knowledge, our work is the first that combines the following: (1) using non-dedicated parallel architecture (GPU), (2) systematic and formal verification of properties at runtime in a parallel fashion. Specifically, in [16] the authors suggest using a dedicated co-processor as a parallel profiling architecture. This co-processor implements some common profiling operations as hardware primitives. Similar to our method, instead of interrupting the processor after every sample, multiple program samples are pre-processed and buffered into a $1K$ buffer before processing. The work in [16], [18] also concentrate on hardware-based monitoring and, hence, the need in a dedicated hardware. On the contrary, our approach utilizes the available many-core platform (GPU or multi-core CPUs) in a computing system.

Finally, in [19] the authors propose using GPU for runtime monitoring, but their approach is limited to data race detection scenarios and not a general formal specification language such as LTL. Consequently, no systematic approach is proposed for generating the code that runs on GPU. For instance, to run experiments the authors have parallelized ERASER and GOLDILOC algorithms and adjusted them to the syntax of CUDA. In addition, the authors do not resolve interdependencies in data frames. More importantly, their approach is not sound and may result in obtaining false positives.

VII. Conclusions

In this paper, we proposed a technique to leverage parallel processing for accelerating and isolating monitors in runtime verification. Our idea is that the program under scrutiny accumulates events that need to be examined by the monitor as a program trace and hands over this trace to the monitor for inspection. The size of the program trace and frequency of triggering the monitor depends on design parameters, such as monitoring scheduling and tolerable latency for detecting a specification violation. Our technique achieves two levels of parallelism: (1) external parallelism between the program and the monitor, and (2) internal parallelism, where a set of monitoring threads take a part of the program trace and verify a set of properties of interest in a parallel fashion. While both levels of parallelism indeed reduce the runtime overhead, the former is particularly valuable in separating the monitoring from functional concerns.

Our formal language to specify logical properties is the 3-valued LTL [10], an extension of the standard linear temporal logic over finite words. We tackled the main technical challenge (i.e., respecting the causal order of events when verifying properties by concurrent threads) by proposing a notion of property history. We also introduced two parallel algorithms for runtime monitoring of LTL_3 properties. However, our parallel algorithms are suitable for any logical formalism where the monitor can be represented by a finite state automaton. Through detailed experiments, we demonstrated that our technique is indeed scalable and limits the runtime overhead effectively. Although our technique performs better on properties that require some level of computational load, other systems with strict power and CPU utilization constraints may benefit as well.

For future work, several research directions exist. We are currently working on extending our algorithms to handle parameterized properties. In addition, identifying fragments of LTL_3 with respect to different history lengths may improve the algorithms performance. Furthermore, one can apply our notion of property history to make distributed monitoring possible in cloud computing. Another open problem is to

extend the results of this paper in the context of timed temporal logics runtime verification for real-time systems.

VIII. Acknowledgement

This research was supported in part by NSERC DG 418396-2012, NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, CFI 20314, CMC, and the industrial partners associated with these projects.

References

[1] S. Colin and L. Mariani, *Run-Time Verification*. Springer-Verlag LNCS 3472, 2005, ch. 18.

[2] A. Pnueli and A. Zaks, "PSL Model Checking and Run-Time Verification via Testers," in *Symposium on Formal Methods (FM)*, 2006, pp. 573–586.

[3] D. Giannakopoulou and K. Havelund, "Automata-Based Verification of Temporal Properties on Running Programs," in *Automated Software Engineering (ASE)*, 2001, pp. 412–416.

[4] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for java," in *Tools and Algorithms for the construction and analysis of systems (TACAS)*, 2005, pp. 546–550.

[5] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok, "Aspect-oriented instrumentation with GCC," in *Runtime Verification (RV)*, 2010, pp. 405–420.

[6] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," in *International Conference on Software Engineering (ICSE)*, 2010, pp. 5–14.

[7] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *Software tools for technology transfer (STTT)*, vol. 14, no. 3, pp. 327–347, 2012.

[8] P. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," *Journal of Automated Software Engineering*, vol. 17, no. 2, pp. 149–180, June 2010.

[9] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *Formal Methods (FM)*, 2011, pp. 88–102.

[10] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[11] J. Holub and S. Stekr, "On parallel implementations of deterministic finite automata," in *Implementation and Application of Automata (CIAA)*, 2009, pp. 54–64.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *International Conference on Software Engineering (ICSE)*, 1999, pp. 411 –420.

[13] E. Bodden, P. Lam, and L. Laurie, "Clara: A framework for partially evaluating finite-state runtime monitors ahead of time," in *Runtime Verification (RV)*, 2010, pp. 183–197.

[14] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," in *Computer Aided Verification (CAV)*, 1999, pp. 172–183.

[15] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multicore hardware," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009, pp. 155–174.

[16] C. B. Zilles and G. S. Sohi, "A programmable co-processor for profiling," in *High Performance Computer Architecture (HPCA)*, 2001, pp. 241–253.

[17] H. Zhu, M. B. Dwyer, and S. Goddard, "Predictable runtime monitoring," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2009, pp. 173–183.

[18] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable COTS-based real-time embedded systems," in *Real-Time Systems Symposium*, 2008, pp. 481 –491.

[19] T. Elmas, S. Okur, and S. Tasiran, "Rethinking runtime verification on hundreds of cores: Challenges and opportunities," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-74, June 2011.