GAIN: GPU-based Constraint Checking for Context Consistency

Jun Sui[†], Chang Xu[*‡], Wang Xi, Yanyan Jiang, Chun Cao[‡], Xiaoxin Ma[‡], Jian Lu[‡]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
[†]smilent_sj@163.com, [‡]{changxu, caochun, xxm, lj}@nju.edu.cn, swangex@gmail.com, jiangyy@outlook.com

***Abstract*—Applications in pervasive computing are often context-aware. However, due to uncontrollable environmental noises, contexts collected by applications can be distorted or even conflicting with each other. This is known as the context inconsistency problem. To provide reliable services, applications need to validate contexts before using them. One promising approach is to check contexts against consistency constraints at the runtime of applications. However, this can bring heavy computations due to tremendous amounts of contexts, thus leading to deteriorated performance to applications. Previous work has proposed incremental or concurrent checking techniques to improve the checking performance, but they heavily rely on CPU computing. In this paper, we propose a novel technique GAIN to exploit GPU computing to improve the checking performance. GAIN can automatically recognize parallel units in a constraint and schedule their checking in parallel on GPU cores. We evaluated GAIN with various constraints under different workloads. Our evaluation results show that, compared to CPU-based computing, GAIN saves CPU computing resources for pervasive applications while checks constraints much more efficiently.**

***Keywords*—GPGPU, constraint checking, pervasive computing**

I. Introduction

Applications in pervasive computing provide context-aware services. Their contexts are collected from environments, which are naturally imperfect. Thus contexts can be distorted or even conflicting with each other. This is known as the *context inconsistency* problem [14]. One promising approach to validating contexts is to check them against pre-specified consistency constraints, which describe necessary properties that must hold about contexts [14]. If any constraint is violated, inconsistency is said "detected". Detected inconsistencies should be resolved before related contexts are used by applications, because otherwise applications can behave abnormally due to such inconsistent contexts [4]. Typically, constraint checking needs to be conducted once any new context is collected, and this checking has to be efficient for applications' context-awareness and timeliness requirements.

Consider a location-aware application that provides smart routing services based on real-time traffic conditions in a city. Each taxi is installed with a GPS sensor to collect its context, which includes the taxi's id, current location, instant speed, service status, and so on. Such context information is sent to a central server continually. Sometimes the context information can be inaccurate or even wrong. For example, GPS sensing can be unreliable when a taxi is near a metal building, and data packets containing context information may be lost during wireless transmission in an underground tunnel. Thus contexts collected by the central server need to be validated before use. Consider an obviously wrong context indicating that a taxi is driving in sea adjacent to this city. To detect this problem, a simple consistency constraint can be specified as: "*any taxi must drive within the scope of the city*" (we omit cross-city traffic for ease of discussion in this paper). Then any collected context indicating a taxi's location beyond the scope of the city would violate this constraint and cause an inconsistency. Besides this simple constraint, the application may also specify complex constraints that concern relationships among multiple contexts (concerning different taxis or the same taxi but during a period of time). Considering that there can be thousands of taxis or more in the city and each taxi sends its context continually to the server, the constraint checking is conducted very frequently and must be very efficient.

Typically, the performance of constraint checking can be improved by incremental checking [13] or concurrent checking [15]. However, these techniques rely fully on CPU computing, which can consume valuable computing resources. These resources should originally be used for context-aware computing by pervasive computing applications. On the other hand, many modern machines own GPU computing facilities. GPU has been designed for concurrently processing pixels on screen. A screen usually contains hundreds of thousands of pixels, and thus GPU is naturally able to process tasks in a high degree of parallelism. We conjecture exploiting GPU computing to support constraint checking. However, GPU architecture is very different from CPU architecture and their programming is totally different. Therefore, GPU-based constraint checking should be carefully re-designed from CPU-based one. In this paper, we propose a novel constraint checking technique called GAIN, which stands for GPU-bAsed consIsteNcy checking. It can automatically recognize parallel computing units in a constraint and distribute them across different GPU cores such that constraint checking can be completed in a high degree of parallelism. Besides, GAIN is independent of a constraint's structure and totally transparent to applications. We evaluated GAIN with various constraints under different workloads. We found that GAIN has satisfactory performance even under heavy workload. For the same consistency constraints, GAIN works much more stably than CPU-based techniques, which can deteriorate quickly when workload increases. For a case study with millions of real-life context data, GAIN saves CPU computing resources for pervasive applications while checks constraints efficiently, as compared to CPU-based ones.

[*]Chang Xu is the corresponding author.

In this paper, we make the following contributions:

- We proposed a novel technique GAIN to check constraints on GPU cores. It guarantees load balance and imposes no restriction on constraints.
- We proposed a two-level storage strategy to concurrently store irregular data, which is used to cope with different constraints that can generate various checking results on GPU cores.
- We evaluated GAIN experimentally and compared it to CPU-based techniques. The experimental results confirmed GAIN's effectiveness in greatly improving the checking performance and its applicability to different structures of constraints.

The rest of this paper is organized as follows. Section II presents background knowledge on constraint checking and GPU programming. Section III uses an illustrative example to overview our GAIN technique. Section IV elaborates on our GAIN. Section V evaluates GAIN and discusses its evaluation results. Section VI presents related work, and finally Section VII concludes this paper.

II. Background

In this section, we briefly introduce constraint checking and CUDA programming.

A. Constraint Checking

In this paper, a *context* can refer to any piece of environmental information, e.g., the location of a taxi at a certain time point. In the process of validating contexts against consistency constraints, we are interested in two questions: (1) Is a certain constraint violated (i.e., its truth value is evaluated to false)? (2) If yes, what has caused this violation? Typically, *truth value evaluation* answers the first question and *link generation* answers the second (each generated link gives a specific reason explaining the violation) [13].

For ease of presentation, we in this paper assume that consistency constraints are specified in the following *first-order logic* (FOL) based language (also used in existing work [10] [11] [13]–[15]):

$$f ::= \forall\gamma \in S(f) \mid \exists\gamma \in S(f) \mid (f)\ and\ (f) \mid (f)\ or\ (f) \mid (f)\ implies\ (f) \mid not(f) \mid bfunc(\gamma_1, \gamma_2, ..., \gamma_n).$$

In this language, S represents a set of contexts and variable γ can take any context from S. Terminal $bfunc$ can be any user-defined function or application-specific function that returns either true or false based on values of its parameters.

Consider our earlier location-aware application. Due to its speed limit, a taxi cannot drive a too long distance within a short period of time. This physical law can be specified as a consistency constraint as follows:

$$\forall taxi_1 \in CITY(\forall taxi_2 \in CITY(Same(taxi_1, taxi_2)\ implies\ Loc(taxi_1, taxi_2))). \quad (*)$$

In this constraint, $CITY$ represents the set of location contexts collected in the last T seconds by the central server (can come from different taxis). The $Same$ function judges whether two contexts concern the same taxi. The Loc function calculates the distance between these two contexts' locations and judges whether this distance is reasonable for duration T with respect to the speed limit.

Each consistency constraint can be represented by a syntax tree [13]. A syntax tree shows the hierarchical structure of a constraint. For example, the syntax tree of the constraint specified by Equation (*) is illustrated in Fig. 1.

In constraint checking, a syntax tree can be associated with contexts under checking to form a runtime tree [15]. In a runtime tree, a formula like $\forall\gamma \in S(f)$ or $\exists\gamma \in S(f)$ can generate multiple branches, whose number is equal to the number of contexts in S, and each branch is associated with a context from S. Consider the syntax tree in Fig. 1. Suppose that the context set $CITY$ contains two contexts $\{ctxt_1, ctxt_2\}$. Thus each $\forall$ node in this syntax tree's corresponding runtime tree would contain two branches, as illustrated in Fig. 2.

The truth value of a constraint is evaluated in a bottom-up manner in its corresponding runtime tree. Leaf nodes represent $bfunc$ functions and their parameters have been associated with certain contexts when the runtime tree is created. Thus, the truth value of a leaf node is its contained function's return value, and the truth value of an internal node is evaluated according to its child nodes' values and the semantics of this node's contained formula.

Links are generated for explaining why a constraint has been violated. Each link takes the form of $\{type, bindings\}$, in which $type$ can be $violated$ or $satisfied$, indicating whether the constraint is violated or satisfied, and $bindings$ is a set of variable-context mappings, disclosing what has caused the violation or satisfaction (in terms of what variables take what values) [13]. Consider the example constraint specified by Equation (*). Suppose that when variable $taxi_1$ takes value $ctxt_1$ and variable $taxi_2$ takes value $ctxt_2$, $Same(taxi_1, taxi_2)$ would hold but $Loc(taxi_1, taxi_2)$ would not, leading to the violation of the whole constraint. Then a link like $\{violated, \{(taxi_1, ctxt_1), (taxi_2, ctxt_2)\}\}$ would be generated for explaining the violation. Links can be generated in a bottom-up way on a constraint's runtime tree [10] [13].

B. CUDA Programming

GPU (*Graphic Processing Unit*) is designed for graphical processing purposes. A GPU contains an array of streaming multiprocessors (SMs) and each SM contains a group of scalar cores that are used to execute GPU threads in parallel. Typically, one GPU contains hundreds of cores (e.g., NVIDIA Geforce GT640 contains 2 SMs and each SM contains 192 cores) and each core can execute a thread individually. Thus GPU can achieve a high level of parallelism.

CUDA (*Compute Unified Device Architecture*) [12] is a parallel computing platform and programming model for general-purpose computing (*GPGPU*). It provides a C-like language called *CUDA-C*, which extends C by allowing programmers to define device functions, called *kernels*, that, when called, are executed independently by multiple CUDA threads in parallel on hundreds of GPU cores [12]. Every 32 threads are grouped as a *warp* to execute on one SM synchronously. Divergence inside a warp is allowed but it will decrease the

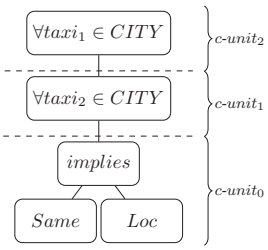


Fig. 1. An example syntax tree

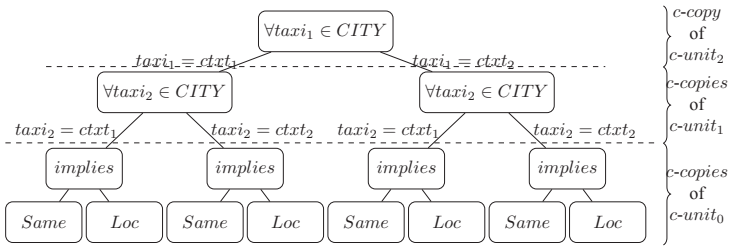


Fig. 2. An example runtime tree

performance severely, because each path will be executed serially. CUDA exposes multithreading and data parallel to users, which means that it is a user's duty to configure the number of GPU threads running for one kernel and arrange data for each thread. This is the challenging part for our GPU-based parallel constraint checking, i.e., we need to make this process transparent to upper-layer constraint checking tasks.

III. Overview

In this section, we overview our GAIN technique.

Generally, constraint checking (i.e., truth value evaluation and link generation) is realized by visiting all nodes in a constraint's corresponding runtime tree recursively (named *Seq-C*). For example, to evaluate a node's truth value, Seq-C first evaluates its children's truth values, and based on them evaluates the truth value of this node according to its associated formula's semantics. Finally, the truth value of the root node of a runtime tree indicates whether the tree's corresponding constraint is satisfied or violated. Link generation works similarly. This process is usually made sequentially (that is why it is named Seq-C). There is also attempt of checking constraints by exploiting CPU computing in parallel like Con-C [15] et al. Con-C identifies *persistently balanced splitting nodes* (PB nodes) in a runtime tree, and distributes their checking tasks to parallel computing threads.

However, Seq-C and Con-C ideas cannot be applied to GPU easily. Seq-C relies on one thread to complete all constraint checking tasks. Con-C requires different threads to cooperate. Unfortunately, no single GPU core is qualified to undertake such computing tasks as a GPU thread is much weaker than a CPU thread. Thus checking a whole runtime tree by a single GPU thread would instead decrease the performance. Besides, a runtime tree cannot be processed in a recursive manner as kernels in CUDA do not support recursion. Therefore, parallelism strategies have to be reconsidered.

Our GAIN can check constraints efficiently on GPU. It automatically extracts parallel computing units (called *c-copies*) from a constraint's runtime tree and schedules their checking in parallel on GPU cores. Fig. 2 illustrates three groups of c-copies divided by two dashed lines. For example, below the second dashed line, there are four c-copies starting with $implies$ nodes. C-copies in every group can be processed by GAIN in parallel. To avoid recursion, GAIN processes the three c-copy groups in Fig. 2 in a bottom-up manner.

When the number of contexts referenced by a runtime tree grows, the tree would become "wider". In this case, GAIN

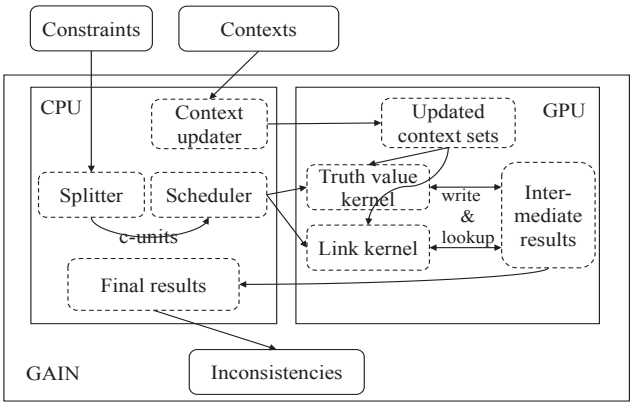


Fig. 3. GAIN architecture

(1) $split[\forall\gamma \in S(f)] \rightarrow cut(\forall\gamma \in S(f));\ split[f];$
(2) $split[\exists\gamma \in S(f)] \rightarrow cut(\exists\gamma \in S(f));\ split[f];$
(3) $split[(f_1)\ and\ (f_2)]$ |
$split[(f_1)\ or\ (f_2)]$ |
$split[(f_1)\ implies\ (f_2)] \rightarrow split[f_1];\ split[f_2];$
(4) $split[not(f)] \rightarrow split[f];$
(5) $split[bfunc(\gamma_1, ..., \gamma_n)] \rightarrow$ stop recursion.

Fig. 4. Syntax tree splitting semantics

can extract more c-copies from the tree, and then distribute more c-copies to GPU cores for parallel processing. In other words, our GAIN can work better when its constraint checking workload increases, as compared to CPU-based computing.

IV. Checking Constraints by GAIN

Fig. 3 illustrates our GAIN's architecture. GAIN takes constraints and contexts as input. Constraints are loaded into GAIN earlier for analysis. For each constraint, **Splitter** converts it into a group of c-units (Fig. 1). At runtime, whenever a new context is received, GAIN will: (1) update all concerned context sets in constraints, (2) (**Scheduler**) schedules c-units and calls **Truth value kernel** and **Link kernel** for checking these c-units' associated c-copies (Fig. 2), and (3) outputs detected inconsistencies (in terms of links) and then checks the next group of c-units (i.e., the next constraint).

A. Constraint Preprocessing

Given a consistency constraint, **Splitter** would automatically split it into a group of parallel units called *c-units*. C-

$$
\begin{array}{lll}
(1) & \tau[\forall\gamma \in S(f)]_\alpha & ::= \{t_i = lookup(\tau[f]_{bind(\alpha,(\gamma,x_i))}) \quad | \quad x_i \in S; \quad return\ \top \wedge t_1 \wedge \cdots \wedge t_m \quad \}; \\
(2) & \tau[\exists\gamma \in S(f)]_\alpha & ::= \{t_i = lookup(\tau[f]_{bind(\alpha,(\gamma,x_i))}) \quad | \quad x_i \in S; \quad return\ \bot \vee t_1 \vee \cdots \vee t_m \quad \}; \\
(3) & \tau[(f_1)\ and\ (f_2)]_\alpha & ::= lookup(\tau[f_1]_\alpha) \wedge lookup(\tau[f_2]_\alpha); \\
(4) & \tau[(f_1)\ or\ (f_2)]_\alpha & ::= lookup(\tau[f_1]_\alpha) \vee lookup(\tau[f_2]_\alpha); \\
(5) & \tau[(f_1)\ implies\ (f_2)]_\alpha & ::= lookup(\tau[f_1]_\alpha) \rightarrow lookup(\tau[f_2]_\alpha); \\
(6) & \tau[not(f)]_\alpha & ::= \neg lookup(\tau[f]_\alpha); \\
(7) & \tau[bfunc(\gamma_1, ..., \gamma_n)]_\alpha & ::= bfunc(get(\alpha, \gamma_1), ..., get(\alpha, \gamma_n)).
\end{array}
$$

Fig. 5. Truth value evaluation semantics in GAIN

$$
\begin{array}{lll}
(1) & \mathcal{L}[\forall\gamma \in S(f)]_\alpha & ::= \{l_i = lookup(\mathcal{L}[f]_{bind(\alpha,(\gamma,x_i))}) \quad | \quad x_i \in S \wedge \tau[f]_{bind(\alpha,(\gamma,x_i))} = \bot; \\
 & & \quad return(\emptyset \cup (\{(violated, (\gamma, x))\} \otimes l_1) \cup ... \cup \{(violated, (\gamma, x))\} \otimes l_m))\}; \\
(2) & \mathcal{L}[\exists\gamma \in S(f)]_\alpha & ::= \{l_i = lookup(\mathcal{L}[f]_{bind(\alpha,(\gamma,x_i))}) \quad | \quad x_i \in S \wedge \tau[f]_{bind(\alpha,(\gamma,x_i))} = \top; \\
 & & \quad return(\emptyset \cup (\{(satisfied, (\gamma, x))\} \otimes l_1) \cup ... \cup \{(satisfied, (\gamma, x))\} \otimes l_m))\}; \\
(3) & \mathcal{L}[(f_1)\ and\ (f_2)]_\alpha & ::= \begin{cases} lookup(\mathcal{L}[f_1]_\alpha) \otimes lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \top; \\ lookup(\mathcal{L}[f_1]_\alpha) \cup lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \bot; \\ lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \top \wedge \tau[f_2]_\alpha = \bot; \\ lookup(\mathcal{L}[f_1]_\alpha) & , \tau[f_1]_\alpha = \bot \wedge \tau[f_2]_\alpha = \top; \end{cases} \\
(4) & \mathcal{L}[(f_1)\ or\ (f_2)]_\alpha & ::= \begin{cases} lookup(\mathcal{L}[f_1]_\alpha) \cup lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \top; \\ lookup(\mathcal{L}[f_1]_\alpha) \otimes lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \bot; \\ lookup(\mathcal{L}[f_1]_\alpha) & , \tau[f_1]_\alpha = \top \wedge \tau[f_2]_\alpha = \bot; \\ lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \bot \wedge \tau[f_2]_\alpha = \top; \end{cases} \\
(5) & \mathcal{L}[(f_1)\ implies\ (f_2)]_\alpha & ::= \begin{cases} flipSet(lookup(\mathcal{L}[f_1]_\alpha)) \otimes lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \top \wedge \tau[f_2]_\alpha = \bot; \\ flipSet(lookup(\mathcal{L}[f_1]_\alpha)) \cup lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \bot \wedge \tau[f_2]_\alpha = \top; \\ lookup(\mathcal{L}[f_2]_\alpha) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \top; \\ flipSet(lookup(\mathcal{L}[f_1]_\alpha)) & , \tau[f_1]_\alpha = \tau[f_2]_\alpha = \bot; \end{cases} \\
(6) & \mathcal{L}[not(f)]_\alpha & ::= flipSet(lookup(\mathcal{L}[f]_\alpha)); \\
(7) & \mathcal{L}[bfunc(\gamma_1, ..., \gamma_n))]_\alpha & ::= \emptyset.
\end{array}
$$

Fig. 6. Link generation semantics in GAIN

units from one constraint are disjointed with each other and they together form this constraint. Fig. 1 shows three c-units of the example constraint specified by Equation (*) earlier. Fig. 4 gives our syntax tree splitting semantics, in which ∀/ ∃ nodes (containing universal or existential formulae) are key nodes indicating boundaries of different c-units. Function cut splits a universal or existential formula into two parts, i.e., its quantifier part and sub-formula part. The splitting recursion stops when reaching any $bfunc$ node.

With c-units, we can efficiently construct runtime trees, which are composed of c-copies. C-copies can be generated by assigning contexts to variables in a constraint. For example, the four c-copies at the bottom in Fig. 2 can be generated by associating contexts $ctxt_1$ and $ctxt_2$ to variables $taxi_1$ and $taxi_2$ in a combinatorial way.

A nice property about c-copy is that it can no longer be divided into smaller parts that are still subject to parallel processing on GPU cores. This is because each c-copy contains no branches generated by universal or existential node. Thus we can realize maximal parallelism by notion of c-unit and c-copy. It could be argued that sub-trees starting with $implies$/ and/ or nodes might also be processed in parallel. However, this would cause divergence since each branch in such sub-trees cannot guarantee to be the same. As a result, each branch has to be processed serially on GPU cores, and this instead decreases the constraint checking performance.

For each c-unit, GAIN reorders the checking sequence of its nodes by a post-order traversal to avoid recursion (recall that GPU threads do not support recursion). For example, after our constraint preprocessing, the syntax tree in Fig. 1 would be converted into the following checking sequence:

$$\{(Same, Loc, implies), (\forall taxi_2 \in CITY), (\forall taxi_1 \in CITY)\}.$$

B. Truth Value Evaluation

GAIN evaluates truth value for each runtime tree node. A node's truth value indicates whether it is satisfied or violated. Fig. 5 gives GAIN's truth value evaluation semantics:

- $\tau[f]_\alpha$: returns the truth value of f under a variable assignment α ("$\top$" stands for true or "$\bot$" for false).
- α : represents a variable assignment, containing a set of variable-context mappings, e.g., $\alpha = \{(taxi_1, ctxt_1), (taxi_2, ctxt_2)\}$ means that variable $taxi_1$ takes context $ctxt_1$ as its value and variable $taxi_2$ takes context $ctxt_2$.
- $bind(\alpha, (\gamma, x_i))$: adds a new variable-context mapping (γ, x_i) to α to form a new variable assignment.
- $get(\alpha, \gamma_i)$: returns the context bound to variable γ_i in variable assignment α.
- $lookup(\tau[f]_\alpha)$: retrieves truth value $\tau[f]_\alpha$.

Note that a constraint's syntax tree has been split and reordered in the aforementioned preprocessing phase. Thus **Truth value kernel** can work in a non-recursive (bottom-up) way. Truth values that have been evaluated earlier can be retrieved through the $lookup$ function. For example, in Fig. 2, when GAIN evaluates truth value for the $\forall taxi_2 \in CITY$ node under variable assignment $\{(taxi_1, ctxt_1)\}$, it can retrieve truth values of the two $implies$ nodes under variable assignments $(\{(taxi_1, ctxt_1), (taxi_2, ctxt_1)\}$ and $\{(taxi_1, ctxt_1), (taxi_2, ctxt_2)\}$, respectively. Based on them, GAIN can evaluate truth value for this $\forall taxi_2 \in CITY$ node. The implementation of the $lookup$ function relies on our storage strategy, which we explain later.

For each c-copy derived from the same c-unit, **Truth value kernel** arranges one GPU thread for processing it. The number of c-copies from one c-unit can be calculated according to the number of contexts in its associated context sets. For example, in Fig. 1, $c\text{-}unit_0$ references two variables in a combinatorial way and they both associate with context set $CITY$, which contains two contexts. Then $c\text{-}unit_0$ is four. Each GPU thread processes nodes according to our truth value evaluation semantics in Fig. 5. Since these c-copies are independent of each other, they can be processed in parallel. Besides, these c-copies are structurally identical, and therefore **Truth value kernel** can process in a way free of divergence (i.e., real parallel).

C. Link Generation

Fig. 6 gives our link generation semantics in GAIN:

- $\mathcal{L}[f]_\alpha$: returns a set of links generated for formula f under variable assignment α, explaining why f is satisfied or violated.
- $\otimes$: makes a Cartesian product between two sets of links. If l_1 (links) explains formula f_1's truth value and l_2 explains formula f_2's truth value, then $l_1 \otimes l_2$ explain both f_1's and f_2's truth values.
- $\cup$: merges two sets of links by set union. If l_1 (links) explains formula f_1's truth value and l_2 explains formula f_2's truth value, then any one from $l_1 \cup l_2$ explains either f_1's or f_2's truth value.
- $flipSet(links)$: changes the type of links from $satisfied$ to $violated$ or from $violated$ to $satisfied$ for all links in a set.
- $lookup(\mathcal{L}[f]_\alpha)$: retrieves links $\mathcal{L}[f]_\alpha$.

Consider a universal formula $\forall\gamma \in S(f)$. If it is violated (i.e., evaluated to false), there must exist at least one context that, if assigned to variable γ, would cause sub-formula f to be evaluated to false. Thus this context should be included into links generated for explaining why this universal formula is violated. Operator $\otimes$ can connect this context with other contexts used by sub-formula f to together explain this formula's violation. Existential formula can be processed similarly. Link generation for formula $(f_1)\ and\ (f_2)$ needs to consider four cases. If f_1 and f_2 are both satisfied, the whole and formula is also satisfied. Thus links generated by f_1 and f_2 need to together explain this formula's satisfaction. If f_1 and f_2 are both violated, any link generated by f_1 or f_2 can explain the formula's violation. If only one sub-formula is violated, the whole and formula is still violated. Then links generated by the violated sub-formula can explain the whole formula's violation. The semantics for $(f_1)\ or\ (f_2)$ and $(f_1)\ implies\ (f_2)$ formulae are similar. To generate links for formula $not\ (f)$, one only needs to reverse the type for links generated by formula f. This is because any link that explains f's violation can also explain $not\ (f)$'s satisfaction.

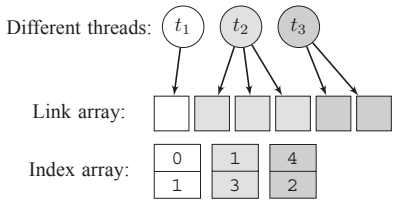


Fig. 7. Example for illustrating our two-level storage strategy

GAIN's parallel strategy for generating links is the same as for truth value evaluation, and thus we omit its details.

D. Storage Strategy

We in the following explain how to realize a conflict-free storage strategy and efficient lookup of previous constraint checking results. This is important because: (1) Results of c-copies should be stored for later lookup purposes. Different c-copies can write results to the same memory address simultaneously, and thus cause conflict. (2) The $lookup$ function has to be realized efficiently as it is called by GAIN frequently as shown in earlier semantics.

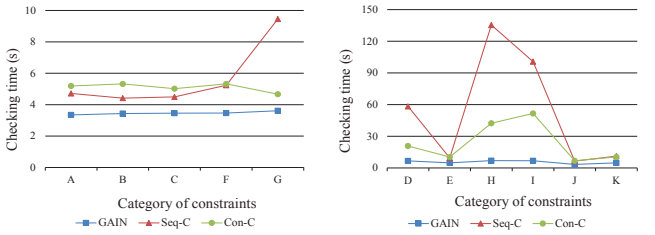
Consider storage of checking results. Truth value results are regular, and this means that each truth value occupies space of the same size in memory. We use an array to store truth values. Given a runtime tree node, the offset from which we store its truth value in the array is proportional to its offset in the tree's reordered checking sequence (Section IV.A). This also makes it very efficient to retrieve previous truth value results. However, links are irregular, and this means that each node can generate a different number of links. We adopt a two-level storage strategy, which uses two arrays: *link array* and *index array*. The former stores all links, and the latter stores indexes for distinguishing links from different nodes. We adopt the prefix sum technique [2] to allocate space in the link array. Given a list of allocation requirements, GAIN calculates offsets from which each thread should start writing. Fig. 7 shows a simple example, in which three threads generate 1, 3, 2 links, respectively. GAIN allocates 1, 3, 2 cells in the link array for corresponding threads. The index array stores these offsets (upper one) and numbers of links (lower one). Given a runtime tree node, the offset from which we store its index in the index array is proportional to its offset in the trees reordered checking sequence. This is similar to the case of storing truth values. Once the index of a node is retrieved, GAIN can efficiently obtain its links according to the number of links stored in the index array.

V. Evaluation

In this section, we evaluate our GAIN and compare it to Seq-C and Con-C.

TABLE I. Different constraint structures

Height	# ∀/∃ quantifiers			# generated constraints
2-level	1 ∀/∃			3 (A)
				1 (A*)
4-level	1 ∀/∃	# nodes $\leq$ 5		3 (B)
		# nodes $\geq$ 6		3 (C)
	2 ∀/∃	nested		3 (D)
				1 (D*)
		not nested		3 (E)
6-level	1 ∀/∃	# nodes $\leq$ 10		3 (F)
		# nodes $\geq$ 20		3 (G)
	2 ∀/∃	nested	**Diff**(c-units) $\leq$ 3	3 (H)
			Diff(c-units) = 5	1 (H*)
			Diff(c-units) $\geq$ 7	3 (I)
	2 ∀/∃	not nested	**Diff**(c-units) $\leq$ 3	3 (J)
			Diff(c-units) $\geq$ 7	3 (K)



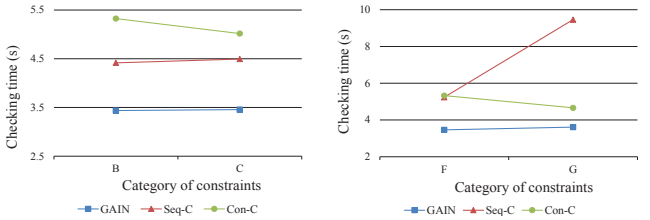
(a) 1-quantifier constraints of different heights (b) 2-quantifier constraints of different heights

Fig. 8. Impact of height

A. Experimental Design

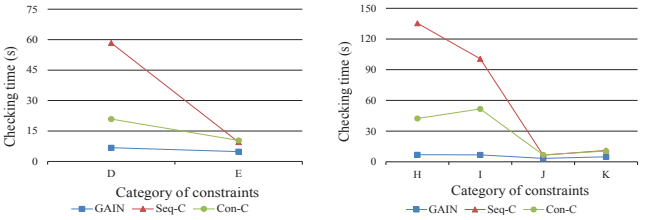
In our experiments, we are interested in the checking performance of these techniques with respect to different constraint structures and under different workloads. Here, *constraint structure* means *structure of a constraint's corresponding syntax tree*. We measure performance by checking time (as dependent variable). We identify three independent variables that can affect checking time, namely, checking technique, constraint structure and workload, as below:

- *Checking technique*: We compare GAIN, Seq-C and Con-C in our experiments. The Con-C we implemented first identifies quantifiers (∀/∃) in a runtime tree, then each branch of a quantifier is handled by one CPU thread. Con-C runs in a recursive way as Seq-C does. Since our CPU contains four cores, we set the maximum number of active threads in Con-C to four.
- *Constraint structure*: We consider three factors concerning a constraint's structure: height of the constraint's syntax tree, number of nodes in this tree, and number of quantifiers (∀/∃) used by this constraint.
- *Checking workload*: The checking workload is mainly determined by the density of received contexts. We consider three groups of contexts (with light, medial and heavy workloads).



(a) 4-level constraints with one quantifier (b) 6-level constraints with one quantifier

Fig. 9. Impact of number of syntax tree nodes



(a) 4-level constraints with two quantifiers (b) 6-level constraints with two quantifiers

Fig. 10. Impact of quantifier nesting

B. Experimental Setup

We conducted experiments on a real-world SUTPC application [13], which aims for smart routing planning as introduced earlier. We used its received contexts for continuous 24 hours, and this accounts for a total of over 1.5 million real taxi data. Each context contains multiple fields including its timestamp, its concerned taxi's id, current location, instant speed, service status, and so on. Intervals between contexts vary from 20 ms to 3,000 ms, with an average value of 55.9 ms, and this decides the workload for constraint checking.

The SUTPC application has its built-in 12 consistency constraints for our experiments. However, they may not necessarily be adequate in covering different constraint structures. Thus we additionally generated 33 constraints randomly, as shown in Table I. We considered two-level, four-level and six-level heights for constraints. For each level of height, we generated constraints with one or two quantifiers (2-level constraints can contain at most one quantifier). Besides, we also considered the number of nodes contained in a constraint, which also affects a constraint's structure and complexity, as shown in Table I. What's more, for constraints containing two quantifiers, we also considered whether the two quantifiers are nested as this also affects a constraint's structure and complexity. For 6-level constraints with two quantifiers, we also additionally considered the maximal difference of their c-units' contained nodes, **Diff**$(c\text{-}units)$, which is an indicator of how a constraint's internal structure differs from that of another. For example, the constraint shown in Fig. 1 contains three c-units with 1, 1, 3 nodes, respectively. Thus **Diff**$(c\text{-}units)$ of this constraint is 2. The last column in Table I gives the number of constraints for each category and their total number is 36. Among them, 33 were randomly generated, and they are labeled from "A" to "K" (11 groups). The remaining three

TABLE II. Three segments of context sequences

Period of time (one hour)	# contexts	Service status: on	Implication
5:00-6:00	69,397	3,895	Light workload
11:00-12:00	63,074	15,638	Medial workload
18:00-19:00	60,736	22,707	Heavy workload

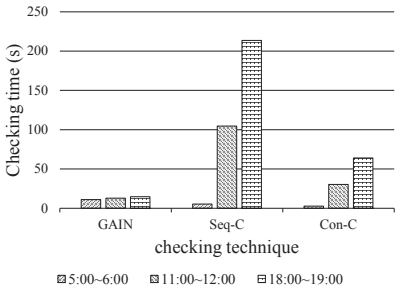


Fig. 11. Impact of checking workload

constraints (A*, D*, H*) are from the original 12 constraints (the other nine constraints are structurally equivalent with the three and thus only three were selected).

Our experiments were conducted on a machine with a quad-core Intel Q9550 CPU @2.83GHz and 4GB RAM, running MS Windows 7. It is equipped with an NVIDIA GeForce GT640 Display Card for supporting GPU computing.

C. Experiment 1: Impact of Constraint Structure

We first study how a constraint's structure affects the checking performance of GAIN, Seq-C and Con-C. We used all 33 randomly generated constraints (11 categories from A to K) and a continuous segment of 5,000 contexts randomly selected from 1.5 million taxi data.

Fig. 8 (a) and (b) compare the checking time of GAIN, Seq-C and Con-C with respect to different heights of syntax trees of constraints (considered by 1-quantifier constraints and 2-quantifier constraints, respectively). We observe that GAIN works most efficiently and its checking time is very stable with respect to different heights of syntax trees of constraints. Seq-C and Con-C spend much more time and are unstable with respect to different heights of syntax trees of constraints. For Seq-C, its time increases dramatically when the number of nodes in a syntax tree is many (G), or quantifiers in constraints are nested (D, H, I). Con-C behaves similarly. Besides, all techniques spend more time for 2-quantifier constraints than for 1-quantifier constraints.

Fig. 9 further studies how the number of syntax tree nodes impacts the checking performance for 1-quantifier constraints. We observe that when the number of syntax tree nodes is close, checking time is almost the same for the three techniques (C vs. B). However, when the number of syntax tree nodes has a large increase, Seq-C becomes very sensitive in checking time (G vs. F), but GAIN and Con-C are stable instead.

Fig. 10 further studies how quantifier nesting impacts the checking performance. We observe that quantifier nesting has a dominant impact on Seq-C's and Con-C's checking time (D vs. E and H/I vs. J/K), but this impact becomes marginal for GAIN. Besides, Fig. 10 (b) also shows that **Diff**$(c\text{-}units)$ has almost no impact on GAIN's checking performance.

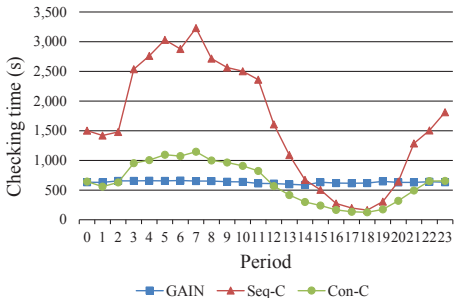


Fig. 12. Comparison of checking time (case study)

D. Experiment 2: Impact of Workload

We then study how the checking workload affects the checking performance of GAIN, Seq-C and Con-C. We used three built-in constraints (A*, D*, H* in Table I), and three segments of context sequences, as listed in Table II. The three segments each includes one hour of taxi data, which are all over 60K. They represent light (early morning before rush hour), medial (noon) and heavy (evening rush hour) workload, respectively. Their respective "service status" data also imply this nature (more "on" data mean larger density of contexts requiring processing).

Fig. 11 compares how the checking workload affects the checking performance of the three techniques. We observe that GAIN works most efficiently for medial and heavy workloads and is very stable for different workloads. Seq-C and Con-C require much more checking time for medial and heavy workloads and are unstable with different workloads. When the workload is light, GAIN needs slightly more checking time than Seq-C and Con-C. This is because when the checking workload is light, most GPU cores are wasted and do not work at all, but overall, GAIN can work much more efficiently.

E. Case Study

We finally compared the three techniques in a case study setting, i.e., using all 1.5 million of taxi data and all its built-in 12 constraints.

Fig. 12 compares the checking performance of GAIN, Seq-C and Con-C for each period (hour). We observe that Seq-C works slowest and it is very unstable to different workloads (different periods), while GAIN and Con-C work much more efficiently (about 61.1% and 61.4% faster, respectively). It is interesting to see that Con-C works very slightly more efficiently than GAIN overall (1.0% relatively). However, we also observe from Fig. 12 that Con-C is unstable, while GAIN is very stable (almost immune to different workloads). Sometimes GAIN works less efficiently than Con-C because when the workload is light, some GPU cores can be wasted as explained earlier. Still, this also reflects that GAIN is scalable as its checking time almost does not increase when the workload grows. This makes it highly applicable to a wider range of application scenarios. Besides, Con-C consumes much

more CPU computing resources (80% on average and up to 98%), while GAIN uses only one CPU core (our machine has four CPU cores and thus the usage is 25% on average and occasionally up to 40%). This also suggests that GAIN can release most CPU computing resources back to applications and this is much appreciated in pervasive computing.

VI. Related Work

In this section, we discuss related work in recent years.

A. Inconsistency Detection

Consistency is an important property of software systems. However, many software artifacts suffer the inconsistency problem. To detect inconsistencies, these artifacts (including contexts) need checking against consistency constraints. One pioneer piece of work is xLinkit [10] [11], which detects inconsistencies in XML documents. GAIN adopts a similar idea of detecting inconsistencies in application contexts. A popular way of improving the checking performance is to conduct incremental checking, which exploits existing intermediate results to reduce unnecessary computing. Examples of incremental checking techniques include UML/Analyzer [5], which checks consistency for evolving UML models, and our previous work PCC [13], which focuses on context consistency for pervasive computing applications. Earlier ConC [15] explored ways of checking consistency constraints in parallel by CPU computing. It maintains balanced workload for all CPU threads, but cannot be applied to GPU computing due to different programming models.

B. GPU Computing

There is a large body of work on GPU computing. Some development aids are proposed to ease GPU programming, and certain GPU algorithms are studied for specific problems. For the former, hiCUDA [6], Mars [7] and Medusa [16] provide high-level abstraction of CUDA programming. They provide APIs or directives to hide underlying details of GPU programming. For the latter, GPU techniques to accelerate specific applications can vary greatly. For example, Bakkum et al. parallelized SQL operation *SELECT* by assigning each row to a GPU thread to execute queries [1]. To facilitate breadth-first search in solving graph problems, two vertex queues are prepared for switching: one for input and another for output, and their roles are reversed in the next iteration [9]. Algorithms based on tree structures can also be implemented by GPU computing. For example, Luo et al. [8] parallelized R-tree queries and constructions, and Burtscher et al. [3] proposed an GPU-based n-body algorithm. Our work studied in this paper differs from these techniques in that we have various tree nodes that have to be processed differently. Besides, links processed by constraint checking can be irregular, and our GAIN addresses it by a two-level storage strategy to realize conflict-free and efficient construction and retrieval.

VII. Conclusion

In this paper, we studied GPU computing for constraint checking. We presented a novel technique GAIN to tackle challenges in doing so. GAIN is applicable to any first-order logic specified consistency constraints. It can automatically recognize parallel units from constraints and arrange them different GPU cores for parallel computing. With a carefully designed two-level storage strategy, GAIN can address irregular checking results, and this makes GAIN general to various constraints and the checking process transparent to applications. We evaluated GAIN experimentally, and the results confirmed GAIN's efficiency as well as its stability and scalability to different workloads. This stability makes GAIN more suitable for tasks with high workload. It also releases valuable CPU computing resources back to applications. Finally, the GAIN idea can be potentially extended for more application scenarios and we are investigating its broader usage.

VIII. Acknowledgment

This work was supported in part by National Basic Research 973 Program (Grant No. 2015CB352202), and National Natural Science Foundation (Grant Nos. 61472174, 91318301, 61321491, 61361120097) of China.

References

[1] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[2] G. E. Blelloch, "Prefix sums and their applications," 1990.

[3] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," *GPU computing Gems Emerald edition*, p. 75, 2011.

[4] C. Chen, C. Ye, and H.-A. Jacobsen, "Hybrid context inconsistency resolution for context-aware services," in *Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on*. IEEE, 2011, pp. 10–19.

[5] A. Egyed, "Instant consistency checking for the uml," in *Proceedings of the 28th ICSE*. ACM, 2006, pp. 381–390.

[6] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 78–90, 2011.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[8] L. Luo, M. D. Wong, and L. Leong, "Parallel implementation of r-trees on the gpu," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 2012, pp. 353–358.

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[10] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelsteiin, "xlinkit: A consistency checking and smart link generation service," *TOIT*, vol. 2, no. 2, pp. 151–185, 2002.

[11] C. Nentwich, W. Emmerich, A. Finkelsteiin, and E. Ellmer, "Flexible consistency checking," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 1, pp. 28–63, 2003.

[12] R. NVIDIA, "Corporation: Nvidia cuda c programming guide version 4.1 (2011)."

[13] C. Xu, S. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *TOSEM*, vol. 19, no. 3, p. 9, 2010.

[14] C. Xu, S.-C. Cheung, and W. Chan, "Incremental consistency checking for pervasive context," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 292–301.

[15] C. Xu, Y. Liu, S. Cheung, C. Cao, and J. Lv, "Towards context consistency by concurrent checking for internetware applications," *Science China Information Sciences*, vol. 56, no. 8, pp. 1–20, 2013.

[16] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," 2013.