

LTL learning on GPUs

Mojtaba Valizadeh^{1,2}, Nathanaël Fijalkow³, and Martin Berger^{1,4}

¹ University of Sussex, Brighton, UK,

² Neubla UK Ltd, Valizadeh.Mojtaba@gmail.com

³ CNRS, LaBRI and Université de Bordeaux, France,
nathanael.fijalkow@gmail.com

⁴ Montanarius Ltd, London, UK, contact@martinfriedrichberger.net

Abstract. Linear temporal logic (LTL) is widely used in industrial verification. LTL formulae can be learned from traces. Scaling LTL formula learning is an open problem. We implement the first GPU-based LTL learner using a novel form of enumerative program synthesis. The learner is sound and complete. Our benchmarks indicate that it handles traces at least 2048 times more numerous, and on average at least 46 times faster than existing state-of-the-art learners. This is achieved with, among others, novel branch-free LTL semantics that has $O(\log n)$ time complexity, where n is trace length, while previous implementations are $O(n^2)$ or worse (assuming bitwise boolean operations and shifts by powers of 2 have unit costs—a realistic assumption on modern processors).

1 Introduction

Program verification means demonstrating that an implementation exhibits the behaviour required by a specification. But where do specifications come from? Handcrafting specifications does not scale. One solution is automatically to *learn* them from example runs of a system. This is sometimes referred to as trace analysis. A trace, in this context, is a sequence of events or states captured during the execution of a system. Once captured, traces are often converted into a form more suitable for further processing, such as finite state automata or logical formulae. Converting traces into logical formulae can be done with program synthesis. Program synthesis is an umbrella term for the algorithmic generation of programs (and similar formal objects, like logical formulae) from specifications, see [8, 15] for an overview. Arguably, the most popular logic for representing traces is linear temporal logic (LTL) [31], a modal logic for specifying properties of finite or infinite traces. The *LTL learning problem* idealises the algorithmic essence of learning specifications from example traces, and is given as follows.

- **Input:** Two sets P and N of traces over a fixed alphabet.
- **Output:** An LTL formula ϕ that is (i) *sound*: all traces in P are accepted by ϕ , all traces in N are rejected by ϕ ; (ii) *minimal*, meaning no strictly smaller sound formula exists.

When we weaken minimality to minimality-up-to- ϵ , we speak of *approximate* LTL learning. Both forms of LTL learning are NP-hard [10, 27]. A different and simpler problem is *noisy* LTL learning, which is permitted to learn unsound formulae, albeit only up-to a give error-rate.

LTL learning is an active research area in software engineering, formal methods, and artificial intelligence [1, 5, 6, 11–14, 18, 19, 22, 24–26, 29, 30, 32, 34–36]. We refer to [6] for a longer discussion. Many approaches to LTL learning have been explored. One common and natural method involves using search-based program synthesis, often paired with templates or sketches, such as parts of formulas, automata, or regular expressions. Another leverages SAT solvers. LTL learning is also being pursued using Bayesian inference, or inductive logic programming. Learning specifically tailored small fragments of LTL often yields the best results in practice [32]. Learning from noisy data is investigated in [14, 26, 30]. All have in common is that they don’t scale, and have not been optimised for GPUs. Traces arising in industrial practice are commonly long (millions of characters), and numerous (millions of traces). Extracting useful information automatically at such scale is currently a major problem, e.g., the state-of-the-art learner in [32] cannot reliably learn formulae greater than size 10. This is less than ideal. Our aim is to change this.

Graphics Processing Units (GPUs) are the work-horses of high-performance computing. The acceleration they provide to applications compatible with their programming paradigm can surpass CPU performance by several orders of magnitude, as notably evidenced by the advancements in deep learning. A significant spectrum of applications, especially within automated reasoning—like SAT/SMT solvers and model checkers—has yet to reap the benefits of GPU acceleration. In order for an application to be “GPU-friendly”, it needs to have high parallelism, minimal data-dependent branching, and predictable data movement with substantial data locality [7, 16, 17]. Current automated reasoning algorithms are predominantly branching-intensive and appear sequential in nature, but it is unclear whether they are inherently sequential, or can be adapted to GPUs.

Research question. *Can we scale LTL learning to at least 1000 times more traces without sacrificing trace length, learning speed or approximation ratio (cost increase of learned formula over minimum) compared to existing work, by employing suitably adapted algorithms on a GPU?*

We answer the RQ in the affirmative by developing the first GPU-accelerated LTL learner. Our work takes inspiration from [33], the first GPU-accelerated minimal regular expression inferencer. Scaling has two core orthogonal dimensions: more traces, and longer traces. We solve one problem [33] left open: scaling to more traces. Our key decision, giving up on learning minimal formula while remaining sound and complete, enables two principled algorithmic techniques.

- **Divide-and-conquer (D&C).** If a learning task has too many traces, split it into smaller specifications, learn those recursively, and combine the learned formulae using logical connectives.

- **Relaxed uniqueness checks (RUCs).** Often generate-and-test program synthesis caches synthesis results to avoid recomputation. [33] granted cache admission only after a uniqueness check. We relax uniqueness checking by (pseudo-)randomly rejecting some unique formulae.

In addition, we design novel algorithms and data structures, representing LTL formulae as contiguous matrices of bits. This allows a GPU-friendly implementation of all logical operations with linear memory access and suitable machine instructions, free from data-dependent branching. Both D&C and RUCs may lose minimality and are thus unavailable to [33], see Appendix D for a comparison. Our benchmarks show that the approximation ratio is typically small.

Contributions. In summary, our contributions are as follows:

- A new enumeration algorithm for LTL learning, with a branch-free implementation of LTL semantics that is $O(\log n)$ in trace length (assuming unit cost for logical and shift operations).
- A CUDA implementation of the algorithm, for benchmarking and inspection.
- A parameterised benchmark suite useful for evaluating the performance of LTL learners, and a novel methodology for quantifying the loss of minimality induced by approximate LTL learning.
- Performance benchmarks showing that our implementation is both faster, and can handle orders of magnitude more traces, than existing work.

2 Formal preliminaries

We write $\#S$ for the cardinality of set S . $\mathbb{N} = \{0, 1, 2, \dots\}$, $[n]$ is for $\{0, 1, \dots, n-1\}$ and $[m, n]$ for $\{m, m+1, \dots, n-1\}$. \mathbb{B} is $\{0, 1\}$ where 0 is falsity and 1 truth. $\mathfrak{P}(A)$ is the *powerset* of A . The *characteristic function* of a set S is the function $\mathbf{1}_S^A : A \rightarrow \mathbb{B}$ which maps $a \in A$ to 1 iff $a \in S$. We usually write $\mathbf{1}_S$ for $\mathbf{1}_S^A$. An *alphabet* is a finite, non-empty set Σ , the elements of which are *characters*. A *string of length* $n \in \mathbb{N}$ over Σ is a map $w : [n] \rightarrow \Sigma$. We write $\|w\|$ for n . We often write w_i instead of $w(i)$, and $v \cdot w$, or just vw , for the concatenation of v and w , ϵ for the empty string and Σ^* for all strings over Σ . A *trace* is a string over powerset alphabets, *i.e.*, $(\mathfrak{P}(\Sigma))^*$. We call Σ the *alphabet* of the trace and write $\text{traces}(\Sigma)$ for all traces over Σ . A *word* is a trace where each character has cardinality 1. We abbreviate words to the corresponding strings, e.g., $\langle \{t\}, \{i\}, \{n\} \rangle$ to *tin*. We say v is a *suffix* of w if $w = uv$, and if $\|u\| = 1$ then v is an *immediate* suffix. We write $\text{sc}(S)$ for the *suffix-closure* of S . S is *suffix-closed* if $\text{sc}(S) \subseteq S$. $\text{sc}^+(S)$ is the *non-empty suffix closure* of S , *i.e.*, $\text{sc}(S) \setminus \{\epsilon\}$. *From now on we will speak of the suffix-closure to mean the non-empty suffix closure.* The *Hamming-distance* between two strings s and t of equal length, written $\text{hamm}(s, t)$, is the number of indices i where $s(i) \neq t(i)$. We write $\text{Hamm}(s, \delta)$ for the set $\{t \in \Sigma^* \mid \text{hamm}(s, t) = \delta, \|s\| = \|t\|\}$.

LTL formulae over $\Sigma = \{p_1, \dots, p_n\}$ are given by the following grammar.

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\phi \mid F\phi \mid G\phi \mid \phi U \phi$$

The *subformulae* of ϕ are denoted $\text{sf}(\phi)$. We say $\psi \in \text{sf}(\phi)$ is *proper* if $\phi \neq \psi$. A formula is in *negation normal form* (NNF) if all subformulae containing negation are of the form $\neg p$. It is *U-free* if no subformula is of the form $\phi \text{ U } \psi$. We write $\text{LTL}(\Sigma)$ for the set of all LTL formulae over Σ . We use **true** as an abbreviation for $p \vee \neg p$ and **false** for $p \wedge \neg p$. We call **X**, **F**, **G**, **U** the *temporal* connectives, \wedge, \vee, \neg the *propositional* connectives, p the *atomic* propositions and, collectively name them the *LTL connectives*. Since we learn from finite traces, we interpret LTL over finite traces [9]. The satisfaction relation $tr, i \models \phi$, where tr is a trace over Σ and ϕ from $\text{LTL}(\Sigma)$ is standard, here are some example clauses: $tr, i \models \text{X}\phi$, if $tr, i+1 \models \phi$, $tr, i \models \text{F}\phi$, if there is $i \leq j < \|tr\|$ with $tr, j \models \phi$, and $tr, i \models \phi \text{ U } \phi'$, if there is $i \leq j < \|tr\|$ such that: $tr, k \models \phi$ for all $i \leq k < j$, and $tr, j \models \phi'$. If $i \geq \|tr\|$ then $tr, i \models \phi$ is always false, and $tr \models \phi$ is short for $tr, 0 \models \phi$.

A *cost-homomorphism* is a map $\text{cost}(\cdot)$ from LTL connectives to positive integers. We extend it to LTL formulae homomorphically: $\text{cost}(\phi \text{ op } \psi) = \text{cost}(\text{op}) + \text{cost}(\phi) + \text{cost}(\psi)$, and likewise for other arities. If $\text{cost}(\text{op}) = 1$ for all LTL connectives we speak of *uniform cost*. So the uniform cost of **true** and **false** is 4. *From now on all cost-homomorphisms will be uniform, except where stated otherwise.*

A *specification* is a pair (P, N) of finite sets of traces such that $P \cap N = \emptyset$. We call P the *positive* examples and N the *negative* examples. We say ϕ *satisfies*, *separates* or *solves* (P, N) , denoted $\phi \models (P, N)$, if for all $tr \in P$ we have $tr \models \phi$, and for all $tr \in N$ we have $tr \not\models \phi$. A *sub-specification* of (P, N) is any specification (P', N') such that $P' \subseteq P$ and $N' \subseteq N$. Symmetrically, (P, N) is an *extension* of (P', N') . We can now make the *LTL learning problem* precise:

- **Input:** A specification (P, N) , and a cost-homomorphism $\text{cost}(\cdot)$.
- **Output:** An LTL formula ϕ that is *sound*, i.e., $\phi \models (P, N)$, and *minimal*, i.e., $\psi \models (P, N)$ implies $\text{cost}(\phi) \leq \text{cost}(\psi)$.

Cost-homomorphisms let us influence LTL learning: e.g., by assigning a high cost to a connective, we prevent it from being used in learned formulae. The *language at i* of ϕ , written $\text{lang}(i, \phi)$, is $\{tr \in \text{traces}(\Sigma) \mid tr, i \models \phi\}$. We write $\text{lang}(\phi)$ as a shorthand for $\text{lang}(0, \phi)$ and speak of the *language* of ϕ . We say ϕ *denotes* a language $S \subseteq \Sigma^*$, resp., a trace $tr \in \Sigma^*$, if $\text{lang}(\phi) = S$, resp., $\text{lang}(\phi) = \{tr\}$. We say two formulae ϕ_1 and ϕ_2 are *observationally equivalent*, written $\phi_1 \simeq \phi_2$, if they denote the same language. Let S be a set of traces. Then we write

$$\phi_1 \simeq \phi_2 \quad \text{mod } S \quad \text{iff} \quad \text{lang}(\phi_1) \cap S = \text{lang}(\phi_2) \cap S$$

and say ϕ_1 and ϕ_2 are *observationally equivalent modulo S* . The following related definitions will be useful later. Let (P, N) be a specification. The *cardinality* of (P, N) , denoted $\#(P, N)$, is $\#P + \#N$. The *size* of a set S of traces, denoted $\|S\|$ is $\sum_{tr \in S} \|tr\|$. We extend this to specifications: $\|(P, N)\|$ is $\|P\| + \|N\|$. The *cost* of a specification (P, N) , written $\text{cost}(P, N)$ is the uniform cost of a minimal sound formula for (P, N) . An extension of (P, N) is *conservative* if any minimal sound formula for (P, N) is also minimal and sound for the extension. We note a useful fact: if ϕ is a minimal solution for (P, N) , and also $\phi \models (P', N')$ then $(P \cup P', N \cup N')$ is a conservative extension.

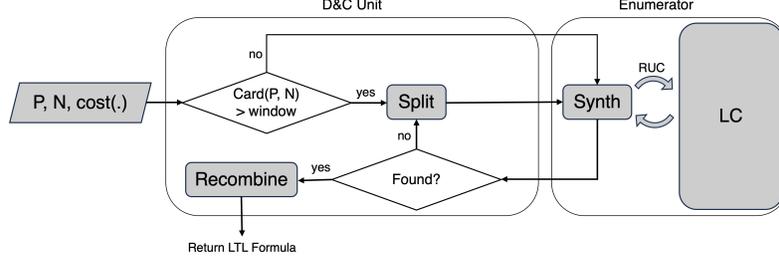


Fig. 1. High-level structure of our algorithm. LC is short for language cache.

Overfitting. It is possible to express a trace tr , respectively a set S of traces, by a formula ϕ , in the sense that $\text{lang}(\phi) = \{tr\}$, resp., $\text{lang}(\phi) = S$. We define the function $\text{overfit}(\cdot)$ on sets of characters, traces, sets and specifications as follows.

- $\text{overfit}(\{a_1, \dots, a_k\}) = (\bigwedge_i a_i) \wedge \bigwedge_{b \in \Sigma \setminus \{a_1, \dots, a_k\}} \neg b$.
- $\text{overfit}(\epsilon) = \neg X(\text{true})$ and $\text{overfit}(a \cdot tr) = \text{overfit}(a) \wedge X(\text{overfit}(tr))$.
- $\text{overfit}(S) = \bigvee_{tr \in S} \text{overfit}(tr)$
- $\text{overfit}(P, N) = \text{overfit}(P)$

The following are immediate from the definitions: (i) For all specifications (P, N) : $\text{lang}(\text{overfit}(P, N)) = P$, (ii) $\text{overfit}(P, N) \models (P, N)$, and (iii) the *cost of overfitting*, i.e., $\text{cost}(\text{overfit}(P, N))$, is $O(\|P\| + \#\Sigma)$. Note that $\text{overfit}(P, N)$ is overfitting only on P , and (ii) justifies this choice.

3 High-level structure of the algorithm

Figure 1 shows the two main parts of our algorithm: the *divide-and-conquer unit*, short D&C-unit, and the *enumerator*. Currently, only the enumerator is implemented for execution of a GPU. For convenience, our D&C-unit is in Python and runs on a CPU. Implementing the D&C-unit on a GPU poses no technical challenges and would make our implementation perform better.

Given (P, N) , the D&C-unit checks if the specification is small enough to be solved by the enumerator directly. If not, the specification is recursively decomposed into smaller sub-specifications. When the recursive invocations return formulae, the D&C-unit combines them into a formula separating (P, N) , see §6 for details. For small enough (P, N) , the enumerator performs a bottom-up enumeration of LTL formulae by increasing cost, until it finds one that separates (P, N) . Like the enumerator in [33], our enumerator uses a language cache to minimise re-computation, but with a novel cache admission policy (RUCs). The language cache is append-only, hence no synchronisation is required for read-access. The key difference from [33], our use of RUCs, is discussed in §4.

The enumerator has three core parameters.

- T = maximal number of traces in the specification (P, N) .
- L = number of bits usable for representing each trace from (P, N) in memory.

- W = number of bits (P, N) hashed *to* during enumeration.

We write $\text{ENUM}(T, L, W)$ to emphasise those parameters. Our current implementation hard-codes all parameters as $\text{ENUM}(64, 64, 128)$ ⁵, but the abstract algorithm does not depend on this. The choice of $W = 128$ is a consequence of the current limitations of WarpCore [20, 21], a CUDA library for high-performance hashing of 32 and 64 bit integers. All three parameters heavily affect memory consumption. We chose $T = 64$ and $L = 64$ for convenient comparison with existing work in §7. While T, L and M are parameters of the abstract algorithm, the implementation is not parameterised: changing these parameters requires changing parts of the code. Making the implementation fully parametric is conceptually straightforward, but introduces a substantial number of new edge cases, primarily where parameters are not powers of 2, which increases verification effort.

We now sketch the high-level structure of `enum`, the entry point of the enumerator, taking a specification and a cost-homomorphism as arguments. For ease of presentation, we use LTL formulae as search space. Their representation in the implementation is discussed in §4.

```

1 language_cache = []
2
3 def enum(p, n, cost):
4     if (p, n) can be solved with Atom then return Atom
5     language_cache.append([Atom])
6     for c in range(cost(Atom)+1, cost(overfit(p, n))):
7         language_cache.append([])
8         for op in [F, U, G, X, And, Or, Not]:
9             handleOp(op, p, n, c, cost)
10    return overfit(p, n)

```

Line 4 checks if the learning problem can be solved with an atomic proposition. If not, Line 5 initialises the global language cache with the representation of atomic propositions, and search starts from the lowest cost upwards. For each cost c a new empty entry is added to the language cache. Line 8 then maps over LTL connectives and calls `handleOp` to construct all formulae of cost c using all suitable lower cost entries in the language cache. When no sound formula can be found with cost less than $\text{cost}(\text{overfit}(P, N))$, the algorithm terminates, returning $\text{overfit}(P, N)$. This makes our algorithm *complete*, in the sense of learning a formula for every specification.

```

11 def handleOp(op, p, n, c, cost):
12     match op:
13     case F:
14         for all phi in language_cache(c-cost(F)): # parallel
15             phi_new = branchfree_F(phi)
16             relaxedCheckAndCache(p, n, c, phi_new)
17     case U:
18         for all (cL, cR) in split(c-cost(U)): # parallel, split(x) gives all (i,j) s.t. i+j = x
19             for all phi_L in language_cache(cL): # parallel
20                 for all phi_R in language_cache(cR): # parallel
21                     phi_new = branchfree_U(phi_L, phi_R)
22                     relaxedCheckAndCache(p, n, c, phi_new)
23     case G: ...
24     case ...

```

The function `handleOp` dispatches on LTL connectives, retrieves all previously constructed formulae of suitable cost from the language cache in parallel (we use `for all` to indicate parallel execution), calls the appropriate semantic function,

⁵ For pragmatic reasons, our implementation uses only 126 bits of $W=128$, and 63 bits of $L = 64$, details omitted for brevity.

detailed in the next section, e.g., `branchfree_F` for F , to construct `phi_new`, and then sends it to `relaxedCheckAndCache` to check if it already solves the learning task, and, if not, for potential caching. Most parallelism in our implementation, and the upside of the language cache’s rapid growth, is the concomitant growth in available parallelism, which effortlessly saturates every conceivable processor.

```

25 def relaxedCheckAndCache(p, n, c, phi_new): # c is a concrete cost
26     if phi_new != (p, n): # check if candidate is sound for (p, n)
27         exit(phi_new) # Terminate learning, return phi_new as learned formula
28     if relaxedUniquenessCheck(phi_new, language_cache):
29         language_cache[c].append(phi_new) # parallel

```

This last step checks if `phi_new` satisfies (P, N) . If yes, the program terminates with the formula corresponding to `phi_new`. Otherwise, Line 28 conducts a RUC, a *relaxed* uniqueness check, described in detail in §5, to decide whether to cache `phi_new` or not. Updating the language cache in Line 29 is done in parallel, and needs little synchronisation, see [33] for details. The satisfaction check in Line 26 guarantees that our algorithm is *sound*. It also makes it trivial to implement noisy LTL learning: just replace the precise check `phi_new != (p, n)` with a check that `phi_new` gets a suitable fraction of the specification right.

4 In-memory representation of search space

Our enumerator does generate-and-check synthesis. That means we have two problems: (i) minimising the cardinality of the search space, *i.e.*, the representation of LTL formulae during synthesis; (ii) making generation and checking as cheap as possible. For each candidate ϕ checking means evaluating the predicate

$$P \subseteq \text{lang}(\phi) \text{ and } N \cap \text{lang}(\phi) = \emptyset. \quad (\dagger)$$

LTL formulae, the natural choice of search space, suffer from the redundancies of syntax: every language that is denoted by a formula at all, is denoted by infinitely many, e.g., $\text{lang}(F\phi) = \text{lang}(FF\phi)$. Even observational equivalence distinguishes too many formulae: the predicate (\dagger) checks the language of ϕ only for elements of $P \cup N$. Formulae modulo $P \cup N$ contain exactly the right amount of information for (\dagger) , hence minimise the search space. However, the semantics of formulae on $P \cup N$ is given compositionally in terms of the non-empty suffix-closure of $P \cup N$, which would have to be recomputed at run-time for each new candidate. Since $P \cup N$ remains fixed, so does $\text{sc}^+(P \cup N)$, and we can avoid such re-computation by using formulae modulo $\text{sc}^+(P \cup N)$ as search space. Inspired by [33], we represent formulae ϕ by characteristic functions $\mathbf{1}_{\text{lang}(\phi)} : \text{sc}^+(P \cup N) \rightarrow \mathbb{B}$, which are implemented as contiguous bitvectors in memory, but with a twist. Fix a total order on $P \cup N$.

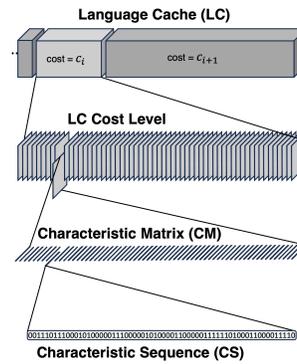


Fig. 2. Data representation in memory (simplified).

- A *characteristic sequence (CS)* for ϕ over tr is a bitvector cs such that $tr, j \models \phi$ iff $cs(j) = 1$. For ENUM(64, 64, 128), CSs are unsigned 64 bit integers.
- A *characteristic matrix (CM)* representing ϕ over $P \cup N$, is a sequence cm of CSs, contiguous in memory, such that, if tr is the i th trace in the order, then $cm(i)$ is the CS for ϕ over tr .

See Appendix A for examples. This representation has two interesting properties, not present in [33]: (i) each CS is *suffix-contiguous*: the trace corresponding to $cs(j+1)$ is the immediate suffix of that at $cs(j)$; (ii) CMs contain *redundancies* whenever two traces in $P \cup N$ share suffixes. Redundancy is the price we pay for suffix-contiguity. Figure 2 visualises our representation in memory.

Logical operations as bitwise operations. Suffix-contiguity enables the efficient representation of logical operations: if $cs = 10011$ represents ϕ over the word $abcaa$, e.g., ϕ is the atomic proposition a , then $X\phi$ is 00110 , *i.e.*, cs shifted one to the left. Likewise $\neg\phi$ is represented by 01100 , *i.e.*, bitwise negation. As we use unsigned 64 bit integers to represent CSs, X and negation are executed as *single* machine instructions! In Python-like pseudo-code:

```

1 def branchfree_X(cm):
2   return [cs << 1 for cs in cm]
1 def branchfree_Not(cm):
2   return [~cs for cs in cm]

```

Conjunction and disjunction are equally efficient. More interesting is F which becomes the disjunction of shifts by *powers of two*, *i.e.*, the number of shifts is logarithmic in the length of the trace (a naive implementation of F is linear). We call this *exponential propagation*, and believe it to be novel in LTL synthesis⁶:

```

1 def branchfree_F(cm):
2   outCm = []
3   for cs in cm:
4     cs |= cs << 1
5     cs |= cs << 2
6     cs |= cs << 4
7     cs |= cs << 8
8     cs |= cs << 16
9     cs |= cs << 32
10  outCm.append(cs)
11  return outCm
1 def branchfree_U(cm1, cm2):
2   outCm = []
3   for i in range(len(cm1)):
4     cs1 = cm1[i]
5     cs2 = cm2[i]
6     cs2 |= cs1 & (cs2 << 1)
7     cs1 ^= cs1 << 1
8     cs2 |= cs1 & (cs2 << 2)
9     cs1 ^= cs1 << 2
10    cs2 |= cs1 & (cs2 << 4)
11    cs1 ^= cs1 << 4
12    cs2 |= cs1 & (cs2 << 8)
13    cs1 ^= cs1 << 8
14    cs2 |= cs1 & (cs2 << 16)
15    cs1 ^= cs1 << 16
16    cs2 |= cs1 & (cs2 << 32)
17    outCm.append(cs2)
18  return outCm

```

See Appendix A for examples. To see why this works, note that $F\phi$ can be seen as the infinite disjunction $\phi \vee X\phi \vee X^2\phi \vee X^3\phi \vee \dots$, where $X^n\phi$ is given by $X^0\phi = \phi$ and $X^{n+1}\phi = XX^n\phi$. Since we work with finite traces, $tr, i \not\models \phi$ whenever $i \geq \|tr\|$. Hence checking $tr, 0 \models F\phi$ for tr of length n amounts to checking

$$tr, 0 \models \phi \vee X\phi \vee X^2\phi \vee \dots \vee X^{n-1}\phi$$

The key insight is that the imperative update $cs \mid= cs \ll j$ propagates the bit stored at $cs(i+j)$ into $cs(i)$ without removing it from $cs(i+j)$. Consider the

⁶ By representing ϕ as a CS, *i.e.*, unsigned integer, we can also read $F\phi$ as rounding up ϕ to the next bigger power of 2 and then subtracting 1, cf. [2].

flow of information stored in $cs(n-1)$. At the start, this information is only at index $n-1$. This amounts to checking $tr, n-1 \models X^{n-1}\phi$. Thus assigning $cs \models cs \ll 1$ puts that information at indices $n-2, n-1$. This amounts to checking $tr, 0 \models X^{n-2}\phi \vee X^{n-1}\phi$. Likewise, then assigning $cs \models cs \ll 2$ puts that information at indices $n-4, n-3, n-2, n-1$. This amounts to checking $tr, 0 \models X^{n-4}\phi \vee X^{n-3}\phi \vee X^{n-2}\phi \vee X^{n-1}\phi$, and so on. In a logarithmic number of steps, we reach $tr, 0 \models \phi \vee X\phi \vee \dots \vee X^{n-1}\phi$. This works uniformly for all positions, not just $n-1$. In the limit, this saves an exponential amount of work over naive shifting.

We can implement U using similar ideas, with the number of bitshifts also logarithmic in trace length. As with F , this works because we can see $\phi U \psi$ as an infinite disjunction

$$\psi \vee (\phi \wedge X\psi) \vee (\phi \wedge X(\phi \wedge X\psi)) \vee \dots$$

We define (informally) $\phi U_{\leq p} \psi$ as: ϕ holds until ψ does within the next p positions, and $G_{\geq p}\phi$ if ϕ holds for the next p positions. The additional insight allowing us to implement exponential propagation for U is to compute both, $G_{\geq 2^i}\phi$ and $\phi U_{\leq 2^i} \psi$, for increasing values of i at the same time. Appendix B proves correctness of exponential propagation.

In addition to saving work, exponential propagation maps directly to machine instructions, and is essentially branch-free code for all LTL connectives⁷, thus maximises GPU-friendliness of our learner. In contrast, previous learners like Flie [29], Scarlet [32] and Syslite [3], implement the temporal connectives naively, e.g., checking $tr, i \models \phi U \psi$ by iterating from i as long as ϕ holds, stopping as soon as ψ holds. Likewise, Flie encodes the LTL semantics directly as a propositional formula. For U this is quadratic in the length of tr for Flie and Syslite.

5 Relaxed uniqueness checks

Our choice of search space, formulae modulo $sc^+(P \cup N)$, while more efficient than bare formulae, still does not prevent the explosive growth of candidates: uniqueness of CMs is *not* preserved under LTL connectives. [33] recommends storing newly synthesised formulae in a “language cache”, but only if they pass a uniqueness check. Without this cache admission policy, the explosive growth of redundant CMs rapidly swamps the language cache with useless repetition. While uniqueness improves scalability, it just delays the inevitable: there are simply too many unique CMs. Worse: with $ENUM(64, 64, 128)$, CMs use up-to 32 times more memory than language cache entries in [33]. We improve memory consumption of our algorithm by relaxing strictness of uniqueness checks: we allow false positives (meaning that CMs are falsely classified as being already in the language cache), but not false negatives. We call this new cache admission policy *relaxed uniqueness checks* (RUCs). False positives mean that less gets cached. False positives are sound: every formula learned in the presence of

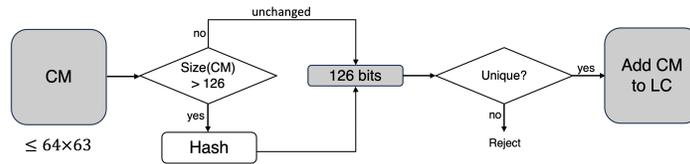
⁷ Including, *mutatis mutandis*, past-looking temporal connectives.

false positives is separating, but no longer necessarily minimal—every minimal solution might have some of its subformulae missing from the language cache, hence cannot be constructed by the enumeration. False positives also do not affect completeness: in the worst case, our algorithm terminates by overfitting.

We implement the RUC using non-cryptographic hashing in several steps.

- We treat each CM as a big bitvector, *i.e.*, ignore its internal structure. Now there are two possibilities.
 - The CM uses more than 126 bits. Then we hash it to 126 bits using a variant of MuellerHash from WarpCore.
 - Otherwise we leave the CM unchanged (except padding it with 0s to 126 bits where necessary).
- Only if this 126 bit sequence is unique, it is added to the language cache.

If the CM is ≤ 126 bits, then the RUC is precise and enumeration performs a full bottom-up enumeration of CMs, so any learned formula is minimal cost. This becomes useful in benchmarking.



Note that RUCs implemented by hashing amount to a (pseudo-)random cache admission policy. See Appendix C for background on cache admission policies. Using RUCs essentially means that hash-collisions (pseudo-)randomly prevent formulae from being subformulae of any learned solution. It is remarkable that this works well in practice, but it probably means that LTL has sufficient redundancy in formulae vis-a-vis the probability of hash collisions. We leave a detailed theoretical analysis as future work.

6 Divide & conquer

The D&C-unit’s job is, recursively, to split specifications until they are small enough to be solved by $\text{ENUM}(T, L, W)$ in one go, and, afterwards recombine the results. A naive D&C-strategy could split (P, N) , when needed, into four smaller specifications (P_i, N_j) for $i, j = 1, 2$, such that P is the disjoint union of P_1 and P_2 , and N of N_1 and N_2 . Then it learned the ϕ_{ij} recursively from the (P_i, N_j) , and finally combine all into

$$(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$$

which is sound for (P, N) , but is not necessarily minimal. E.g., whenever ϕ_{11} implies ϕ_{12} , then $\phi_{11} \vee (\phi_{21} \wedge \phi_{22})$ is lower cost⁸. Thus it might be tempting to

⁸ Such redundancies can be eliminated, for example, by using theorem provers.

minimise D&C-steps. Alas, the enumerator may run out-of-memory (OOM): the parameters in $\text{ENUM}(T, L, W)$ are *static* constraints, pertaining to data structure layout, and do not guarantee successful termination. Let us call the maximal cardinality $\#(P, N)$ that the D&C-unit sends directly to the enumerator, the *split window*. In order to navigate the trade-offs between avoiding OOM and minimising the approximation ratio, our refined D&C-units below use search to find as large as possible a split window. We write *win* for the split window parameter. Both implementations split specifications until they fit into the split window, *i.e.*, $\#(P, N) \leq \text{win}$, and then invoke the enumerator. The split window is then successively halved, until the enumerator no longer runs OOM but returns a sound formula.

Deterministic splitting. The idea behind $\text{detSplit}(P, N, \text{win})$ is: if $\#(P, N) \leq \text{win}$, we send (P, N) directly to the enumerator. Otherwise, assume P is $\{p_1, \dots, p_n\}$. Then $P_1 = \{p_1, \dots, p_{n/2}\}$ and $P_2 = \{p_{n/2+1}, \dots, p_n\}$ are the new positive sets, and likewise for N . (If the specification is given as two lists of traces, this is deterministic.) We then make 4 recursive calls, but remove redundancies in the calls' arguments.

- $\phi_{11} = \text{detSplit}(P_1, N_1, \text{win})$,
- $\phi_{12} = \text{detSplit}(P_1, N_2 \cap \text{lang}(\phi_{11}), \text{win})$,
- $\phi_{21} = \text{detSplit}(P_2 \setminus L, N_1, \text{win})$,
- $\phi_{22} = \text{detSplit}(P_2 \setminus L, N_2 \cap \text{lang}(\phi_{21}), \text{win})$,

Here $L = \text{lang}(\phi_{11}) \cup \text{lang}(\phi_{12})$. Assuming that none of the 4 recursive calls returns OOM, the resulting formula is $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$. Otherwise we recurse with $\text{detSplit}(P, N, \text{win}/2)$.

Random splitting. This variant of the algorithm, written $\text{randSplit}(P, N, \text{win})$, is based on the intuition that often a small number of traces already contain enough information to learn a formula for the whole specification. (E.g., the traces are generated by running the same system multiple times.)

- $\phi_{11} = \text{aux}(P, N, \text{win})$
- $\phi_{12} = \text{randSplit}(P \cap \text{lang}(\phi_{11}), N \cap \text{lang}(\phi_{11}), \text{win})$
- $\phi_{21} = \text{randSplit}(P \setminus \text{lang}(\phi_{11}), N \setminus \text{lang}(\phi_{11}), \text{win})$
- $\phi_{22} = \text{randSplit}(P \setminus \text{lang}(\phi_{11}), N \cap \text{lang}(\phi_{11}), \text{win})$

The function $\text{aux}(P, N, \text{win})$ first construct a sub-specification (P_0, N_0) of (P, N) as follows. Select two random subsets $P_0 \subseteq P$ and $N_0 \subseteq N$, such that the cardinality of (P_0, N_0) is as large as possible but not exceeding *win*; in addition we require the cardinalities of P_0 and N_0 to be as equal as possible. Then (P_0, N_0) is sent the enumerator. If that returns OOM, $\text{aux}(P, N, \text{win}/2)$ is invoked, then $\text{aux}(P, N, \text{win}/4), \dots$ until the enumerator successfully learns a formula. Once ϕ_{11} is available, the remaining ϕ_{ij} can be learned in parallel. Finally, we return $(\phi_{11} \wedge \phi_{12}) \vee (\phi_{21} \wedge \phi_{22})$.

Our benchmarks in §7 show that deterministic and random splitting display markedly different behaviour on some benchmarks.

7 Evaluation of algorithm performance

This section quantifies the performance of our implementation. We are interested in a comparison with existing LTL learners, but also in assessing the impact on LTL learning performance of different algorithmic choices. We benchmark along the following quantitative dimensions: number of traces the implementation can handle, speed of learning, and cost of inferred formulae. In our evaluation we are facing several challenges.

- We are comparing a CUDA program running on a GPU with programs, sometimes written in Python, running on CPUs.
- We run our benchmarks in *Google Colab Pro*. It is unclear to what extent Google Colab Pro is virtualised. We observed variations in CPU and GPU running times, for all implementations measured.
- Existing benchmarks are too easy. They neither force our implementation to learn costly formulae, nor terminate later than the *measurement threshold* of around 0.2 seconds, a minimal time the Colab-GPU would take on any task, including toy programs that do nothing at all on the GPU.
- Lack of ground-truth: how can we evaluate the price we pay for scale, *i.e.*, the loss of formula minimality guarantees from algorithmic choices, when we do not know what this minimum is?

Hardware and software used for benchmarking. Benchmarks below run on GOOGLE COLAB PRO. We use Colab Pro because it is a widely used industry standard for running and comparing ML workloads. **Colab CPU parameters:** Intel Xeon CPU (“cpu family 6, model 79”), running at 2.20GHz, with 51 GB RAM. **Colab GPU parameters:** Nvidia Tesla V100-SXM2, with System Management Interface 525.105.17, with 16 GB memory. We use Python version 3.10.12, and CUDA version: 12.2.140. All our timing measurements are end-to-end (from invocation of $\text{learn}(P, N)$ to its termination), using Python’s `time` library.

Benchmark construction. Since existing benchmarks for LTL learning are too easy for our implementation, we develop new ones. A good benchmark should be tunable by a small number of explainable parameters that allows users to achieve hardness levels, from trivial to beyond the edge-of-infeasibility, and any point in-between. We now describe how we construct our new benchmarks.

- By $\text{BENCHBASE}(\Sigma, k, lo, hi)$ we denote the specifications generated using the following process: uniformly sample $2 \cdot k$ traces from $\{tr \in \text{traces}(\Sigma) \mid lo \leq \|tr\| \leq hi\}$. Split them into two sets (P, N) , each containing k traces.
- By $\text{SCARLET}(\Sigma, \phi, k, lo, hi)$ we mean using the sampler coming with Scarlet [32] to sample specifications (P, N) that are separated by ϕ , where ϕ is a formula over the alphabet Σ . Both, P and N , contain k traces each, and for each $tr \in P \cup N$ we have $lo \leq \|tr\| \leq hi$. The probability distribution Scarlet implements is detailed in Appendix E.

- By $\text{SAMPLING}(i, k, c)$, where $i, k \in \mathbb{N}$ and $c \in \{\text{conservative}, \text{-conservative}\}$, we mean the following process, which we also call *extension by sampling*.
 1. Generate (P, N) with $\text{BENCHBASE}(\mathbb{B}, i, 2, 5)$.
 2. Use our implementation to learn a minimal formula ϕ for (P, N) that is U -free and in NNF (for easier comparison with Scarlet, which can neither handle general negation nor U).
 3. Next we sample a specification (P', N') from $\text{SCARLET}(\mathbb{B}, \phi, k, 63, 63)$.
 4. The final specification is given as follows:
 - $(P \cup P', N \cup N')$ if $c = \text{conservative}$. Hence the final specification is a conservative extension of (P, N) and its cost is $\text{cost}(P, N)$.
 - (P', N') otherwise.

Note that the minimal formula required in Step 2 exists because for $i \leq 8$, any (P, N) generated by $\text{BENCHBASE}(\mathbb{B}, i, 2, 5)$ has the property that $\#\text{sc}^+(P \cup N) \leq 80 < 126$, so our algorithm uses neither RUCs nor $\text{D}\&\text{C}$, but, by construction, does an exhaustive bottom-up enumeration that is guaranteed to learn a minimal sound formula.
- By $\text{HAMMING}(\Sigma, l, \delta)$, with $l, \delta \in \mathbb{N}$, we mean specifications $(\{tr\}, \text{Hamm}(tr, \delta))$, where tr is sampled uniformly from all traces of length l over Σ .

Benchmarks from $\text{SAMPLING}(k, i, c)$ are useful for comparison with existing LTL learners, and to hone in on specific properties of our algorithm. But they don't fully address a core problem of using random traces: they tend to be too easy. One dimension of "too easy" is that specifications (P, N) of random traces often have tiny sound formulae, especially for large alphabets. Hence we use binary alphabets, the hardest case in this context. That alone is not enough to force large formulae. $\text{HAMMING}(\Sigma, l, \delta)$ works well in our benchmarking: it generates benchmarks that are hard even for the GPU. We leave a more detailed investigation why as future work. Finally, in order to better understand the effectiveness of MuellerHash in our RUC, we use the following deliberately simple map from CMs to 126 bits.

FIRST-K-PERCENT (FKP). This scheme simply takes the first $k\%$ of each CS in the CM. All remaining bits are discarded. The percentage k is chosen such that the result is as close as possible to 126 bits. e.g., for a $64 \cdot 63$ bit CM, $k = 3$.

Comparison with Scarlet. In this section we compare the performance of our implementation against Scarlet [32], in order better to understand how much performance we gain in comparison with a state-of-the-art LTL learner. Our comparison with Scarlet is implicitly also a comparison with Flie [29] and Syslite [3] because [32] already benchmarks Scarlet against them, and finds that Scarlet performs better. We use the following benchmarks in our comparison.

- All benchmarks from [32], which includes older benchmarks for Flie and Syslite.
- Two new benchmarks for evaluating scalability to high-cost formulae, and to high-cardinality specifications.

Table 1. Comparison of Scarlet with our implementation on existing benchmarks. Timeout is 2000 seconds. On the existing benchmarks our implementation never runs OOM or out-of-time (OOT), while Scarlet runs OOM in 5.9% of benchmarks and OOT in 3.8%. In computing the average speedup we are conservative: we use 2000 seconds whenever Scarlet runs OOT, if Scarlet runs OOM, we use the time to OOM. The “Lower Cost” column gives the percentages of instances where our implementation learns a formula with lower cost than Scarlet, and likewise for “Equal” and “Higher”. Here and below, “Ave” is short for the *arithmetic mean*. The column on the right reports the average speedup over Scarlet of our implementation.

D&C / Hsh / Win			Lower Cost	Equal Cost	Higher Cost	Ave Speed-up
Rand Split	FKP	64	11.2%	81.9%	6.9%	> 515x
		32	10.7%	79.8%	9.5%	> 483x
		16	10.9%	76.3%	12.8%	> 320x
	Mueller	64	12.3%	77.9%	9.8%	> 58x
		32	11.4%	72.7%	15.9%	> 433x
		16	11.2%	67.2%	21.6%	> 236x
Det Split	FKP	64	11.4%	74.4%	14.2%	> 173x
		32	10.4%	70.5%	19.2%	> 103x
		16	10.5%	66.1%	23.3%	> 52x
	Mueller	64	12.4%	78.1%	9.5%	> 263x
		32	10.5%	72.5%	16.9%	> 114x
		16	10.5%	66.5%	23.0%	> 46x

In all cases, we learn U-free formulae in NNF for easier comparison with Scarlet. This restriction hobbles our implementation which can synthesise cheaper formulae in unrestricted LTL.

Scarlet on existing benchmarks. We run our implementation in 12 different modes: D&C by deterministic, resp., random splitting, with two different hash functions (MuellerHash and FKP), and three different split windows (16, 32, and 64). The results are visualised in Table 1. We make the following observations. On existing benchmarks, our implementation usually returns formulae that are roughly of the same cost as Scarlet. They are typically only larger on benchmarks with a sizeable specification, e.g., 100000 traces, which forces our implementation into D&C, with the concomitant increase in approximation ratio due to the cost of recombination. However the traces are generated by sampling from trivial formulae (mostly Fp, Gp or $G\neg p$). Scarlet handles those well. [32] defines a parameterised family ϕ_{seq}^n that can be made arbitrarily big by letting n go to infinity. However in [32] $n < 6$, and even on those Scarlet run OOM/OOT, while our implementation handles all in a short amount of time. On existing benchmarks, our implementation runs on average at least 46 times faster. We believe that this surprisingly low worst-case average speedup is largely because the existing benchmarks are too easy, and the timing measurements are dominated by GPU startup latency. The comparison on harder benchmarks below shows this.

Table 2. On the left, comparison between Scarlet and our implementation on HAMMING(\mathbb{B}, l, δ) benchmarks with $l = 3, 6, 9, \dots, 48$ and $\delta = 1, 2$. Timeout is 2000 sec. Reported percentage is fraction of specifications that were successfully learned. On the right, comparison on benchmarks from SAMPLING($5, 2^k, conservative$) for $k = 3, 4, 5, \dots, 17$. All benchmarks were run to conclusion, Scarlet’s OOMs occurred between 1980.21 sec for $(2^{17}, 2^{17})$, and 16568.7 sec for $(2^{13}, 2^{13})$.

D&C / Hsh / Win		Delta=1	Delta=2	
Rand Split	FKP	64	100%	100%
		32	100%	100%
		16	100%	100%
	Mueller	64	100%	75%
		32	100%	75%
		16	100%	100%
Det Split	FKP	64	100%	100%
		32	100%	100%
		16	100%	100%
	Mueller	64	100%	88%
		32	100%	100%
		16	100%	100%
Scarlet			7%	7%

(# P, # N)	Our impl. Time (Cost)	Scarlet Time (Cost)
$(2^3, 2^3)$	0.31s (12)	1532.85s (19)
$(2^4, 2^4)$	0.32s (12)	1463.67s (17)
$(2^5, 2^5)$	0.36s (12)	2867.47s (17)
$(2^6, 2^6)$	0.34s (12)	5691.98s (17)
$(2^7, 2^7)$	0.63s (20)	OOM
$(2^8, 2^8)$	0.95s (19)	OOM
$(2^9, 2^9)$	0.72s (19)	OOM
$(2^{10}, 2^{10})$	1.09s (19)	OOM
$(2^{11}, 2^{11})$	1.32s (19)	OOM
$(2^{12}, 2^{12})$	1.66s (19)	OOM
$(2^{13}, 2^{13})$	2.46s (19)	OOM
$(2^{14}, 2^{14})$	4.62s (20)	OOM
$(2^{15}, 2^{15})$	8.35s (19)	OOM
$(2^{16}, 2^{16})$	15.52s (19)	OOM
$(2^{17}, 2^{17})$	30.49s (19)	OOM

Scarlet and high-cost specifications. The existing benchmarks can all be solved with small formulae. This makes it difficult to evaluate how our implementation scales when forced to learn high-cost formulae. In order to ameliorate this problem, we create a new benchmark using HAMMING(\mathbb{B}, l, δ) for $l = 3, 6, 9, \dots, 48$ and $\delta = 1, 2$. We benchmark with the aforementioned 12 modes. The left of Table 2 summarises the results. This benchmark clearly shows that Scarlet is mostly unable to learn bigger formulae, while our implementation handles all swiftly.

Scarlet and high-cardinality specifications. The previous benchmark addresses scalability to high-cost specifications. The present comparison with Scarlet seeks to quantify an orthogonal dimension of scalability: high-cardinality specifications. Our benchmark is generated by SAMPLING($i, 2^k, conservative$) for $i = 5$, and $k = 3, 4, 5, \dots, 17$. Using $i = 5$ ensures getting a few concrete times from Scarlet rather than just OOM/OOT; $k \leq 17$ was chosen as Scarlet’s sampler makes benchmark generation too time-consuming otherwise. The choice of parameters also ensures that the cost of each benchmark is moderate (≤ 20). This means any difficulty with learning arises from the sheer number of traces. Unlike the previous two benchmarks, we run our implementation only in one configuration: using MuellerHash, and random splitting with window size 64 (the difference between the variants is too small to affect the comparison with Scarlet in a substantial way). The results are also presented on the right of Table 2. This benchmark clearly shows that we can handle specifications at least 2048 times larger, despite Scarlet having approx. 3 times more memory available. Moreover,

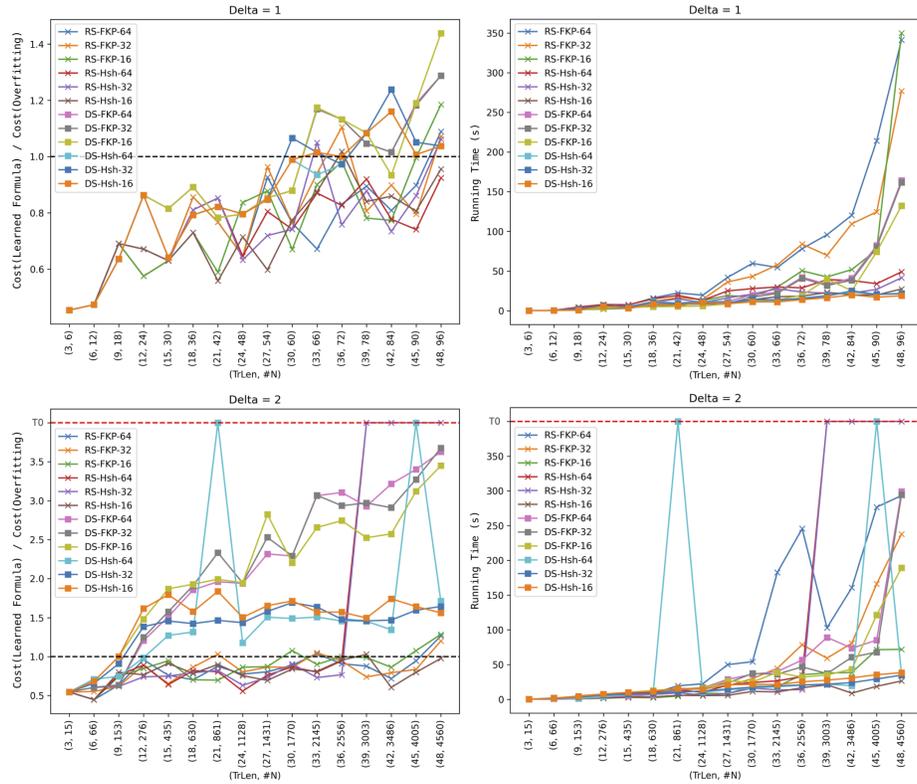


Fig. 3. Here RS means random-splitting, DS deterministic-splitting. The numbers 16, 32, 64 are the used splitting window. Hsh is short for MuellerHash. The x-axis is annotated by $(trLen, \#N)$, giving the length of the single trace in P , and the cardinality $\#N$ of N . TO denotes timeout. Timeout is 2000 seconds. On the left, the y-axis gives the ratio $\frac{\text{cost of learned formula}}{\text{cost of overfitting}}$, the dotted line at 1.0 is the cost of overfitting.

not only is our implementation much faster and can handle more traces, it also finds substantially smaller formulae in all cases where a comparison is possible.

Hamming benchmarks. We have already used HAMMING(...) in our comparison with Scarlet. Now we abandon existing learners, and delve deeper into the performance of our implementation by having it learn costly formulae. This benchmark is generated using HAMMING(\mathbb{B}, l, δ) for $l = 3, 6, 9, \dots, 48$ and $\delta = 1, 2$. As above, the implementation learns U-free formulae in NNF. Table 3 gives a more detailed breakdown of the results. The uniform cost of overfitting on each benchmark is given for comparison:

Length of tr	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
Cost of overfitting	22	38	55	73	92	111	129	147	164	182	201	219	238	256	274	292

This benchmark shows the following. Hamming benchmarks are hard for our implementation, and sometimes run for > 3 minutes: we successfully force our

Table 3. All run times are below the measurement threshold.

(# P, # N)	FKP		MuellerHash	
	AveExtraCost	OOM	AveExtraCost	OOM
(8, 8)	0.0%	0.0%	0.0%	0.0%
(12, 12)	2.4%	12.8%	1.9%	24.5%
(16, 16)	4.1%	19.3%	0.7%	32.6%
(20, 20)	4.1%	22.4%	0.4%	32.9%
(24, 24)	3.9%	24.0%	0.1%	33.6%
(28, 28)	3.6%	24.9%	0.0%	32.9%
(32, 32)	2.7%	26.3%	0.2%	40.3%

implementation to synthesise large formulae, and that has an effect on running time. The figures on the left show that random splitting typically leads to smaller formulae in comparison with deterministic splitting, especially for $\delta = 2$. Indeed, we may be seeing a sub-linear increase in formula cost (relative to the cost of overfitting) for random splitting, while for deterministic splitting, the increase seems to be linear. In contrast, the running time of the implementation seems to be relatively independent from the splitting mechanism. It is also remarkable that the maximal cost we see is only about 3.5 times the cost of overfitting: the algorithm processes P and N , yet over-fitting happens only on P , which contains a single trace. Hence the cost of over-fitting (292 in the worst case) is not affected by N , which contains up to 4560 elements (of the same length as the sole positive trace).

Benchmarking RUCs. Our algorithm uses RUCs, a novel cache admission policy, and it is interesting to gain a more quantitative understanding of the effects of (pseudo-)randomly rejecting some CMs. We cannot hope to come to a definitive conclusion here. Instead we simply compare MuellerHash with FKP, which neither distributes values uniformly across the hash space (our 126 bits) to minimise collisions, nor has the avalanche effect where a small change in the input produces a significantly different hash output. This weakness is valuable for benchmarking because it indicates how much a hash function can degrade learning performance. (Note that for the edge case of specifications that can be separated from just the first $k\%$ alone, FKP should perform better, since it leaves the crucial bits unchanged.) The benchmark data is generated with `SAMPLING(8, 24, conservative)`. Table 3 summarises our measurements. We note the following. The loss in formula cost is roughly constant for each hash: it stabilises to around 0.2% for MuellerHash, and a little above 2.5% for FKP. Hence MuellerHash is an order of magnitude better. Nevertheless, even 2.5% should be irrelevant in practice, and we conjecture that replacing MuellerHash with a cryptographic hash will have only a moderate effect on learning performance. A surprising number of instances run OOM, more so as specification size grows, with MuellerHash more than FKP. We leave a detailed understanding of these phenomena as future work.

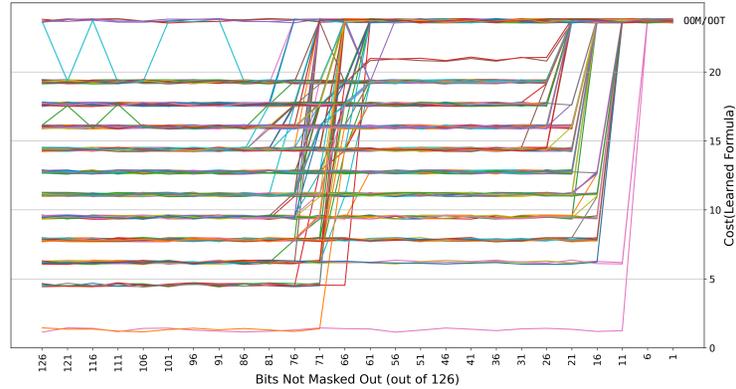


Fig. 4. Effects of masking on formula cost. Timeout is 200 seconds. Colours correspond to different (P, N) . The slight 'wobble' on all graphs is deliberately introduced for readability, and is not in the data.

Masking. The previous benchmarks suggested that naive hash functions like FKP sometimes work better than expected. Our last benchmark seeks to illuminate this in more detail and asks: can we relate the information loss from hashing and the concomitant increase in formula cost? A precise answer seems to be difficult. We run a small experiment: after MuellerHashing CMs of size 64×63 bits to 126 bits, we add an additional information loss phase: we *mask out* k bits, *i.e.*, we set them to 0. This destroys all information in the k masked bits. After masking, we run the uniqueness check. We sweep over $k = 1, \dots, 126$ with stride 5 to mask out benchmarks generated with `SAMPLING(8, 32, \neg conservative)`. Figure 4 shows the results. Before running the experiments, the authors expected a gradual increase of cost as more bits are masked out. Instead, we see a phase transition when approx. 75 to 60 bits are not masked out: from minimal cost formulae before, to OOM/OOT after, with almost no intermediate stages. Only a tiny number of instances have 1 or 2 further cost levels between these two extremes. We leave an explanation of this surprising behaviour as future work.

8 Conclusion

The present work demonstrates the effectiveness of carefully tailored algorithms and data structures for accelerating LTL learning on GPUs. We close by summarising the reasons why we achieve scale: high degree of parallelism inherent in generate-and-test synthesis; application of divide-and-conquer strategies; relaxed uniqueness checks for (pseudo-)randomly curtailing the search space; and succinct, suffix-contiguous data representation, enabling exponential propagation where LTL connectives map directly to branch-free machine instructions with predictable data movement. All but the last are available to other learning tasks that have suitable operators for recombination of smaller solutions.

LTL and GPUs, a match made in heaven.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining Specifications. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 4–16. POPL '02, Association for Computing Machinery, New York, NY, USA (2002). <https://doi.org/10.1145/503272.503275>
2. Anderson, S.E.: Bit twiddling hacks: Round up to the next highest power of 2 (2005), <https://graphics.stanford.edu/~seander/bithacks.html>
3. Arif, M.F., Larraz, D., Echeverria, M., Reynolds, A., Chowdhury, O., Tinelli, C.: SYSLITE: Syntax-Guided Synthesis of PLTL Formulas from Finite Traces. In: Proceedings of the International Conference on Formal Methods in Computer Aided Design, FMCAD. pp. 93–103 (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_16
4. Bernardi, O., Giménez, O.: A Linear Algorithm for the Random Sampling from Regular Languages. *Algorithmica* **62**(1–2), 130–145 (feb 2012). <https://doi.org/10.1007/s00453-010-9446-5>
5. Camacho, A., McIlraith, S.A.: Learning Interpretable Models Expressed in Linear Temporal Logic. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2019, Berkeley, CA, USA, July 11–15, 2019. pp. 621–630. AAAI Press (2019), <https://ojs.aaai.org/index.php/ICAPS/article/view/3529>
6. Camacho, A., McIlraith, S.A.: Learning interpretable models expressed in linear temporal logic. *International Conference on Automated Planning and Scheduling, ICAPS* **29** (2019), <https://ojs.aaai.org/index.php/ICAPS/article/view/3529>
7. Dally, W.J., Turakhia, Y., Han, S.: Domain-Specific Hardware Accelerators. *Commun. ACM* **63**(7), 48–57 (jun 2020). <https://doi.org/10.1145/3361682>
8. David, C., Kroening, D.: Program Synthesis: Challenges and Opportunities . *Philosophical Transactions A* **375** (2017)
9. De Giacomo, G., Vardi, M.Y.: Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. p. 854–860. *IJCAI '13*, AAAI Press (2013)
10. Fijalkow, N., Lagarde, G.: The complexity of learning linear temporal formulas from examples. In: Proceedings of the Fifteenth International Conference on Grammatical Inference. *Proceedings of Machine Learning Research*, vol. 153, pp. 237–250. PMLR (23–27 Aug 2021), <https://proceedings.mlr.press/v153/fijalkow21a.html>
11. Gabel, M., Su, Z.: Javert: Fully automatic mining of general temporal properties from dynamic traces. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 339–349. *SIGSOFT '08/FSE-16*, Association for Computing Machinery, New York, NY, USA (2008). <https://doi.org/10.1145/1453101.1453150>
12. Gabel, M., Su, Z.: Symbolic Mining of Temporal Specifications. In: Proceedings of the 30th International Conference on Software Engineering. p. 51–60. *ICSE '08*, Association for Computing Machinery, New York, NY, USA (2008). <https://doi.org/10.1145/1368088.1368096>
13. Gabel, M., Su, Z.: Online Inference and Enforcement of Temporal Properties. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1. p. 15–24. *ICSE '10*, Association for Computing Machinery, New York, NY, USA (2010). <https://doi.org/10.1145/1806799.1806806>

14. Gaglione, J., Neider, D., Roy, R., Topcu, U., Xu, Z.: Maxsat-based temporal logic inference from noisy data. *Innovations in Systems and Software Engineering* **18**(3), 427–442 (2022). <https://doi.org/10.1007/S11334-022-00444-8>
15. Gulwani, S., Polozov, O., Singh, R.: *Program Synthesis*. Now Foundations and Trends (2017), <http://ieeexplore.ieee.org/document/8187066>
16. Hennessy, J., Patterson, D.: *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann (2017)
17. Hwu, W.M.W., Kirk, D.B., Hajj, I.E.: *Programming Massively Parallel Processors*. Morgan Kaufmann (2022)
18. Ielo, A., Law, M., Fionda, V., Ricca, F., De Giacomo, G., Russo, A.: Towards ILP-Based LTL_f Passive Learning. In: *Inductive Logic Programming*. pp. 30–45. Springer Nature Switzerland, Cham (2023)
19. Jeppu, N., Melham, T., Kroening, D., O’Leary, J.: Learning Concise Models from Long Execution Traces. In: *Proceedings of the 57th ACM/IEEE Design Automation Conference, DAC*. pp. 1–6 (2020). <https://doi.org/10.1109/DAC18072.2020.9218613>
20. Jünger, D.: WARPCORE: Hashing at the speed of light on modern CUDA-accelerators. <https://github.com/sleepyjack/warpcore> (November 2022)
21. Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., Schmidt, B.: WarpCore: A Library for fast Hash Tables on GPUs. In: *Proceedings of the 27th International Conference on High Performance Computing, Data, and Analytics, HiPC*. pp. 11–20 (2020). <https://doi.org/10.1109/HiPC50609.2020.00015>
22. Kim, J., Muise, C., Shah, A., Agarwal, S., Shah, J.: Bayesian inference of linear temporal logic specifications for contrastive explanations. In: *International Joint Conference on Artificial Intelligence, IJCAI* (2019). <https://doi.org/10.24963/ijcai.2019/776>
23. Kirilin, V., Sundarajan, A., Gorinsky, S., Sitaraman, R.K.: RL-Cache: Learning-Based Cache Admission for Content Delivery. In: *Proceedings of the 2019 Workshop on Network Meets AI & ML*. p. 57–63. NetAI’19, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3341216.3342214>
24. Lemieux, C., Beschastnikh, I.: Investigating Program Behavior Using the Texada LTL Specifications Miner. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. pp. 870–875. IEEE Computer Society, Los Alamitos, CA, USA (nov 2015). <https://doi.org/10.1109/ASE.2015.94>
25. Lemieux, C., Park, D., Beschastnikh, I.: General LTL Specification Mining. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. pp. 81–92. IEEE Computer Society, Los Alamitos, CA, USA (nov 2015). <https://doi.org/10.1109/ASE.2015.71>
26. Luo, W., Liang, P., Du, J., Wan, H., Peng, B., Zhang, D.: Bridging LTL_f Inference to GNN Inference for Learning LTL_f Formulae. *Proceedings of the AAAI Conference on Artificial Intelligence* **36**(9), 9849–9857 (Jun 2022). <https://doi.org/10.1609/aaai.v36i9.21221>
27. Mascle, C., Fijalkow, N., Lagarde, G.: Learning temporal formulas from examples is hard (2023). <https://doi.org/10.48550/arXiv.2312.16336>
28. Mittal, S.: A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* **49**(2) (aug 2016). <https://doi.org/10.1145/2907071>
29. Neider, D., Gavran, I.: Learning linear temporal properties. In: *Formal Methods in Computer Aided Design, FMCAD*. pp. 1–10 (2018). <https://doi.org/10.23919/FMCAD.2018.8603016>

30. Peng, B., Liang, P., Han, T., Luo, W., Du, J., Wan, H., Ye, R., Zheng, Y.: PURLTL: Mining LTL Specification from Imperfect Traces in Testing. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1766–1770. IEEE Computer Society, Los Alamitos, CA, USA (Sep 2023). <https://doi.org/10.1109/ASE56229.2023.00202>
31. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, FOCS. pp. 46–57 (1977). <https://doi.org/10.1109/SFCS.1977.32>
32. Raha, R., Roy, R., Fijalkow, N., Neider, D.: Scalable Anytime Algorithms for Learning Fragments of Linear Temporal Logic. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS. pp. 263–280. Springer International Publishing, Cham (2022)
33. Valizadeh, M., Berger, M.: Search-Based Regular Expression Inference on a GPU. Proceedings of the ACM on Programming Languages, Issue PLDI 7 (jun 2023). <https://doi.org/10.1145/3591274>, technical report available at <https://arxiv.org/abs/2305.18575>, implementation: <https://github.com/MojtabaValizadeh/paresy>
34. Weimer, W., Necula, G.C.: Mining Temporal Specifications for Error Detection. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 461–476. TACAS’05, Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_30
35. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining Temporal API Rules from Imperfect Traces. In: Proceedings of the 28th International Conference on Software Engineering. p. 282–291. ICSE ’06, Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1134285.1134325>
36. Yogananda Jeppu, N.: Learning symbolic abstractions from system execution traces. Ph.D. thesis, University of Oxford (2022)

A Worked examples of CS manipulation by bitshifts

In this section we illustrate constructing and using CSs by examples.

Characteristic sequences. Consider the atomic proposition g where $g \in \Sigma$ is a character. Here are some examples of CS for formulae $X^n g$ over the word “squeegee”.

ϕ	cs
g	00000100
Xg	00001000
XXg	00010000
$XXXg$	00100000
$XXXXg$	01000000
$XXXXXg$	10000000
$XXXXXXg$	00000000
$XXXXXXXg$	00000000

The next example is also CSs over “squeegee”, but now for formulae of the form $\bigvee_n X^n g$, which are closely related to F.

ϕ	cs
g	00000100
$g \vee Xg$	00001100
$g \vee Xg \vee X^2g$	00011100
$g \vee Xg \vee X^2g \vee X^3g$	00111100
$g \vee Xg \vee X^2g \vee X^3g \vee X^4g$	01111100
$g \vee Xg \vee X^2g \vee X^3g \vee X^4g \vee X^5g$	11111100
$g \vee Xg \vee X^2g \vee X^3g \vee X^4g \vee X^5g \vee X^6g$	11111100
$g \vee Xg \vee X^2g \vee X^3g \vee X^4g \vee X^5g \vee X^6g \vee X^7g$	11111100

Exponential propagation for F and U. To see exponential propagation in action consider a formula ϕ that is represented by cs over some trace. We begin by computing the CS for $F\phi$ with exponential propagation. In order to visualise how the CSs are changed by assignments, we use the notation $\{pre\} P \{post\}$ from Hoare logic. It should be read as: if, before executing of program P , the state is correctly described by pre , then, after P has executed, the assertion $post$ holds. Assume the initial state is $cs = 0000000000000100$, which is a CS of length 16. Then we compute the CS for $F\phi$, as given by (the body of) `branchfree_F` with the following steps.

$$\begin{aligned}
\{cs = 0000000000000100\} \text{ cs } | = \text{cs} \ll 1 & \{cs = 0000000000001100\} \\
\{cs = 0000000000001100\} \text{ cs } | = \text{cs} \ll 2 & \{cs = 0000000000111100\} \\
\{cs = 0000000000111100\} \text{ cs } | = \text{cs} \ll 4 & \{cs = 0000001111111100\} \\
\{cs = 0000001111111100\} \text{ cs } | = \text{cs} \ll 8 & \{cs = 1111111111111100\}
\end{aligned}$$

At the end of this computation we reach $cs = 1111111111111100$ and that is the CS for $F\phi$, as required. In order to see exponential propagation for U in action, let us assume we are given two CSs, $cs_1 = 11111110$ which represents ϕ and

$cs_2 = 00000001$ representing ψ , both over some trace of length 8. We construct the CS representing $\phi \cup \psi$. We carry out exponential propagation as given by (the body of) `branchfree_U` with the following steps.

```
{cs1 = 11111110, cs2 = 00000001} cs2 |= cs1 & (cs2 << 1) {cs1 = 11111110, cs2 = 00000011}
{cs1 = 11111110, cs2 = 00000011} cs1 &= (cs1 << 1) {cs1 = 11111100, cs2 = 00000011}
{cs1 = 11111100, cs2 = 00000011} cs2 |= cs1 & (cs2 << 2) {cs1 = 11111100, cs2 = 00001111}
{cs1 = 11111100, cs2 = 00001111} cs1 &= (cs1 << 2) {cs1 = 11110000, cs2 = 00001111}
{cs1 = 11110000, cs2 = 00001111} cs2 |= cs1 & (cs2 << 4) {cs1 = 11110000, cs2 = 11111111}
```

Now $cs_2 = 11111111$ is the CS for $\phi \cup \psi$, as required.

B Correctness and complexity of branch-free semantics for temporal operators

We reproduce here a slightly more general version (for any length) of the exponentially propagating algorithms for computing F and U.

<pre>1 def branchfree_F(cs): 2 L = len(cs) 3 for i in range(log(L)+1): 4 cs = cs << 2**i 5 return cs</pre>	<pre>1 def branchfree_U(cs1, cs2): 2 L = len(cs1) 3 for i in range(log(L)+1): 4 cs2 = cs1 & (cs2 << 2**i) 5 cs1 &= cs1 << 2**i 6 return cs2</pre>
---	--

They are lifted to CMs pointwise. As a warm-up, let us start with F.

Lemma 1. *Let cs be the characteristic sequence for ϕ over a trace of length L . The algorithm above computes the characteristic sequence for $F\phi$. Assuming bitwise boolean operations and shifts by powers of two have unit costs, the complexity of the algorithm is $O(\log(L))$.*

To ease notations, let us introduce $F_{\leq p}$ with the semantics $tr, j \models F_{\leq p}\phi$ if there is $j \leq k < \min(j+p, \|tr\|)$ with $tr, k \models \phi$.

Proof. Let us write cs for the characteristic sequence for ϕ over tr , and cs_i for the characteristic sequence after the i th iteration. We write $\log(x)$ as a shorthand for $\lfloor \log_2(x) \rfloor$. We write Fcs for $F\phi$, and $F_{\leq p}cs$ for $F_{\leq p}\phi$. We show by induction that for all $i \in [0, \log(L) + 1]$, for all tr of length L we have:

$$\forall j \in [0, L], tr, j \models cs_i \iff tr, j \models F_{\leq 2^i}cs.$$

This is clear for $i = 0$, as it boils down to $cs_0 = cs$. Assuming it holds for i , by definition $cs_{i+1}(j) = cs_i(j) \vee (cs_i \ll 2^i)(j) = cs_i(j) \vee cs_i(j + 2^i)$, hence

$$\begin{aligned} tr, j \models cs_{i+1} &\iff tr, j \models cs_i, \text{ or } tr, j + 2^i \models cs_i \\ &\iff tr, j \models F_{\leq 2^i}cs, \text{ or } tr, j + 2^i \models F_{\leq 2^i}cs \\ &\iff tr, j \models F_{\leq 2^{i+1}}cs. \end{aligned}$$

This concludes the induction proof. For $i = \log(L)$ we obtain

$$\forall j \in [0, L], tr, j \models cs_i \iff tr, j \models F_{\leq L}cs \iff tr, j \models Fcs,$$

since clearly $F = F_{\leq L}$, when restricted to traces not exceeding L in length. \square

We now move to \mathbf{U} .

Lemma 2. *Let cs_1, cs_2 the characteristic sequences for ϕ_1 and ϕ_2 , both over traces of length L . The algorithm above computes the characteristic sequence for $\phi_1 \mathbf{U} \phi_2$. Assuming bitwise boolean operations and shifts by powers of two have unit costs, the complexity of the algorithm is $O(\log(L))$.*

Again to ease notations, let us introduce $\mathbf{U}_{\leq p}$ with the semantics $tr, j \models \phi_1 \mathbf{U}_{\leq p} \phi_2$ if there is $j \leq k < \min(j+p, \|tr\|)$ such that $tr, k \models \phi_2$ and for all $i \leq k' < k$ we have $tr, k' \models \phi_1$. We will also need $\mathbf{G}_{\geq p}$ defined with the semantics $tr, j \models \mathbf{G}_{\geq p} \phi$ if for all $j \leq k < \min(j+p, \|tr\|)$ we have $tr, k \models \phi$.

Proof. Let us write $cs_{1,i}$ and $cs_{2,i}$ for the respective characteristic sequences after the i th iteration. We show by induction that for all $i \in [0, \log(L) + 1]$, for all tr of length L , for all $j \in [0, L]$, we have:

- $tr, j \models cs_{1,i} \iff tr, j \models \mathbf{G}_{\geq 2^i} cs_1$, and
- $tr, j \models cs_{2,i} \iff tr, j \models cs_1 \mathbf{U}_{\leq 2^i} cs_2$.

This is clear for $i = 0$, as it boils down to $cs_{1,0} = cs_1$ and $cs_{2,0} = cs_2$. Assume it holds for i . Let us start with $cs_{1,i+1}$: by definition

$$\begin{aligned} cs_{1,i+1}(j) &= cs_{1,i}(j) \wedge (cs_{1,i} \ll 2^i)(j) \\ &= cs_{1,i}(j) \wedge cs_{1,i}(j + 2^i), \end{aligned}$$

hence it is the case that

$$\begin{aligned} tr, j \models cs_{1,i+1} &\iff tr, j \models cs_{1,i}, \text{ and } tr, j + 2^i \models cs_{1,i} \\ &\iff tr, j \models \mathbf{G}_{\geq 2^i} cs_1, \text{ and } tr, j + 2^i \models \mathbf{G}_{\geq 2^i} cs_1 \\ &\iff tr, j \models \mathbf{G}_{\geq 2^{i+1}} cs_1. \end{aligned}$$

Now, by definition $cs_{2,i+1}(j) = cs_{2,i}(j) \vee (cs_{1,i}(j) \wedge (cs_{2,i} \ll 2^i)(j))$, which is equal to $cs_{2,i}(j) \vee (cs_{1,i}(j) \wedge cs_{2,i}(j + 2^i))$. Hence

$$\begin{aligned} tr, j \models cs_{2,i+1} &\iff tr, j \models cs_{2,i}, \text{ or } (tr, j \models cs_{1,i}, \text{ and } tr, j + 2^i \models cs_{2,i}) \\ &\iff tr, j \models cs_1 \mathbf{U}_{\leq 2^i} cs_2, \text{ or } \\ &\quad (tr, j \models \mathbf{G}_{\geq 2^i} cs_1, \text{ and } tr, j + 2^i \models cs_1 \mathbf{U}_{\leq 2^i} cs_2) \\ &\iff tr, j \models cs_1 \mathbf{U}_{\leq 2^{i+1}} cs_2. \end{aligned}$$

This concludes the induction proof. For $i = \log(L)$ we obtain

$$\forall j \in [0, L], tr, j \models cs_{2,i} \iff tr, j \models cs_1 \mathbf{U}_{\leq L} cs_2 \iff tr, j \models cs_1 \mathbf{U} cs_2,$$

since clearly $\mathbf{U} = \mathbf{U}_{\leq L}$ for all sufficiently short traces. \square

C Related work: cache admission policies

Caches are one of the most widely used concepts in computer science for increasing a system’s performance. Caches reserve memory to store data for future retrieval, a space-time trade-off. The fundamental issue with caching is thus dealing with the limitations of the available memory. There are two main points in time where this is done:

- Before caching: a cache *admission* policy (CAP) decides if an item is worth caching.
- After caching: a cache *replacement* policy (CRP) decides which existing entry need to go to make space for a new one.

They are not mutually exclusive. Both have in common that they seek to make a lightweight prediction on whether a piece of data will be likely used again soon in order to decided what to admit / replace. In hardware processors, the CAP is usually trivial: cache everything⁹. This is based on the empirical observation of *temporal locality*: for most programs, the fact that a piece of data is used now is a reasonable predictor for it being used again soon. In contrast with their simple CAPs, processors use sophisticated CRPs, including least-recently-used, most-recently-used, FIFO, or LIFO. All require that the cache associates meta-data with cached elements, and do not make sense as CAP. The only exception is the (pseudo-)random CRP.

The situation is quite different in network-based caching, such as *content delivery networks* like Akamai, which cache frequently requested data to make the internet faster (see, e.g., [23]). Here the CAP is usually based on the fact that network requests on the internet adhere to some reasonably well understood probability distribution (e.g., Zipf-like) and it is reasonably cheap to decide where in the distribution a candidate for cache entry sits.

To the best of our knowledge, our RUC of (pseudo-)randomly rejecting potential CMs, is new. This is because, unlike the caching discussed above, each CM is accessed equally often by future stages of the algorithm, so predicting how often a cache entry will be used is pointless. The relevant question in LTL learning is: does this CM play a necessary role in a minimal (or low-cost) LTL formula? Answering this question is hard, probably as hard as as LTL learning itself, hence infeasible as a CAP. Understanding better why our (pseudo-)random CAP is effective is an interesting question for future work, and the surprising

⁹ An exception are processors with NUMA caches where the CAP question “*should* this be placed in the cache?” is refined to “*where* in the cache should this be placed?” so as to minimise the time required to transport cached data to the processor element that needs it. In addition, modern processors have dedicated *prefetching* components [28], which analyse the stream of memory reads, detect patterns in it (e.g., if memory access to addresses x , $x + 8$, $x + 16$, $x + 24$ becomes apparent, a prediction is made that the next few memory reads will be to $x + 32$, $x + 40$, $x + 48$ and so on) and request data items at addresses extending this pattern ahead of time, which places them in the cache.

benchmarks in §7 might be taken to indicate that more radical CAPs might be viable.

D Related work: detailed comparison with Paresy [33]

PARESY [33], the first GPU-accelerated regular expression inferencer, was the main influence on the present work and it is important to clarify how both relate. Table 4 discusses the key points. In summary, both have a similar high-level structure (e.g., bottom-up enumeration, language cache, bitvector representation), but are completely different in low-level details (e.g., staging, redundant representation, shift-based). This opens interesting avenues for further work. It is clear that regular expression inference, and indeed any form of search based program synthesis can benefit from D&C and from RUCs, albeit at the cost of losing minimality guarantees. A more complex question is: can the costs arising from the redundant representation of suffixes be avoided, for example by a staged guide table for LTL connectives, similar to the staged guide table in [33], and what would be the effects on performance of such a dramatic change of algorithm be?

E Scarlet’s sample generation

Our $\text{SAMPLING}(i, k, c)$ is built on Scarlet’s sampler [32] which implements the algorithm proposed in [4]. We now sketch how it works. For sampling positive traces, proceed as follows.

- Given a suitable formula, convert the formula into an equivalent DFA.
- Sampling starts from the initial state of the DFA.
- From a state, sample uniformly at random a transition, register the character, and move to the corresponding target state of the transition.
- If the state is accepting, there’s a non-zero probability to stop (uniform, counted as a transition). Otherwise proceed with sampling more characters.

For negative traces, the same algorithm is used, albeit on the complement DFA.

Table 4. Comparison of PARESY [33] and the present work regarding core algorithmic features.

Feature	PARESY	Present paper
GPU-based	Yes	Yes
What is learned	Regular expressions	LTL formulae
Generate-and-test	Yes	Yes
Generation	Bottom-up enumeration	Bottom-up enumeration
Exhaustive	Yes	No
Source of incomplete generation	N/A	RUC
Closure	Infix	Suffix
Search space	(P, N) quotiented by infix-closure	(P, N) quotiented by suffix-closure
Irredundant representation of candidates	Yes	No
Bitvector representation	Yes	Yes
Suffix-contiguous representation of candidates	No	Yes
Max size of bitvector in given implementation	126 bits	64*63 bits
Language cache	Yes	Yes
Cache admission policy	Precise uniqueness check	Relaxed uniqueness check (false positives allowed)
D&C	No	Yes
Semantics of candidates	Staged with guide table	Exponentially propagated bitshifts
Staging w.r.t. (P, N)	High (e.g., guide table)	Low
Scalable	No	Yes
Sound	Yes	Yes
Complete	Yes	Yes
Minimality guarantees	Yes	No
Causes of non-minimality	N/A	RUC and D&C