Secco: Codesign for Resource Sharing in Regular-Expression Accelerators

Jackson Woodruff
University of Edinburgh
Edinburgh, Scotland
J.C.Woodruff@sms.ed.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, Scotland
sam.ainsworth@ed.ac.uk

Michael F.P. O'Boyle
University of Edinburgh
Edinburgh, Scotland
mob@inf.ed.ac.uk

Regular expressions have applications in fields from network intrusion detection to bioinformatics. This has led to the wide-spread development of FPGA-based hardware accelerators. However, reprogramming these accelerators for different regular expressions is slow and difficult due to FPGA toolchain overheads. Translation overlays that enable fast repurposing of existing accelerator layouts have been proposed. However, these assume the underlying accelerator design exists and is well-designed. For many domains, this is impossible to do by-hand: requiring simple patterns and significant programmer effort.

We present Secco, a compiler targeting symbol-only reconfigurable architectures, which codesigns the (fast-reconfigured) translation overlay and the (slow-reconfigured) underlying accelerator layout, by reusing the same hardware when simple overlay translations are available, and generating new hardware otherwise. This allows significant efficiency improvements: Secco enables 5.9x more expressions using the same resources across all ANMLZoo benchmarks compared to regular expression tool chains like REAPR. This enables large numbers of diverse regular expressions to be accelerated with context-switching overheads in the milliseconds.

***Index Terms*—regex, hardware accelerator, compiler**

I. Introduction

Regular expressions are central to a number of fields, from verification [1] to bioinformatics to network intrusion detection [2]. Finding faster, more scalable and more energy efficient accelerators for regular expressions has been an active area [3]. FPGA accelerators provide a significantly more energy-efficient way to accelerate regular expressions compared with CPU-based alternatives such as Hyperscan [4], by taking advantage of the MISD nature of the task. However, FPGAs have limited space, with small FPGAs incapable of accelerating the thousands of regular expression patterns found in network intrusion detection rulesets such as Snort [5].

Reconfiguration allows more regular expressions to be considered at the cost of run-time overhead, but fully reprogramming FPGAs at run-time is typically infeasibly slow, and synthesising all configurations even slower [6]. *Partial reconfiguration* leveraging the similarity between different expressions for the same task by generating *translation overlays* can address this problem. This *symbol-only-reconfiguration* [6] has previously been used to enable fast partial reconfiguration of a REAPR [7] regular expression accelerator. It enables near-zero-overhead reprogrammability for regular-expression accelerators, given a suitably generic base accelerator. These techniques are not limited to FPGAs — Micron's Automata Processor implements a similar technique called Symbol Replacement [8] to overcome the high latency of reconfiguration.

Currently, however, such translation-overlay architectures lack automation. They require generic base accelerators and overlay configurations to be designed by-hand. This is already a challenging task for domains with simple, repetitive, accelerator structures [6] but is impossible for domains with millions of unique patterns such as bioinformatics applications.

Other overlays such as *stateless translation* [9] have compilers that can target arbitrary underlying accelerators, but these are significantly less powerful than symbol-only reconfiguration. They require the use of a CPU to check matches, harming scalability, and have more rigid similarity constraints on translations, harming compression rates. We show that symbol-only reconfiguration can enable 5.9x more expressions, a 3.4x improvement in expression density compared with the state-of-the-art [9]. Further, existing compilers for stateless translators can only target pre-designed accelerators, rather than choosing, configuring and co-optimising the accelerator.

In this work, we present Secco (Structural Expression Codesign Compiler), a compiler capable of taking a large ruleset with many regular expressions, and producing base accelerators and reconfiguration sets to be used in a symbol-only-reconfiguration architecture. We build upon the infrastructure in RXPSC [9], a compiler that enables multiple regular expressions to share the same accelerator using stateless translation, to design new mechanisms to automatically compile *symbol-only reconfiguration* [6] translation layers instead. We also provide a new backend for RXPSC that can *generate* new accelerators rather than just being provided them as input, allowing Secco to codesign the translation overlay with the accelerator for both stateless translation and symbol-only reconfiguration.

We demonstrate the applicability of our compiler across the regex benchmarks in ANMLZoo [2] and provide an in-depth analysis of a bioinformatics use case, showing that we can represent 5.9× more regular expressions than are possible with a basic REAPR accelerator, and 3.4× more than competing compression techniques [9]. This compression is vital both to minimise the area of the accelerator while maximising coverage, and as a proxy measure for how Secco can successfully generate generic, reusable accelerator logic.

This is a preprint of the article in ASP-DAC 2024.

We make the following contributions:

- We present a novel compilation technique targeting symbol-only reconfigurable architectures.
- We develop a method for co-design of generic symbol-only reconfigurable hardware accelerators and stateless translation accelerators from sets of regular expressions.
- We demonstrate that there are sufficient patterns in existing rulesets to be exploited with symbol-only reconfiguration architectures.

II. Motivation

Secco introduces a co-design framework: enabling automated design and reconfiguration of symbol-only reconfigurable accelerators. The size, and number of, regular expression accelerators dictates the resources required on an FPGA, and the number of resources consumed dictates the number of expressions that can be accelerated. In this section we describe how existing partial reconfiguration approaches work and how symbol-only reconfiguration is more effective for regular expression pattern matching.

A. Regular Expressions for Network Intrusion Detection

Regular expressions are a well-known way of representing text-matching patterns. They are constructed from a series of characters, with operators like `*` to represent zero or more and `|` to represent either or. For example, the expression `ab*(c|d)` matches strings `ac`, `abbc`, `abbd` etc.

B. Existing Approaches

Two key approaches to resource sharing for regular expressions exist, prefix merging, in which expressions with identical prefixes are merged and stateless translation, in which an overlay is introduced allowing different expressions to use the same accelerator.

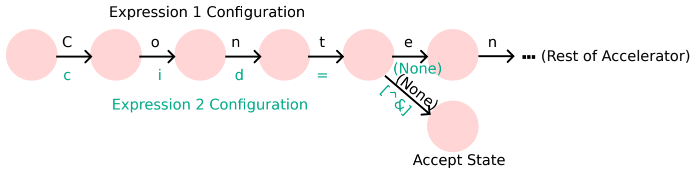
1) Prefix Merging: Prefix merging [2] is a technique where expressions that share prefixes share accelerator resources. For example, the two expressions `ab*` and `ac*` share a common prefix, `a`.

Prefix-merging techniques are limited by their requirements for exact equality of prefixes. For example although the expressions `ab*` and `cd*` share significant similarity, they cannot share resources using prefix merging.

2) Stateless Translators: Stateless translators [9] are a state of the art approach that can automatically run regular expressions on accelerators for different expressions. In stateless translation, a symbol lookup implemented as a BRAM is used to translate the character input stream character-by-character. Figure 3a shows this architecture. This allows two different expressions to share much of the same underlying hardware. While stateless translation can handle more cases than prefix merging, it is still limited. For example, the expression `aa*` cannot use an accelerator for `ab*`, as this would require translation of `a` to both `a` and `b`.

C. Symbol-Only Reconfiguration

Symbol-only reconfiguration is a technique where edges within a non-deterministic finite automata (NFA) are reprogrammed to support different symbols, introduced by Bo [6]. This enables fast reprogramming of regular expression accelerators, and is supported by FPGA accelerators [6] and



Expression 1 Configuration
C
o
n
t
e
n
... (Rest of Accelerator)
c
i
d
=
(None)
(None)
[^&]
Expression 2 Configuration
Accept State

Fig. 1: Expressions (from table I) merged to share an accelerator using symbol-only reconfiguration.

`Content-Type\s*\x3A[^\n\x3A]{256}`	`cid=[^&]`

TABLE I: Two rules from the Snort ruleset.

ASICs [10]. Figure 3b shows an FPGA-based symbol-only reconfigurable design.

Symbol-only reconfiguration offers more power than stateless translation. A stateless translator cannot accelerate `aa*` using an accelerator for `ab*` as it involves a symbol clash (where `a` must be translated to `a`, and also to `b`). In contrast, symbol-only reconfiguration enables reprogramming at a finer-grained level, allowing a mapping to `a` at the first symbol and a mapping to `b` in the second symbol.

Architectures that implement this style of reconfigurability typically compile expressions to *homogeneous* NFAs, which are NFAs where every edge into a node has the same symbol. This allows state-activation checks to be simplified, and decouples the structure of the NFA from the symbols, enabling such symbol-only reconfiguration.

D. Example

Figure 2 shows how Secco generates generic expression accelerators that can be quickly updated.

On the left, we see prefix merging and stateless translation reducing the resources required by an accelerator for both input expressions, `a(b|c)` and `a(d|e)`. On the right, the regular expressions show less similarity. As there is no common prefix between the expressions `a(a|b)` and `d(e|f)`, prefix merging can't operate. Similarly, the expressions are not compatible with symbol only reconfiguration, as `a` cannot be translated to both `d` and `e`. Using symbol-only reconfiguration, Secco generates a relabelling so that `a(a|b)` and `d(e|f)` can be multiplexed onto a single accelerator.

For a concrete example (Table I) we use the Snort ruleset [5], an open-source set of rules developed to enable network intrusion detection.

Although these rules appear visibly different, we can design a common accelerator that supports both using symbol-only reconfiguration to enable the accelerator to run either pattern (e.g. Figure 1). In this case, this is particularly useful as the first expression is intended to be run on UDP packets into port 554, while the second is intended to run on TCP packets into port 80. Rule compression is particularly relevant for embedded targets, which benefit from low-power FPGA-based scanning and do not have the resources required to run the full ruleset on-device.

Given a set of such regular expressions Secco builds a set of accelerators by merging expressions with similar structures. For example, given `a`, `bc*` and `d|e`, Secco first merges `a` and `bc*` to create an accelerator for `ac*` — this can be configured

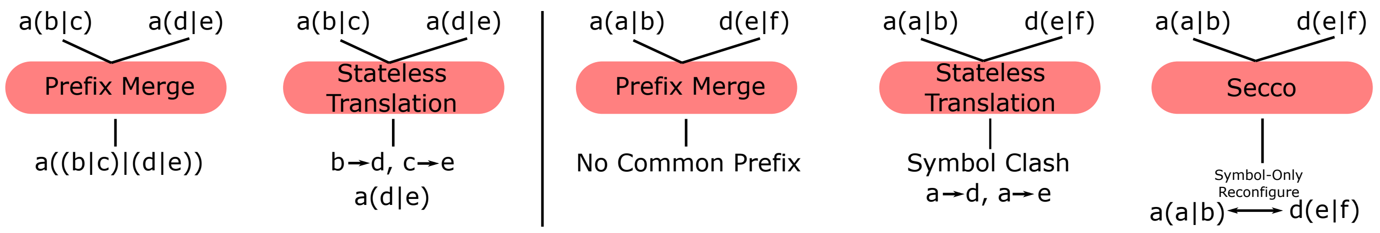
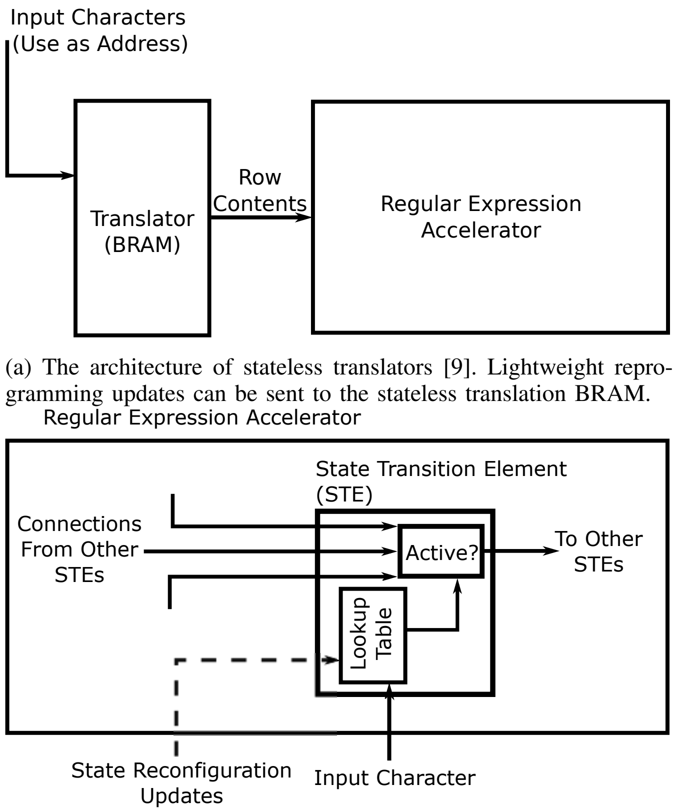


Fig. 2: Expressions can share accelerators using prefix merging and symbol-only reconfiguration techniques. Provided expressions do not have to run simultaneously, symbol-only reconfiguration techniques can merge many more accelerators than either prefix-merging or stateless translation techniques.



(a) The architecture of stateless translators [9]. Lightweight reprogramming updates can be sent to the stateless translation BRAM.

(b) A symbol-only reconfigurable architecture [6]. One state transition element (STE) is used for each state in the compiled NFA. The STE is active if any of the incoming edges are active, and the input symbol activates the state.

Fig. 3: Stateless translators [9] use a BRAM lookup table to translate all occurrences of a given symbol into another, to reuse accelerators. Symbol-only reconfiguration [6] is more general, as each State Transition Element can be reprogrammed to map to any chosen character.

to either `a` or `bc*`. Secco then explores the expression `d|e` to create an accelerator for the expression `a(c*|e)`. This process of building accelerators continues as long as more expressions need accelerators.

III. Implementation

Secco introduces a greedy algorithm for synthesising generic *sets* of accelerators that support many expressions (section III-A). We introduce a novel unification algorithm for merging similar accelerators (section III-B). Secco supports two different architectures: stateless translation [9] and symbol-only reconfiguration [6].

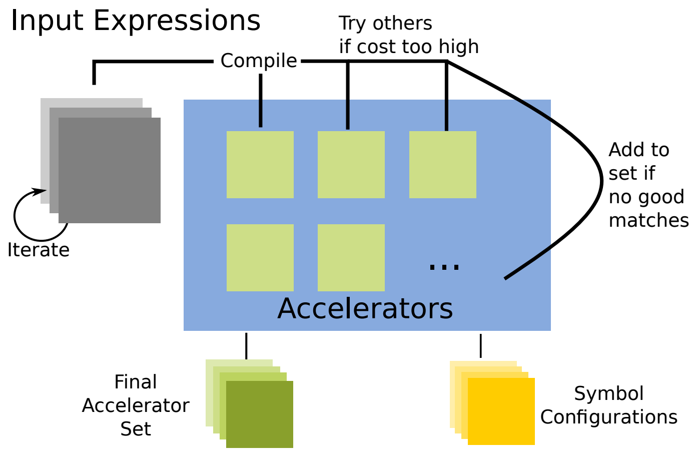


Fig. 4: Secco constructs a set of accelerators by greedily iterating through its input expressions, attempting to match the current input expression to each accelerator in turn with symbol-only reconfiguration and light modification of the accelerator up to a threshold. If this fails, then the same modification is attempted on the next accelerator. If no accelerators can be converted, a new one is added to the set.

A. A Greedy Approach to Merging Accelerators

Rulesets may have thousands to millions of expressions. Given a set of expressions to accelerate, we iterate through the expressions, creating a set of accelerators.

For each new expression, the set of existing accelerators is inspected, and we apply algorithm 1 to obtain a number of possible accelerators and mappings. To do this, both the existing accelerators and new expression are converted into accepting-path algebras (see section III-B1) We choose the accelerator that requires the fewest modifications. If the cost of mapping this expression to the accelerator is too high (i.e. it requires too many modifications), we try the next accelerator in the set. If no current accelerator can be modified with below-threshold changes, then a new accelerator is added. This algorithm is shown in figure 4.

To use our greedy algorithm, we require a pairwise algorithm that can produce single accelerators that can be shared by two expressions. The pairwise algorithm modifies the accelerator in a way that preserves the behavior of previous configurations and produces a configuration that can be used to configure the accelerator for the regular expression.

B. Compiling Between Regular Expressions

We develop an algorithm that abstracts symbols from regular expressions (and regular expression accelerators), and matches the underlying structures to each other. We build on the accepting-path algebra from [9].

1) Accepting-Path Algebra: The accepting-path algebra is an abstract representation of regular expressions where symbols are abstracted away and expressions are represented only by their structure.

There are a number of terms in the accepting-path algebra. We use t to refer to sub-terms:

n	This refers to a sequence of n consecutive characters. E.g., the expression `abc` has accepting-path algebra 3.
a	This refers to an accepting position within the expression.
e	This refers to the end of an expression.
$t_1, \ldots, t_n$	This represents a number of options, of which any can be taken.
$t*$	This means that t may repeat zero or more times.
$t_1 + t_2$	Represents t_1 followed by t_2. For example, we could write the algebra for `ab` as $1 + 1$

2) Structural Compilation Algorithm: To compile for symbol-only reconfigurable architectures, we define a unification function $\leftrightarrow$ on algebras that *merges* two algebras provided that there is sufficient similarity.

Algorithm 1 shows this unification, via pattern matching. This algorithm is exponential, in-particular the SumEq and BranchEq cases. We use a greedy strategy with cut-offs that control the maximum exploration depth for either of these cases to achieve reasonable performance. An example application of this algorithm is shown in figure 5. The two patterns in green can both be made compatible by deriving an underlying accelerator in yellow, using symbol-only reconfiguration to convert or disable symbols at run-time to switch between the two configurations.

3) Generating Reconfiguration Sets:

a) Symbol-Only Reconfiguration: As discussed in section III-C1 to introduce symbol-only reconfigurable sets, we derive a mapping between characters. Each node in the algebra corresponds to an edge in an underlying NFA. If terms are connected to each other by a rule, then we set the edges in the underlying NFA to the symbols in the expression that correspond to that edge.

This architecture is more flexible than stateless translation as it is not vulnerable to symbol conflicts (figure 2). As we see in section IV, this alone accounts for a 3.4× smaller set of accelerators.

b) Stateless Translation: Given a mapping, we can generate a stateless translation using the existing algorithm in [9]. For each symbol, we derive the set of terms it must activate to preserve the accelerator's behavior. We then unify these sets, creating a single, consistent, translation table.

The produced compiler is limited by the power of the stateless translators. As the size of an expression increases, the likelyhood of a clash in a stateless translator also increases, where one character must (impossibly) be translated to two different characters to correctly unify the accelerator. However, this approach is implementation-independent, allowing any accelerator design to be used.

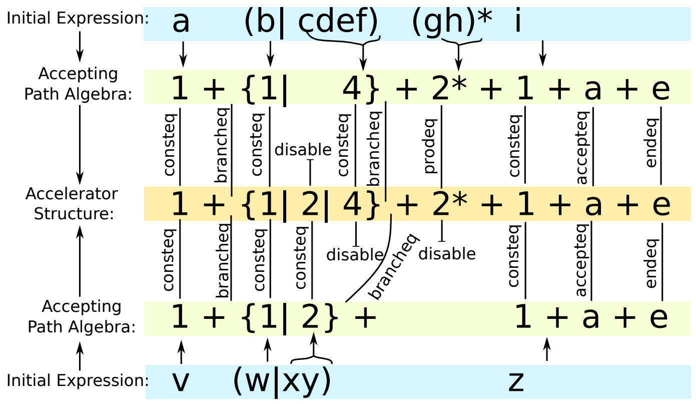


Fig. 5: An example compiling from expressions `a(b|cdef)(gh)*i` and `v(w|xy)z` (in blue) to the accepting path algebra (in green) and then a generic structure (in yellow) capable of accelerating both using a symbol-only reconfiguration.

Algorithm 1 The expression-merging algorithm, expressed recursively using pattern matching. The input is two different accepting-path algebras where the first represents an expression, and the second represents an accelerator, and the output is a single accelerator that can run either pattern, and a mapping for each accelerator. Some cases (e.g. the SumEq case) result in more than one valid result, in which case the merging with the fewest changes to the accelerator is selected.

```
function A ↔ B:
    |<Pattern> : <Result>                                   ▷ <RuleName>
    | $a \leftrightarrow a$: $a$                            ▷ AcceptEq
    | $e \leftrightarrow e$: $e$                            ▷ EndEq
    | $m \leftrightarrow m, m \in \mathbb{N}$: $m$          ▷ ConstEq
    | $m + x \leftrightarrow n + y$:                        ▷ AddBranch
      $(m \leftrightarrow n) + \{x, y\}$
    | $x* \leftrightarrow y*$: $(x \leftrightarrow y)*$     ▷ ProductEq
    | $x* \leftrightarrow y$: $x * + y$                     ▷ AddProduct
    | $\{x_0, \ldots, x_n\} \leftrightarrow \{y_0, \ldots, y_m\}$:   ▷ BranchEq
      A set $K$ such that each element is either $x_i$, $y_i$ or
    $(x_i \leftrightarrow y_j)$ and each $x_i$ and $y_i$ appears exactly once.
    | $x_0 + \cdots + x_n \leftrightarrow y_0 + \cdots + y_{n'}$:   ▷ SumEq
      A sequence $s_1 + \cdots s_m$ such that each $s_i$ is some
    $x_j + \cdots + x_k \leftrightarrow y_{j'} + \cdots + y_{k'}$ with monotonically increasing
    $j, k, j', k'$.
    | Otherwise: Fail
end function
```

C. Overview of Target Architectures

Secco can target two different styles of regular-expression overlay: stateless translators and symbol-only reconfigurable architectures. For both, Secco generates two key components: the accelerator structure (section III-A) and the reconfiguration sets (section III-B3).

1) Symbol-only Reconfigurable Architectures: Symbol-only reconfigurable architectures [6] enable expressions that share the same structure to also share the same accelerator. To compile a regular expression to a symbol-only reconfigurable accelerator, we first apply the unification algorithm to the expression and the accelerator. This produces a set of mappings between states in the accelerator and the expression — each state in the accelerator should be programmed with the symbol in the corresponding state in the expression.

2) Stateless Translation: Stateless translator architectures [9] use a BRAM-based lookup table to translate each input symbol one-by-one. To compile a regular expression to a stateless translator we use the algorithm described in [9]. In summary, the unified expressions produce a set of constraints on which characters must be translated to which other characters. These constraints either produce a stateless translator that maps the expression, or fail if there are symbol clashes.

D. Simulator Backend

We also introduce a Python-based backend to RXPSC that enables the creation of Python simulators for regular expressions: as raw accelerators, with symbol-only reconfiguration overlays and with stateless translation overlays. This enables rapid development and debugging of unification algorithms.

IV. Results

We evaluate Secco against another reconfigurable architecture that can support expression compression, stateless translation. We compare these techniques to prefix merging, a well-known way of enabling hardware sharing between regular expressions.

A. Compression Results

In this section, we examine the ANMLZoo benchmark suite [2], a regular expression benchmark suite covering varied domains, from network intrusion detection to bioinformatics. In-line with previous work [9], we assume that all expressions can share hardware with all other expressions. This assumption applies well to domains where a relatively small fraction of the expressions must be run on each input (such as network intrusion detection). We compare against stateless translators [9] using the same assumptions, although we note that stateless translators do not fully accelerate patterns that they compress, as they allow for approximation — depending on the field this may require additional computation.

Figure 6 shows the reduction in the number of states that Secco achieves. Stateless translation improves over prefix merging in the resource sharing it enables, providing an increase of 1.7× in the number of expressions that can be represented with a fixed number of states. We can see that symbol-only reconfiguration provides the most powerful model of reconfiguration, increasing the number of expressions that can be represented with a fixed number of states by a factor of 5.9x.

Figure 7 shows the reduction in the number of expressions that must be present on the accelerator to be able to run any from the set. Stateless translation reduces the number of regexes that must be implemented by 53% , while symbol-only reconfiguration improves on this, reducing the number of accelerators that must be implemented by a factor of 88%.

These factors are greater than the state reduction factors as removing an accelerator often requires adding states to another accelerator. We will explore this in section IV-B.

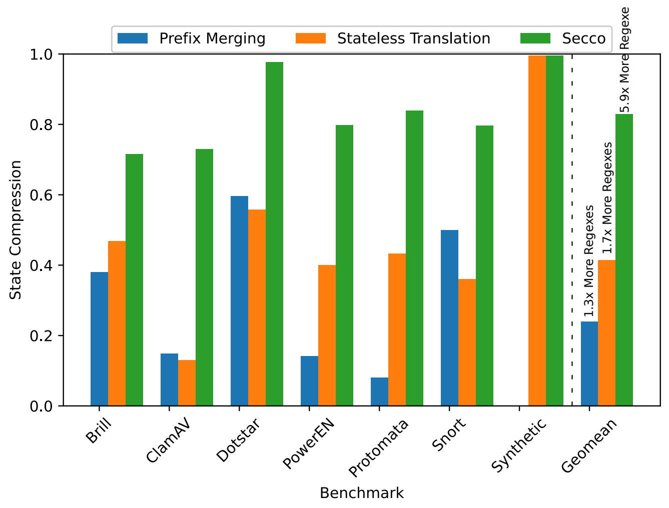


Fig. 6: Reduction in the number of states that must be implemented to support all expressions in a set (higher is better). Secco compresses expressions 3.4x more efficiently than stateless translation, which enables 5.9x more expressions to fit.

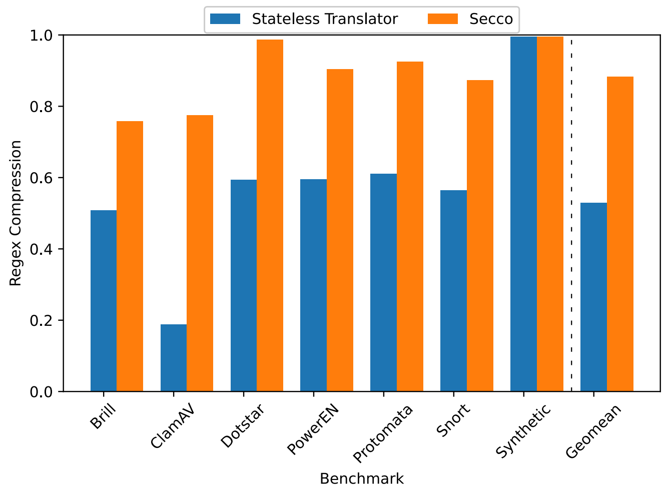


Fig. 7: Reduction in the number of accelerators that must be implemented to support all expressions in a set (higher is better). As smaller accelerators are more likely to be mergeable, the compression numbers using this metric are higher than in figure 6.

B. Analysis of Accelerator Construction

a) Expressions per Accelerator: Figure 8 shows the number of expressions that each implemented accelerator is expected to support. We can see that the number of patterns each accelerator is expected to support follows a power law distribution, with few accelerators supporting many expressions, and many accelerators only supporting a few expressions. This is an artefact of our greedy algorithm, that tries to add each new expression to the same accelerator. We can also see discrepancies between different benchmarks depending on the complexity of the expressions in the benchmark (e.g. Protomata largely contains relatively simple expressions, so a

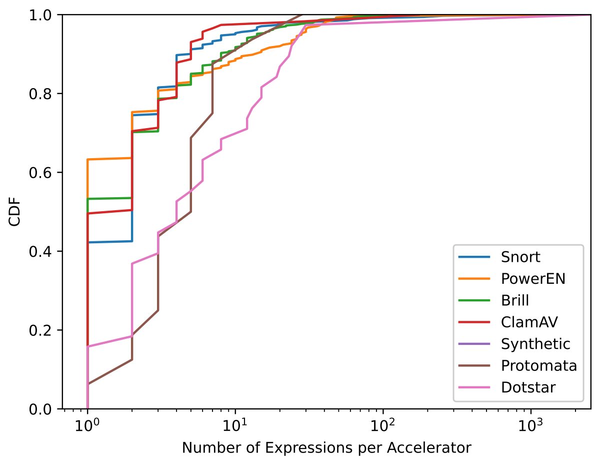


Fig. 8: The number of expressions using each accelerator. The power law-like distribution is a result of the greedy algorithm we use to select which accelerators should be shared.

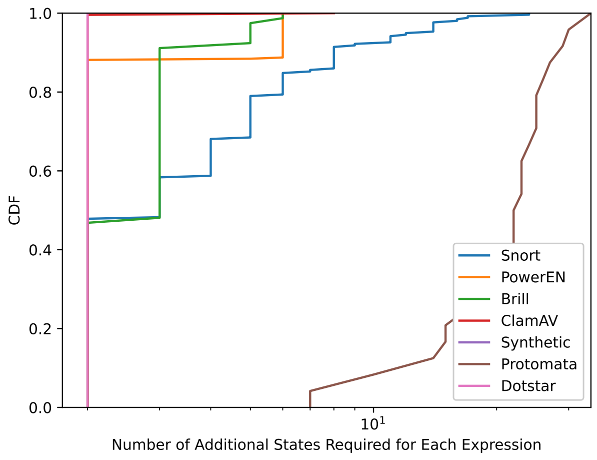


Fig. 9: The number of additional states for each expression. This graph shows that for most pairs of expressions, very few states need to be added to merge them. However, in many cases, many 10s of states do need to be added to the underlying accelerators for each expression.

lot of sharing is possible).

b) States per Expression: Figure 9 shows the number of additional states required when merging two expressions. In general, the fewer additional states required the better. We can see that the number of states added is highly benchmark dependent. This reflects how good our unification algorithm is at isolating differences between expressions and accelerators and overcoming them. For example, our algorithm better identifies these modifications in protomata than dotstar.

V. Conclusion

Secco enables compilation of sets of regular expressions to specialized overlays. We demonstrate the application of our algorithm to two existing FPGA overlays, stateless translation and symbol-only-reconfiguration and show that the number of regular expressions that can be supported with a fixed quantity of resources increases by factors of 1.7× and 5.9x respectively over prefix merging techniques. Our work introduces a generic accelerator-merging algorithm that opens the design-space for new designs of regular expression overlays.

References

[1] C. Colombo and G. J. Pace, "Regular expressions," in *Runtime Verification*, pp. 87–108, Springer, 2022.

[2] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures," *IISWC*, 2016.

[3] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 2991–3029, 2016.

[4] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi, "Demystifying automata processing: GPUs, FPGAs or Micron's AP?," 2017.

[5] The Snort Project, "SNORT users manual: 2.9.16," 2020.

[6] C. Bo, V. Dang, T. Xie, J. Wadden, M. Stan, and K. Skadron, "Automata processing in reconfigurable architectures," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, pp. 1–25, 6 2019.

[7] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *FPL*, IEEE, 9 2017.

[8] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the micron automata processor," in *the ACM International Conference*, ACM Press, 5 2016.

[9] J. Woodruff and M. F. P. O'Boyle, "New regular expressions on old accelerators," *DAC 2021*, 2021.

[10] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy II, J. Wadden, M. Stan, and K. Skadron, "An overview of micron's automata processor," 2016.