Approaching Symbolic Parallelization by Synthesis of Recurrence Decompositions

Grigory Fedyukovich and Rastislav Bodík
Computer Science and Engineering, University of Washington, Seattle, Washington, USA

We present SYMPA, a novel approach to perform automated parallelization relying on recent advances of formal verification and synthesis. SYMPA augments an existing sequential program with an additional functionality to decompose data dependencies in loop iterations, to compute partial results, and to compose them together. We show that for some classes of the sequential prefix sum problems, such parallelization can be performed efficiently.

1 Introduction

Parallelization of software is important for improving its effectiveness and productivity. Industrial applications (studied, e.g., in [11]) operate with large inputs and therefore could run days. But even a small program that iterates over a single array and incrementally computes a single numerical output could be challenging for parallelization. It would require partitioning of the input data into a sequence of segments, processing each segment separately, and aggregating the partial outputs for the segments.

Data dependencies prevent parallel processing of the segments. Therefore, the parallel loop should break the dependencies by devising an alternative function. This additional processing could invoke some analysis either *before* the parallel execution: in order to move segment boundaries, or *after* the parallel execution: in order to repair the outputs broken by violated dependencies. In this paper, we address the challenge of synthesizing such additional code automatically.

Preserving equivalence is a crucial requirement to automatic parallelization, and it aims at confirming that pairwise equivalence of inputs implies pairwise equivalence of outputs [8, 13, 2, 5, 6, 12, 3, 4]. Rather than merely relying on the existing solutions to prove equivalence between programs, we aim at constructing a parallel program P that is equivalent to the given sequential program S. Given arbitrary segments of the input data, we build on an assumption that P can always adjust segment boundaries, and the actual loop computation borrowed from S can be performed on the updated segments.

Program synthesis is an approach to generate a program implementation from the given specification. State-of-the-art synthesis tools employ the Counter-Example-Guided Inductive Synthesis (CEGIS) [14] paradigm, i.e., assume a space of candidate implementations and check whether there exists a candidate among them that matches the given specification. We follow the SMT-based paradigm for bounded model checking and template-based synthesis provided by ROSETTE [15, 16].

Our novel approach SYMPA automatically generates the search space of candidate decompositions of S, each of those consists of a *prefix* function (that would identify during runtime how the given segment boundaries should be moved), and a *companion* function (that would identify how the partial outputs should be aggregated). Given a bound for size of inputs, ROSETTE chooses a candidate decomposition and explicitly considers all possible input values to check whether the candidate is a valid decomposition. Our preliminary experiments with SYMPA confirm that for finite-state S and reasonably small inputs the decomposition can be found in seconds.

Submitted to: SYNT 2016

2 Sequential Recurrence Decomposition

We start by introducing the functional notation of array-manipulating programs, and proceed by formulating the parallelization criteria for them. For simplicity, we stick to functions taking a single finite-sized array as input, and recurrently computing a single output.

An *input* and an *output* are the designated variables respectively of the type *In* and *Out*. An n-sized *array* is a finite sequence of inputs:

$$A : In^n$$

In this paper, we consider functions of the type:

$$f : D \times In \to D$$

where D is a domain of any type. An element d of type D is called a *state*. Intuitively, f takes a state and updates it with respect to a given input. The state calculated by function f can further be taken by a function h to compute *output*:

$$h : D \to Out \qquad\qquad h(f(d, input)) = output$$

In the scope of the paper, we are interested in iterative application of function f to the elements of an n-sized array by means of the higher-order function *fold*:

$$fold : (D \times In \to D) \times D \times In^n \to D$$

For example, if there is an array $A = (input_1, \ldots, input_n)$, then in the first iteration, f would be applied to the first element $input_1$ and the initial state d_0. Then, the updated state would be taken by f again and updated with respect to the second element $input_2$. The result of n iterative applications of f to d_0 is called a *final* state and is represented as the following first-order *recurrent relation*:

$$fold(f, d_0, A) = f(input_n, f(input_{n-1}, \ldots, f(input_2, f(input_1, d_0))))$$

Notably, *output* has to be computed only for the final state:

$$h(fold(f, d_0, A)) = output$$

Throughout the paper, we rely on an operator that concatenates m arrays:

$$append : In^{n_1} \times \ldots \times In^{n_m} \to In^{n_1 + \ldots + n_m}$$

It is important to ensure the following functional property of *append*:

$$fold(f, d_0, append(A_1, \ldots A_m)) = fold(f, fold(f, \ldots fold(f, d_0, A_1), \ldots, A_{m-1}), A_m) \tag{1}$$

The left-hand-side of the equation denotes the initial state d_0 iteratively updated by f with respect to all elements of $append(A_1, \ldots A_m)$. The right-hand-side of the equation consists of m groups of consequent applications of *fold* to each of the arrays $\{A_i\}$. That is, the final state obtained for an i-th application of *fold* is further used for the $(i+1)$-th application of *fold*. We refer to this property to as *sequential recurrence decomposition*, since it guarantees equivalence between a single application of *fold* to $append(A_1, \ldots A_m)$ and m recurrent applications of *fold*.

Trivially, the equivalence of final states entails equivalence of outputs computed out of these states.

3 Parallel Recurrence Decomposition

Application of *fold* to each of the arrays $\{A_i\}$ in parallel requires the recurrent relation to be decomposed. We assume that each application of *fold* to A_i takes the same initial state d_0, and refer to the corresponding final states $\{d_i\}$ as to *partial* states (and to the corresponding outputs as to *partial* outputs).

$$\forall i \cdot d_i \triangleq fold(f, d_0, A_i) \qquad \forall i \cdot output_i \triangleq h(d_i)$$

The question is how to aggregate those partial outputs, so the aggregation result is equivalent to the output of sequential computation. Classic literature [7] refers to such function as to *companion* function, existence of which is equivalent to sound parallelization.

$$companion : Out^m \to Out$$

In the rest of the section, we aim at establishing the property of *parallel* (as opposed to sequential) *recurrence decomposition* that binds together all ingredients of the parallel processing of m arrays. Interestingly, there are several possible ways of defining this property, depending on existence of *companion* for each particular f. We consider three such cases.

3.1 Direct Decomposition

The first case assumes existence of a *companion* that is *directly* applicable to all partial outputs obtained after applications of *fold* to each A_i:

$$h(fold(f, d_0, append(A_1, \ldots A_m))) = companion(output_1 \ldots, output_m) \tag{2}$$

Example. Consider a function `array-max` that calculates the maximal element of a given array in Racket[1] (shown in Fig. 1). A function `f` updates a single argument `current-max` and in each iteration returns the result of application of the built-in function `max` to the current `element` of the array `A` and to `current-max`. Function `fold` is straightforward: it takes `f`, initial state $-\infty$ and the array of numbers. For some partitioning `A = (append` $A_1 \ldots$ A_m `)`, the parallel application of `fold` yields the array `out` of maximal elements of each A_i. It is easy to see that there exists a companion function for this case, and it is equal to `array-max`. □

We return to this scheme in Sect. 5 and show how the *companion* function can be synthesized.

3.2 Decomposition with Constant Prefixes

When no companion function meeting the condition (2) exists, then the computations $fold(f, d, A_i)$ and $fold(f, d, A_{i+1})$ depend on each other and cannot be *correctly* performed from the speculative initial state $d = d_0$. If we are in the lucky situation that the impact of the speculative initial state is localized to a prefix of A_i, we can repair the incorrect execution by recomputing the affected prefix. The scheme of this subsection performs such a repair on a constant-size prefix, if one exists. First, computations $fold(f, d, A_i)$ are performed in parallel from the initial state $d = d_0$, computing the partial result d_i. Subsequently, the scheme reruns the computations on a constant-size prefix of A_{i+1}, starting from the

[1] We provide *fold*-like implementation to comply with the chosen formalization. An alternative implementation using `apply` is also possible (similar to `max` in Fig. 3).

array-max

```
; given functions:
   (define (f element current-max) (max element current-max))
   (define (array-max A) (foldl f -inf.0 A))

; function needed for parallel decomposition:
   (define (companion-max out) (array-max out))
```

Figure 1: Calculating a maximal element of the array and the companion function in Racket.

state d_i. The result of processing the prefix, called $d_i^{completed}$, is then supplied to a suitable companion function instead of d_i.

We say that A' is a *prefix* of A if $A = append(A', A'')$ for some A''. We allow the prefix to be an empty array. We denote by $prefix(A)$ the prefix of A of the predetermined length $prefix_{length}$. We call the prefix the *constant prefix*.

Assuming existence of constant prefixes for each A_{i+1}, the constant-prefix scheme proceeds as follows:

$$\forall i \cdot d_i^{completed} \triangleq fold(f, d_i, prefix(A_{i+1})) \qquad \forall i \cdot output_i^{completed} \triangleq h(d_i^{completed})$$

Assuming the existence of a suitable *companion*, the repaired partial results can be combined:

$$h(fold(f, d_0, append(A_1, \ldots A_m))) = companion(output_1^{completed}, \ldots, output_{m-1}^{completed}, output_m) \quad (3)$$

Note that the partial output produced for the last array A_m is always completed since no subsequent array needs to be repaired.

An important observation is that computing each $d_i^{completed}$ requires processing $prefix(A_{i+1})$ twice, with the second processing serialized after d_i has been computed. The inefficiency is mitigated by the observation that the prefixes can be processed in parallel, so the critical path of the computation grows only by the processing of the constant prefix.

This scheme is applicable when there exists a constant prefix of each A_{i+1} that limits the scope of the necessary recomputation. Crucially, the prefix must not cover the whole A_{i+1} or else the repair of $fold(f, d_i, prefix(A_{i+1}))$ would modify d_{i+1}, which would in turn necessitate the repair of $fold(f, d_{i+1}, A_{i+2})$, serializing the repairs.

Example. Consider a function `is-sorted` (shown in Fig. 2) that returns 1 if for each pair of consequent elements of the array `A`, the former is smaller than the latter, and returns 0 otherwise. A function `f` updates two arguments with values of the previous element and the current output.

The companion function is `min`: it takes the array `out` of zeroes and ones, and returns 1 if all arrays $\{\mathtt{A}_i\}$ are sorted (i.e., all elements of `out` are ones), and 0 - if at least one array is not sorted (i.e., at least one element of `out` is zero). The prefix of each $\mathtt{A}_{i+1}$ used for completing `out` is defined as the array containing only the first element of $\mathtt{A}_{i+1}$.

As we mentioned earlier, the prefixes should be processed twice, e.g., for checking that `(append` $\mathtt{A}_i$ `(take` $\mathtt{A}_{i+1}$ `1))`[2] is sorted and for checking that $\mathtt{A}_{i+1}$ itself is sorted. Both results are necessary but

[2]Following the Racket notation, `(take` $\mathtt{A}_{i+1}$ `1)` stands for the sub-array of $\mathtt{A}_{i+1}$ containing just its first element.

is-sorted

```
; given functions:
    (define (f element state)
      (define prev-element (first state))
      (define output (second state))
      (cond
        [(>= element prev-element) '(element output)]
        [else '(0 0)]
      )
    )
    (define (is-sorted A) (foldl f '(-inf.0 1) l))

; functions needed for parallel decomposition:
    (define (companion-sorted out) (apply min out))
    (define (prefix-length-sorted) 1)
```

Figure 2: Checking if the array is sorted, the corresponding *companion* and *prefix* functions.

not sufficient premises for concluding that (append $\mathtt{A}_i$, $\mathtt{A}_{i+1}$) is sorted. It remains to establish that all elements of $\mathtt{A}_i$ are smaller than any element of $\mathtt{A}_{i+1}$. Without including the prefix into both arrays, such check would require accessing the partial state of $\mathtt{A}_i$ which in practice would end up in a more complicated companion function. □

We return to this scheme in Sect. 5 and show how the *companion* function as well as the value of $prefix_{length}$ can be synthesized.

3.3 Decomposition with Conditional Prefixes

When there is no companion function meeting condition (3) for all possible constant prefixes, then there could exist (but not necessarily does) a more complicated function to express the length of prefixes. Indeed, the number of elements in the beginning of some A_i might depend on the intermediate states computed while recurrently applying f to the elements of A_i. The problem of finding such *conditional* prefixes can be reduced to iterative evaluating of a predicate for each element of the array and the corresponding state obtained by *fold*:

$$prefix_{cond} : In \times D \to bool$$

That is, for each i, the length of $prefix(A_i)$ is the position number k of some element in A_i, such that $prefix_{cond}$ evaluates to *true* for the k-th element, and $prefix_{cond}$ evaluates to *false* for all j-th elements in A_i where $j < k$. Thus, contrary to constant prefixes that could be identified by some static analysis of f, the calculation of conditional prefixes requires running f for the particular given arrays. For the tasks enjoying the existence of the appropriate predicate $prefix_{cond}$ and existence of the appropriate *companion*, the parallel recurrence decomposition property has the same form as (3). The difference is the way of

seen-2-after-1

```
; given functions:
    (define (f element state)
      (define seen-one (first state))
      (define seen-two (second state))
      (cond
        [(= element 1) '(1 seen-two)]
        [(= element 2) (if (= seen-one 1) '(seen-one 1) '(0 0))]
        [else '(seen-one seen-two)]
      )
    )
    (define (seen-2-after-1 A) (foldl f '(0 0) A))

; functions needed for parallel decomposition:
    (define (companion-search out) (apply max out))
    (define (prefix-cond-search element) (= element 2))
```

Figure 3: Searching if "2" appeared in the array some time after "1" and the possible implementation of *companion* and *prefix*.

computing prefixes, whose length is not constant any longer:

$$prefix_{length} : (In \times D \to bool) \times (D \times In \to D) \times D \times In^n \to int$$

Example. Consider a function `seen-2-after-1` that checks whether "2" appeared in the array some time after "1" (shown in Fig. 3). A function `f` updates two flags indicating whether "1" or "2" was already seen in the array.

The companion function is `max`: it takes the array `out` of zeroes and ones, and returns 1 if at least one array has "2" appeared some time after "1". The prefix of each $\mathtt{A}_{i+1}$ contains all elements from the beginning of $\mathtt{A}_{i+1}$ until the first appearance of "2". Indeed, consider a case when $m = 2$, $\mathtt{A}_1$ contains "1", but does not contain "2", and $\mathtt{A}_2$ contains "2", but does not contain "1". In this case, it is important to keep searching "2" in $\mathtt{A}_2$ (i.e., traverse all elements until "2" is found. In contrast, processing $\mathtt{A}_1$ and $\mathtt{A}_2$ solely only lead to incorrect outputs. □

We return to this scheme in Sect. 5 and show how the *companion* function and the $prefix_{cond}$ predicate can be synthesized.

4 Bringing It All Together

Assuming existence of the *prefix* and *companion* for some f, we show how f can be parallelized in Fig. 4. The diagram considers three processors and represents different segments (also referred to as (sub-)arrays) of data by means of rectangular boxes, and functions to iterate over data (including the *prefix* and *companion* functions delivered by SYMPA) – by means of ovals.

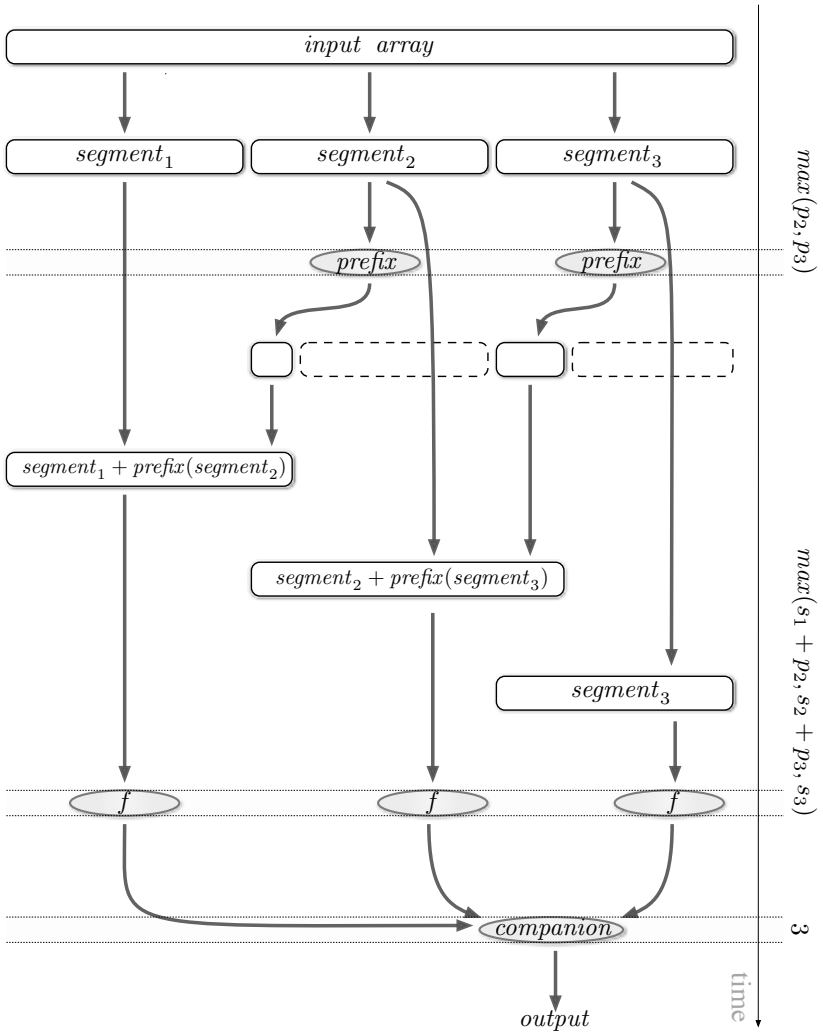

Figure 4: Executing a parallelized function.

We estimate the time spent at different stages of the parallel algorithm as a means of number of *fold*-iterations. Due to (1), the total time for three sequential *fold*-s required for three segments is:

$$T_s = s_1 + s_2 + s_3$$

For all arrays except the first one, a prefix should be calculated. Since in the worst case the prefix calculation is an iterative process, the time is linear. Notably, every process should wait until prefixes for all arrays are found.

$$T_p = max(p_2, p_3)$$

Then, for each array (already updated with prefixes), the corresponding *fold* should proceed. Similarly, it expects to wait until the computation in all processes is done:

$$T_f = max(s_1 + p_2, s_2 + p_3, s_3)$$

The last step for executing *companion* takes time 3, since it just aggregates three integers:

$$T_c = 3$$

Finally, the speedup earned by the parallel version of the function compared to the sequential one can be calculated by the following formula:

$$X \triangleq \frac{T_s}{T_p + T_f + T_c}$$

5 Parallelization Synthesis Problem

Given function f and initial state d_0, as declared in Sect. 2, we wish to find such function implementations for *prefix* and *companion*, as declared in Sect. 3, so for any possible sequence of input arrays, *prefix* and *companion* correctly decompose recurrence relations foisted by f:

$$\begin{aligned}
&\exists prefix, companion \cdot \forall A_1, \ldots A_m \cdot \\
&\quad h(fold(f, d_0, append(A_1, \ldots A_m))) = \\
&\qquad companion(h(fold(f, d_0, append(A_0, prefix(A_1)))), \ldots, \\
&\qquad\qquad h(fold(f, d_0, append(A_{m-1}, prefix(A_m)))), h(fold(f, d_0, A_m)))
\end{aligned}$$

5.1 Our Solutions

We present SYMPA, an algorithm to deliver solutions to the synthesis problem, and outline its pseudocode in Alg. 1. SYMPA aims at parallelizing iterative applications of f to d_0 for any possible input arrays $A_1, \ldots, A_m$. It treats each A_i nondeterministically, by allowing them to contain only symbolic elements. Thus, while parallelizing $fold(f, d_0, append(A_1, \ldots A_m))$, the algorithm considers all possible resolution of nondeterminism in each A_i and if a solution is found, it is guaranteed to be general enough to satisfy all arrays containing numeric elements. To ensure the finiteness of the search space of the solutions, the lengths of A_i are bounded.

SYMPA exploits our observations made in Sect. 3, and gradually attempts synthesizing *companion* and *prefix* functions for f and d_0 under the following hypotheses:

1) there exists a *companion* that makes (2) hold, so there is no need for prefixes (SYNTNOPREFIX method),
2) there exist a *companion* and a constant *prefix* with the fixed length $prefix_{length}$ that make (3) hold (SYNTCONSTANTPREFIX method),
3) there exist a *companion* and a conditional *prefix* defined by iterative evaluation of the predicate $prefix_{cond}$ that make (3) hold (SYNTCONDITIONALPREFIX method).

Each of the three methods verifies whether the corresponding hypothesis is true. In particular, it traverses the search space of candidate implementations of *companion* (and, for the hypotheses 2 and 3, *prefix*) function and checks whether one of those candidates is a witness for the hypothesis. The increasing complexity of the methods allows saving time while parallelization, and delivering simple solutions first. To further improve efficiency, SYMPA bounds the search space of *companion* to contain relatively

Algorithm 1: SYMPA (f, d_0)

Input: Function f, initial state d_0
Output: *companion*, [*prefix*]

```
1 A_1,...,A_m ← NONDET()
2 Try return SYNTNOPREFIX(f, d_0, A_1, ..., A_m)                    ▷ see Sect. 3.1
3 Try return SYNTCONSTANTPREFIX(f, d_0, A_1, ..., A_m)              ▷ see Sect. 3.2
4 Try return SYNTCONDITIONALPREFIX(f, d_0, A_1, ..., A_m)           ▷ see Sect. 3.3
5 return "unknown"
```

simple operators (e.g., "+", "*min*", "*max*"), and the search space of $prefix_{cond}$ to contain conjunctions of simple terms in linear arithmetic with equality. Notably, the efficiency comes with a price: the more restrictions are applied to the search space, the more risks are taken by the algorithm to produce "unknown" output.

Given solutions to the synthesis problem, it is straightforward to compile the synthesized functions into a self-contained parallelized function that is equivalent to the sequential one, but behaves in a fashion described in Sect. 4.

5.2 Current Limitations and Future Improvements

Since prefixes are the key to identify the amount of overhead of parallel computation against sequential computation, SYMPA can be required to minimize the length of discovered prefixes. In case of constant prefixes, the algorithm can enumerate positive numbers starting from 0 and check whether the current number constitutes a sufficient prefix length for all arrays.

Interestingly, in case of conditional prefixes, it is not obvious how to construct the minimal prefix. Since depending on evaluation of $prefix_{cond}$, the length of each prefix can vary from array to array. Furthermore, $prefix_{cond}$ can be evaluated to *false* for all iterations of the i-th application of *fold*, implying that the entire A_{i-1} and A_i should be concatenated, and construction of prefixes should be continued with A_{i+1}. In general, existence of $prefix_{cond}$ does not even guarantee that the prefix can actually be found. And in the worst case, the entire input array should be processed sequentially.

As we mentioned in Sect. 3.2, for each i, the elements of $prefix(A_i)$ are processed twice: for completing the $(i-1)$-th, and for starting the i-th applications of *fold*. For the big picture, we should also mention the routine for calculating $prefix(A_i)$ itself that requires yet another iterating over these elements. To avoid this redundancy, the prefix calculation engine can be augmented by construction of symbolic summaries that would capture the effect of applying of f to the given prefix and an arbitrary initial state. Despite this idea is borrowed from [11], SYMPA enables its application in a new context. That is, as opposed to summarizing the effect of application of *fold* to entire arrays, we would like to summarize the effect of application of *fold* just to the prefixes, while leaving the remaining elements to be iterated by *fold*. The computed summaries could be applied then directly to complete the $(i-1)$-th, and to start the i-th applications of *fold*, thus avoiding duplicate iterations.

Benchmark	# Vars	Hypothesis	*companion*	$prefix_{length}$ / $prefix_{cond}$	Synt time (*s.*)
`array-max`	1	SYNTNOPREFIX	`max`	—	3
`is-sorted`	1	SYNTCONSTPREFIX	`min`	`1`	3
`alternation-of-1-2`	1	SYNTCONSTPREFIX	`min`	`1`	2
`number-of-112`	2	SYNTCONSTPREFIX	`+`	`2`	3
`seen-2-after-1`	2	SYNTCONDPREFIX	`max`	`(= element 2)`	3
`alternation-of-11-22`	3	SYNTCONDPREFIX	`min`	`(= element 3)`	4

Table 1: Preliminary evaluation of SYMPA.

6 Evaluation

We implemented a prototype of SYMPA on top of ROSETTE, a function synthesizer for Racket. Given a specification in a form of array-handling function f, ROSETTE exploits CEGIS paradigm, that maintains a space of candidate decompositions of f and verifies equivalence between f and each candidate separately. A candidate decomposition that fulfils the specification is returned by ROSETTE as output.

We evaluated SYMPA on a set of Racket implementations of some prefix sum problems. We used a bound of 15 for the length of the input array. Interestingly, this bound was sufficient, and gave correct decompositions also for bigger arrays. In general, of course, the soundness of recurrence decomposition for big arrays is not guaranteed.

Table 1 summarizes our preliminary results of SYMPA for six benchmarks. It measures the complexity of each benchmark by giving the number of variables in the state of each f ("# Vars"). The results of recurrence decomposition are reported by the hypothesis which was true for the function, the operator for synthesized *companion* function, and function used by calculating the length of prefixes (i.e., $prefix_{length}$ for the constant prefixes, and $prefix_{cond}$ for the conditional prefixes). Finally, we provide the total time spent by ROSETTE for realizability check of the correspondent hypothesis (i.e., to realize the necessity of either constant or conditional prefixes), and for synthesis of the witnessing functions.

Functions `array-max`, `is-sorted` and `seen-2-after-1` were already discussed in Examples 1, 2, and 3 respectively. Functions `alternation-of-1-2` and `alternation-of-11-22` check if the entire array if an alternation of "1" and "2" and "11" and "22" respectively. Function `number-of-112` searches for appearances of pattern "112". Interestingly, the synthesis time for all decompositions was insignificant. In future work, we plan to enhance the set of experiments by more challenging benchmarks.

7 Related work and Conclusion

In this paper we addressed the problem of synthesizing decompositions of recurrence relation that is known to be crucial for the tasks of automated parallelization [7]. Our approach applies for array-handling functions and is motivated by the fact that arrays are often split into segments, and accessible by different processors. Our main idea is to allow each processor first, to perform computation on its own segment and, if needed, to move segment boundaries.

We are approaching the goal of automated parallelization through formal verification and synthesis. We showed that the function that realizes the necessity of the prefixes (as well as the function computing the prefixes themselves, and the function aggregating the results by all processors) can be discovered using the state-of-the-art synthesizer ROSETTE. We presented a prototype of the algorithm SYMPA to automatically synthesize decompositions of small programs in Racket, and we envision its further im-

provements in future. We believe, this is the first synthesis-driven approach to parallelization as opposed to [17, 10, 1, 9, 11] and many others.

Acknowledgments. This work is supported in part by the SNSF Fellowship P2T1P2_161971, NSF Grants CCF–1139138, CCF–1337415, and NSF ACI–1535191, a Grant from U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers program under Award Number FOA–0000619, and grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032 , as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[2] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[3] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Automated discovery of simulation between programs. In *LPAR*, volume 9450, pages 606–621. Springer, 2015.

[4] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. Property directed equivalence via abstract simulation. In *CAV*. Springer, 2016. to appear.

[5] Benny Godlin and Ofer Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.

[6] Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.

[7] Peter M. Kogge. Parallel solution of recurrence problems. *IBM Journal of Research and Development*, 18(2):138–148, 1974.

[8] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *DAC*, pages 263–268. IEEE, 1997.

[9] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel finite-state machines. In *ASPLOS*, pages 529–542. ACM, 2014.

[10] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. Translating imperative code to MapReduce. In *OOPSLA*, pages 909–927. ACM, 2014.

[11] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, pages 153–167. ACM, 2015.

[12] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Incremental Upgrade Checking by Means of Interpolation-based Function Summaries. In *FMCAD*, pages 114–121. IEEE, 2012.

[13] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. Data-driven equivalence checking. In *OOPSLA*, pages 391–406. ACM, 2013.

[14] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

[15] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with Rosette. In *Onward!*, pages 135–152. ACM, 2013.

[16] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, page 54. ACM, 2014.

[17] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260. ACM, 2009.