Revealing Parallel Scans and Reductions in Recurrences through Function Reconstruction

Peng Jiang
The Ohio State University
Columbus, OH, USA
jiang.952@osu.edu

Linchuan Chen
Google
Mountain View, CA, USA
linchuan@google.com

Gagan Agrawal
The Ohio State University
Columbus, OH, USA
agrawal@cse.ohio-state.edu

Abstract

Many sequential loops are actually *recurrences* and can be parallelized across iterations as scans or reductions. Many efforts over the past 2+ decades have focused on parallelizing such loops by extracting and exploiting the hidden scan/reduction patterns. These approaches have largely been based on a heuristic search for closed-form composition of computations across loop iterations.

While the *search-based* approaches are successful in parallelizing many recurrences, they have a large search overhead and need extensive program analysis. In this work, we propose a novel approach called *sampling-and-reconstruction*, which avoids the search for closed-form composition and has the potential to cover more recurrence loops. It is based on an observation that many recurrences can have a *point-value* representation. The loop iterations are divided across processors, and where the initial value(s) of the *recurrence variable(s)* are unknown, we execute with several chosen (*sampling*) initial values. Then, correct final result can be obtained by reconstructing the function from the outputs produced on the chosen initial values. Our approach is effective in parallelizing *linear*, *rectified-linear*, *finite-state* and *multivariate* recurrences, which cover all of the test cases in previous works. Our evaluation shows that our approach can parallelize a diverse set of sequential loops, including cases that cannot be parallelized by a state-of-the-art static parallelization tool, and achieves linear scalability across multiple cores.

CCS Concepts • **Theory of computation → Parallel computing models**; **Program analysis**;

Keywords Loop Parallelization, Recurrence, Reduction

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243204

ACM Reference Format:
Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2018. Revealing Parallel Scans and Reductions in Recurrences through Function Reconstruction. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243176.3243204

1 Introduction

Scan (prefix sum) and reduction are well-known computation patterns that can be parallelized by exploiting associativity [6, 18], despite dependencies. As an example, consider the evaluation of a polynomial expression ($y = \sum_{i=0}^{N}(a[N-i]\cdot c^i)$) by Horner's method:

```
y = 0; for(i=0; i<N; i++) { y = c * y + a[i]; }
```

The loop is inherently sequential at first glance because there is a carried dependence on y. However, the computation is a first-order *recurrence* and the parallelization of such recurrences has been well studied [7, 13, 17]. Specifically, if we symbolically combine the computation of two consecutive iterations, we can obtain the value of y at the end of iteration $i+1$ (y_{i+1}) as a function of the value of y at the beginning of iteration i (y_{i-1}):

$$y_{i+1} = c * y_i + a[i+1] = c * (c * y_{i-1} + a[i]) + a[i+1]$$
$$= \boxed{c * c} * y_{i-1} + \boxed{c * a[i] + a[i+1]} \quad (1)$$

Since this symbolic composition does not depend on the actual value of y and is associative, it can be conducted in parallel as a reduction. As one possible parallelization approach, the loop can be divided into portions of consecutive iterations, and each processor performs the symbolic composition independently on one such portion. After all processors have finished the symbolic execution, the correct value of y is propagated across processors, i.e., from the first one to the last one, sequentially.

Overall, there are two steps in the parallelization above: 1) deriving the associative and *closed-form* composition of computation across iterations, as in Expression 1, and 2) applying parallel scan/reduction to the associative composition. For effective parallelization, symbolic composition has to be closed-form. This is because it ensures that a dominant fraction of the computation is distributed and conducted on different processors, and the sequential propagation of the values for y incurs only a constant overhead. It turns out that for the example loop above obtaining closed-form symbolic

$$
\begin{array}{lll}
e \in \texttt{Exp} ::= e \oplus e' & e, e' \in \texttt{Exp} & \text{Expressions} \\
\quad |x|k & x \in \texttt{Var}, k \in \mathbb{Z}, \mathbb{Q}, \mathbb{R} & \\
\quad |s[e] & s \in \texttt{SeVar}, e \in \texttt{Exp} & \\
\quad |\text{if } be \text{ then } e \text{ else } e' & & \\
be \in \texttt{BExp} ::= e \otimes e' & e, e' \in \texttt{Exp} & \text{Boolean Exps} \\
\quad |be \wedge be' | \neg be & be, be' \in \texttt{BExp} & \\
\quad |t[e] & t \in \texttt{BSVar}, e \in \texttt{Exp} & \\
\quad |\text{true}|\text{false} & & \\
\texttt{Program} ::= c; c' & c, c' \in \texttt{Program} & \\
\quad |x := e & x \in \texttt{Var}, e \in \texttt{Exp} & \\
\quad |b := be & b \in \texttt{BVar}, be \in \texttt{BExp} & \\
\quad |\text{if}(be)\{c\}\text{else}\{c'\} & be \in \texttt{BExp}, c, c' \in \texttt{Program} & \\
\quad |\text{for}(i \in \mathcal{I})\{c\} & i \in \texttt{Iterator} &
\end{array}
$$

Figure 1. Program syntax for recurrence loops from [11]. The binary operator $\oplus$ represents any arithmetic operation (+,-,*,/), $\otimes$ represents any comparator (<,≤,>,≥,=,≠).

composition is relatively simple. However, for more complex cases such as those involving conditionals, it is nontrivial to derive the closed-form composition.

1.1 Problem Formulation

Our goal is to apply parallel scan/reduction to more complex loops that can be modeled as recurrence equations. Formally, we define our target loops with the program syntax introduced by Farzan *et al.* [11] as shown in Figure 1. The program is assumed to be written in simple imperative language with basic constructs for branching and looping. Nested loops are allowed in the syntax, but we always focus on one inner loop at a time. Farzan *et al.* conclude that *"every non-nested loops in the program model in Figure 1 can be modeled by a system of recurrence equations"* [11]. While the syntax does not imply true dependence in the loops, we are specifically interested in parallelizing loops with true carried dependences (other cases are usually simpler). Note that because a loop with carried dependence can be modeled as recurrence equations does not necessarily mean it can be parallelized. In fact, the loops that have been successfully parallelized in previous works are in a small subset of recurrences defined by the model. Our goal is to parallelize not only the loops that could be handled by previous efforts, but possibly a larger set.

1.2 Previous Works

There has been continuing interest in parallelizing loops involving recurrences [3, 5, 7, 11–13, 19, 20, 23]. Despite the differences in formalism and detailed techniques, all of the previous works follow the same two steps as for the example above. The approaches in earlier efforts were mostly *template-based* [3, 7], i.e., if the recurrences in the original loops (or sliced recurrences after loop distribution) match the template recurrences whose scan/reduction patterns are obvious (e.g., linear and finite-state recurrences), the loop can be parallelized. The major limitation of these early works is that they cannot handle variables with mutual dependences except for linear cases.

More recent techniques for extracting scan/reduction patterns in recurrences are mostly *search-based*. Fisher *et al.* [13] use aggressive symbolic analysis to search for closed-form compositions of the loop body functions – however, their method only works for loops that can be sliced into first-order recurrences. Raychev *et al.* use symbolic execution to derive the combined function at runtime [22] – however, the runtime maintenance of the combined function involves frequent path-merging, which is nontrivial and expensive. Farzan *et al.* use an SMT solver to search for the combined function [11], and thus their parallelization tool depends on the ability of the underlying solver. Though the *search-base* approaches have shown greater potential over the earlier approaches, their applicability has not been clearly characterized. Specifically, it is unclear for what recurrences the closed-form compositions can be obtained in a reasonable amount of time.

1.3 Our Work

In this paper, we propose a novel approach for extracting scan/reduction parallelism in recurrence loops without any search. The insight is to exploit a *point-value* view of the recurrences. We observe that in many cases, the combined function across iterations for a recurrence loop usually have a fixed *geometric shape* and can be precisely represented by one or more input-output pairs. Thus, instead of deriving or searching for the explicit form of the combined function, we only verify its type (or shape) at compile time, which can be done by simple program analysis. Then we generate a sampling-and-reconstruction strategy for the combined function based on the function type. The loops are parallelized by simply running the iterations on a processor with multiple initial values of the recurrence variable(s), (i.e, a *sampling* phase) and reconstructing the function with the outputs on these initial values afterwards (i.e, a *reconstruction* phase). Specifically, we present sampling-and-reconstruction strategies for four types of recurrences: *linear*, *rectified-linear*, *finite-state*, and *multivariate*. We further show algorithms that can verify the function types and generate the parallelized codes automatically. The evaluation results show that our approach can parallelize a diverse set of sequential loops, including cases that cannot be parallelized by the state-of-the-art static tool from Farzan *et al.* [11].

2 Illustration of the Idea

Before getting into technical details, we describe our idea of point-value representation of loop bodies and parallelization with an example. Consider again the polynomial evaluation by Horner's method:

```
y = 0; for(i=0; i<N; i++) { y = c * y + a[i]; }
```

```
i0: y0=2*0+1=1
i1: y1=2*y0-5=-3
i2: y2=2*y1+3=-3
i3: y3=2*y2-7=-13
i4: y4=2*y3-1=-27
i5: y5=2*y4-10=-64
```

(a) Sequential

```
        P0                              P1
i0: y0=2*0+1=1                  i3: y30=2*0-7=-7
i1: y1=2*y0-5=-3                i3: y31=2*1-7=-5
i2: y2=2*y1+3=-3                i4: y40=2*y30-1=-15
                                i4: y41=2*y31-1=-11
                                i5: y50=2*y40-10=-40
                                i5: y51=2*y41-10=-32
-------------------------------------------------
        y5=(y51-y50)*y2+y50=8*(-3)-40=-64
```

(b) Parallel

Figure 2. Execution of the polynomial evaluation loop based on Horner's method: "`y=0;for(i=0;i<N;i++){y=c*y+a[i];}`", with $a[6] = \{1, -5, 3, -7, -1, -10\}$ and $c = 2$.

In each iteration, y is a linear function of its initial value. If we compose the computation of two adjacent iterations as in Expression 1, the value of y_{i+1} is a linear function of y_{i-1} with a coefficient $c*c$ and an offset $c*a[i]+a[i+1]$. The basic idea here is that the composition of two linear functions is still linear. By inductive reasoning, we can see that the result of y after an arbitrary number of iterations is a linear function of its initial value, i.e. $y_j = Linr(y_i) = A * y_i + B, \forall i \leq j < N$.

To make the example more concrete, let us consider the computation of polynomial "$f(x) = x^5 - 5 * x^4 + 3 * x^3 - 7 * x^2 - x - 10$" at $x = 2$ by the loop of Horner's method. Here, the array $a[6] = \{1, -5, 3, -7, -1, -10\}$ and $c = 2$. Figure 2a shows the sequential execution of the loop with this particular input. It produces the correct result of $f(2) = -64$.

Now suppose we have two processors (P_0 and P_1) and we want processor P_0 to compute the first three iterations of the loop and processor P_1 to compute the remaining three iterations simultaneously. The initial value of y is known to be 0, so we can directly start the computation on processor P_0. Since the value at the end of iteration 2 is unknown before P_0 finishes, we do not know the actual initial value to start execution from iteration 3 on processor P_1. To perform the computation on the two processors in parallel, we start the execution on processor P_1 with two initial values 0 and 1, which produce two sampled values of y ($y50, y51$) at the end of iteration 5. Because the underlying function is linear, two values turn out to be sufficient. (We will explain why this works in more detail in the next section.) As shown in Figure 2b, the correct result of y can be computed by the two sampled values $y50, y51$ on processor P_1 and the resulting value ($y2$) on P_0, using an expression we derive later. Though the computation on P_1 is doubled compared with the original sequential loop, the benefits of this parallelization pattern will manifest when more processors are used. Moreover, available SIMD parallelism can be used for executing each pair of instructions (same computations with different initial values) in parallel.

```
pushed = False;
count = 0;
for(i=0; i<N; i++) {
   type = Inputs[i].type;
   if(type=="PUSH") {
      pushed = True;
      count = 0;
   } else if(type=="PULL" && pushed) {
      pushed = False;
      Print(count);
   } else if(pushed) {
      count += 1; }}
```

Figure 3. An example of user-defined aggregation that counts the events between "PUSH" and "PULL"

3 Point-value Representation of Loop Body Functions

Building on the example above, this section describes our parallelization strategies for recurrences constructed by four function types: *linear*, *rectified-linear*, *finite-state* and *multi-variate*, which are commonly seen in real loops.

3.1 Linear Functions

Besides a computation like Horner's method for polynomial evaluation discussed above, it turns out that functions in certain seemingly complex loops are actually linear. Figure 3 shows an example of a user-defined aggregation, which is adopted from Raychev *et al.* [22]. Regardless of the dependence on *pushed*, the value of *count* over iterations is a linear function of its initial value. This is because the two assignments to *count*, as shadowed in Figure 3, are linear functions and any composition of the two linear functions is still a linear function. The dependence on *pushed* adds another dimension to the function for *count*, but it does not affect the linear relation itself.

Formally, we define the linear relation between two values y_j (the value of recurrence variable y at the end of iteration j) and x_i (the value of recurrence variable x at the end of iteration i) for $j \geq i$:

Definition 3.1. (Linear Relation) If $y_j = C_1 * x_i + C_2$ where C_1 does not depend on any recurrence variable and C_2 does not depend on x, then y_j is a linear function of x_i, denoted as $y_j = Linr(x_i)$.

Note y and x can be the same recurrence variable in the definition. To verify the linear relation between values of recurrence variables, we can first check if the recurrence variables have linear relation in a single iteration, and then prove the linear relation across iterations by induction. Take the loop in

Figure 3 as an example, *count* is a linear function to itself in all branches of the loop body, i.e., $count_{i+1} = Linr_i(count_i), \forall i < N$. From the base case, suppose $count_k = Linr_k(count_i)$, we know $count_{k+1} = Linr_{k+1}(count_k) = Linr_{k+1}(Linr_k(count_i)) = Linr_{k+1,k}(count_i)$. Thus, $count_j = Linr(count_i), \forall i \leq j < N$. That is, the value of *count* at the end of iteration j is a linear function of its value at the end of iteration i, $\forall i \leq j < N$. The verification process generally works for all of the function types we are considering in this work and can be done by simple program analysis. We leave the detailed algorithms of this verification process in §4. In this section, we focus on discussing the sampling-and-reconstruction strategies for different function types and show our main idea for achieving parallelization.

Sampling and Reconstruction Assume the linear relation between y_j and x_i is verified, two sample points are sufficient to reconstruct it because the function is a line on a plane. Thus, we can start the execution of iteration $i+1$ with $x_i[0] = 0$ and $x_i[1] = 1$ when the actual value of x_i is unknown. Suppose the values of y_j produced by the two inputs are $y_j[0]$ and $y_j[1]$ respectively, the combined function from iteration $i+1$ to iteration j can be represented as:

$$y_j = (y_j[1] - y_j[0]) * x_i + y_j[0] \tag{2}$$

That is, we can directly compute the correct value of y_j by Expression 2 once the actual value of x_i is available from the previous neighboring processor. This was the idea used in the example pertaining to Horner's method presented earlier.

3.2 Rectified-linear Functions

Rectified-linear function arises in loops that have recurrence variables in conditionals. An example is the loop that computes the maximum segmented sum of an array, as shown in Figure 4. In each iteration, the update of s is a function of itself. Intuitively, the function consists of two pieces – the left piece is $s = 0$ with the condition *s+A[i]<0* and the right piece is *s=s+A[i]* with condition *s+A[i]>=0*. If plotted, the function will have the shape as shown in Figure 5a.

Because the composition of any two rectified-linear functions of the shape shown in Figure 5a still holds the same shape, the resulting value of s after an arbitrary number of iterations is a rectified-linear function of its initial value with the shape as in Figure 5a. This can be seen from Fisher *et al.* [13], which has this loop as a running example of heuristic search of closed-form composition of recurrences. They give a closed-form representation for s across iterations as

$$s_{res} = (s_{init} + C_1 < 0)?C_2 : s_{init} + C_1 + C_2 \tag{3}$$

The closed-form representation has the shape as shown in Figure 5a.

Generalizing beyond the example above, a useful property of rectified-linear functions is that their compositions are

```
m = INT_MIN; s = 0;
for(i=0; i<N; i++) {
    if(s + A[i] < 0) s = 0; else s += A[i];
    if(m < s) m = s; }
```

Figure 4. A loop that computes maximum segmented sum of an array

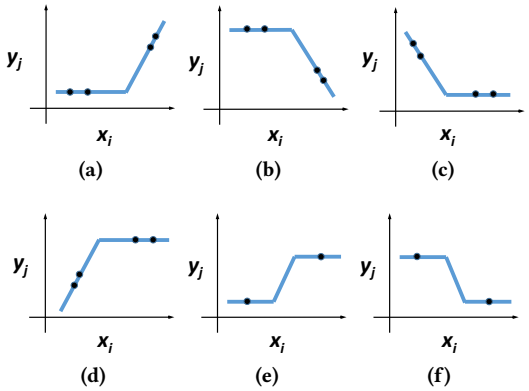


Figure 5. Rectified-linear functions and their combinations

either still a rectified-linear function or a simple piecewise-linear function. Figure 5 shows all possible shapes of rectified-linear functions and their compositions. Certain compositions may be straight lines depending on the actual values, but they can be considered as special cases of the listed shapes. The functions resulting from compositions of two functions are summarized in Table 1, where each cell indicates the resulting shape of the composition of its row and column. For example, if we compose a rectified-linear function of shape b with another function of shape a, i.e., $a(b(x)) = (b \circ a)(x)$, the composed function will have the shape of Figure 5f, as indicated by the cell at row b and column a in Table 1.

Formally, we define the rectified-linear relation between y_j (the value of recurrence variable y at the end of iteration j) and x_i (the value of recurrence variable x at the end of iteration i) for $j \geq i$:

Definition 3.2. (Rectified-Linear Relation) y_j is a rectified-linear function of x_i, denoted as $y_j = RectLinr(x_i)$, if

$$y_j = \begin{cases} Linr(x_i) & , \quad x_i \oslash B \\ Linr(B) & , \quad x_i \obslash B \end{cases}$$

where B is not dependent on recurrence variables, $\oslash$ can be any of $<$, $>$, $\leq$ and $\geq$, and $\obslash$ is the inverse of $\oslash$.

Note y and x can be the same recurrence variable in the definition. The rectified-linear relation among values of recurrence variables can be verified by a similar process as for linear relation (§4.2).

Sampling and Reconstruction Assume the rectified-linear relation between y_j and x_i is verified (i.e., the function has one of the shapes in Figure 5a to 5d), four sample points are

	a	*b*	*c*	*d*	*e*	*f*
a	a	b	f	e	e	f
b	f	e	a	b	f	e
c	c	d	e	f	f	e
d	e	f	c	d	e	f
e	e	f	f	e	e	f
f	f	e	e	f	f	e

Table 1. Composition of shapes listed in Figure 5 – each cell indicates the resulting shape of the composition of its row and column

sufficient to reconstruct it. Let the four points be $(x_i[0], y_j[0])$, $(x_i[1], y_j[1])$, $(x_i[2], y_j[2])$ and $(x_i[3], y_j[3])$ where $x_i[0] < x_i[1] < x_i[2] < x_i[3]$, as long as $x_i[0]$ and $x_i[1]$ are small enough to reside in the left piece of the function and $x_i[2]$ and $x_i[3]$ are large enough to fall in the right piece, we can compute any point in the function by:

$$y_j = \begin{cases} max(l, r) & , \quad y_j[0] > y_j[1] \;||\; y_j[2] < y_j[3] \\ min(l, r) & , \quad otherwise \end{cases} \tag{4}$$

Here, assuming x_i falls in the left piece of the function that is a line determined by $(x_i[0], y_j[0])$ and $(x_i[1], y_j[1])$, l is computed as

$$l = (y_j[1] - y_j[0])/(x_i[1] - x_i[0]) * (x_i - x_i[0]) + y_j[0]$$

Next, assuming x_i falls in the right piece that is determined by $(x_i[2], y_j[2])$ and $(x_i[3], y_j[3])$, r is computed as

$$r = (y_j[3] - y_j[2])/(x_i[3] - x_i[2]) * (x_i - x_i[2]) + y_j[2]$$

The final value is either the greater or the smaller one of l and r, depending on the shape of the function. More efficiently, when the function passes origin and/or has slope of 1 or -1, fewer sample points are required. For example, if we know the function between y_j and x_i has shape 5a and the function has slope 1 on the right piece, then we only need two sample points – $x_i[0]$ on the left piece and $x_i[3]$ on the right piece, and the value at any point of the function can be computed as

$$y_j = max(y_i[0], x_i + y_j[3] - x_j[3]) \tag{5}$$

If we further know that the function passes origin, then only one sample point $x_i[0]$ on the left piece is needed and the value at any point of the function can be computed as

$$y_j = max(y_i[0], x_i) \tag{6}$$

For piecewise functions with shapes shown in Figure 5e and 5f, it is difficult to keep all the information with sampling for the general cases because the slope and boundary of the middle piece are unknown. However, for most loops, we can usually get more information about the middle piece (e.g., the middle piece or the extended line passes origin and has

```
i0: s0=max(0+A[0],0)=2;     m0=max(INT_MIN,s0)=2
i1: s1=max(s0+A[1],0)=5;       m1=max(m0,s1)=5
i2: s2=max(s1+A[2],0)=1;       m2=max(m1,s2)=5
i3: s3=max(s2+A[3],0)=0;       m3=max(m2,s3)=5
i4: s4=max(s3+A[4],0)=5;       m4=max(m3,s4)=5
i5: s5=max(s4 A[5],0)=4;       m5=max(m4,s5)=5
```

(a) Sequential

```
            P0                              P1
i0: s0=max(0+A[0],0)=2    i3: s30=max(-100+A[3],0)=0
i1: s1=max(s0+A[1],0)=5   i3: s31=max(100+A[3],0)=98
i2: s2=max(s1+A[2],0)=1   i4: s40=max(s30+A[4],0)=5
                          i4: s41=max(s31+A[4],0)=103
                          i5: s50=max(s40+A[5],0)=4
                          i5: s51=max(s41+A[5],0)=102
-----------------------------------------------------
              s5=max(s50,s2+s51-100)=4
```

(b) Parallel

Figure 6. Execution of the maximum segmented sum loop in Figure 4 with $A[6] = \{2, 3, -4, -2, 5, -1\}$.

slope of 1 or -1). In this case, two sample points at the two ends of the function are sufficient.

Runtime Checking The sampling strategy above requires that $x_i[0]$, $x_i[1]$ are "small enough" and $x_i[2]$, $x_i[3]$ are "large enough". We cannot guarantee that at compile time because the turning points of the rectified-linear functions are unknown. Instead, we ensure the correctness of sampling at runtime. Specifically, we check whether the actual value of the recurrence variable (once available) is between $x_i[1]$ and $x_i[2]$. If it is true, Expression 4 will produce the correct result. If the actual value does not fall between $x_i[1]$ and $x_i[2]$, we simply rollback and re-execute that portion of iterations with the correct initial value. Note that the result is correct even if $x_i[0]$, $x_i[1]$, $x_i[2]$, $x_i[3]$ are not perfectly placed as in Figure 5. For example, if $x_i[0]$, $x_i[1]$, $x_i[2]$ are on the left piece and $x_i[3]$ on the right piece of Figure 5a, the interpolation in range $[x_i[1], x_i[2]]$ is always precise though some accuracy is lost in $(x_i[2], \infty)$, but this does not affect the correctness. The same argument applies for other possible cases. Since $x_i[1]$ is small and $x_i[2]$ is large, the actual value of the recurrence variable is very likely to fall in $[x_i[1], x_i[2]]$. In fact, we did not encounter any rollback in our evaluation.

Example As a concrete example, let us consider the "maximum segmented sum" loop in Figure 4 with input $A[6] = \{2, 3, -4, -2, 5, -1\}$. In this loop, m and s are both recurrence variables. Figure 6a shows the sequential execution of the loop with this particular input. It produces the correct result $m = 5$, which is the sum of $A[0]$ and $A[1]$. Figure 6b shows the parallel computation of s on two processors with processor P_0 executing the first three iterations and processor P_1 executing the remaining three iterations simultaneously. The initial value of s at iteration 0 is known to be 0, so we can directly start the computation on processor P_0. The

actual initial value of s at iteration 3 is unknown before P_0 finishes, so we start the execution on processor P_1 with two sampling values: -100 and 100. Only two sample points are needed in this case because the function has shape as shown in Figure 5a with the right piece having slope of 1. Also, -100 and 100 are small and large enough to sample the left and right piece of the function for this particular input. With the two sampled results $s50 = 4$ and $s51 = 102$, the parallel version computes the correct result of $s5$ using $max(s50, s2 + s51 - 100)$, according to Expression 5. The parallel computation of m is not included in Figure 6b because m depends on both s and itself. That is, m is a *bivariate* function of the initial value of both itself and s. We will discuss the sampling-and-reconstruction strategy for multivariate functions and complete this example in §3.4.

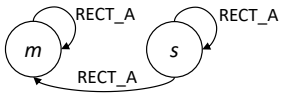
3.3 Finite-state Functions

Finite-state functions are functions whose domain and range are both finite sets. An example is a function on boolean recurrence variable such as *pushed* in Figure 3. The input and output of the function are either $True$ or $False$. Another example is the transition on a finite-state machine (FSM) where the domain and range are both the set of all states. The finite-state relations can be determined by checking equality conditionals in the loop body (§4.2).

Sampling and Reconstruction The sampling and reconstruction for finite-state functions are straightforward: simply enumerate all the possible input states and the function is the one-to-one mapping between the inputs and outputs. Take the loop in Figure 3 as an example, we start each portion of iterations with two initial values for *pushed* ($True$ and $False$). The correct result is then the state produced by the actual initial value. It will be inefficient to enumerate all of the states when the state set is large. In this case, the sampling can be combined with speculation to exploit the convergence of state transition. We discussed enumerative speculation in a recent work on parallelizing FSM [16].

3.4 Multivariate Functions

It is common in real loops that a recurrence variable can depend on multiple recurrence variables. Take the loop in Figure 4 as an example, from the code "`if(m < s) m = s;`" – it is easy to see that the value of m at the end of iteration i (m_i) is a rectified-linear function of its value at the end of iteration $i-1$ (m_{i-1}). We can also tell that m_i is a rectified-linear function of s_i (the value of s at the end of iteration i). Next, because s_i is a rectified-linear function of s_{i-1} as discussed in §3.2, we can tell that m_i is a bivariate function of both s_{i-1} and m_{i-1}. The type of the function along both dimensions are rectified-linear of shape in Figure 5a. Then, based on the composition closure of rectified-linear functions, we can determine that m_j is a bivariate function of s_i and m_i with shape in Figure 5a in each dimension for all $j \geq i$. The relation can be represented in a graph as



In the Figure above, the nodes represent the recurrence variables m and s, and an edge from s to m indicates that m_j is a rectified-linear function of s_i for all $j \geq i$. It is similar to the dependence graph of a loop with function relations among the recurrence variables stored in the edges.

Formally, we define *function relation graph* to represent the relations among recurrence variables:

Definition 3.3. (Function Relation Graph) A function relation graph for a recurrence loop is a graph $G = (V, E)$ in which a node $u \in V$ represents a recurrence variable, and an edge $(u, v) \in E$ represents that v_j (the value of v at the end of iteration j) is a function of u_i (the value of u at the end of iteration i) $\forall j \geq i$. The function type is stored in the edge (u, v). If a node v has multiple incoming edges, it indicates that v_j is a multivariate function of each of the source nodes of incoming edges (with the function type along each dimension stored in the corressponding incoming edge).

Note that in the definition above we assume that there is at most one edge between any two nodes. For a multivariate function, it indicates that the function along the dimension of each input variable has a unique function type (thus a unique sampling-and-reconstruction strategy). This condition commonly holds in the recurrence loops we are targeting, and it is important for the efficacy of our sampling-and-reconstruction method. We will discuss the algorithms for building a function relation graph for a recurrence loop in §4.3. We now assume the function relation graph is already available, and focus on discussing its sampling-and-reconstruction strategy in this section.

Sampling The initial values for sampling a multivariate function are the *Cartesian product* of the initial sampling values of all its dependent variables. To see this, let us consider an example of 2-D linear function. Suppose a recurrence variable z has two incoming nodes x and y in the function relation graph and the function types stored in the two incoming edges are both linear. Thus, we have a function of the form $z = a * x + b * y + c$ where a, b, and c have unknown values that are not dependent on any recurrence variables. To obtain these values, we need at least three independent linear equations. Our key idea is that the equations can be obtained by computing z with four pairs of initial values: *{x:0, y:0}*, *{x:0,y:1}*, *{x:1,y:0}*, and *{x:1,y:1}*, in other words, the cartesian product of $(x : \{0, 1\})$ and $(y : \{0, 1\})$. With four corresponding output values of z (z_0, z_1, z_2, z_3), we have four equations, from which 3 unknown values can be determined. And then, once the actual values of x and y are available, we can calculate the actual output value of z directly.

This sampling strategy generally works for multivariate functions of other types discussed in this section as long as the *unique edge property* is satisfied in the function relation

graph. Given a function relation graph for a recurrence loop, the procedure to determine the initial values of all recurrence variables for sampling includes the following steps:

1) For a recurrence variable v, determine its initial values for sampling the function represented by each of its outgoing edges, denoted as set $S_1[v][e]$.
2) Determine the initial values of v for sampling functions of all of its outgoing edges by calculating the union of $S_1[v][*]$, denoted as set $S_2[v]$.
3) The initial values for sampling a recurrence variable x are the Cartesian product of $S_2[v]$ for all v that has edge pointing to x, denoted as $S_3[x][v_1..v_n]$.
4) If a recurrence variable v belongs to multiple connected components and the initial values for v are different in different components, calculate the union of initial values for different components and extend the initial values for other variables in each component with arbitrary values.
5) Repeat Step 4 until a consistent set of initial values for sampling of all variables is achieved.

Reconstruction In the example above, the function $z = a * x + b * y + c$ can be reconstructed explicitly by solving for a, b, and c. However, a more efficient approach is to reconstruct the function by *interpolation*. That is, given the values of a function at certain sampled points, we want to directly determine the value of the function at any point without computing the function parameters. Consider again the function $z = f(x, y) = a * x + b * y + c$. The value of z at any point (x, y) can be computed in the following steps:

1) It is already known that $f(0, 1) = z_1$ and $f(1, 1) = z_3$. Because function $f(x, 1)$ is linear, it can be determined by z_1, z_3 as $f(x, 1) = (z_3 - z_1) * x + z_1$ according to Expression 2.
2) Similar to the first step, we can obtain $f(x, 0) = (z_2 - z_0) * x + z_0$.
3) Because $f(x, y)$ is a linear function of y determined by $f(x, 0)$ and $f(x, 1)$, we have the reconstructed function as $f(x, y) = (f(x, 1) - f(x, 0)) * y + f(x, 0)$.

This procedure is commonly used to interpolate bilinear function of the form $z = a_3 * xy + a_2 * x + a_1 * y + a_0$ [1]; in this example, the coefficient of term xy is zero.

The key idea in the above interpolation procedure is to decouple the dependent variables one-by-one. It works for multivariate functions of other types discussed in this section as long as the function relation graph has only unique edges between any two nodes. The order for decoupling does not matter as the function type along each dimension is unique. The decoupling of each variable involves replacing the samples with their reconstruction functions on that dimension (Step 1 and 2 in the above example). Our general algorithm involves repeating the replacing step until all variables are decoupled and all of samples are in one formula, which is the reconstructed multivariate function (Step 3 in the above example).

P1

```
i3: s30=max(-100+A[3],0)=0;    m30=max(-100,s30)=0
i3: s31=max(100+A[3],0)=98;    m31=max(-100,s31)=98
i4: s40=max(s30+A[4],0)=5;     m40=max(m30,s40)=5
i4: s41=max(s31+A[4],0)=103;   m41=max(m31,s41)=103
i5: s50=max(s40+A[5],0)=4;     m50=max(m40,s50)=5
i5: s51=max(s41+A[5],0)=102;   m51=max(m41,s51)=103
-------------------------------------------------
          m5=max(max(m50,s2+m51-100),m2)=5
```

Figure 7. Computation of m on processor P_1 for execution example in Figure 6.

Example Consider again the loop in Figure 4 and its execution in Figure 6. We now explain the parallel computation of m. We have known that m is a bivariate function of s and m with shape as shown in Figure 5a – this is along both dimensions and after an arbitrary number of iterations. Because the rectified-linear function on s has slope 1 on the right piece, two sample points (say, -100 and 100) are sufficient. Because the rectified-linear function on m has slope 1 on the right piece and the right piece (or extended line) passes origin, one sample point (say, -100) on the left piece suffices. The initial values for sampling are thus the cartesian product of $(s : \{-100, 100\})$ and $(m : \{-100\})$, which are $(s[0] = -100, m[0] = -100)$ and $(s[1] = 100, m[1] = -100)$. Figure 7 (together with Figure 6b) shows the full computation on processor P_1, and it produces the correct result of m. The expression for computing $m5$ is constructed based on the decoupling rule described above. If we decouple dimension s first, according to Expression 5, the two sample points $m50$ and $m51$ are reduced to one sample point $max(m50, s2 + m51 - 100)$ along the dimension m. It is further reduced to $max(max(m50, s2 + m51 - 100), m2)$ according to Expression 6, which is the bivariate function of $m5$ from $m2$ and $s2$. It turns out that decoupling along dimension m first will produce an equivalent expression.

3.5 Comparison with Previous Works

Representing combined functions as input-output pairs provides a mechanical way to reconstruct the functions without any search. Though the function types discussed in this section seem limited, they occur quite frequently in practice, and to the best of our knowledge, they cover all of the test cases used in previous works. In addition, our approach has two advantages over the previous works.

More Efficient Parallel Codes First, for loops with acyclic dependence graphs (e.g., the loop in Figure 4), a conventional parallelization approach is to apply *loop distribution* to slice the loop into first-order recurrences and parallelize them one-by-one [13]. However, loop distribution usually requires multiple passes over the entire iteration space or storing the intermediate values in the parallelized codes, which can be expensive. Our approach parallelizes such loops in one pass. Second, the multiple sampling is not an overhead of our approach. They are actually the (near) minimum amount

```
ma = FLT_MIN; mi=FLT_MAX; m = FLT_MIN;
for(i=0; i<N; i++) {
    if(A[i] >= 0) {
        ma = max(ma*A[i], A[i]);
        mi =min(mi*A[i], A[i]);
    } else {
        tmp = max(mi*A[i], A[i]);
        mi = min(ma*A[i], A[i]);
        ma = tmp; }
    m = max(m, ma); }
```

Figure 8. A loop that computes maximum segmented product of an array

of computation needed for achieving parallelization. All of the previous approaches have to do such *redundant* computation. For example, for parallel computing of s in the loop in Figure 4, Fisher *et al.* [13] need to do four additions and two comparisons in each iteration in the parallelized code to maintain the template variables C_1 and C_2 in Expression 3. In comparison, our sampling in Figure 6b only involves two additions and two comparisons. While the recent work of Farzan *et al.* [11] parallelizes this loop with two additions and one comparison in each iteration, our approach can easily benefit from SIMD processing and achieve the performance of one addition and one comparison in each iteration. If a loop requires multiple samplings with our approach, Farzan *et al.* [11] also need to do *auxiliary computations* in each iteration in the parallelized code. The parallel codes generated by Farzan *et al.* [11], however, cannot easily benefit from SIMD processing as the auxiliary computations they require are different from the loop body.

Supporting Mutual Dependences Our approach not only works for loops that can be sliced into first-order recurrences, but we can handle variables with mutual dependences for more general cases than linear systems. For example, Figure 8 shows a loop that computes the maximum segmented product of an array. We can see that *ma* and *mi* are mutually dependent across iterations and the computation cannot be expressed as matrix multiplication in max-plus semiring [24] or sliced into first-order recurrences. The template-based approaches in early works certainly cannot parallelize this loop. It is unclear whether the search-based approaches in more recent works can find a closed-form composition of function modeling this loop. We show in the evaluation that the most recent available tool from Farzan *et al.* [11], which searches combined function based on an SMT solver, fails to parallelize this loop. With our approach, the parallelization and reasoning are much more straightforward. In one iteration, ma_i is a rectified-linear function of either ma_{i-1} with shape in Figure 5a or mi_{i-1} with shape in Figure 5c, and mi_i is a rectified-linear function of either mi_{i-1} with shape in Figure 5d or ma_{i-1} with shape in Figure 5b. According to the composition rules in Table 1, we can decide by induction that ma_j is a rectified-linear function of both ma_i (with shape in Figure 5a) and mi_i (with shape in Figure 5c), and mi_j is a rectified-linear function of both mi_i (with shape in Figure 5d) and ma_i (with shape in Figure 5b) after an arbitrary number of iterations, for any $j \geq i$. With this information, we can parallelize this loop by simply starting the execution on each portion of iterations with the cartesian product of four sampling initial values for each of *ma* and *mi*, and the results on different portions can be merged by Expression 4.

4 Inferring Function Types

In the previous section, we have shown strategies for parallelizing recurrence loops based on the assumption that we already know the function types among the recurrence variables. This section describes three steps to obtain the function types from the sequential codes automatically: 1) identify the recurrence variables (i.e., which variables we should sample); 2) build a partial dependence graph for the recurrence variables based on one iteration of the loop; 3) compute transitive closure of the partial dependence graph to obtain a function relation graph for the loop. We discuss the three steps based on the syntax in Figure 1.

4.1 Identifying Recurrence Variables

The recurrence variables are variables with loop-carried dependences. More specifically, suppose `RVar` is the set of all recurrence variables, a variable $v \in$ `RVar` if it meets both of the following two conditions: 1) There is no assignment ($v = e$) dominating all of the uses of v, and 2) v is both assigned and used in the loop body, or v is only assigned but there exist at least one execution path in the loop body where no assignment occurs (i.e., its value can possibly come from previous iteration).

4.2 Analysis of a Single Iteration

A recurrence loop can be represented as a set of condition-assignment pairs of the form <*cond*, $v = e$>, where $v = e$ is an assignment of a recurrence variable v, e is an expression, and *cond* is a boolean expression for entering the branch that contains the assignment instruction [10, 11]. A traversal on the control flow graph (CFG) of the loop body can be used to obtain this, specifically, by storing the conditions along the traversal tree and the assignment instructions for each recurrence variable. Once we have the set of condition-assignment pairs, they are all analyzed to obtain the function type(s).

Verifying Linear Relation First, we verify the linear relations among recurrence variables without considering the conditions. For an assignment $v = e$, Algorithm 1 can find the recurrence variables that v depends on and verify if the relations between v and the recurrence variable(s) is/are linear. The output of the algorithm is a set of relation records storing all the recurrence variables that e depends on and the signs and coefficients of the linear functions if available. If e is a recurrence variable, it directly returns the recurrence variable along with a function sign of *LINR_P* (indicating it

Algorithm 1 Check Linear Relations

```
1: procedure CheckLinear(e)                          ▷ Input: e ∈ Exp
2:     if e ::= x and x ∈ Var then
       ▷ Case#1: e is a recurrence variable
3:         if x ∈ RVar then return {[x, LINR_P, 1]}
4:         else                    ▷ Case#2: e is a non-recurrence variable
5:             de ← GetReachingDefinition(x)
6:             return CheckLinear(de)
7:         end if
8:     else if e is a binary expression then
       ▷ Case#3: e ::= e1 ⊕ e2
9:         r1 ← CheckLinear(e1)
10:        r2 ← CheckLinear(e2)
11:        if e ::= e1 + e2 then return r1 ∪* r2
12:        else if e ::= e1 − e2 then return r1 ∪* FlipSign(r2)
13:        else if e ::= e1 ∗ e2 then
14:            if r1 == ∅ then return UpdateSign(r2, e1)
15:            else if r2 == ∅ then return UpdateSign(r1, e2)
16:            else Abort()                      ▷ Not a linear function
17:            end if
18:        else if e ::= e1 ÷ e2 then
19:            if r2 == ∅ then return UpdateSign(r1, e2)
20:            else Abort()                      ▷ Not a linear function
21:            end if
22:        end if
       ▷ Case#4: e ::= k, k ∈ ℤ, ℚ, ℝ or e ::= s[e], s ∈ SeVar
23:    else return ∅
24:    end if
25: end procedure
```

is a positive linear function) and a coefficient of 1. If e is a variable but not a recurrence variable, it obtains the reaching definition of the variable and checks linear relation for the assigning expression de in the reaching definition.

If e is a binary expression $e ::= e_1 \oplus e_2$, it recursively checks the linear relations for its two operands. Suppose the binary operator is '+', $e_1 = Linr_1(v_1) = A_1 * v_1 + B_1$ and e_2 does not depend on v_1, then $e = A_1 * v_1 + B_1 + e_2$, indicating that e is a linear function of v_1 according to Definition 3.1. It is obvious that the relation records for e are the union of the relation records for its two operands e_1 and e_2 if they do not depend on the same recurrence variable(s). If e_1 and e_2 both depend on v_1 (i.e., $e_1 = Linr_1(v_1) = A_1 * v_1 + B_1$ and $e_2 = Linr_2(v_1) = A_2 * v_1 + B_2$), then $e_1 = (A_1 + A_2) * v_1 + (B_1 + B_2)$, indicating that e is still a linear function of v_1, though the sign of the function in this case can be unknown. This combining procedure of the relation records for e_1 and e_2 is denoted as $\cup^*$ in the algorithm. Other types of binary expressions can be analyzed with a similar approach. The remaining cases are e is a constant or e is a reference to a non-recurrence array, for which the algorithm simply returns an empty set. Note that because the algorithm returns an output whenever a recurrence variable is found, a recurrence variable x in the relation records of e indicates that e is a linear function of the reaching definition of x rather than the initial value of x at the beginning of the iteration. Storing the partial dependences in the relation records simplifies the dependence analysis and yet preserves sufficient information for inferencing function types among the recurrence variables.

Building Partial Dependence Graph Based on Algorithm 1, we take the conditions into account to verify rectified-linear and finite-state functions and build a partial dependence graph for the loop body function. The procedure includes the following steps:

1) For each condition-assignment pair <$cond, v = e$>, first identify all recurrence variables that v depends on and check their linear relations by invoking Algorithm 1.

2) If $cond$ is $true$, which means the linear relations in the assignment are unconditional, directly add the linear relations as edges in the graph.

3) If the assignment is conditional, separate $cond$ into sub-conditions for different recurrence variables and analyzes the sub-conditions one by one.

4) For one sub-condition $x \circledcirc e'$, if the comparison operator is equality, which means e is assigned to v only when x is equal to e', then v is a finite-state function of x.

5) If the comparison operator is $>$ or $\geq$, which means e is assigned to v only when x is greater than e', then only the right pieces of the linear functions for $v = e$ are valid.
- If v is a positive linear function of x under condition $x > e'$, then x to v is the right piece of a rectified-linear function of shape in Figure 5a.
- If v is a negative linear function of x under condition $x > e'$, then x to v is the right piece of Figure 5b.
- If v is not related to x (i.e., v is a constant with respect to x under condition $x > e'$), then x to v is the right piece of either Figure 5c or 5d.

6) A similar decision procedure applies to the cases where the comparison operator is $<$ or $\leq$.

7) After all of the condition-assignment pairs are processed, check symbolic equivalence of the left and right pieces at the connecting points and see if they construct valid rectified-linear functions. Any non-connecting pieces will abort the analysis.

The procedure returns a partial dependence graph with nodes representing recurrence variables and edges representing dependences among them.

4.3 Reasoning across Iterations

To verify the function relations among recurrence variables across iterations, we want to obtain a function relation graph for a recurrence loop. This is achieved by computing the transitive closure of the partial dependence graph. Recall that an edge in the partial dependence graph represents a relation between two assignments of two recurrence variables x and y in one iteration. Thus, a path from node x to node y in the partial dependence graph indicates that, over a certain number of iterations, the value of y after an assignment is a function of the value of x at an earlier assignment. An edge from node x to node y in the transitive closure has

two implications: 1) there is at least one path from x to y in the partial dependence graph; 2) the function relations represented by different paths from x to y have a unique type or are special cases of a unique type. According to Definition 3.3, we can see that the transitive closure of the partial dependence graph is a function relation graph for a recurrence loop.

The computation of the transitive closure of a partial dependence graph can be implemented as an iterative matrix-matrix multiplication based on two operators on function types: *composition* and *merge*. The function represented by a path in the partial dependence graph is the composition of the functions along its edges. We use $\boxtimes$ to denote the composition of two functions. If *FST* represents finite-state function, *ANY* represents any type of function, *LINR_P* and *LINR_N* represent linear positive and linear negative function, *RECT_*$*$ represent rectified-linear function of shapes in Figure 5, the composition rules are as follows:

- $FST \boxtimes ANY = FST$ • $LINR\_P|N \boxtimes LINR\_P|N = LINR\_P$
- $LINR\_P|N \boxtimes LINR\_N|P = LINR\_N$
- $LINR\_P \boxtimes RECT\_ANY = RECT\_ANY$
- $RECT\_ANY \boxtimes LINR\_P = RECT\_ANY$
- $LINR\_N \boxtimes RECT\_A|B|C|D|E|F = RECT\_C|D|A|B|F|E$
- $RECT\_A|B|C|D|E|F \boxtimes LINR\_N = RECT\_B|A|D|C|F|E$
- Compositions of rectified-linear functions are shown in Table 1.
- $\boxtimes$ is associative but not commutative.

If there is a path from node x to node y where the edges cannot be composed by the rules above, i.e., the composed function may have a type that is not one of the three function types discussed in §3, the analysis will abort and the loop cannot be parallelized by our approach.

There might be multiple paths between two nodes in the partial dependence graph. We use $\boxplus$ to denote the merge of two function types represented by different paths. The merge rules are as follows:

- $T \boxplus T = T$ • $LINR\_P \boxplus RECT\_A|D|E = RECT\_A|D|E$
- $LINR\_N \boxplus RECT\_B|C|F = RECT\_B|C|F$
- $\boxplus$ is commutative and associative.

To elaborate, merge of two identical function types is still the same function type, and merge of a linear function and a rectified-linear function with the same direction is the rectified-linear function (linear function can be considered as a special case of a rectified-linear function). If the paths between two nodes in the partial dependence graph cannot be merged by the rules above, which means that the function relations between the two recurrence variables do not have a unique type, the analysis will abort and the loop cannot be parallelized by our approach.

The edges in the partial dependence graph are first merged and stored in a matrix M. With the multiplication $\boxtimes$ and addition $\boxplus$ of the matrix elements defined above, the transitive closure of the partial dependence graph is the power of M at convergence. That is, $G = M \times M \times \ldots \times M$ and $G \times M == G$.

```
/* Shared Array for storing sampled results */
S_Vm[_N_THREADS][_N_SAMP];
S_Vs[_N_THREADS][_N_SAMP];
...
/* Sampling on each processor */
Vm[2] = {SMALL, SMALL};
Vs[2] = {SMALL, LARGE};
/* Outer loop: compute a portion of iterations */
for(i=start_pos; i<end_pos; i++) {
  /* Inner loop: compute each of the samples */
  for(_i_s=0; _i_s<_N_SAMP; _i_s++) {
    if(Vs[_i_s] + A[i] < 0) { Vs[_i_s] = 0; }
    else { Vs[_i_s] += A[i];}
    if(Vm[_i_s] < Vs[_i_s]) {Vm[_i_s] = Vs[_i_s];}}
/* Store the sampled results */
memcpy(S_Vm[tid], Vm, sizeof(Vm));
memcpy(S_Vs[tid], Vs, sizeof(Vs));
```

Figure 9. Code for parallel sampling on each processor for the "maximum segmented sum" loop in Figure 4

5 Evaluation

In this section, we apply our approach to a diverse set of sequential loops, evaluating the effectiveness, and also examine the efficiency of parallelization.

5.1 Experimental Setup

Platform Our experiments are conducted on an Intel Xeon Phi 7250 (Knights Landing) processor, which contains 68 cores running at 1.4GHz, each with four hardware threads. Though the techniques presented in this paper are not restricted to any specific architecture, this processor was chosen because of high degree of parallelism both at MIMD and SIMD levels. We use multicores to execute multiple portions of loop iterations in parallel and utilize SIMD to hide the overhead of sampling at each core. To fully exploit the wide SIMD vectors (512-bit), we further execute multiple (portions of) iterations in one SIMD vector on a single core, and obtain additional speedups. Both the sequential and the parallelized codes are compiled with Intel ICC 17.0.4.

Implementation We implement the program analysis algorithms in §4 as an LLVM IR pass, which is used within Clang (version 6.0.0). The input is the source code of the sequential loop written in C++. The pass analyzes the sequential loop and produces a function relation graph for the loop (or aborts if the loop cannot be parallelized with our approach). We also implement a standalone source-to-source transformation tool with Clang libTooling [4], which generates the parallel codes automatically based on the function relation graph. The generation of the sampling part of the parallelized code is straightforward. We add an inner loop of *_N_SAMP* iterations to the origin loop and replace each recurrence variable x with a reference to its corresponding vector *Vx[1..._N_SAMP]*. As an example, Figure 9 shows the code for parallel sampling generated for the "maximum segmented sum" loop in Figure 4. The composed function across

Loop	Function Types	#Sampling Sets	Parsynt?
mps	linear; rectified-linear	1×1	yes
mss	linear; rectified-linear	2×1	yes
mts-p	linear; rectified-linear	2×1	yes
ss	linear; rectified-linear	1×1	yes
bal	linear; rectified-linear	1×1	yes
ses	linear	2	no
des	linear	2×2	no
msp	linear; rectified-linear	$4 \times 4 \times 1$	no
agg1	linear; finite-state	2×2	no
agg2	linear	2	no

Table 2. Test cases used for evaluation with their function types and number of initial value sets for sampling in our approach and whether they can be parallelized by the program synthesis tool from [11]

iterations for a recurrence loop is generated according to the reconstruction procedures discussed in §3.

SIMD Optimization The inner loop for sampling in our parallelized code (e.g. in Figure 9) can be vectorized across iterations. This indicates that the overhead for computing on multiple initial value sets can be eliminated if they can be accommodated in the SIMD vector. We apply this optimization to our codes whenever possible to improve the overall parallelization performance.

5.2 Benchmarks

We collected the comprehensive test cases from several previous research efforts in this area [11, 22, 24] and added certain additional loops. As listed in Table 2, the test cases include:

- mps (maximum prefix sum), mss (maximum segmented sum), mts-p (maximum tail sum with position returned), ss (second smallest), and bal (balanced-()) which checks if a string of brackets is balanced, are commonly used test cases from previous papers [11, 19, 24, 25].
- ses (single exponential smoothing) and des (double exponential smoothing) are common techniques for smoothing time series data [2].
- msp (maximum segmented product), as shown in Figure 8, returns the subarray with maximum product.
- agg1 and agg2 are two user-defined aggregation queries adopted from Raychev *et al.* [22]. agg1 was shown earlier in Figure 3 and counts the events between "PUSH" and "PULL" on a repository in github datasets. agg2 counts the number of operations in each session that is determined by whether the time interval between two consecutive operations are smaller than a threshold.

The function types in these loops are also shown in Table 2. They can be parallelized using our approach with the number of initial value sets as shown in the third column of Table 2 – here each multiplicand indicates the number initial values for sampling of each recurrence variable.

As a comparison, we try to parallelize all the test cases with the program synthesis based tool Parsynt[1] from Farzan *et al.* [11]. The results are shown in the last column of Table 2. As expected, mps, mss, mts-p, ss, and bal, which are included in their benchmarks, can be successfully parallelized by Parsynt. However, for ses, des, msp, agg1 and agg2, Parsynt fails to produce a parallelization. The codes are successfully translated to functional representations and given to the SMT solver, but the solver fails to find solutions for these functions and finally aborts after a long time of searching. Even for the loops that Parsynt can parallelize, it takes tens of seconds to minutes to generate the parallelized codes. Our tool however parallelizes all of the loops almost instantly (in less than 1 sec) as we construct the combined function directly without any search.

5.3 Parallel Performance

There are three aspects in our performance evaluation: 1) scaling across cores with our approach, 2) performance comparison with the codes generated by Parsynt [11], and 3) the extra benefits from SIMD processing of multiple portions of iterations.

For mps, mss, mts-p, ss and bal, which are the five loops that can be parallelized by Parsynt [11], we compare the performance of the parallel code generated by our approach and Parsynt. The basic version of our approach (our-nonvec) only uses multithreading to process different portions of loop iterations and does not use any SIMD optimization (as shown in Figure 9). The first optimized version of our approach (our-vec1) vectorizes the inner loop of the basic version. That is, the multiple sampling are performed simultaneously in SIMD vectors. For mps, ss, bal, because our approach only needs one sample point (i.e., the inner loop has only one iteration), our-vec1 is not applicable. As SIMD units are commonly supported in modern CPUs and the inner loop is quite easy to vectorize (either automatically or manually), we use this version as the default version of our approach. The second optimized version of our approach (our-vec2) vectorizes both the inner and the outer loop of the basic version. Because the iterations in different portions of the outer loop are independent, they can be processed simultaneously in SIMD vectors. According to the number of sampling sets required in each case as shown in Table 2, we apply the optimization to mps, mss, mts-p, ss, bal, ses and des. The number of portions processed per SIMD vector is the number of lanes in the vector divided by the number of sampling sets required for the loop. It is difficult for a compiler to do this vectorization automatically, so we manually generate this version with an API[2] for SIMD programming on Intel Xeon Phi processors without explicitly writing Intel Intrinsics [15]. For agg1, the input is the log of January

[1] https://github.com/victornicolet/parsynt

[2] https://github.com/lcchen008/irreg-simd/tree/master/SSE_API_Package/

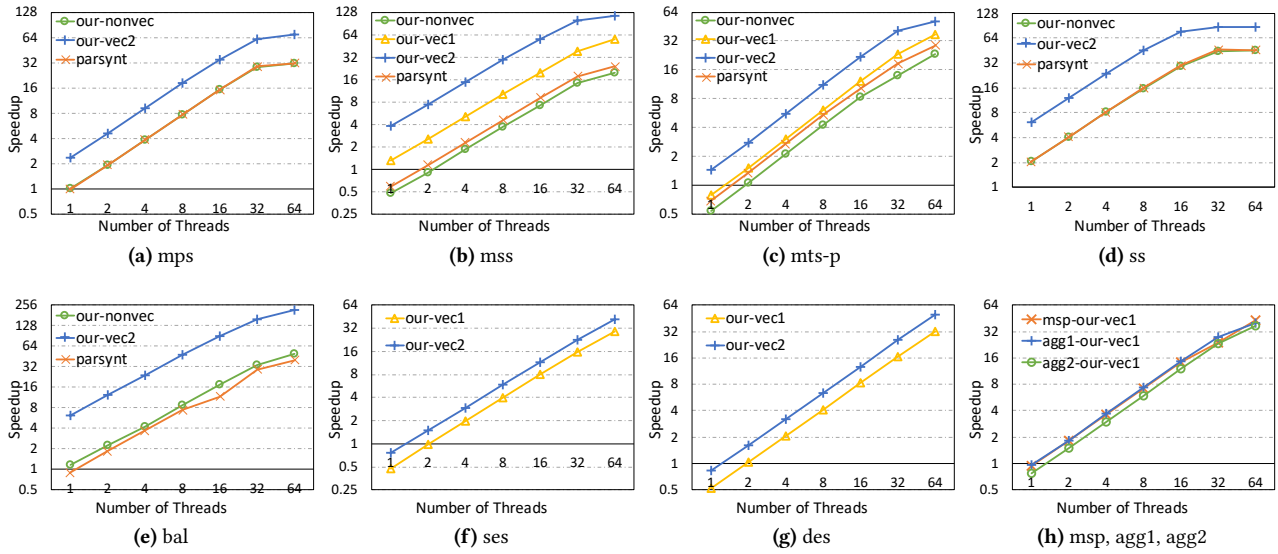


Figure 10. Speedups over sequential codes for all the test cases

2015 in github archive[3], which has about 12M entries. For agg2, the input is a randomly generated sequence of 64M event-timestamp pairs. For other loops, we use randomly generated arrays of 64M entries as the input.

Figure 10 shows the speedups of the parallelized codes against the original sequential codes. Our approach without SIMD optimization (our-nonvec) achieves the same performance with Parsynt for mps and ss as the parallelized codes generated by Parsynt also do not require *auxiliary computation*. For bal, our-nonvec achieves slightly better performance than Parsynt because Parsynt needs to do auxiliary computation while we do not. For mss and mts-p, two sample points are needed in our approach. The basic version our-nonvec runs slightly slower than Parsynt because the extra sampling in our approach incurs a bit more operations than the codes generated by Parsynt (see §3.5). However, this overhead is easily hided by SIMD processing of the multiple sampling as our-vec1 outperforms Parsynt.

Parsynt fails to parallelize the rest of the test cases. For ses and des, because all of the intermediate results are required (i.e., the loops are scans), we have to take two passes over the inputs, so the speedups of our-vec1 are half the number of the computing threads. The two user-defined aggregations (agg1 and agg2) also involve intermediate outputs, but the number of outputs is much smaller than the number of iterations. Similar to the idea of SymVector in Raychev *et al.* [22], we store the intermediate sample values for the outputs during the parallel execution and calculate the actual output values at the end of the entire loop. This makes the two scan-like loops parallelizable as reductions. For this reason, we only apply inner-loop vectorization (our-vec1) to the parallel codes for the two loops and we achieve near linear speedup across multiple cores. For msp, our-vec1 achieves a similar speedup.

For the test cases except msp, agg1 and agg2, our-vec2 achieves an extra 1.5x to 5.4x speedups over our-vec1. The speedups may not be the optimal because there is tradeoff between cache locality and propagation overhead that are controlled by the block size. We simply choose 8KB entries as our block size to show the efficacy of this optimization.

6 Related Work

Static Approaches There have been a long series of efforts on extracting parallelism in loops by compiler techniques. A seminal work that parallelizes complex scans and reductions was proposed by Fisher *et al.* [13]. Their key idea was to use aggressive symbolic analysis to search for closed-form combinations of the function of the loop body. Many followup efforts presented different formalizations and improvements extending this idea. For example, Sato *et al.* abstracted the computation in certain loops as matrix multiplication on a semiring and parallelized these loops as matrix chain [24]. Morita *et al.* formalized the loops with list homomorphism and used the third homomorphism theorem to derive parallel programs [19]. More recently, Farzan *et al.* presented a more practical solution also formalized with list homomorphism [11]. They used an SMT solver to synthesize the *join operators* and *auxiliary accumulators* that meet the list homomorphism properties and parallelized the loops in a divide-and-conquer fashion. In another recent effort, Fedyukovich *et al.* proposed to gradually synthesize the composed functions across loop iterations. They relied on the same SMT solver to generate the parallel codes and ended up with the same results [12]. Halide [26] is another parallelization tool based on SMT solver. Ginsbach *et al.* presented techniques to discover reductions including simple scalar reductions

[3] https://www.githubarchive.org/

and complex histogram reductions [14]. As shown in our evaluation, the state-of-the-art tool in this line of research, `Parsynt` from Farzan *et al.* [11] cannot parallelize certain loops that our approach can work on.

Runtime Approaches A commonly used method to break dependences in loops is thread-level speculation (TLS) [8, 9, 28]. Many of the TLS efforts are based on the idea that certain dependences in programs that are diagnosed at compile-time rarely happen at runtime. During the speculative execution, the systems check whether the dependent variables are modified; if so, they simply rollback and re-execute the speculation work [8]. For variables with constant dependences, value prediction is often used to improve speculation success rate [21, 27]. One successful application of value prediction is in parallelizing finite-state machines (FSMs). Zhao *et al.* leveraged the state convergence in FSMs and successfully parallelize a broad class of FSMs with *principled speculation* [29, 30]. However, for more general cases such as those considered in this paper, speculation is not likely to work because predicting the starting state for integers or floating-points of arbitrary values is difficult. A recent work on parallelizing user-defined aggregations shows the potential of using symbolic execution to break dependences at runtime [22]. However, the efficacy of symbolic execution is dependent on the existence and efficient maintenance of the *canonical forms* of recurrence variables.

7 Conclusion

This paper has presented a methodology to extract scan or reduction parallelism in recurrence loops. The key idea is a *point-value* representation of the loop body functions. We observe that with outputs on a few sampling inputs, we can preserve the information of the functions modeling the recurrent loop bodies, and can reconstruct the combined functions over iterations. We discussed several of the most common function types and their sampling and reconstruction techniques. We further explain the program analysis algorithms to automate our parallelization process. The experiments show that our approach can parallelize a diverse set of sequential loops, and the parallelized codes achieve linear scalability across multicores.

A Artifact appendix

A.1 Abstract

The artifact includes all of the programs that are needed to reproduce the experiments in the paper "Revealing Parallel Scans and Reductions in Recurrences through Function Reconstruction". There are mainly two parts in the artifact: (1) a compiler tool for parallelizing recurrence loops and (2) the codes for performance evaluation. The compiler tool is implemented as an LLVM IR pass and is used within Clang; this tool should be able to work on any platform with proper LLVM installation. The parallelized codes for performance evaluation should be executed on an Intel Xeon Phi 7250 (Knights Landing) processor as the codes use AVX-512 instruction set. An Intel ICC compiler is also required for compiling the codes on the Xeon Phi KNL processor.

A.2 Artifact check-list (meta-information)

- **Program:** (1) a LLVM IR pass to determine the function types in recurrence loops as described in §4, (2) a code generation tool to produce the parallelized codes as described in §5.1, and (3) the parallelized and vectorized codes for reproducing the results in Figure 10.
- **Output:** Program results including the execution time and speedups are printed out to the console; the speedups are also saved in a txt file and a python script is provided to plot the speedups similar to those in Figure 10.
- **Experiment workflow:** A bash script is provide to automatically compile and run the performance evaluation on the Intel Xeon Phi processor. Details of how to setup and use the compiler tool are in its Github repository.
- **Artifacts publicly available?:** Yes
- **Artifacts functional?:** Yes
- **Artifacts reusable?:** Yes
- **Results validated?:** Yes

A.3 Description

A.3.1 How delivered

The artifact is publicly available at https://github.com/pengjiang030/pact2018_artifact.git

A.3.2 Hardware dependencies

An Intel Xeon Phi 7250 (Knights Landing) processor is required.

A.3.3 Software dependencies

Intel ICC 17.0.0, LLVM 6.0.0 and Clang 6.0.0 are required.

A.3.4 Data sets

Most of the programs use randomly generated data. Only `agg1` uses data from github archive as described in §5.3. We preprocessed the data and stored them on Dropbox (please make sure you have access to Dropbox). The bash script provided in the artifact will download the data automatically.

A.4 Installation

The performance evaluation part does not need installation – simply download the codes to a machine with Intel Xeon Phi 7250 processor. The compiler part requires installation of LLVM 6.0.0 and Clang 6.0.0. Please see the Github repository for detailed instructions to setup the compiler tool.

A.5 Experiment workflow

Performance Evaluation Part, on an Intel Xeon Phi 7250:

1) Download the codes:
`git clone https://github.com/pengjiang030/pact2018_artifact.git`

2) Open to the folder for performance evaluation:
`cd pact2018_artifact/performance_eval`

3) Compile the codes, download the input data, and run the programs with the provided bash script:

chmod +x run_test.sh; ./run_test.sh.
The execution time and speedup for each test case will be printed out to the console.

4) Wait for the programs to finish (should be less than 30 min), the speedups will be saved in a file 'speedups.txt'. Make sure 'speedups.txt' is generated and has content, then plot the speedups: python plot.py

5) Images of pdf format will be generated in the folder. Open the images (you may need to download them to a laptop) and compare the generated images with Figure 10. (The speedups should be close; note that the generated images may have different colors to Figure 10. Figure 10 was originally plotted by Excel, we provide the 'plot.py' here just to facilitate the artifact evaluation.)

Compiler Part:
See the Github repository for detailed instructions to setup the environment and install the tool.

A.6 Evaluation and expected result

The speedups should be close to those reported in the paper. The compiler tool should be able to parallelize all the test loops.

A.7 Experiment customization

The compiler tool works on any machine with installation of LLVM 6.0.0 and Clang 6.0.0. Please follow the instructions in the Github repository if you want to install it on your local machine.

A.8 Notes

The compiler tool is currently a prototype, especially the code generation part. The automatically generated codes use one more samples for 'msp.cpp' and 'ss.cpp' compared with the codes for performance evaluation. This is because the code generation part needs more engineering efforts to implement this optimization, while it is much easier to do manually based on the function relation graph. Also, for 'msp.cpp', we use 16 samples for performance evaluation as reported in the paper. However, we later find that the *rectified-linear* function (or its extended line) of this loop passes origin, so we actually need only 4 samples for parallelization. This optimization has been implemented in our compiler tool, so you may notice that the automatically generated 'msp_par.cpp' uses only 4 samples instead of 16.

Acknowledgements

This work was supported by NSF award CCF-1526386.

References

[1] Online. Bilinear Interpolation. (Online). https://en.wikipedia.org/wiki/Bilinear_interpolation

[2] Online. Exponential Smoothing. (Online). https://en.wikipedia.org/wiki/Exponential_smoothing

[3] Zahira Ammarguellat and W. L. Harrison, III. 1990. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/93542.93583

[4] Eli Bendersky. 2014. Modern source-to-source transformation with Clang and libTooling. (2014). https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling

[5] G. E. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.* 38, 11 (Nov 1989), 1526–1538. https://doi.org/10.1109/12.42122

[6] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

[7] D. Callahan. 1992. Recognizing and parallelizing bounded recurrences. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–185.

[8] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. 2007. Software Behavior Oriented Parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 223–234. https://doi.org/10.1145/1250734.1250760

[9] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *ACM Comput. Surv.* 49, 2, Article 22 (June 2016), 39 pages. https://doi.org/10.1145/2938369

[10] Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design (FMCAD '15)*. FMCAD Inc, Austin, TX, 57–64. http://dl.acm.org/citation.cfm?id=2893529.2893544

[11] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of Divide and Conquer Parallelism for Loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 540–555. https://doi.org/10.1145/3062341.3062355

[12] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3062341.3062382

[13] Allan L. Fisher and Anwar M. Ghuloum. 1994. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/178243.178255

[14] P. Ginsbach and M. F. P. O'Boyle. 2017. Discovery and exploitation of general reductions: A constraint based approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 269–280. https://doi.org/10.1109/CGO.2017.7863746

[15] Xin Huo, Bin Ren, and Gagan Agrawal. 2014. A Programming System for Xeon Phis with Runtime SIMD Parallelization. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM, New York, NY, USA, 283–292. https://doi.org/10.1145/2597652.2597682

[16] Peng Jiang and Gagan Agrawal. 2017. Combining SIMD and Many/Multi-core Parallelism for Finite State Machines with Enumerative Speculation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/3018743.3018760

[17] Peter M. Kogge and Harold S. Stone. 1973. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Trans. Comput.* 22, 8 (Aug. 1973), 786–793. https://doi.org/10.1109/TC.1973.5009159

[18] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (Oct. 1980).

[19] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-conquer Parallel Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 146–155. https://doi.org/10.1145/1250734.1250752

[20] Shlomit S. Pinter and Ron Y. Pinter. 1991. Program Optimization and Parallelization Using Idioms. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 79–92. https://doi.org/10.1145/99583.99597

[21] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. 2008. Spice: Speculative Parallel Iteration Chunk Execution. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)*. ACM, New York, NY, USA, 175–184. https://doi.org/10.1145/1356058.1356082

[22] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/2815400.2815418

[23] Xavier Redon and Paul Feautrier. 1993. Detection of Recurrences in Sequential Programs with Loops. In *In PARLE 93, Springer LNCS*. 132–145.

[24] Shigeyuki Sato and Hideya Iwasaki. 2011. Automatic Parallelization via Matrix Multiplication. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 470–479. https://doi.org/10.1145/1993498.1993554

[25] R. Shi, S. Potluri, K. Hamidouche, X. Lu, K. Tomko, and D. K. Panda. 2013. A scalable and portable approach to accelerate hybrid HPL on heterogeneous CPU-GPU clusters. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. Indianapolis, IN, USA, 1–8.

[26] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 281–291. http://dl.acm.org/citation.cfm?id=3049832.3049863

[27] Chen Tian, Min Feng, and Rajiv Gupta. 2010. Speculative Parallelization Using State Separation and Multiple Value Prediction. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1806651.1806663

[28] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. 2011. Enhanced Speculative Parallelization via Incremental Recovery. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/1941553.1941580

[29] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2694344.2694369

[30] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations Through Principled Speculation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 543–558. https://doi.org/10.1145/2541940.2541989