# Applying Temporal Blocking with a Directive-based Approach

Shota Kuroda*
Tokyo Institute of Technology

Toshio Endo
Tokyo Institute of Technology
endo@is.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology
matsu@acm.org

## ABSTRACT

Stencil kernels are important, iterative computation patterns heavily used in scientific simulations and other operations such as image processing. The performance of stencil kernels is usually bound by memory bandwidth, and the common method of overcoming this is to apply Temporal Blocking (TB) as a form of bandwidth reducing algorithm. However, applying TB to existing code incurs high programming cost due to real-life codes embodying complex loop structures, and moreover, multitudes of parameters and blocking schemes involved in TB complicating the tuning process. We propose an automated, directive-based compiler approach for TB by extending the polyhedral compilation in the Polly/LLVM framework, significantly reducing programming cost as well as being easily subject to auto-tuning. Evaluation of the performance of our generated stencil codes on Core i7 and Xeon Phi show that the auto-generated stencil kernels achieve performance that is close to and often on par with hand TB-converted and optimized codes.

## CCS CONCEPTS

• **Software and its engineering** → **Compilers**;

## KEYWORDS

Temporal blocking, Stencils, Polyhedral compilers, Compiler directives, LLVM, Manycore, Cache optimization

**ACM Reference format:**
Shota Kuroda, Toshio Endo, and Satoshi Matsuoka. 2017. Applying Temporal Blocking with a Directive-based Approach. In *Proceedings of LLVM-HPC'17: Fourth Workshop on the LLVM Compiler Infrastructure in HPC, Denver, CO, USA, November 12–17, 2017 (LLVM-HPC'17),* 11 pages.
DOI: 10.1145/3148173.3148190

## 1 INTRODUCTION

Stencil computations are important kernels heavily used for scientific simulations including fluid dynamics and structural analysis. In such kernels, computation and memory access have the same order of complexity, thus they tend to be memory-intensive. Since typical stencil implementations scan the entire target array for each iteration, they have low cache locality and impose high load to main memory.

Generally, blocking techniques have been used in order to reduce main memory load for performance improvement, since they improve data reuse to harness memory hierarchy efficiently. For stencil computations, while spatial blocking has been well known, *temporal blocking* (TB) have been proposed[1–4] as a technique to improve locality beyond spatial blocking. With TB, when the computation of an array part begins, we compute the part for several temporal steps locally at once. TB has been used not only for reduction of main memory access, but for the reduction of communication between GPUs and CPUs[5, 6] and SSD access costs[7].

In spite of those results, TB imposes high programming costs, as it introduces complex loop structures to application codes. Unlike blocking in dense matrix computations, TB introduces "skewed" block shapes, in order to preserve data dependency between adjacent points, which significantly complicates programming. To resolve this issue, some researchers have taken an approach with DSLs integrated with TB[8–10]. However, this approach requires applications already written in specific DSLs. Contrastingly, our approach is compiler-based, and focuses on applications written in general purpose languages, such as C/C++. We use polyhedral compiler technology, such as Pluto compiler[11] and Polly/LLVM[12, 13], in order to execute loop transformations.

Even when the loop transformation is achieved, users still need fine tuning according to the characteristics of applications and underlying architecture. The optimal temporal block size, the spatial block sizes are difficult to predict. Additionally, several block shapes have been proposed, including wavefront, overlapped, trapezoid and diamond; an appropriate shape should be selected among them.

This paper describes our compiling tool chain based on directives that enables users to tune TB parameters including block sizes and shapes, while specifying code fragments to be transformed for TB. It is also expected to make auto-tuning easier. Our tool chain is implemented as extensions to the LLVM compiler framework. The extensions mainly consist of:

(1) Directive-based parameter tuning mechanism as an extension to the Clang frontend[14],
(2) Loop transformation mechanism as an extension to the Polly polyhedral optimizer. As a byproduct using Polly, the resultant tool executes automatic OpenMP parallelization, in addition to introducing TB.

Even with this tool, we have found that some programming idioms that frequently appear in stencil computations,

* Currently Sony


*LLVM-HPC'17, Denver, CO, USA*

DOI: 10.1145/3148173.3148190

including double buffering, work adversarily and prohibits loop transformation when they are described in a straightforward fashion. We also demonstrate several case study and our current solutions.

We demonstrate the performance results of stencil codes optimized by our tool chain. The benchmark programs include simple one-dimension and two-dimension stencils, and platforms include a Core i7 (Sandy Bridge) server and a Xeon Phi (Knights Landing) server. The results show that generated code exhibits up to 1.8 times speedup on Xeon and up to 3.6 times speedup on Xeon Phi, which are comparable to hand TB-converted and optimized codes.

## 2 STENCIL COMPUTATIONS AND OPTIMIZATIONS

### 2.1 Stencil Computations

Stencil computations are iterative kernels that scan the entire target arrays to compute new arrays. Each value of new arrays are computed using the corresponding and adjacent values of old arrays. The following formula shows the computation of a single value in the simplest stencil form, called one-dimensional three-point (1D3P) stencil:

$$f_{t+1,x} = a_{-1}f_{t,x-1} + a_0 f_{t,x} + a_1 f_{t,x+1}$$

The largest distance used for computation of each value is called stencil *radius*, which is 1 in the above kernel.

When stencil computations are implemented in a naive fashion, each iteration requires $O(N)$ data to compute $O(N)$ array elements, where $N$ is the spatial array size. Thus memory channel becomes the bottleneck, prohibiting harnessing of computation power of modern processors with low Bytes/Flops ratio, especially in multi/many-core cases.

To alleviate the issue, spatial blocking has been proposed for multi-dimensional stencils. Here the spatial array is divided into smaller blocks to improve data reuse. However, the reuse is still limited inside a single iteration. For more aggressive data reuse, loop transformation over multiple iterations are necessary.

### 2.2 Temporal Blocking

Temporal blocking introduces blocks divided both in spatial and temporal dimensions. When we start computation of a spatial block, we continue computation of that block for several steps locally at once, before moving to other spatial blocks. This improves access locality significantly over spatial blocking. However, we have to be careful to preserve data dependency enclosed in the original computation, which significantly complicates the programming.

In the example in Section 2.1, the computation of $f_{t+1,x}$ depends not only on $f_{t,x}$ but on $f_{t,x-1}$, $f_{t,x+1}$. Thus the spatial size that can be computed locally shrinks by the stencil radius for each iteration shown as red blocks in Figure 1. This figure shows the one-dimensional case; here the blocking shape is called "trapezoid" and the temporal block size is 3, and the spatial block size is 9. After computations of red blocks in time steps 1 to 3, we can compute blue blocks in the same time steps. Then we can go to next time steps.

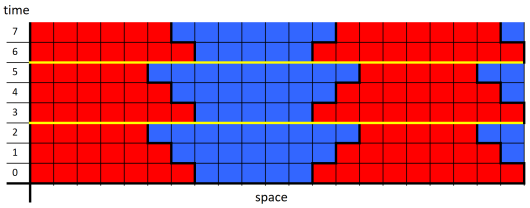


**Figure 1: An example of temporal blocking. The block shape is "trapezoid".**

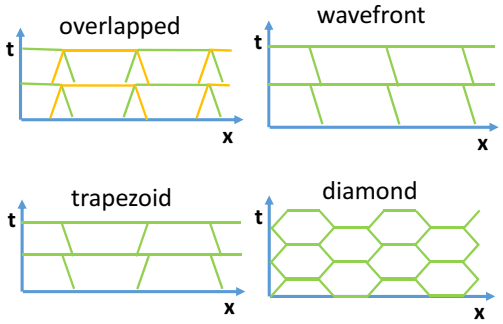


**Figure 2: Examples of block shapes.**

The intention of TB is to reduce main memory access. When $k$ is the temporal block size and the spatial block size is enough small to be accommodated in cache, main memory access is expected to be reduced to $1/k$.

The block shape that preserves data dependency is not unique; Figure 2 shows several examples of such shapes, with varying tradeoffs: overlapped blocking introduces redundant computations, while the programming costs are lower than others. Among the other three, while wavefront blocking does not have limitation on temporal block size $k$, the spatial blocks must be computed sequentially (left to right in the Figure). Trapezoid and diamond blocking enables parallel block computation, while temporal block size is limited by spatial block size. In this paper, our tool omits support of overlapped blocking; we consider this is not a large flaw since its importance is low due to inefficiencies of redundant computations.

Through the above discussions, we observe TB incurs significant load to users in programming and tuning, resulting in low adoptions in real codes despite the advantages. Since TB introduces "skewed" blocks, TB requires complex loop structures, especially for multiple dimensional cases. Also the appropriate block shape and temporal/spatial block sizes may depend on characteristics of kernels and underlying architecture. Thus a tool chain that performs loop transformation for TB on existing codes enabling easy tuning are required for practical use.

# 3 LLVM

## 3.1 LLVM Overview

In order to construct a tool chain for loop transformation and directive-based tuning mechanisms, we use the LLVM compiler infrastructure[13] as our basis. LLVM is a collection of compiler modules that are designed to be reusable. Among the many available modules, we mainly extend the Clang frontend [14] for directive-based tuning mechanisms, and the Polly optimizer [12] for loop transformation.

The key feature of LLVM, which supports various languages and architectures, is a common intermediate language, called LLVM-IR (IR, hereafter). The compilation flow of source programs consists of three steps:

(1) Front-end takes a source program as input and translates it to IR. Clang is a front-end module for C/C++ language.
(2) Middle-end consists of various optimization modules. Each module, called a pass, takes an IR code as input and generates an optimized IR code. Polly is included in the middle-end modules.
(3) Back-end translates IR to the target machine code, such as x86 code.

While the front-end module translates source code to IR, it can handle information given by directives, which are expressed by `#pragma` in C/C++. The front-end can add such information to IR as additional metadata. Middle-end and back-end modules can use the metadata for their optimization or analysis. In our tool chain, information about block sizes and shapes are passed as metadata from the front-end to the loop optimizer based on Polly.

## 3.2 Polly

Our transformation module for temporal blocking is implemented as an extension to Polly, which is a loop transformation tool for LLVM-IR based on the polyhedral model. Polly mainly consists of the following middle-end passes, which are tightly coupled with each other (while the actual implementation has more complex structure, the following descriptions are simplified).

(1) *SCoP detection pass* takes IR code as input, and detects code regions that Polly can transform. Such regions are called Static Control Part of program (SCoP), which typically corresponds to a (nested) loop.
(2) *Polyhedral model construction pass* generates a polyhedral expression described below for each SCoP. The polyhedral expression for a SCoP is stored into a `.jscop` file.
(3) *Loop transform pass* modifies polyhedral expressions, which corresponds to loop transformation.
(4) *Code generation pass* takes modified polyhedral expressions and the original IR code as input. It verifies validity of the modified expression, and if it is valid, transformed IR code is generated. It also inserts OpenMP parallelization code if appropriate.

While Polly has a strong capability for loop transformation and analysis, code region that can be transformed (SCoP) must satisfy several conditions. SCoP detection pass finds such code regions. The followings are conditions related to our context.

- The code region has an entry point and an exit point.
- Control statements:
  - Allowable control statements are `for` and `if`.
  - Each `for` loop has a single induction variable (IV). IV is incremented by a constant value for each iteration.
  - The lower bound and upper bound of a `for` loop is expressed as affine expressions of parameters (constants and variables that are not modified during execution of the region) and IVs of outer loops.
  - Condition of each `if` statement is a comparison of two affine expressions.
- Non-control statements:
  - Statements are equivalent to an assignment of a value to an element of an array. The value is the computed result of an expression that includes operators or functions without side effects whose operands are array elements, parameters or IVs.
  - Array indices are affine expressions of parameters and IVs.

For each detected SCoP, polyhedral model construction pass generates a polyhedral expression. Here, an $n$-level nested loop is mapped to a domain in $n$-dimensional integer space $\mathbb{Z}^n$, and a single iteration corresponds to a point in the space. An iteration domain must be a convex set for loop transformation, optimization and dependency analysis.

A polyhedral expression of a SCoP consists of the following elements.

- name: name of the SCoP
- context: constraints on parameters given to the SCoP
- statement[] : a set of statements in the SCoP
  - name: name of the statement
  - domain: a $n$-dimensional integer set that consists of possible value for $n$ loop induction variables (IV)
  - schedule: an integer map that assigns to a IV vector a multi-dimensional point in time in order to define execution order of statements
  - accesses[] : a set of memory accesses in the statement
    - kind : kind of memory access. read, may write or write
    - relation : an integer map, which maps from the domain of the statement to memory spaces to be accessed

Figure 3 shows a simple example that includes a single loop (`LOOP1`) treated as a SCoP. Figure 4 shows the (simplified) polyhedral expression of `LOOP1`. In this example, a

```
void calc(float *d,float *s,const int nx){
  const float alpha = 1.0f / 3.0f;
  for(int x=0 ; x<nx ; ++x){                  // LOOP1
    d[x] = alpha * (s[x-1] + s[x] + s[x+1]); // S1
  }
}
```

**Figure 3: An code example that can be transformed by Polly.**

```
"name": "LOOP1",
"context" : "[nx]",
"statements" : [ {
  "name" : "S1",
  "domain" : "[nx] -> {S1[x] : 0 <= x < nx}",
  "schedule" : "[nx] -> {S1[x] -> [x]}"
  "accesses" : [ {
    "kind" : "read",
    "relation" : "[nx] -> {S1[x] -> s[-1 + x]}"
   },{
    "kind" : "read",
    "relation" : "[nx] -> {S1[x] -> s[x]}"
   },{
    "kind" : "read",
    "relation" : "[nx] -> {S1[x] -> s[1 + x]}"
   },{
    "kind" : "write",
    "relation" : "[nx] -> {S1[x] -> d[x]}"
   } ],
 } ]
```

**Figure 4: The polyhedral expression of the loop in Figure 3 in Polly**

parameter `nx` is defined outside of the SCoP, and described in the context. This SCoP includes a single statement, `S1`. The domain of `S1` corresponds to the set of possible value for IV vector. In this example, there is a single IV `x` and the domain is given by "`0 <= x < nx`".

The schedule maps the IV vector to a vector that expresses execution order. Each statement is executed according to lexicographical order of the resultant vector. In the example, the vector is `[x]`, thus iterations are simply executed in ascending order of `x`. "Accesses" show a set of memory accesses in the statement in each iteration. In the example, each iteration executes three reads and one write. The relation shows the memory locations to be accessed.

In the loop transform pass, the polyhedral expression, especially "schedule" clause, is modified to change the execution order of the iterations. Finally code generation pass reconstructs new IR code by using modified SCoP and the original IR code.

While Polly has been successful for blocking of dense matrix kernels or spatial blocking of stencils, the current loop transform pass does not support temporal blocking with skewed blocks. Thus we propose and implement an extension to support temporal blocking as shown in Section 4.

Also we have found that the conditions for SCoP detection prevent users from writing stencil codes in a simple fashion. For example, swapping two pointers for double buffering is not allowed. We will discuss this issue in Section 5.

# 4 PROPOSED TOOL CHAIN

## 4.1 Overview

Figure 5 shows overview of our tool chain based on LLVM. Our new modules, indicated by the red area, are extensions to the Clang front-end and loop transform pass in the Polly optimizer.

The extended Clang front-end takes a source program, parsing our directives proposed for temporal blocking. The parameters specified in the directives are embedded into IR code as metadata.

The extended loop transform pass, hereafter called *TB-Pass*, takes those parameters as input, and modifies "schedule" information of the polyhedral expression so that temporal blocking is applied. The remaining SCoP detection, polyhedral model construction and code generation are done by original passes in Polly.

## 4.2 Design of Directives

Our motivation of introducing new directives is to enable flexible tuning of temporal blocking parameters, including block sizes and block shapes for both manual and auto tuning. We assume that our directive is inserted before the outermost loop of the target nested loop, since typically the outermost loops corresponds to a temporal loop.

A directive line starts with prefix `#pragma tb`, which is followed by several clauses for configuring parameters as follows:

- `tile_size(bt, b1, b2 ,...)`: Block sizes
- `radius(r1, r2, ...)`: Stencil radii
- `scheme(s1, s2, ...)`: Block shapes

Each clause generally takes multiple parameters to support multiple dimensional stencil computations. The `tile_size` takes parameters for block sizes; the first one is for temporal (outermost) loop, which is denoted as `bt` above. The following parameters are block sizes for spatial loops.

The `radius` clause is to specify stencil radii for spatial dimensions. The `scheme` clause is used to specify block shapes for spatial dimensions. Each `s`$i$ is one of the following keywords:

- `none`: Blocking is not applied to the dimension
- `trapezoid`: Trapezoid blocking is applied
- `wavefront`: Wavefront blocking is applied

Figure 6 shows an example stencil code with our directives [1]. The code includes a triply nested loop, which consists of temporal loop, $y$-dimensional loop and $x$-dimensional loop. Three directive lines are attached just before the outermost temporal loop. The `tile_size` clause indicates that temporal block size is 16 and spatial block sizes are 64 along Y-axis and 128 along the X-axis. The `radius` clause indicates that the stencil radii are 1 along the Y-axis and 2 along the X-axis. The `scheme` clause indicates that trapezoid TB should be applied only along the Y-axis.

[1] Actually, this sample cannot be transformed as is, though it is used for explanation of directives. This issue is discussed in Section 5

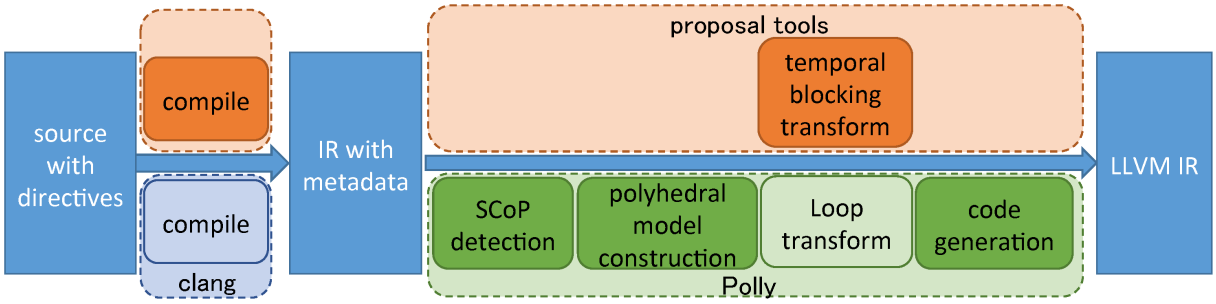


**Figure 5: Overview of our tool chain as an extension to LLVM**

```
#define IDX(x,y) ( (y) * stride + (x) )
float *f[2];
const float alpha;

f[0] = malloc( ... );
f[1] = malloc( ... );

#pragma tb tile_size(16,64,128)
#pragma tb radius(1,2)
#pragma tb scheme(trapezoid,none)
for (int t=0; t<nt ; ++t){
  int src = t%2;
  int dst = (t+1)%2;
  for (int y=0; y<ny ; ++y)
    for (int x=0; x<nx ; ++x)
        f[dst][IDX(x,y)] =
             alpha * ( f[src][IDX(x  ,y-1)]
                     + f[src][IDX(x-2,y  )]
                     + f[src][IDX(x-1,y  )]
                     + f[src][IDX(x  ,y  )]
                     + f[src][IDX(x+1,y  )]
                     + f[src][IDX(x+2,y  )]
                     + f[src][IDX(x  ,y+1)]);
}
```

**Figure 6: An example stencil code with our directives for temporal blocking**

## 4.3 Extended Clang Front-end

We have extended the Clang front-end to accept directive syntax describes above. The role of the extended Clang is to pass parameters specified with the directives to the resultant LLVM-IR code as metadata.

In LLVM, an IR instruction can have some metadata, each of which is expressed as a tuple of strings, integers, and references to other metadata. The extended Clang attaches metadata related to temporal blocking to an branch instruction corresponding to the backedge of the outermost loop. For example, the directives in Figure 6 are transformed into:

- `!{!"temporalblocking.tilesizes", i32 16, i32 64, i32 128}`
- `!{!"temporalblocking.radiuses", i32 1, i32 2}`
- `!{!"temporalblocking.schemes", !"trapezoid", !"none"}`

```
[param_1, ... , param_p] -> {
    Stmt[iv_1, ... , iv_n] ->
        [ts_1_1 , ... , ts_1_t] : condition_1 ;

        ... ;

    Stmt[iv_1, ... , iv_n] ->
        [ts_m_1 , ... , ts_m_t] : condition_m
}
// param: parameters specified in the context
// iv_*: loop induction variables
// ts_*: elements in timestamp vector
// condition_* : a condition which makes
                 the schedule applicable
```

**Figure 7: Format of a schedule string**

## 4.4 TB-Pass

Our TB-Pass module, an extension to Polly loop transform pass, has a role to apply temporal blocking to the code region. This pass takes a polyhedral expression generated by the preceding SCoP detection and the polyhedral model construction passes. It also uses temporal blocking parameters if they are found as metadata in IR code of the corresponding SCoP.

Among the polyhedral expressions in Polly, as exemplified in Figure 4, TB-Pass focuses on the "schedule" string. TB-Pass transforms the schedule so that temporal blocking is applied[2].

More generally, a schedule string is expressed as in Figure 7. The main components of a schedule are maps between a vector of induction variables `[iv_1, ... , iv_n]` embedded in the domain space and a timestamp vector `[ts_j_1 , ... , ts_j_t]`. As described in Section 3.2, a timestamp vector denotes the execution order of the iterations.

A schedule may have multiple (m in this case) maps, each of which has a condition that shows when the corresponding map is valid. We use this mechanism with condition to express blocks with different shapes, such as red blocks and blue blocks in Figure 1.

Hereafter we take one-dimension three-point (1D3P) stencil as an example, and explain how TB-Pass transforms the

[2]Polly internally represents polyhedral expressions as data structures defined in the integer set library (ISL)[15], such as `isl_map`. However, we implement schedule modification as string operation

```
// Schedule before transformation
[nt, nx] -> { S1[t, x] -> [t, x] }

// Schedule after transformation
[nt, nx] -> {
  S1[t, x] -> [TBID, 0, SBID, t, x] :                              // Red blocks
    ( TBID = floor(t / 3) and                                      // (RC1)
      SBID = floor(( x + 1 * ((3-1) - (t-3*T))) / 14) and          // (RC2)
      floor(( x + 1 * ((3-1) - (t-3*T))) / 14)
        = floor(( x - 9 + 1 * ((3-1) + (t-3*T)) + 14) / 14)) ;     // (RC3)
  S1[t, x] -> [TBID, 1, SBID, t, x] :                              // Blue blocks
    ( TBID = floor(t / 3) and                                      // (BC1)
      SBID = floor(( x + 1 * ((3-1) - (t-3*T))) / 14) and          // (BC2)
      floor(( x + 1 * ((3-1) - (t-3*T))) / 14)
        != floor(( x - 9 + 1 * ((3-1) + (t-3*T)) + 14) / 14)) }    // (BC3)
```

**Figure 8: An example of transformation of a schedule string in one-dimensional stencil. The block shape is "trapezoid", temporal block size is 3, spatial block size (longer base of a trapezoid) is 9, and stencil radius is 1.**

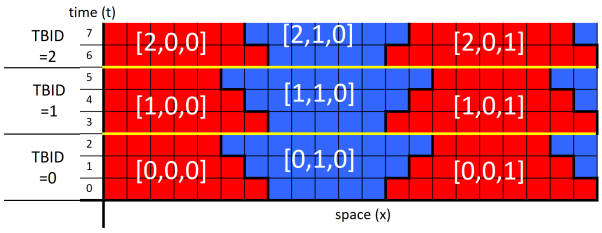


**Figure 9: Scheduling of blocks with trapezoid shapes shown in Figure 1. For each block, the first (TBID), second (red-or-blue) and the third (SBID) elements of the five-dimensional timevectors are illustrated.**

schedule string. We assume that the target code region is a doubly nested loop, which consists of an outer temporal (t-dimension) loop and an inner spatial (x-dimension) loop. We also assume that the directive syntax specifies the block shape, temporal block size, spatial block size (longer basis of a trapezoid) and stencil radius as "trapezoid", 3, 9 and 1, respectively as in Figure 1.

For such a code region, TB-Pass obtains the original schedule as shown in upper part in Figure 8. In this schedule, an IV vector `[t,x]` is simply mapped to an timestamp vector `[t,x]`. In TB-Pass, we transform the timestamp as in the lower part of the figure, whose concept is illustrated in Figure 9.

First, we introduce temporal block ID `TBID`, which is calculated by $\lfloor t/temporal\_block\_size \rfloor$ and shown in `(RC1)`, `(BC1)` lines (both are the same conditions). Since all iterations should be calculated in ascending order of `TBID`, the first element of the new timestamp vector is `TBID`.

Next, we need to classify the iterations with the same `TBID` into two groups, red ones and blue ones in Figure 1. We use the second dimension of the timestamp for this purpose. As the borderlines are skewed, the conditions are rather complicated as shown in `(RC3)` for red, and `(BC3)` for blue. Here the denominator of 14 in the formulae comes from the summation of longer basis and shorter basis sizes, which is $9 + (9 - (3 - 1) \times 2)$. Since the red iterations should have earlier timestamps than the blue ones, we make a distinction in the second dimension of the timestamp; we assign 0 to red ones and 1 to blue ones.

The third dimension of the timestamp is spatial block ID `SBID`, calculated in `(RC2)`, `(BC2)`. The fourth and fifth dimensions give execution order inside a block. Since all the iterations are scheduled according to the lexicographical order of the new timestamp vector, we achieve the desired scheduling with TB.

For the above one-dimensional example, we use two block types; in $n$-dimensional cases, we introduce $2^n$ block types.

The schedule string transformed by TB-Pass is passed to the following passes to generate IR code with temporal blocking. If a wrong directive parameter that may break data dependencies is given, a validation error is output by the code generation pass. An example of such cases would be specifying stencil radius as being 1, while the actual statement refers wider area of arrays.

## 5 FUTHER CODE TRANSFORMATIONS

The previous section has described the proposed tool chain to transform loop structure of stencil code for temporal blocking. While it is desirable to work with a wide variety of stencil codes, we have found some code patterns, such as double buffering that frequently appear, become obstacles for optimization when written in natural ways. Some patterns mitigate performance improvements, while others event prohibit loop transformations completely. This section discusses three such patterns and shows that systematic rewriting of user code recovers the ability for optimization.

### 5.1 Double Buffering

Double buffering is a standard and well known technique to reduce memory consumption of stencils significantly. Instead of holding spatial arrays for every time step, it is sufficient to hold arrays only for the latest two time steps. Code in Figure 6 shows a simple implementation of double buffering,

where two arrays `f[0]` and `f[1]` are updated alternately. To simplify the code, a variable `src` indicates 0 or 1, and `dst` indicates its counterpart, in order to express which is being written and read.

However, this implementation introduces indirect references to `f`, which breaks conditions described in Section 3.2, and thus the SCoP detection pass fails to recognize the code region as a SCoP. Thus we cannot transform this code for temporal blocking as is.

We have found that SCoP detection works well if the code is rewritten to remove indirect accesses. An example of resultant code is in Figure 10. Here odd iterations and even iterations are separated statically, although this rewrite makes the code longer. As a result, every memory reference has only constant base address, `f[0]` or `f[1]`, thus the enclosing loop can be treated as a SCoP.

Although temporal blocking is successfully introduced to the abovementioned modified code, we have noticed its execution performance is lower than expected. Through inspection of the result IR code and binary, we have observed that the innermost loop includes more memory access instructions than manually blocked code, for references to the array `f`, although `f[0], f[1]` are constant during the loop.

Figure 11 shows an example where references to the array `f` are removed from the loop. In this case, it is the programmers' responsibility to assure that memory regions `f[0], f[1]` do not have aliases, and the compiler knows `f0, f1` are constant during the loop execution. The performance improvement by this modification will be demonstrated in Section 11.

## 5.2 Multiple Spatial Loops

In real applications, it is normal that there are multiple spatial loops are included in the outer temporal loop. In an example of Figure 12, each of the three spatial loops may update different arrays, which makes SCoP detection fail. Figure 13 is a modified version to solve the issue. Here a new induction variable `t3` instead of original `t` is introduced so that `t`$= \lfloor$`t3`$/\#\_of\_loops \rfloor$ holds. Spatial loops to be executed is switched according to (`t3` $\%\#\_of\_loops$).

## 5.3 Discussion

The code modifications shown in this section have been applied manually in the experiments in next section, which increases programming costs. Of course, it is desirable that they are applied automatically and we are currently working to do so. We expect it is straightforward to provide middle-end passes for these code transformations. It would be more challenging to automatically detect which code region should be transformed, and what transformation should be applied; we expect that our directive-based approach will work well such for decision making.

```
f[0] = malloc( ... );
f[1] = malloc( ... );

#pragma tb tile_size(16,64,128)
#pragma tb radius(1,2)
#pragma tb scheme(trapezoid,none)
for (int t=0 ; t < nt ; ++t) {
  if (t % 2 == 0)
    for (int y=0 ; y<ny ; ++y)
      for (int x=0 ; x<nx ; ++x)
          f[1][IDX(x,y)] =
                alpha * ( f[0][IDX(x  ,y-1)]
                        + f[0][IDX(x-2,y  )]
                        + f[0][IDX(x-1,y  )]
                        + f[0][IDX(x  ,y  )]
                        + f[0][IDX(x+1,y  )]
                        + f[0][IDX(x+2,y  )]
                        + f[0][IDX(x  ,y+1)]);
  else
    for (int y=0 ; y<ny ; ++y)
      for (int x=0 ; x<nx ; ++x)
          f[0][IDX(x,y)] =
                alpha * ( f[1][IDX(x  ,y-1)]
                        + f[1][IDX(x-2,y  )]
                        + f[1][IDX(x-1,y  )]
                        + f[1][IDX(x  ,y  )]
                        + f[1][IDX(x+1,y  )]
                        + f[1][IDX(x+2,y  )]
                        + f[1][IDX(x  ,y+1)]);
}
```

**Figure 10: A modified version of Figure 6, which SCoP detection pass works with (TB-auto)**

```
void new_func(float * restrict f0,
              float * restrict f1, ...)
{
  #pragma tb tile_size(16,64,128)
  #pragma tb radius(1,2)
  #pragma tb scheme(trapezoid,none)
  for (int t=0 ; t < nt ; ++t) {
    if (t % 2 == 0)
      for (int y=0 ; y<ny ; ++y)
        for (int x=0 ; x<nx ; ++x)
            f1[IDX(x,y)] =
                  alpha * ( f0[IDX(x  ,y-1)]
                          + ... );
    else
      for (int y=0 ; y<ny ; ++y)
        for (int x=0 ; x<nx ; ++x)
            f0[IDX(x,y)] =
                  alpha * ( f1[IDX(x  ,y-1)]
                          + ... );
  }
}

   :
f[0] = malloc( ... );
f[1] = malloc( ... );
new_func(f[0], f[1], ...);
   :
```

**Figure 11: A further modified version of Figure 10 where references to `f` is removed from inner loop (TB-auto-2)**

```
for (t = 0; t < nt; t++){
  for (x = 1; x < nx - 1; x++)
    for (y = 1; y < ny - 1; y++)
      B[x][y] = 0.2 * ( A[x][y-1] + A[x][y+1]
            + A[x-1][y] + A[x+1][y] + A[x][y]);
  for (i = 1; i < nx - 1; i++)
    for (j = 1; j < ny - 1; j++)
      C[x][y] = 0.2 * ( B[x][y-1] + B[x][y+1]
            + B[x-1][y] + B[x+1][y] + B[x][y]);
  for (i = 1; i < nx - 1; i++)
    for (j = 1; j < ny - 1; j++)
      A[x][y] = 0.2 * ( C[x][y-1] + C[x][y+1]
            + C[x-1][y] + C[x+1][y] + C[x][y]);
}
```

**Figure 12: An example with multiple spatial loops**

```
for (t3 = 0; t3 < nt*3; t3++){
  if (t3 % 3 == 0)
    for (x = 1; x < nx - 1; x++)
      for (y = 1; y < ny - 1; y++)
        B[x][y] = 0.2 * ( A[x][y-1] + A[x][y+1]
              + A[x-1][y] + A[x+1][y] + A[x][y]);
  else if (t3 % 3 == 1)
    for (i = 1; i < nx - 1; i++)
      for (j = 1; j < ny - 1; j++)
        C[x][y] = 0.2 * ( B[x][y-1] + B[x][y+1]
              + B[x-1][y] + B[x+1][y] + B[x][y]);
  else
    for (i = 1; i < nx - 1; i++)
      for (j = 1; j < ny - 1; j++)
        A[x][y] = 0.2 * ( C[x][y-1] + C[x][y+1]
              + C[x-1][y] + C[x+1][y] + C[x][y]);
}
```

**Figure 13: A modified version to Figure 12**

# 6 PERFORMANCE EVALUATION

## 6.1 Evaluation Conditions

We evaluate performance of stencil benchmark codes with temporal blocking applied by our tool chain. We use two servers described in Table 1. One is a Core i7 processor of Sandy Bridge generation, and the other is Xeon Phi Knights Landing (KNL) processor with 64 cores. The current implementation of our tool chain is based on LLVM 4.0. When stencil codes are compiled, compile options related to base optimizations include `-O3 -march=native`.

The benchmark codes include a one-dimensional three-point (1D3P) stencil and a two-dimensional five-point (2D5P) stencil[3]. Double buffering technique is used, and the datatype of array elements is float. For each stencil type, performances of the following implementations are compared.

- Original: the original code without blocking
- SB-manual (only for 2D5P): spatial blocking is manually applied
- TB-manual: temporal blocking is manually applied
- TB-auto: temporal blocking is applied automatically by our tool chain. Double buffering is implemented in a fashion of Figure 10.

[3]Additionally, evaluations with three-dimensional code based on finite-difference time-domain method are ongoing.

**Table 1: Evaluation Environments**

| | SandyBridge | KNL |
|---|---|---|
| CPU | Core i7-3930K | Xeon Phi 7210 |
| CPU Clock | 3.2GHz | 1.3GHz |
| # of cores | 6 | 64 |
| # of HW threads | 12 | 256 |
| L1D cache size | 32kB | 32kB |
| Last level cache size | 12MB | 32MB |
| Memory bandwidth | 51.2GB/s | 102GB/s |

**Table 2: The numbers of threads used in evaluation**

1D3P

| | SandyBridge | KNL |
|---|---|---|
| Original | 6 | 64 |
| TB-manual | 12 | 64 |
| TB-auto | 12 | 64 |
| TB-auto-2 | 12 | 128 |

2D5P

| | SandyBridge | KNL |
|---|---|---|
| Original | 6 | 128 |
| SB-manual | 6 | 64 |
| TB-manual | 6 | 64 |
| TB-auto | 6 | 64 |
| TB-auto-2 | 6 | 64 |

- TB-auto-2: improvement of TB-auto, as shown in Figure 11.

The performance evaluation is done for multi-threaded cases parallelized with OpenMP. For the origial case and manual cases, we have used `#pragma openmp parallel` directives compiled by the Clang front-end. On the other hand, for the TB-auto case, we have relied on Polly's OpenMP auto-parallelization mechanism. The reason for this difference is that threaded IR codes generated by Clang does not pass the SCoP detection pass. The numbers of threads in the evaluation have been determined through preliminary measurements, which are done by varying the number of threads per core. The determined numbers of threads are shown in Table 2.

In all the cases, we used `-fno-vectorize` compile option to disable vectorization optimization. This is due to that we have observed vectorization does not currently work with our tool chain, but this only would underestimate the effectiveness of TB and our tool chain, as Bytes/Flops demand will only increase due to the amplified computational density with vectorization, i.e. with vectorization TB results would be better when vectorization would work in the future.

## 6.2 1D3P Stencil

For one-dimensional three-point stencil, we let spatial size be 16,777,216 and the number of time steps be 2048. In

TB-manual, TB-auto and TB-auto-2 cases, trapezoid block shapes are applied.

Figures 14 and 15 shows the impacts of temporal/spatial block sizes onto performance of TB-manual and TB-auto-2, respectively.

According to the graphs, optimal spatial block sizes are around 2048. This is explained by the fact that a spatial block size considering double buffering, $4B(float) \times 2 \times 2,048 = 16kB$ fits in the L1D cache. When comparing SandyBridge and KNL, temporal block size (TB) tends to have larger impact on KNL. While $TB \geq 4$ achieves almost optimal performance on SandyBridge, $TB = 8$ improves performance more than $TB = 4$ on KNL. This is likely due to the fact that since KNL has more cores, hiding memory access costs requires more aggressive blocking.

Figure 16 compares the performances of different implementations. The Y axis denotes the execution time and X axis denotes the temporal block size. For each case, the spatial block size has been configured for optimal values. In all cases with temporal blocking, performances are better than the original case, while we see differences between the implementations. On SandyBridge, TB-auto-2 achieves 1.9 times speedup compared with original, while TB-manual shows 2.1 times speedup. On KNL we observe a larger gap; TB-auto-2 and TB-manual achieve 2.3 times and 3.7 times, respectively. We are currently investigating its reason in detail. We suspect that we have not totally eliminated redundant memory access or that we are suffering from cache conflict misses more on KNL than SandyBridge. Compared with TB-auto-2 and TB-manual, TB-auto is less effective, which shows importance of eliminating redundant memory accesses during loop execution.

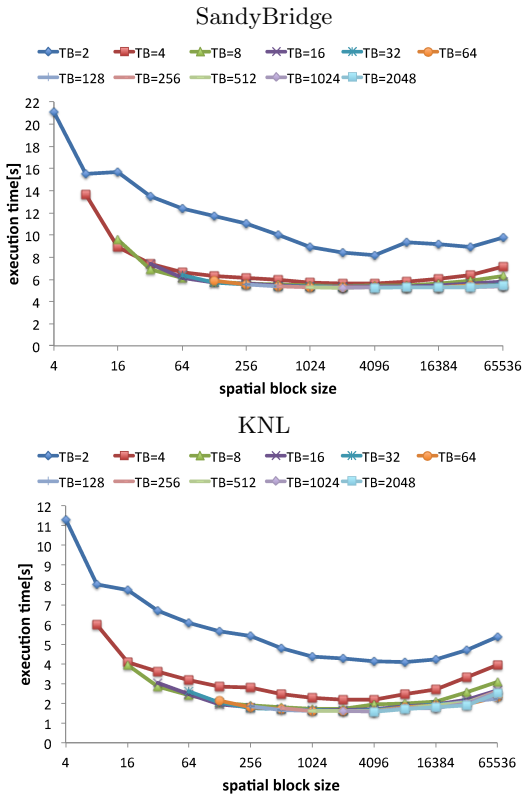


**Figure 14: Impacts of block sizes to 1D3P TB-manual performance (nx=16777216,nt=2048)**

## 6.3 2D5P Stencil

For two-dimensional five-point stencil, we let spatial size be $4096 \times 4096$ and number of time steps be 2048. When we use temporal blocking, trapezoid block shapes are applied.

Figure 17 compares performance of different implementations. Spatial block sizes have been configured to optimal ones. We observe TB-auto, which includes redundant memory accesses, shows the worst performance, worse than Original on SandyBridge. On SandyBridge, both TB-auto-2 and TB-manual achieve around 10% speedup than Original. On KNL, the temporal blocking has more impact; TB-auto-2 and TB-manual achieve 40% and 80% speedup, respectively.

Generally, in multi-dimensional stencils, avoiding blocking along the innermost loop may improve performance, for larger loop size. The SandyBridge graph includes such a case denoted by "TB-manual-Y", which shows 21% speedup compared to the Original [4].

# 7 RELATED WORK

Temporal blocking has been proposed as a technique to improve locality of stencil computations[1–4]. While there are reports for its application for improvement of cache usage, some researchers have used temporal blocking for reduction of PCI-Express communication between GPUs and CPUs[5, 6] and SSD access costs[7].

Malas et al. proposed a temporal blocking method called multi-threaded wavefront diamond blocking (MWD) to achieve higher performance for three-dimensional stencils[4]. It adopts wavefront shape for Z-axis, while diamond shape is used for Y-axis. One of motivations of our approach is to express such a complex blocking method as directives.

To alleviate issues regarding programming costs, some researchers have taken an approach with DSLs integrated with TB[8–10]. On top of them, the main task of programmers is to write operations for update of a single point. Blocked and parallelized code is generated by the DSL compiler. This approach requires applications already written in specific DSLs. Contrarily, our approach focuses on applications written in general purpose languages.

While our tool chain is implemented as an extension to LLVM, it is closer to Pluto[11], a source-to-source polyhedral translator. With Pluto, code regions enclosed by directives are transformed for blocking, including temporal blocking. Our tool chain has more flexibility in tuning, since it allows to configure different parameters for code regions. The

[4] We will evaluate these cases for TB-auto-2 soon, by applying `none` keyword in "`tb scheme`" directive

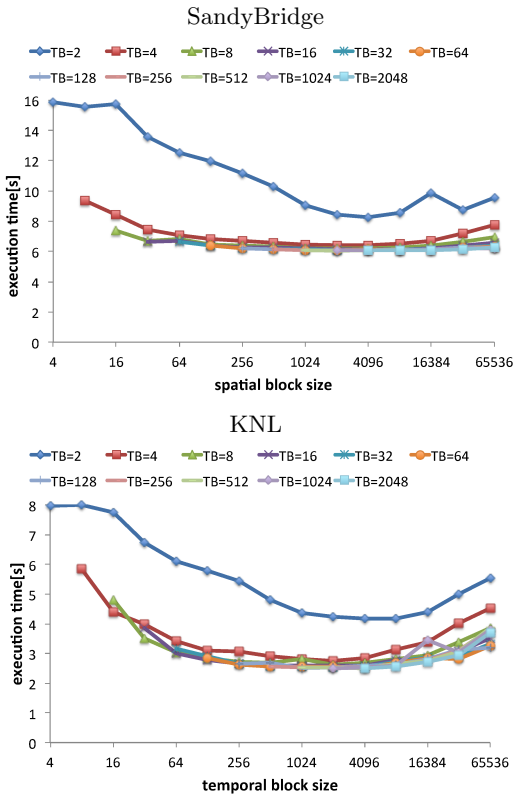


**Figure 15: Impacts of block sizes to 1D3P TB-auto-2 performance (nx=16777216,nt=2048)**

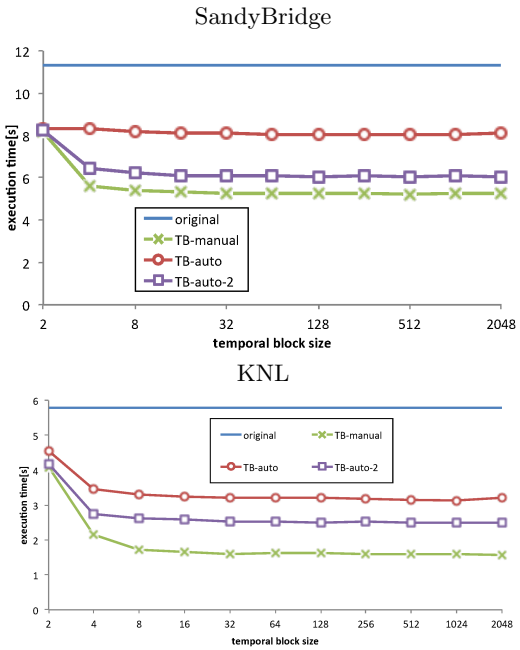


**Figure 16: Performance comparison of 1D3P implementations (nx=16,777,216,nt=2048)**

reason we have not used Pluto as basis is that we observed difficulty in transform of pseudo multiple dimensional arrays as in Figure 6.

## 8 CONCLUSION

We proposed a compiler tool chain that automatically applies temporal blocking to existing stencil codes written in C/C++. By providing directive based syntaxes, it allows flexible (auto) tuning of blocking sizes and shapes, on existing codes, not proprietary DSLs. The tool chain is implemented as an extension to the LLVM compiler framework. Through performance evaluation of generated codes, we demonstrated that they achieve comparable performance with manually optimized codes on a Sandy Bridge machine and a Knights Landing machine. We have also observed that programming fashions that frequently appear in stencil programming prohibit loop transformation, and we have discussed further transformation methodologies to rewrite codes to enable TB transformation automatically.

As future work, we are going to support the abovementioned code modifications, by providing middle-end passes for those modifications. More important direction is to apply our approach to real-world stencil applications. They may include complex loop structures spanning across multiple functions or source files, complex data structures, where arrays to be updated are embedded in deep structures, and so on. We will improve and demonstrate the applicability of the tool chain by investigating code patterns in such applications that prohibits transformation by polyhedral compilers.

## ACKNOWLEDGMENTS

This research is supported by JST-CREST, "Software Technology that Deals with Deeper Memory Hierarchy in Post-petascale Era" and JST-CREST, "EBD: Extreme Big Data - Convergence of Big Data and HPC for Yottabyte Processing".

## REFERENCES

[1] M. E. Wolf and M. S. Lam: A Data Locality Optimizing Algorithm. ACM PLDI 91, pp. 30–44 (1991).
[2] M. Wittmann, G. Hager, and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages (2010).
[3] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. IEEE/ACM SC'10, 13 pages (2010).
[4] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes: Multicore-optimized wavefront diamond blocking for optimizing stencil updates, SIAM Journal on Scientific Computing, 37 (4), C439-C464 (2015).
[5] L. Mattes, S. Kofuji: Overcoming the GPU memory limitation on FDTD through the use of overlapping subgrids. International Conference on Microwave and Millimeter Wave Technology (ICMMT), pp.1536–1539 (2010).
[6] G. Jin, T. Endo, S. Matsuoka: A Parallel Optimization Method for Stencil Computation on the Domain that is Bigger than

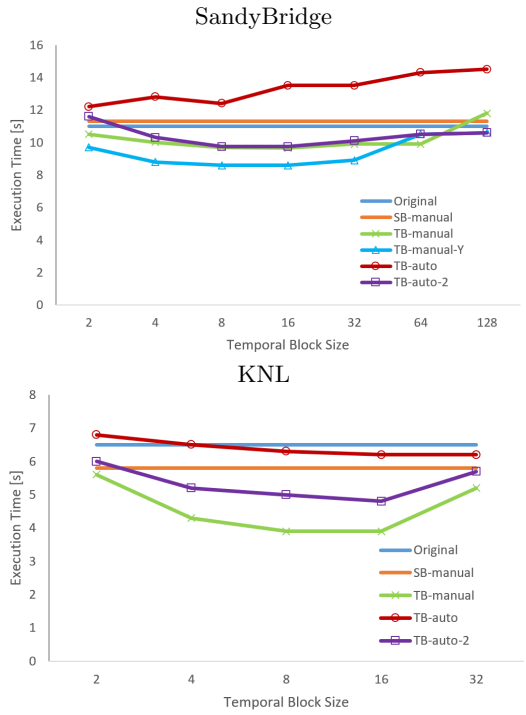


**Figure 17: Performance comparison of 2D5P implementations (nx=4096, ny=4096, nt=2048)**

Memory Capacity of GPUs. IEEE Cluster Computing (CLUSTER2013), 8 pages (2013).

[7] Hiroko Midorikawa, Hideyuki Tan: Evaluation of Flash-based Out-of-core Stencil Computation Algorithms for SSD-Equipped Clusters, The 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS2016), pp.1031-1040 (2016).

[8] T. Tang, R. Chowdhury, B. C. Kuszmaul, C. Luk, C. E. Leiserson: The pochoir stencil compiler, Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pp.117-128 (2011).

[9] N. Maruyama, T. Nomura, K. Sato, S. Matsuoka: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers, IEEE/ACM SC'11, 12pages (2011).

[10] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, Y. Nakamura: Simulations of Below-Ground Dynamics of Fungi: 1.184 Pflops Attained by Automated Generation and Autotuning of Temporal Blocking Codes, IEEE/ACM SC'16, 11pages (2016).

[11] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan: PLuTo: A practical and fully automatic polyhedral program optimization system, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), pp.101-113 (2008).

[12] T. Grosser, A. Groesslinger, C. Lengauer: Polly - Performing polyhedral optimizations on a low-level intermediate representation, Parallel Processing Letters, 22 (04), (2012).

[13] C. Lattner and V. Adve: LLVM: A compilation framework for lifelong program analysis and transformation. IEEE/ACM International Symposium on Code Generation and Optimization, p.75 (2004).

[14] C. Lattner: LLVM and Clang: Next generation compiler technology, The BSD Conference (2008).

[15] S. Verdoolaege: isl: An Integer Set Library for the Polyhedral Model, In Mathematical Software - ICMS 2010, LNCS Vol. 6327, pp.299-302 (2010).