Pipelined Multithreading Generation in a Polyhedral Compiler

Harenome Razanajato
hrazanajato@unistra.fr
University of Strasbourg and Inria
France

Cédric Bastoul
bastoul@unistra.fr
University of Strasbourg and Inria
France

Vincent Loechner
loechner@unistra.fr
University of Strasbourg and Inria
France

```
1  for (int i = 1; i < N; ++i)
2    A[i] = f1(A[i], A[i - 1]);
3  for (int i = 1; i < N; ++i)
4    B[i] = f2(A[i], B[i - 1]);
5  /* ... */
6  for (int i = 1; i < N; ++i)
7    Z[i] = f4(Y[i], Z[i - 1]);
```

(a) Original program

```
1  #pragma omp parallel
2  {
3      #pragma omp for schedule(static) ordered nowait
4      for (int i = 1; i < N; ++i)
5        #pragma omp ordered
6        A[i] = f1(A[i], A[i - 1]);
7      #pragma omp for schedule(static) ordered nowait
8      for (int i = 1; i < N; ++i)
9        #pragma omp ordered
10       B[i] = f2(A[i], B[i - 1]);
11     /* ... */
12     #pragma omp for schedule(static) ordered nowait
13     for (int i = 1; i < N; ++i)
14       #pragma omp ordered
15       Z[i] = f4(Y[i], Z[i - 1]);
16 }
```

(b) Pipelined OpenMP target program

Figure 1. Goal: automatically generate **(b)** from **(a)** to take advantage of pipelined multithreading with OpenMP

Abstract

State-of-the-art automatic polyhedral parallelizers extract and express parallelism as isolated parallel loops. For example, the Pluto high-level compiler generates and annotates loops with "`#pragma omp parallel for`" directives. Our goal is to take advantage of pipelined multithreading, a parallelization strategy allowing to address a wider class of codes, currently not handled by automatic parallelizers.

Pipelined multithreading requires to interlace iterations of some loops in a controlled way that enables the parallel execution of these iterations. We achieve this using OpenMP clauses such as `ordered` and `nowait`. The sketch of our method is to: (1) schedule a SCoP using traditional techniques such as Pluto's algorithm; (2) detect potential pipelines in groups of sequential loops; (3) fine-tune the schedule; and (4) generate the resulting code.

The fully automatic generation is ongoing work, yet we show on a small set of experiments how pipelined multithreading permits to parallelize programs which would otherwise not be parallelized.

Keywords automatic code generation, OpenMP, pipeline, polyhedral model

IMPACT 2020, January 22, 2020 , Bologna, Italy
.

1 Introduction

There has been extensive research on software pipelines focusing mostly on hardware generation, low level implementation (targeting for example CPUs) and VHDL generation (targeting for example field-programmable devices). In most of these researches the input programs are already in pipelined tasks format, and the goal is to efficiently map them on the available hardware. The two difficulties addressed by previous works are (1) to characterize the streams types and sizes between tasks and (2) to allocate the streams and the tasks on the fixed size hardware.

Our interest here is higher level pipelined multithreading code. In multithreading programming environments, streams and tasks allocation and size are not a problem since we benefit from multithreading support and a shared memory between threads. The question we try to answer is: how to make an automatic parallelizing compiler generate pipelined multithreaded code? Within a polyhedral compiler the powerful dependence analysis makes it possible to characterize the flow graph between potential pipelined tasks. However, state-of-the-art polyhedral compilers do not yet generate pipelined multithreaded code.

The two main issues that we address are:

1. how to automatically identify pipelines in a polyhedral compiler;

2. how to generate pipelined code using the OpenMP high level multithreading API.

Our proposal is based on the following steps. We start by a traditional polyhedral scheduling using Pluto for example; then we detect potential pipelines; and finally we modify and improve the scheduling by fusing loops. In the code generation step, we make extensive use of the OpenMP "`ordered`" and "`nowait`" clauses.

The teaser example of Fig. 1 presents the pipelined OpenMP code that we can automatically generate from a program. In this example, the original code (1a) is obviously pipeline-able: each function can be assigned to a task and executed successively in the following loops. In the pipelined version of this code (1b) the group of threads is created when entering the parallel region (line 1). Then, each thread will execute a block of successive iterations of the current loop (lines 4-6 for example) before executing the same iterations of the next loop (lines 8-10), without waiting for the first loop to be completed thanks to the `nowait` clause. Another thread will execute the following iterations of the first loop, and so on. The `ordered` clause ensures that the code contained in each loop is executed in order, since the loops carry a dependence and can not be parallelized.

This paper is organized as follows: Sec. 2 reviews the OpenMP constructs that we will be using. In Sec. 3 we describe the core of our method for generating pipelined code in a polyhedral compiler. Some preliminary experimental results are given in Sec. 4. We discuss some related work in Sec. 5 and conclude in Sec. 6.

2 Pipelines in OpenMP

There have been numerous proposals of dataflow parallelism, stream processing and process networks libraries and languages that permit to exploit pipeline parallelism. They usually rely on the support of low level or OS provided FIFOs. In multithreading environments like multicore general purpose processors, using FIFOs implies useless memory copies and synchronization overheads that could be avoided: very simple synchronization mechanisms are often sufficient to realize a software pipeline in a shared memory environment. Although the class of codes that we address is not as general as with tasks communicating through FIFOs, we propose to use asynchronous parallelism and ordered execution of parallel loop iterations to implement pipelines in nested loops.

On the other hand, source-to-source polyhedral parallelizers usually generate multithreaded parallel code using OpenMP [10]. For example, the Pluto parallelizer and locality optimizer [2] generates "`#pragma omp parallel for`" directives to enclose parallel loops. But OpenMP can also be used to generate pipelined multithreading, using the clauses that will be detailed hereafter. The advantage of using OpenMP is that it is well suited and efficient for shared memory architectures and it is already integrated and widely used in polyhedral compilers.

2.1 Parallel regions

Parallel execution begins at the starting point and sequential execution is resumed at the ending point of the *parallel regions*. When the master thread reaches a parallel region, worker threads are requested to execute the code within that region along with the master thread. Worker threads are stopped and the master thread continues the execution alone when all threads have reached the end of the parallel region. In OpenMP, the pragma "`omp parallel`" is used to specify that the work enclosed in the construct is a parallel region. Parallel regions come with a creation and management overhead. Optimized parallel codes should merge parallel regions when it is possible, eventually using synchronization mechanisms inside of the parallel region when necessary.

All available threads execute the code within a parallel region unless a *work sharing construct* is specified. Since we are dealing with polyhedral codes the most common work sharing construct that we will be using is the loop distribution among the threads. In OpenMP the pragma "`omp for`" is used to distribute the loop iterations of the loop enclosed in the construct among the available threads.

2.2 Ordered execution

For a piece of code contained in a parallel loop to be executed in order, OpenMP provides the "`ordered`" clause. It has to be provided at two levels: in the loop distribution work sharing construct and to enclose the code to be executed in order, as if the loop was sequential. The latter "`omp ordered`" clause may appear at most once in the parallel loop.

2.3 Synchronizations and asynchronicity

Work sharing constructs have synchronization barriers implied at the end: all threads have to wait for all the others before continuing their execution. We will explicitly remove them in order to generate pipelined loops. In OpenMP the "`nowait`" clause may be specified along with a work sharing construct to remove its implicit barrier at the end.

For the iterations of two such successive parallel loops to follow in order, the OpenMP specification enforces the use of the "`schedule(static)`" clause in the loop distribution work sharing construct. An optional explicit chunk size can be specified, but it must be the same one among pipelined parallel loops.

To perform explicit synchronizations, OpenMP also allows the declaration of a mutex using the `omp_lock_t` type. The mutex can then be accessed using the classical lock/unlock functions: `omp_set_lock()` and `omp_unset_lock()`.

3 Pipelined Multithreading Generation

Our approach takes as input an already scheduled SCoP and targets loops that are still sequential. The steps are: (1) performing loop fission on sequential loops, (2) annotating groups of consecutive sequential loops with the `ordered` and `nowait` clauses, (3) performing loop fusion between parallel and ordered sequential loops and (4) performing loop blocking. Note that our method also requires the scheduling policy of `for` loop iterations to be set to static.

3.1 Sequential loop fission

A given loop may contain multiple statements. Interlacing iterations of loops using the `ordered` and `nowait` requires separate loops. To maximize the potential for interlacing, we propose to perform loop fission on such loops. However, loop fission can not be performed on each and every statement of such loops: loop fission changes the scheduling. Groups of statements with loop-carried dependencies must belong to the same loop while loop fission may be applied on groups of statements with loop-independent dependencies.

Since we take as input a SCoP which should have already been optimized for locality, we do not want to reorder the statements from the input loop (apart from the reordering caused by the loop fission). Determining how to perform loop fission for our pipelining method amounts to computing strongly connected components from the dependence graph: each strongly connected component may belong to a standalone loop.

```
1  for (int i = 2; i < N; ++i) {
2    a[i] = h[i - 1] + R[i]; // S1
3    b[i] = a[i - 1] + a[i]; // S2
4    c[i] = b[i - 1] + b[i]; // S3
5    d[i] = c[i - 1] + c[i]; // S4
6    e[i] = d[i - 2] + d[i - 1]; // S5
7    f[i] = e[i - 2] + e[i - 1]; // S6
8    g[i] = f[i] + X[i]; // S7
9    h[i] = g[i] + Y[i]; // S8
10 }
11 for (int i = 2; i < N; ++i) {
12   u[i] = v[i - 1] + d[i]; // S9
13   v[i] = u[i] + Z[i]; // S10
14 }
```

Figure 2. Loop fission performed on Van Dongen

For example, in Figure 2, statements `S1` to `S8` are grouped in the same loop while statements `S9` and `S10` are placed in another loop. There is a loop-carried dependence between statements `S1` and `S8`: `S1`, `S8` and all statements in between belong to the same loop. In the same vein, `S9` and `S10` must be placed together but can be separated from statements `S1` to `S8` since there is no loop-carried dependence between the two groups.

3.2 Relaxed conditions on the `nowait` clause

According to the OpenMP specification, the `nowait` clause may be safely used on a `for` construct which is followed by another `for` construct if the following conditions are met: (1) both iteration domain sizes are equal, (2) *chunk sizes* are equal or not specified, (3) both `for` loops are bound to the same parallel region, (4) the loops are not associated with a SIMD construct.

Previous work [15] explains how to generate code using the `nowait` clause for consecutive parallel `for` loops. This method can only be applied when the dependencies between loops link the same logical iterations. Using the `ordered` clause allows us to relax the restrictions. Once a given thread executes an iteration i_n of a loop, all previous iterations i_m ($m < n$) have been executed. In this case, the allowed dependencies between loops include more than dependencies between identical logical iterations.

Let $S1$ and $S2$ be two statements scheduled in separate consecutive `ordered for` loops over dimensions d_{S1} and d_{S2}. Given the dependencies $\delta_{S1,S2}$ between $S1$ and $S2$, ensuring that the `nowait` clause can be used in conjunction with the `ordered` clause amounts to verifying that:

$$\delta_{S1,S2} \setminus \mathcal{P}_{S1,S2} \equiv \emptyset$$

where:

$$\mathcal{P}_{S1,S2}(\vec{p}) = \left\{ \vec{\imath}_{S1} \rightarrow \vec{\imath}_{S2} \;\middle|\; \left(d_{S1} \leq d_{S2} \right) \right\}$$

For nested loops, the dependencies between inner loops can be projected over d_{S1} and d_{S2}.

3.3 Annotations

3.3.1 Annotating sequential loops

To annotate sequential loops with the `ordered` and `nowait` clauses, the method from [15] can be used with the following modifications: (1) all sequential loops are annotated with `#pragma omp for ordered`, (2) the bodies of `ordered` loops are enclosed in `#pragma omp ordered` regions, (3) the validity of the `nowait` clause is determined as described in Section 3.2.

The teaser example of Fig. 1 gives an example of such a generated loop nest.

3.3.2 Extension to parallel loops and cleanup

Section 3.3.1 explains how to add the `nowait` clause on loops that precede another loop of the same kind: either a parallel loop preceding another parallel loop or a sequential loop preceding another sequential loop. Determining whether a parallel loop preceding a sequential loop — or vice versa — can be annotated with the `nowait` clause amounts to selecting the appropriate set of requirements: if the considered loop is a parallel loop, the stricter conditions on the use of the `nowait` clause apply, otherwise, the relaxed conditions are sufficient.

Finally, any `ordered` loop without a `nowait` clause that is neither preceded by another `ordered` loop nor precedes another `ordered` loop should be reverted to an annotation-free loop enclosed in a `#pragma omp single` region.

3.4 Manual synchronization of blocks

An alternative to generating code with `#pragma omp ordered` and `nowait` annotations is to block loops over a given block size, fuse the resulting loops over the blocking dimension, distribute the iterations, with a `chunk_size` of 1, over this dimension and manually synchronize threads using OpenMP locks as shown in the example in Figure 3.

The pipelining occurs as follows: (1) for each thread, each stage of the pipeline is associated with a lock (thus, $n \times m$ locks are required for n threads and m stages), (2) for each stage i, thread t attempts to own lock$[t\%n][i]$ at stage entry and releases lock$[(t+1)\%n][i]$ at stage exit (hence, each thread will wait for its predecessor to complete a given stage of the pipeline), (3) except for the first thread, all locks are locked at the beginning of the parallel region.

Additionnal code must be generated at the start and the end of the `parallel` region to allocate, initialize, destroy and free the locks. It must be noted that we set the scheduling policy to `static` and the `chunk_size` to 1: otherwise pipelining opportunities may be lost as a given thread may execute two or more subsequent blocks of a given stage. The `block_size` value in the example in Figure 3 approaches the `chunk_size` which would be chosen by OpenMP implementations for the code example shown in Figure 1b.

```
1  omp_lock_t** locks;
2  #pragma omp parallel
3  {
4    const size_t num_threads = (size_t) omp_get_num_threads();
5    const size_t block_size = (N/num_threads)+1;
6    const size_t block_count = ((N+block_size-1)/block_size);
7
8    /* Code to allocate, initialize and set the locks. */
9    /* Wait for the lock setup completion. */
10   #pragma omp barrier
11
12   #pragma omp for schedule(static, 1)
13   for (size_t block = 0; block < block_count; ++block) {
14     /* Compute local loop bounds. */
15     const size_t start = 1 + block * block_size;
16     const size_t end = MIN(start + block_size, N);
17     /* Compute self and next thread indexes. */
18     const size_t self = block % num_threads;
19     const size_t next = (block + 1) % num_threads;
20
21     omp_set_lock(&locks[self][0]);
22     for (size_t i = start; i < end; ++i) {
23       A[i] = f(A[i], A[i-1]);
24     }
25     omp_unset_lock(&locks[next][0]);
26
27     /* Other stages of the pipeline */
28
29     omp_set_lock(&locks[self][5]);
30     for (size_t i = start; i < end; ++i) {
31       F[i] = f(E[i], F[i-1]);
32     }
33     omp_unset_lock(&locks[next][5]);
34   }
35   /* Code to destroy and free locks */
36 }
```

Figure 3. Manual synchronization of blocks

4 Experimental Results and Discussion

Our experiments were conducted on an Intel Xeon E5-2620v3 @ 2.40GHz (6 cores, 12 threads) running Linux 5.3.7. The benchmarks were compiled with options `"-O3 -march=native -fopenmp"` using gcc 9.2.0 and clang 9.0.0. Linux FIFO scheduling was enabled and process priority was set to 75.

Benchmarks `van_dongen` (Fig. 6a) and `wdf` (Fig. 6b) come from Fimmel and Müller [7] who derive these examples from Van Dongen et al. [18] and Fettweis [6]. The third benchmark (Fig. 6c) is a code example which contains both parallel loops and sequential loops. The last benchmark (Fig. 6d) is based on our teaser example where all statements execute a 1 nanosecond nanosleep and then return the sum of the two operands. The aim is to simulate long sequential computations.

We used Pluto [2] version 0.11.4 and `autoPar` from the Rose [12] Compiler version 0.9.12.0 on our code samples and can confirm these tools do not expose parallelism on our benchmarks (apart from the parallel loop in our third benchmark).

We generated multiple versions of the benchmarks: the original sequential code, the sequential code naively annotated with the ordered clause, the code annotated with our approach, with and without blocking.

A tempting alternative to our approach is to use OpenMP `task` constructs. Indeed, the `task` construct (in conjunction with the depend annotation) allows finer control over execution order. We manually wrote two versions: a naive version where a task is created for each and every statement and another version where tasks are created for each loop body after we applied loop fission as described in Section 3.1.

Figure 4 presents results observed on the code compiled with gcc with options `-O3 -march=native -fopenmp`. Speedup values of 0 are displayed for `task+fission` when no corresponding version was generated for a given benchmark because it would not differ from the `task` version. Except for the `mix` example (which can be partially parallelized), Pluto does not change the code: the observed speedups are very close to the speedups observed over the original sequential version.

As expected, merely adding the `ordered` clause without further modification and then executing the loops in an

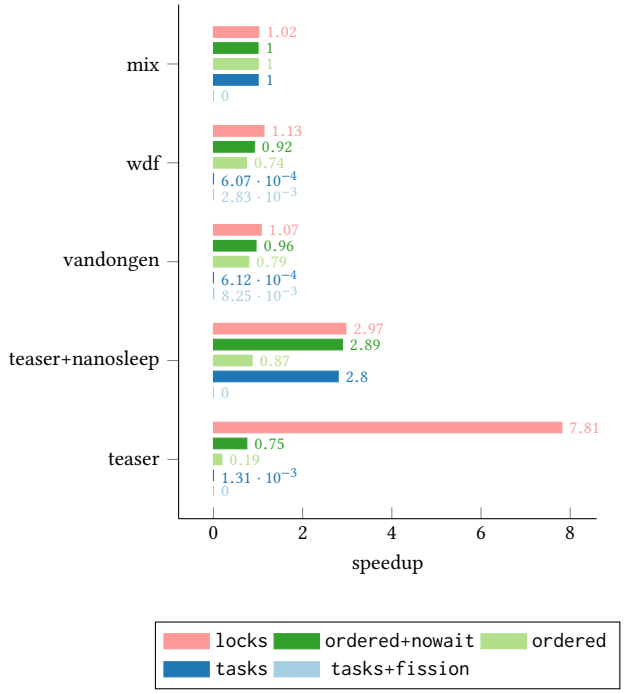

Figure 4. Speedups or slowdowns over sequential version
$N = 100,000$ for `teaser`,
$N = 1,000,000$ `wdf`, `van_dongen`
$N = 3000$ and $M = 3000$ for `mix`
gcc 9.2.0, options: `-O3 -march=native -fopenmp`

Figure 5. Speedups or slowdowns over sequential version
$N = 100,000$ for `teaser`, `wdf`, `van_dongen`
$N = 3000$ and $M = 3000$ for `mix`
clang 9.0.0, options: `-O3 -march=native -fopenmp`

OpenMP parallel region does not increase performance: all iterations are executed in the exact same order as the sequential order with the added overhead of the `ordered` synchronizations.

Both our approach and the tasks versions allow our `teaser` example to significantly outperform the sequential version. Moreover, it can be noticed that `task` creation competes with our approach in situations where the number of tasks remains contained. However, the introduced overhead can tremendously impede the resulting program as the number of tasks grows larger. This can clearly be seen as the task creation after we apply our loop fission — albeit still too costly — is much lighter than with the greedy task creation approach. The overhead comes from the growing amount of tasks to create and the dependencies which must be checked at runtime. The drastic impact of this overhead can also be observed when comparing the performance of `tasks` on our teaser with and without the call to `nanosleep` which was introduced to mimic long compute times.

Manual blocking and locks outperform the `ordered+nowait` versions and even allows our teaser example to exhibit speedups without the call to `nanosleep`. The results show the speedups for block sizes similar to the default chunk size for non blocked versions. We have observed even greater speedups with different block sizes (for instance a speedup of 9.6 with a block size of 2048 on our teaser code). The values were guessed based on the input code and cache size of our test environment. Further work is required to determine how to find optimal block sizes for a given pipeline and environment.

The measurements were also conducted over code compiled with clang and the results are presented in Figure 5. In this case, we limited N to 100, 000 for `wdf` and `van_dongen`: it can already be seen that even for a smaller N value, the `task` versions are worse with clang/libomp than with gcc/libgomp. For $N = 1,000,000$ we stopped the experiment after more than 10 minutes spent in the task+fission code (as opposed to approximately 1 minute for the gcc/libgomp version).

5 Related Work

Extensive work has been done on software pipelining [1, 5, 7–9, 14]. It used to be applied to architectures with specific features, while we propose to pipeline loops on multithreading environments. Software pipelining of a loop body can still be used on a given core, in the cases where a loop body still contains more than one statement after our algorithm has been applied.

There has also been a lot of research focusing on low-level pipelined code generation, usually based on the concept of process network [4, 17, 19]. The main concerns of these papers are the characterization of stream types and sizes and

the efficient placement of tasks on a controlled sized static hardware. These issues are not relevant in a multithreaded general purpose environment like OpenMP.

Raman et al. [13] construct pipelines on a similar idea based on strongly connected components however their work does not target SCoPs.

Previous work generate other forms of parallelism using OpenMP constructs and clauses and the polyhedral model. Sbîrlea et al. [16] parallelize doaccross loops using the OpenMP `ordered` construct and clause. However, this transformation is based on specialized input: the input programs are expressed in a Data-Flow Graph Language and provide extended information on dependencies. Chatarasi et al. [3] propose a framework which can handle OpenMP `tasks` and `ordered` constructs. This framework expects input programs where the parallelism has already been explicitly exposed with OpenMP annotations and checks the correctness of the input or optimizes the suggested program. Pop et al. [11] propose OpenStream as an extension of OpenMP. The corresponding compiler can generate code which can exploit pipeline parallelism. However, it requires help from the programmer to annotate and expose parallelism in the input code. Compared to these approaches, our method could be applied to sequential loops in simple annotation-free SCoPs.

6 Conclusion

We presented in this paper a technique to interlace and execute in parallel the iterations of otherwise sequential loops. Using the OpenMP `ordered` and `nowait` clauses and thanks to dependence analysis through the polyhedral model, we are able to expose pipelined multithreading in programs which current state-of-the-art automatic polyhedral parallelizers fail to parallelize.

Our study shows that programs with multiple consecutive sequential loops and with long enough sequential iterations can benefit from pipelined multithreading. Indeed, pipelined multithreading introduces control overhead which can only be mitigated with long enough compute times.

Integrating our algorithm in an automatic parallelizer is a work in progress. The synchronizations may impose a great toll on the resulting program. We plan to investigate whether specifying chunk sizes or tiling multithreaded pipelined loops may reduce this synchronization overhead.

Our experiments led us to manually write alternative versions using OpenMP `tasks`. Although our observations showed that such constructs were not appropriate in some of our tests cases, it can be noticed that — in ideal conditions — using tasks was as efficient as our method. Future work should investigate the requirements for task efficiency and how to automatically extract task parallelism from simple SCoPs.

References

[1] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. 1995. Software pipelining. *ACM Computing Surveys (CSUR)* 27, 3 (1995), 367–432.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices* 43, 6 (2008), 101–113. https://doi.org/10.1145/1379022.1375595

[3] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 213–226.

[4] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. 2008. High-level synthesis of loops using the polyhedral model. In *High-level synthesis*. Springer, 215–230.

[5] Paul Feautrier. 1994. Fine-grain scheduling under resource constraints. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1–15.

[6] Alfred Fettweis. 1986. Wave digital filters: Theory and practice. *Proc. IEEE* 74, 2 (1986), 270–327.

[7] Dirk Fimmel and Jan Müller. 2001. Optimal software pipelining under resource constraints. *International Journal of Foundations of Computer Science* 12, 06 (2001), 697–718.

[8] Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM Sigplan Notices*, Vol. 23. ACM, 318–328.

[9] Qi Ning and Guang R Gao. 1993. A novel framework of register allocation for software pipelining. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 29–42.

[10] OpenMP Architecture Review Board. [n. d.]. OpenMP API version 4.5. http://openmp.org

[11] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 53.

[12] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[13] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J Bridges, and David I August. 2008. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 114–123.

[14] B Ramakrishna Rau. 1994. Iterative module scheduling: An algorithm for software pipelining loops. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 63–74.

[15] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. 2017. Lifting Barriers Using Parallel Polyhedral Regions. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 338–347.

[16] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2015. Polyhedral optimizations for a data-flow graph language. In *Languages and Compilers for Parallel Computing*. Springer, 57–72.

[17] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. 2004. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*. IEEE Computer Society, 10340.

[18] Vincent H Van Dongen, Guang R Gao, and Qi Ning. 1992. A polynomial time method for optimal software pipelining. In *Parallel Processing: CONPAR 92—VAPP V*. Springer, 613–624.

[19] Sven Verdoolaege. 2013. Polyhedral process networks. In *Handbook of Signal Processing Systems*. Springer, 1335–1375.

A Benchmarks

```
for (int i = 2; i < N; ++i) {
  a[i] = h[i - 1] + R[i]; // S1
  b[i] = a[i - 1] + a[i]; // S2
  c[i] = b[i - 1] + b[i]; // S3
  d[i] = c[i - 1] + c[i]; // S4
  e[i] = d[i - 2] + d[i - 1]; // S5
  f[i] = e[i - 2] + e[i - 1]; // S6
  g[i] = f[i] + X[i]; // S7
  h[i] = g[i] + Y[i]; // S8
  u[i] = v[i - 1] + d[i]; // S9
  v[i] = u[i] + Z[i]; // S10
}
```

(a) Van Dongen: original code

```
for (int i = 1; i < N; ++i) {
  a[i] = X[i] + e[i - 1]; // S1
  b[i] = a[i] - g[i - 1]; // S2
  c[i] = b[i] + e[i]; // S3
  d[i] = gamma1 * b[i]; // S4
  e[i] = d[i] + e[i - 1]; // S5
  f[i] = gamma2 * b[i]; // S6
  g[i] = f[i] + g[i - 1]; // S7
  Y[i] = c[i] - g[i]; // S8
}
```

(b) WDF: original code

```
for (int i = 1; i < N; ++i)
  for (int j = 0; j < M; ++j)
    for (int k = 0; k < M; ++k)
      C[i] = C[i] + B[j] + A[k] + 1.;
for (int i = 1; i < N; ++i) {
  D[i] = D[i - 1] * C[i];
  E[i] = E[i - 1] * D[i];
}
```

(c) Mix: original code

```
for (int i = 1; i < N; ++i)
  A[i] = f(A[i], A[i-1]);
for (int i = 1; i < N; ++i)
  B[i] = f(A[i], B[i-1]);
for (int i = 1; i < N; ++i)
  C[i] = f(B[i], C[i-1]);
for (int i = 1; i < N; ++i)
  D[i] = f(C[i], D[i-1]);
for (int i = 1; i < N; ++i)
  E[i] = f(D[i], E[i-1]);
for (int i = 1; i < N; ++i)
  F[i] = f(E[i], F[i-1]);
```

(d) Teaser: original code

Figure 6. Benchmarks