Pipelined Multithreading Generation in a Polyhedral Compiler

January 22nd 2020, IMPACT'20, HiPEAC, Bologna, Italy

Harenome Ranaivoarivony-Razanajato, Cédric Bastoul, Vincent Loechner

University of Strasbourg and Inria Nancy Grand Est

Team **ICPS** | Scientific and Parallel Computing
University of **Strasbourg**

Motivating Example

```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program

Motivating Example

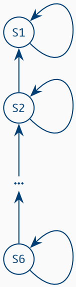
```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program



(b) Dependency Graph

Motivating Example

S1(1), thread 1

```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program

(b) Pipelined Execution

Motivating Example

```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program

(b) Pipelined Execution

Motivating Example

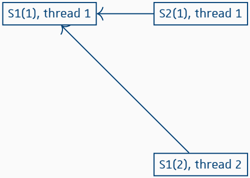
```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program



(b) Pipelined Execution

Motivating Example

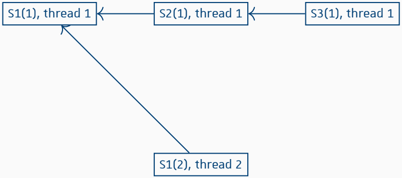
```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program



(b) Pipelined Execution

Motivating Example

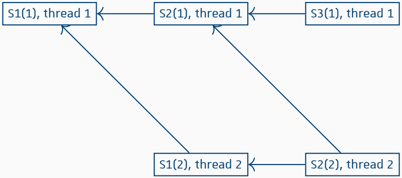
```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program



(b) Pipelined Execution

Motivating Example

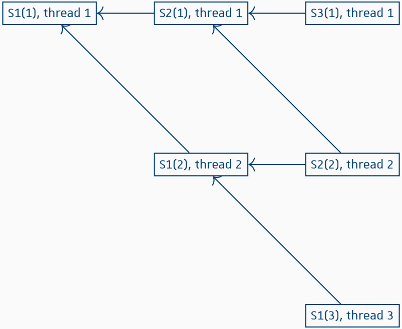
```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program



(b) Pipelined Execution

Motivating Example

```c
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program

```c
#pragma omp parallel
{
    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      A[i] = f1(A[i], A[i - 1]); // S1
    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      B[i] = f2(A[i], B[i - 1]); // S2

    /* ... */

    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      F[i] = f6(E[i], F[i - 1]); // S6
}
```

(b) Pipelined OpenMP target program

Motivating Example

```
for (int i = 1; i < N; ++i)
  A[i] = f1(A[i], A[i - 1]); // S1

for (int i = 1; i < N; ++i)
  B[i] = f2(A[i], B[i - 1]); // S2

/* ... */

for (int i = 1; i < N; ++i)
  F[i] = f6(E[i], F[i - 1]); // S6
```

(a) Sequential Program

```
#pragma omp parallel
{
    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      A[i] = f1(A[i], A[i - 1]); // S1
    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      B[i] = f2(A[i], B[i - 1]); // S2

    /* ... */

    #pragma omp for schedule(static) ordered nowait
    for (int i = 1; i < N; ++i)
      #pragma omp ordered
      F[i] = f6(E[i], F[i - 1]); // S6
}
```

(b) Pipelined OpenMP target program

Speedup: 2.89

6 stages on an Intel Xeon E5-2620v3 @ 2.40 GHz, with $N = 100,000$

Goals

- Identifying software pipelines in a polyhedral compiler
- Generate pipelined multithreading using OpenMP

OpenMP

- `#pragma` based API for shared memory parallelism
- Worksharing constructs
  - `#pragma omp for`
  - `#pragma omp task`

OpenMP

- `#pragma` based API for shared memory parallelism
- Worksharing constructs
  - `#pragma omp for`
  - `#pragma omp task`
- Synchronization
  - `#pragma omp barrier`: explicit synchronization barrier
  - `omp_set_lock()` and `omp_unset_lock()`: explicit lock mechanism

OpenMP

- `#pragma` based API for shared memory parallelism
- Worksharing constructs
  - `#pragma omp for`
  - `#pragma omp task`
- Synchronization
  - `#pragma omp barrier`: explicit synchronization barrier
  - `omp_set_lock()` and `omp_unset_lock()`: explicit lock mechanism
- Clauses
  - `nowait` clause on worksharing constructs: omit the implicit barrier at the end of a worksharing construct
  - `ordered` clause on worksharing constructs: sequentialize a region

Sequential Loop Fission

- Goal: maximize the number of pipeline stages
- Dependence analysis: identify Strongly Connected Components

Sequential Loop Fission

```
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) Original loop body

```
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
}
for (int i = 2; i < N; ++i) {
  u[i] = v[i - 1] + d[i]; // S9
  v[i] = u[i] + Z[i]; // S10
}
```

(b) Fission of Strongly Connected Components

Conditions on the nowait clause for parallel

The safe use of the `nowait` clause between two **parallel** loops requires that there are no dependencies between the loops or that:

- the sizes of the iteration domains are equal
- the chunk size is either the same or not specified
- both loops are bound to the same parallel region
- none of the loops is associated with a SIMD construct
- the second loop depends only on the same logical iteration of the first loop

Relaxed conditions on the nowait clause for ordered loops

The safe use of the `nowait` clause between two **ordered** loops requires that there are no dependencies between the loops or that:

- the sizes of the iteration domains are equal
- the chunk size is either the same or not specified
- both loops are bound to the same parallel region
- none of the loops is associated with a SIMD construct
- ~~the second loop depends only on the same logical iteration of the first loop~~

Relaxed conditions on the nowait clause for ordered loops

The safe use of the `nowait` clause between two **ordered** loops requires that there are no dependencies between the loops or that:

- the sizes of the iteration domains are equal
- the chunk size is either the same or not specified
- both loops are bound to the same parallel region
- none of the loops is associated with a SIMD construct
- ~~the second loop depends only on the same logical iteration of the first loop~~
- the second loop depends on the same logical iteration **or previous logical iterations** of the first loop

Relaxed conditions on the nowait clause for ordered loops

```c
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i = 0; i < N; ++i)
    A[i] = f1(A[i]);
  #pragma omp for
  for (int i = 0; i < N; ++i)
    B[i] = f2(B[i], A[i]);
}
```

(a) Parallel `for` and `nowait`

```c
#pragma omp parallel
{
  #pragma omp for ordered nowait
  for (int i = 0; i < N; ++i)
    #pragma omp ordered
    A[i] = f1(A[i]);
  #pragma omp for ordered
  for (int i = 0; i < N; ++i)
    #pragma omp ordered
    B[i] = f2(B[i], A[i-1]);
}
```

(b) Ordered `for` and `nowait`

Annotations

- Annotate sequential loops with `#pragma omp for ordered`
- Enclose sequential loop bodies in `#pragma omp ordered` regions
- Annotate loops with `nowait` where possible
- Optimize by reverting `ordered` loops without `nowait` clauses to `#pragma omp single` regions

Explicit synchronization

- Loop blocking and loop fusion
- `#pragma omp for schedule(static, 1)` on the blocking loop
- `omp_set_lock()` and `omp_unset_lock()` before and after each loop of the pipeline
- up to $n \times m$ locks required for m pipeline stages over n threads

Explicit synchronization

```
#pragma omp parallel
{
    #pragma omp for ordered nowait
    for (size_t i = 1; i < N; ++i)
      #pragma omp ordered
      A[i] = f(A[i], A[i - 1]);

    /* Other stages */

    #pragma omp for ordered
    for (size_t i = 1; i < N; ++i)
      #pragma omp ordered
      F[i] = f(E[i], F[i - 1]);
}
```

(a) Original program

```
omp_lock_t** locks;
#pragma omp parallel
{
  /* Choose num_threads, block_size, block_count. */
  /* Allocate, initialize and set the locks. */
  #pragma omp for schedule(static, 1)
  for (size_t block = 0; block < block_count; ++block)
    /* Local loop bounds and indexes. */
    const size_t start = 1 + block * block_size;
    const size_t end = MIN(start + block_size, N);
    const size_t self = block % num_threads;
    const size_t next = (block + 1) % num_threads;

    omp_set_lock(&locks[self][0]);
    for (size_t i = start; i < end; ++i)
      A[i] = f(A[i], A[i-1]);
    omp_unset_lock(&locks[next][0]);
    /* Other stages of the pipeline */
    omp_set_lock(&locks[self][5]);
    for (size_t i = start; i < end; ++i)
      F[i] = f(E[i], F[i-1]);
    omp_unset_lock(&locks[next][5]);
  }
  /* Destroy and free locks. */
}
```

(b) Pipelined OpenMP target program

Future Work

Experimental Setup

- Tested on an Intel Xeon E5-2620v3 @ 2.40 GHz, linux 5.3.7
- Code compiled using `gcc 9.2.1` and `clang 9.0.0` with options `-O3 -march=native -fopenmp`
- FIFO scheduling enabled and process priority set to 75

Benchmarks

benchmark	parallel loops	stages
teaser	0	5
van_dongen[1]	0	2
wdf[2]	0	2
mix	1	2

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

[2] (Alfred Fettweis. "Wave digital filters: Theory and practice". In: *Proceedings of the IEEE* 74.2 [1986], pp. 270–327)

Results – gcc/libgomp

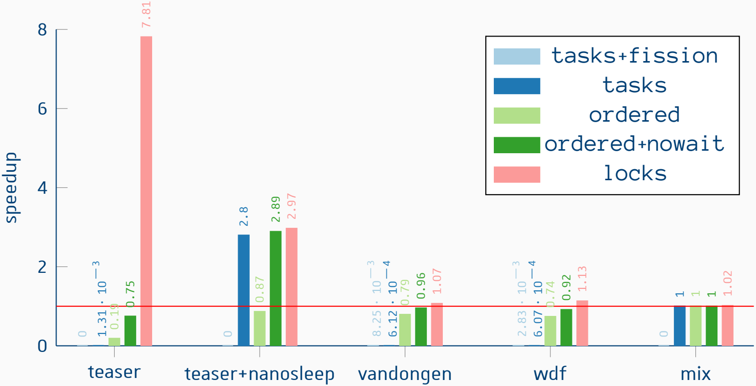


Figure 7: Speedups or slowdowns over sequential version

Results – clang/libomp

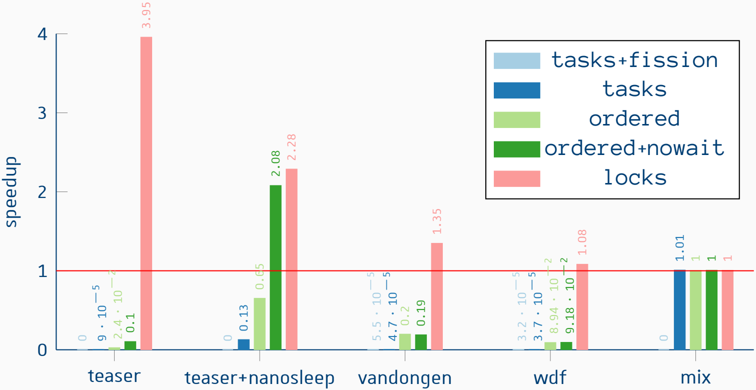


Figure 8: Speedups or slowdowns over sequential version

Future Work

Contributions and Future Work

- Contributions:
  - Identifying software pipelines in a polyhedral compiler
  - Generating pipelined multithreading

Contributions and Future Work

- Contributions:
  - Identifying software pipelines in a polyhedral compiler
  - Generating pipelined multithreading
- Future work:
  - Integration in an automatic parallelizer
  - Investigate methods to determine optimal block sizes

Appendix

References i

[1] Alfred Fettweis. “Wave digital filters: Theory and practice”. In: *Proceedings of the IEEE* 74.2 (1986), pp. 270–327.

[2] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. “Pipelined Multithreading Generation in a Polyhedral Compiler”. In: *IMPACT 2020, 10th International Workshop on Polyhedral Compilation Techniques*. Bologna, Italy, Jan. 2020.

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