Superloop Scheduling: Loop Optimization via Direct Statement Instance Reordering

Cedric Bastoul
University of Strasbourg
France

Alain Ketterlin
University of Strasbourg and Inria
France

Vincent Loechner
University of Strasbourg and Inria
France

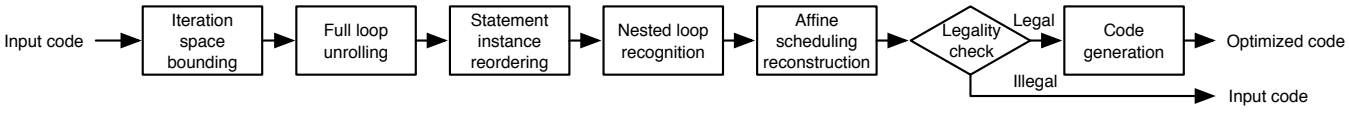

Figure 1. Superloop Scheduling Processing Flow

Abstract

Loop optimization in the polyhedral model is supported by the expressiveness of affine scheduling functions to model statement iteration ordering. Discovering the best scheduling remains a grand challenge reinforced by the need to build *affine* functions. Automatic techniques based on solving systems of affine constraints and/or composing affine scheduling primitives are limited either by the affine modeling or by their primitive set.

In this paper we propose a radically different approach to affine scheduling construction called superloop scheduling. We leverage basic-block-level instruction reordering techniques and the capacity to agglomerate statement instances into "super" loops offered by modern trace analysis tools. This approach enables deepest possible reasoning about instruction ordering and a global approach to loop optimization, e.g., where tiling and intra-tile optimization is considered along with other transformations.

1 Introduction

Superloop scheduling enables a number of breakthroughs:

- we demonstrate that loop-level transformations, even applied to parametric loops, can be derived by reasoning on a subset of their instruction instances;
- we show how affine scheduling can be driven and built by non-affine optimization techniques and abstractions, where regularity comes from target architecture properties and degrees of freedom rather than as a limitation of the scheduling algorithm;
- we reconcile loop-level and instruction-level optimization, enabling advanced approaches such as superword level parallelism [7] to impact whole loops;
- we present an approach where tiling is enabled within the global optimization space rather than being addressed separately, e.g., after tilable loops have been extracted as in Pluto-like techniques [3].

IMPACT 2023, January 16, 2023 , Toulouse, France

.

2 Superloop Scheduling

Superloop scheduling is a 7-step loop optimization process outlined in Fig. 1 and illustrated using matrix multiplication in Fig. 2. It takes as input a static-control code and provides as output an optimized version if a semantically correct scheduling is found. The successive steps are:

(a) Iteration space bounding limits the number of iterations and replaces the parameters with known values. Values should be chosen to expose optimization opportunities, keep next steps tractable and enable later parameter recovery.

(b) Full loop unrolling transforms the code to a sequence of statement instances as shown for matrix multiplication (with all parameters set to 2) in Fig. 2b.

(c) Statement instance reordering transforms the unrolled code via direct reasoning and manipulation of the statement instances. Trivial parallel block extraction and internal reordering to enable vectorization is possible and would result in the new ordering in Fig. 2c for our example. Elaborate basic block level techniques such as superword level parallelisation [7] may be leveraged as well to enable possibly unprecedented loop vectorization opportunities. Reordering policies should include constraints or mechanisms to favor some regularity when possible.

(d) Nested loop recognition (NLR) recovers loops in a fast and incremental way [6]. It takes as input a trace comprised of tagged vectors of numbers, as shown in the comments in Fig. 2b and Fig. 2c. On detecting a linear progression over blocks of input vectors, it builds a loop with a single instance of the block, where raw numbers have been replaced with affine functions of the loop counter. Blocks of such loops are then themselves subjected to linear interpolation when possible. NLR is thus able to recover arbitrarily deep and/or complex affine loop nests from their traces, an example of which is shown in Fig. 2d for the trace in Fig. 2c.

(e) Affine scheduling reconstruction builds an affine scheduling expression from the loop structure and the mapping information present in NLR's output. It may also attempt to recover parameters, e.g. through pattern-matching.

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++) {
    C[i][j] = 0.;                    // S0
    for (k = 0; k < P; k++)
      C[i][j] += A[i][k] * B[k][j]; // S1
  }
```

(a) Input Code

```
C[0][0] = 0;                    // S0  0  0
C[0][0] += A[0][0] * B[0][0]; // S1  0  0  0
C[0][0] += A[0][1] * B[1][0]; // S1  0  0  1
C[0][1] = 0;                    // S0  0  1
C[0][1] += A[0][0] * B[0][1]; // S1  0  1  0
C[0][1] += A[0][1] * B[1][1]; // S1  0  1  1
C[1][0] = 0;                    // S0  1  0
C[1][0] += A[1][0] * B[0][0]; // S1  1  0  0
C[1][0] += A[1][1] * B[1][0]; // S1  1  0  1
C[1][1] = 0;                    // S0  1  1
C[1][1] += A[1][0] * B[0][1]; // S1  1  1  0
C[1][1] += A[1][1] * B[1][1]; // S1  1  1  1
```

(b) Full Unrolling (`M = N = P = 2`)

```
C[0][0] = 0;                    // S0  0  0
C[0][1] = 0;                    // S0  0  1
C[0][0] += A[0][0] * B[0][0]; // S1  0  0  0
C[0][1] += A[0][0] * B[0][1]; // S1  0  1  0
C[0][0] += A[0][1] * B[1][0]; // S1  0  0  1
C[0][1] += A[0][1] * B[1][1]; // S1  0  1  1
C[1][0] = 0;                    // S0  1  0
C[1][1] = 0;                    // S0  1  1
C[1][0] += A[1][0] * B[0][0]; // S1  1  0  0
C[1][1] += A[1][0] * B[0][1]; // S1  1  1  0
C[1][0] += A[1][1] * B[1][0]; // S1  1  0  1
C[1][1] += A[1][1] * B[1][1]; // S1  1  1  1
```

(c) Optimized Statement Instance Order

```
for i0 = 0 to 1
  for i1 = 0 to 1
    val S0 , 1*i0 , 1*i1
  for i1 = 0 to 1
    for i2 = 0 to 1
      val S1 , 1*i0 , 1*i2 , 1*i1
```

(d) Loops Recovered from Trace by NLR

```
S0(i, j)    = (i, 0, j)
S1(i, j, k) = (i, 1, k, j)
```

(e) Extracted Scheduling

```
#pragma omp parallel for private(j, k)
for (i = 0; i < M; i++) {
  #pragma vector always
  for (j = 0; j < N; j++)
    C[i][j] = 0.;                    // S0
  for (k = 0; k < P; k++)
    #pragma vector always
    for (j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j]; // S1
}
```

(g) Generated Code

Figure 2. Superloop Scheduling of Matrix Multiplication

In our example, the scheduling extracted from NLR's output is shown in Fig. 2e. In this case the loop structure contributes the second scheduling dimension (separated `i1` loops) while the mapping expressions contribute the others.

(f) Legality check verifies the scheduling correctness on the original input code using, e.g., the Candl tool[1], and may assess its properties such as parallel and vector dimensions.

[1] https://github.com/periscop/candl

(g) Code generation produces the optimized code that implements the scheduling using, e.g., CLooG [2]. The result for our example is shown in Fig. 2g, demonstrating that optimization of parametric loops can be done via reordering of a subset of their statement instances. Notably, the optimization is different than both Feautrier [4] (which favors `k, i, j` version with internal parallelism) and Pluto without tiling [3] (which splits `S0` and `S1` in two loop nests to enable `i, k, j` order for `S1`, or uses less CPU-efficient `i, j, k`).

3 Related Work

We may identify two families of techniques for affine scheduling construction. The first family computes the scheduling by solving systems of affine constraints. This approach has been proposed by Feautrier in his seminal work [4] and has set the ground for many later techniques, notably the Pluto algorithm designed by Bondhugula et al. to extract outermost parallelism, data locality and tilable loops [3], and a number of variants that differ in the way the affine constraint system is built. The second family builds the affine scheduling by composition of basic primitives, as suggested by Kelly and Pugh [5] and improved by many authors, e.g., Baghdadi et al. who propose a rich scheduling language for the Tiramisu framework [1]. Differently, we present the seed for a new family that builds affine scheduling from finest-grain statement instance reordering and loop reconstruction.

4 Conclusion

In this paper we present a radically new scheduling canvas. We do not expose all details but rather present and demonstrate a new route to loop optimization for the community to explore. We strongly believe it has a significant potential, in particular to exploit custom vector instructions.

References

[1] R. Baghdadi, J. Ray, M. Ben Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM Intl. Symp. on Code Generation and Optimization, CGO, Washington, USA.*

[2] C. Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques.* Juan-les-Pins, France, 7–16.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08).* Tucson, AZ, USA.

[4] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[5] W. Kelly and W. Pugh. 1993. *A framework for unifying reordering transformations.* Technical Report UMIACS-TR-92-126.1. University of Maryland Institute for Advanced Computer Studies.

[6] A. Ketterlin and P. Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *Sixth Intl. Symp. on Code Generation and Optimization, CGO, Boston, USA.*

[7] C. Mendis and S. Amarasinghe. 2018. goSLP: globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 110:1–110:28.