When Polyhedral Transformations Meet SIMD Code Generation

Martin Kong
Ohio State University
kongm@cse.ohio-state.edu

Richard Veras
Carnegie Mellon University
rveras@cmu.edu

Kevin Stock
Ohio State University
stockk@cse.ohio-state.edu

Franz Franchetti
Carnegie Mellon University
franzf@ece.cmu.edu

Louis-Noël Pouchet
University of California Los Angeles
pouchet@cs.ucla.edu

P. Sadayappan
Ohio State University
saday@cse.ohio-state.edu

Abstract

Data locality and parallelism are critical optimization objectives for performance on modern multi-core machines. Both coarse-grain parallelism (e.g., multi-core) and fine-grain parallelism (e.g., vector SIMD) must be effectively exploited, but despite decades of progress at both ends, current compiler optimization schemes that attempt to address data locality and both kinds of parallelism often fail at one of the three objectives.

We address this problem by proposing a 3-step framework, which aims for integrated data locality, multi-core parallelism and SIMD execution of programs. We define the concept of *vectorizable codelets*, with properties tailored to achieve effective SIMD code generation for the codelets. We leverage the power of a modern high-level transformation framework to restructure a program to expose good ISA-independent vectorizable codelets, exploiting multi-dimensional data reuse. Then, we generate ISA-specific customized code for the codelets, using a collection of lower-level SIMD-focused optimizations.

We demonstrate our approach on a collection of numerical kernels that we automatically tile, parallelize and vectorize, exhibiting significant performance improvements over existing compilers.

Categories and Subject Descriptors D 3.4 [*Programming languages*]: Processor — Compilers; Optimization

General Terms Algorithms; Performance

Keywords Compiler Optimization; Loop Transformations; Affine Scheduling; Program synthesis; Autotuning

1. Introduction

The increasing width of vector-SIMD instruction sets (e.g., 128 bits in SSE, to 256 bits in AVX, to 512 bits in LRBni) accentuates the importance of effective SIMD vectorization. However, despite the significant advances in compiler algorithms [15, 26, 28, 30, 31, 43] over the last decade, the performance of code vectorized by current compilers is often far below a processor's peak.

A combination of factors must be addressed to achieve very high performance with multi-core vector-SIMD architectures: (1) effective reuse of data from cache — the aggregate bandwidth to main memory on multi-core processors in words/second is far lower than the cumulative peak flop/second; (2) exploitation of SIMD parallelism on contiguously located data; and (3) minimization of load, store and shuffle operations per vector arithmetic operation.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

While hand tuned library kernels such as GotoBLAS address all the above factors to achieve over 95% of machine peak for specific computations, no vectorizing compiler today comes anywhere close. Recent advances in polyhedral compiler optimization [5, 11] have resulted in effective approaches to tiling for cache locality, even for imperfectly nested loops. However, while significant performance improvement over untiled code has been demonstrated, the absolute achieved performance is still very far from machine peak. A significant challenge arises from the fact that polyhedral compiler transformations to produce tiled code generally require auxiliary transformations like loop skewing, causing much more complex array indexing and loop bound expressions than the original code. The resulting complex code structure often leads to ineffective vectorization by even sophisticated vectorizing compilers such as Intel ICC. Further, locality enhancing transformations, e.g. loop fusion, can result in dependences in the innermost loops, inhibiting vectorization.

In this paper, we present a novel multi-stage approach to overcome the above challenges to effective vectorization of imperfectly nested loops. First, data locality at the cache level is addressed by generating tiled code that operates on data footprints smaller than L1 cache. Code within L1-resident tiles is then analyzed to find affine transformations that maximize stride-0 and stride-1 references in parallel loops as well as minimization of unaligned loads and stores. This results in the decomposition of a L1-resident tile into *codelets*. Finally, a specialized back-end codelet optimizer addresses optimization of register load/store/shuffle operations, maximization of aligned versus unaligned load/stores, as well as register level reuse. Target-specific intrinsics code is generated for the codelet by the back-end optimizer.

This paper makes the following contributions. (1) It presents a novel formalization for an affine loop transformation algorithm that is driven by the characteristics of codelets.(2) Unlike most previous approaches that focus on a single loop of a loop-nest for vectorization, it develops an approach that performs integrated analysis over a multi-dimensional iteration space for optimizing the vector-SIMD code. (3) It represents the first demonstration of how high-level polyhedral transformation technology can be effectively integrated with back-end codelet optimization technology through a model-driven approach, with significant performance improvement over production and research compilers.

This paper is organized as follows. Sec. 2 gives a high-level overview of our approach, and Sec. 3 defines the concept of vectorizable codelets, the interface between a high-level loop transformation engine and a low-level ISA-specific SIMD code generator. Sec. 4 details our new framework for automatically extracting maximal codelets. Sec. 5 details the framework for ISA-specific SIMD code generation. Experimental results are presented in Sec. 6, and related work is discussed in Sec. 7.

2. Overview

The key to our approach to vectorization is a separation of tasks between a high-level program restructuring stage and an ISA-specific SIMD code generation stage. High-level program transformation frameworks such as the polyhedral/affine framework excel at finding loop transformation sequences to achieve high-level, cost-model-based objectives. Examples such as maximizing data locality [7], maximizing parallelism (fine-grain or coarse-grain [11, 16, 29]) and possibly balancing the two [33, 34] illustrate the ability of the polyhedral framework to aggressively restructure programs. On the other hand, SIMD-specific concerns such as ISA-specific vector instruction selection, scheduling, and register promotion have been implemented using frameworks that depart significantly from classical loop transformation engines [15, 28, 30].

In this work we formalize the interface between these two optimization stages — the vectorizable codelet: a tile of code with specific, SIMD-friendly properties. We develop a novel and complete system to generate and optimize vectorizable codelets.

High-level overview of algorithm The data locality optimization algorithm in a system like Pluto [11] is geared towards exposing parallelism and tilability for the outer-most loops first, through aggressive loop fusion. This is critical for achieving good L1 cache data locality through tiling, as well as coarse-grained multi-core parallelization. However, for L1-resident code, those objectives are often detrimental: for effective exploitation of vector-SIMD ISA's, we need inner-most parallel loops, and the addressing of stride and alignment constraints. Other work has looked at the impact of unroll-and-jam for SIMD vectorization [12], but they do not consider the problem of restructuring programs to maximize the applicability of unroll-and-jam, nor any access stride constraint.

In this work, we develop an integrated system aimed at reconciling the different considerations in optimizing for coarse-grain parallelism and cache data locality versus effective SIMD vectorization. Our end-to-end strategy is as follows.

1. Transform the program for cache data locality (see Sec. 4.2), using: (1) a model-driven loop transformation algorithm for maximizing data locality and tilability [11]; (2) a parametric tiling method to tile all tilable loops found [5]; and (3) an algorithm to separate partial tiles and full tiles [1].
2. For each full tile, transform it to expose maximal vectorizable codelets (see Sec. 4), using: (1) a new polyhedral loop transformation algorithm for codelet extraction; (2) unroll-and-jam along permutable loops to increase data reuse potential in the codelet(s); and (3) an algorithm for abstract SIMD vectorization, addressing hardware alignment issues.
3. For each vectorizable codelet, generate high-performance ISA-specific SIMD code (see Sec. 5), using a special *codelet compiler* akin to FFTW's `genfft` or the SPIRAL system. Further, use instruction statistics or runtime measurements to autotune the codelets if alternative implementations exist.

3. Vectorizable Codelets

the original program, which satisfy some specific properties that enable effective SIMD code to be synthesized for it using an ISA-specific back-end code generator. In order to illustrate the properties, we proceed backwards by first describing the process of abstract SIMD vectorization on a code satisfying those properties.

3.1 Abstract SIMD Vectorization

We use the term abstract SIMD vectorization for the generation of ISA-independent instructions for a vectorizable innermost loop. The properties to be satisfied by this inner-most loop are summarized in the following definition:

DEFINITION 1 (Line codelet). *A line codelet is an affine innermost loop with constant trip count such that:*

1. *there is no loop-carried dependence,*
2. *all array references are stride-1 (fastest varying array dimension increments by one w.r.t. the innermost loop) or stride-0 (innermost loop absent from array index expressions),*
3. *unaligned load/stores are avoided whenever possible.*

To illustrate the process of abstract SIMD vectorization, Fig. 1 shows a code example which satisfies the above requirements. It is the transformed output of a sample full tile by our codelet exposure algorithm, to be discussed later. Fig. 3 shows its abstract vector variant, after performing a basic unroll-and-jam as shown in Fig. 2. The abstract vector code generation involves peeling the vectorizable loop with scalar prologue code so the first store (and all subsequent ones) will be aligned modulo V, the vector length. Similarly, an epilogue with scalar code is peeled off. All arithmetic operators are replaced by equivalent vector operations, stride-0 references are replicated in a vector by "splatting", stride-1 references are replaced by vload/vstore calls, and the vectorized loop has its stride set to the vector length V. The arrays in this example, and the remainder of this paper, are assumed to be padded to make each row a multiple of the vector length V.

```
for (i = lbi; i < ubi; ++i)
  for (j = lbj; j < ubj; ++j) {
R:  A[i-1][j] = B[i-1][j];
S:  B[i][j] = C[i]*A[i-1][j];
  }
```

Figure 1. After transfo. for codelet exposure

```
for (i = lbi; i < ubi; i += 2)
  for (j = lbj; j < ubj; ++j) {
R:  A[i-1][j] = B[i-1][j];
S:  B[i][j] = C[i]*A[i-1][j];
R:  A[i][j] = B[i][j];
S:  B[i+1][j] = C[i+1]*A[i][j];
  }
```

Figure 2. After unroll-and-jam 2×1

```
for (i = lbi; i < ubi; i += 2)
  // Prolog: peel for alignment.
  lbj2 = &(A[i-1][j]) % V
  for (j = lbj; j < lbj + lbj2; ++j) {
     A[i-1][j] = B[i-1][j];
     B[i][j] = C[i]*A[i-1][j];
  }
  // Body: codelet (abstract vectorization)
  ubj2 = (lbj2 - ubj) % V
  for (j = lbj2; j < ubj - ubj2; j += V) {
     vstore(A[i-1][j], vload(B[i-1][j]));
     vstore(B[i][j], vmul(vsplat(C[i]),
                          vload(A[i-1][j])));
     vstore(A[i][j], vload(B[i][j]));
     vstore(B[i+1][j], vmul(vsplat(C[i+1]),
                          vload(A[i][j])));
  }
  // Epilog: peel for multiple of V.
  for (j = ubj - ubj2; j < ubj; ++j) {
     A[i-1][j] = B[i-1][j];
     B[i][j] = C[i]*A[i-1][j];
  }
```

Figure 3. After abstract SIMD vect.

Our objective in this work is to automatically transform full tiles into code fragments satisfying Definition 1 (i.e., Fig. 1), possibly interleaved with other code fragments not matching those requirements. Each vectorizable inner-loop is a *vectorizable codelet*. We show in Sec. 4 how to leverage the restructuring power of the polyhedral transformation framework to automatically reshape the loop nests to satisfy the requirements when possible.

3.2 Vectorizable Codelet Extraction

Efficient vector code generation requires to exploit data reuse potential. Specifically, given a program segment that is L1-resident, we should maximize the potential for register reuse while increasing the number of vector computations in the codelet. To do so we use register tiling, or unroll-and-jam, which amounts to unrolling outer loops and fusing the unrolled statements together.

Our framework is well suited to perform this optimization: the polyhedral framework can restructure a loop nest to maximize the number of *permutable loops*, along which unroll-and-jam can be applied. Arbitrarily complex sequences of fusion/distribution/skewing/shifting may be required to make unroll-and-jam possible, and we show in Sec. 4 how to automatically generate effective sequences for a given program to maximizes the applicability of unroll-and-jam. As a result of our framework, the innermost vectorizable loops (the line codelets) will contain an increased number

of vector operations and operands for the ISA-specific synthesizer to optimize. Definition 2 lists the various optimization objectives that drive the high-level transformation engine to ensure the creation of good (i.e. large) candidate vector codelets.

DEFINITION 2 (Maximal Codelet Extraction). *Given a program P, maximal codelets are obtained by applying polyhedral loop transformations on P to obtain a program P′ such that:*

1. *the number of innermost loop(s) iterations which are parallel is maximized;*
2. *the number of references in P′ which are not stride-0 or stride-1 is minimized;*
3. *the number of permutable dimensions is maximized.*
4. *the number of unaligned stores is minimized;*
5. *the number of unaligned loads is minimized;*

In this work, we apply maximal codelet extraction on each full tile in the program. We then apply unroll-and-jam on all permutable loops, apply abstract SIMD vectorization on all candidate codelets, before synthesizing ISA-specific SIMD code for each of them.

4. Framework for Codelet Extraction

We now present our high-level framework to automatically transform the program to extract candidate vectorizable codelets.

4.1 Polyhedral Framework

To meet our goal of effectively transforming a program (region) to expose maximal codelets, we use a powerful and expressive mathematical framework for program representation: the polyhedral model. In the present work, we leverage recent developments on expressing various optimization constraints (e.g. loop permutability, data reuse, parallelism, etc.) into a single optimization problem that can then be optimally solved using Integer Linear Programming [34, 39]. We first review the basis of program representation and optimization in the polyhedral model, using the same notation as in [34].

Background and program representation The polyhedral framework is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [16, 19], roughly defined as a set of consecutive statements such that all loop bounds and conditional expressions are affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution (whose values are unknown at compile-time). Numerous scientific kernels exhibit those properties; they are found frequently in image processing filters (such as medical imaging algorithms) and dense linear algebra operations.

Unlike the abstract syntax trees used as internal representation in traditional compilers, polyhedral compiler frameworks internally represent imperfectly nested loops and their data dependence information as a collection of parametric polyhedra. Programs in the polyhedral model are represented using four mathematical structures: each statement has an *iteration domain*, each memory reference is described by an affine *access function*, data dependences are represented using *dependence polyhedra* and finally the program transformation to be applied is represented using a *scheduling function*.

Iteration domains For all textual statements in the program (e.g. R and S in Fig. 1) the set of its dynamic instances is described by a set of affine inequalities. When the statement is enclosed by one or more loops, all iterations of the loops are captured in the iteration domain of the statement. Each executed instance of the statement corresponds to a point in the iteration domain; the coordinate of this point is defined by the value of the surrounding loop iterators when the statement instance is executed. Parametric polyhedra are used to capture loop bounds whose values are invariant through loop execution but unknown at compilation time. These polyhedra use *parameters* in the inequalities defining their faces, and are a natural extension to standard polyhedra. For instance, in Fig. 1, the iteration domain of R is:

$$\mathcal{D}_R = \{(i,j) \in \mathbb{Z}^2 \mid \texttt{lbi} \leq i < \text{ubi} \wedge \texttt{lbj} \leq j < \text{ubj}\}.$$

We denote by $\vec{x}_R$ the vector of the surrounding loop iterators; for R it is (i,j) and takes values in $\mathcal{D}_R$.

Access functions They represent the location of the data accessed by the statement, in the form of an affine function of the iteration vector. For each point in $\mathcal{D}_R$, the access function $F_A^R(\vec{x_R})$ returns the coordinate of the cell of A accessed by this instance of the memory reference. We restrict ourselves to subscripts of the form of affine expressions which may depend on surrounding loop counters and global parameters. For instance, the subscript function for the read reference `A[i-1][j]` of statement R is $F_R^A(i,j) = (i-1, j)$.

Data dependences The sets of statement instances between which there is a producer-consumer relationship are modeled as equalities and inequalities in a *dependence polyhedron*. Dependences are defined at the reference granularity of an array element. If two instances $\vec{x}_R$ and $\vec{x}_S$ refer to the same array cell and one of these references is a write, a data dependence exists between them. To respect program semantics, the producer instance must be executed before the consumer instance. Given two statements R and S, a dependence polyhedron, written $\mathcal{D}_{R,S}$, contains all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$. Multiple dependence polyhedra may be required to capture all dependent instances, at least one for each pair of array references accessing an array (scalars being a particular case of array). It is possible to have several dependence polyhedra per pair of textual statements, as they may contain multiple array references.

Program transformation Program transformations in the polyhedral model are represented by a scheduling function. This function is used to reorder the points in the iteration domain, and the corresponding source code can be generated using polyhedral code generation [6]. A scheduling function captures, in a single step, what may typically correspond to a sequence of several tens of basic loop transformations [19]. It takes the form of a scheduling matrix Θ^R, whose coefficients drive the program restructuring to be performed.

DEFINITION 3 (Affine multi-dimensional schedule). *Given a statement R, an affine schedule Θ^R of dimension m is an affine form of the d loop iterators (denoted $\vec{x}_R$) and the p global parameters (denoted $\vec{n}$). $\Theta^R \in \mathbb{Z}^{m \times (d+p+1)}$ can be written as:*

$$\Theta^S(\vec{x}_R) = \begin{pmatrix} \theta_{1,1} & \cdots & \theta_{1,d+p+1} \\ \vdots & & \vdots \\ \theta_{m,1} & \cdots & \theta_{m,d+p+1} \end{pmatrix} \cdot \begin{pmatrix} \vec{x_R} \\ \vec{n} \\ 1 \end{pmatrix}$$

The scheduling function Θ^R maps each point in $\mathcal{D}_R$ to a multidimensional time-stamp (a vector) of dimension m. In the transformed code, the instances of R defined in $\mathcal{D}_R$ will be executed in lexicographic order of their associated time-stamp. Multidimensional timestamps can be seen as logical clocks: the first dimension similar to days (most significant), the next one to hours (less significant), etc.

The optimization of a full program requires a collection of affine schedules, one for each syntactic program statement. Even seemingly non-linear transformations like loop tiling (also known as blocking) and unrolling can be modeled [19].

4.2 Program Transformation for L1-resident Blocking

Tiling for locality involves grouping points in an iteration space into smaller blocks (tiles) [43], enabling reuse in multiple directions when the block fits in a faster level of the memory hierarchy (registers, L1, or L2 cache). Tiling for coarse-grained parallelism partitions the iteration space into tiles that may be executed concurrently on different processors with a reduced frequency and volume of inter-processor communication. A tile is atomically executed on a processor, with communication required only before and after execution. The tiling hyperplane method [11] is an effective approach for tiling of imperfectly nested affine loops. However, it can only generate tiled code for tile sizes that are fixed at compile-time. Alternatively, *parametric tiling* [5, 22], used in the present work, enables run-time selection of tile sizes. It has the advantage that tile sizes can be adapted to the problem size and machine used without recompiling the code.

The first stage of our optimization flow is as follows. Given an input program, we first apply a model-driven optimization geared towards maximizing the applicability of tiling [11]. Then parallel parametric tiling is applied, followed by full-tile separation. The process of separating full tiles leads to rectangular blocks of code whose loop bounds are functions of the tile sizes (which are run-time parameters). Each full tile is a potential candidate for effective SIMD execution using the codelet extraction and synthesis approach detailed below. We note that the codelet extraction and synthesis techniques are not limited to tilable programs. However, in our experiments in Sec. 6 we use benchmarks which can be processed by parallel parametric tiling.

The downside of parametric tiling is the generation of very complex loop bounds for the loops iterating on the various blocks (inter-tile loops), typically involving complex compositions of *ceil*(), *floor*() and *round*() expressions. Current production compilers often fail to successfully analyze the dependences in such loops, and therefore are unable to automatically generate the best SIMD code for the full tiles. This leads to significant performance loss in the generated program. However, using the information extracted by the polyhedral compilation framework, full tiles can be manipulated, transformed and vectorized using our proposed approach.

4.3 Convex Set of Semantics-Preserving Transformations

A key reason for the power and effectiveness of the polyhedral transformation framework is the ability to formulate, with a single set of (affine) constraints, a set of *semantics-preserving (affine) program transformations* [34, 39]. An optimization problem whose solutions are subject to these constraints will necessarily lead to a semantics-preserving program transformation.

To build such a set of constraints, the reasoning is as follows. First, we observe that for all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$, the dependence is strongly satisfied if $\Theta(\vec{x}_R) < \Theta(\vec{x}_S)$, that is if the producer is scheduled before the consumer. As Θ is in practice a multi-dimensional function, we can more precisely state that the dependence is strongly satisfied if $\Theta(\vec{x}_R) \prec \Theta(\vec{x}_S)$, where $\prec$ is the lexicographic precedence operator. This can be rewritten as $\Theta(\vec{x}_S) - \Theta(\vec{x}_R) \succ \vec{0}$. Alternatively, given Θ_p, row p of Θ, we have $\forall p,\ \Theta_p(\vec{x}_S) - \Theta(\vec{x}_R) \geq \delta_p$ with $\delta_p \in \{0,1\}$. We note that once a dependence has been strongly satisfied at a dimension d, then it does not contribute to the semantics constraint and so we have instead $\forall p > d,\ \Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -\infty$ (that is, a "void" constraint with no impact on the solution space).

The constraints for semantics preservation are summarized in Def. 4, and are the starting basis for our new optimization algorithm. For each row p of the scheduling matrix Θ and each dependence polyhedron $\mathcal{D}_{R,S}$, we associate a Boolean decision variable $\delta_p^{\mathcal{D}_{R,S}}$; these decision variables are used to encode semantics-preserving properties of a schedule so that for each pair of instances in dependence the source instance will be scheduled necessarily before the target instance. We bound the coefficients of Θ to be in some arbitrary range, so that we can find some arbitrarily large $K \in \mathbb{Z}$ such that the form $(K.\vec{n}+K)$ is an upper bound of the schedule latency [34], therefore $\Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -(K.\vec{n}+K)$ is equivalent to $\Theta_p(\vec{x}_S) - \Theta_p(\vec{x}_R) \geq -\infty$.

DEFINITION 4 (Semantics-preserving affine schedules). *Given a set of affine schedules $\Theta^R, \Theta^S \ldots$ of dimension m, the program semantics is preserved if the three following conditions hold:*

$$(i) \qquad \forall \mathcal{D}_{R,S},\ \forall p,\ \delta_p^{\mathcal{D}_{R,S}} \in \{0,1\}$$

$$(ii) \qquad \forall \mathcal{D}_{R,S},\ \sum_{p=1}^{m} \delta_p^{\mathcal{D}_{R,S}} = 1 \tag{1}$$

$$(iii) \qquad \forall \mathcal{D}_{R,S},\ \forall p \in \{1,\ldots,m\},\ \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, \tag{2}$$

$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \geq -\sum_{k=1}^{p-1} \delta_k^{\mathcal{D}_{R,S}}.(K.\vec{n}+K) + \delta_p^{\mathcal{D}_{R,S}}$$

As Feautrier proposed, the affine form of the Farkas lemma is used to build the set of all non-negative functions over a polyhedron [16], a straightforward process as Eq. 2 represents a non-negative function and $\mathcal{D}_{R,S}$ is a polyhedron. As a result, a convex set of constraints linking the various $\theta_{i,j}$ coefficients and the δ variables is obtained, such that each point in this space is a semantics-preserving schedule [34].

4.4 Codelet-Specific Cost Functions

We now present the set of additional constraints we use to formulate our scheduling optimization problem. The complete scheduling algorithm works in two steps: first we find a schedule for the program that maximizes parallelism, the number of stride-0/1 references, and permutability. Then we apply a second algorithm for minimizing the number of unaligned load/stores on the transformed program.

4.4.1 Additional Properties on the Schedule

In the present work, we impose specific properties on Θ: $\theta_{i,j} \in \mathbb{N}$. Enforcing non-negative coefficients greatly simplifies the process of finding good schedules. Large coefficients for Θ can lead to complex code being generated, containing modulo operations [6], and small values for the coefficients are usually preferred [32]. Our ILP formulation attempts to minimize the value of coefficients, therefore making them as close to 0 as possible when $\theta_{i,j} \geq 0$.

In addition, we use the so-called 2d+1 schedule form [19], which forces the scheduling function to be an interleaving of linear and constant dimensions. The number of rows of Θ^S is set to $2d+1$ if S is surrounded by d loops in the original sub-program, and every odd row of Θ^S is made constant (that is, $\theta_{i,j} = 0$ for all j except $j = d+n+1$). For instance, in this 2d+1 form, the original schedule of Fig 1 is $\Theta^R : (0,i,0,j,0)$ and $\Theta^S : (0,i,0,j,1)$. Those are computed simply by building the AST of the sub-program with loops and statements as nodes, and edges between a parent and its children in the AST. Edges are labeled by the syntactic order of appearance of the children, and to compute the original schedule one simply takes the path from the root to the statement, collecting the node labels and the edge labels along this path.

While not technically required for this work, the 2d+1 form provides a standardized notation for the schedules, ensures that at least one schedule exists in the considered search space (the identity schedule, as shown above), and tends to simplify the generated loop bounds when implementing full distribution through the scalar dimensions.

4.4.2 Minimizing Dependence Distance

The first objective we encode is designed to find solutions with good data locality. It also helps with other objectives. The minimization finds a function that bounds the dependence distance [11], for each non-constant scheduling level (the even rows). This function is a form of the program parameters $\vec{n}$, and requires additional constraints on Θ: all coefficients associated with the parameters $\vec{n}$ are set to 0. This only removes parametric shifts, and is therefore not a limiting factor unless there are dependences of parametric distances in the program, a rare case. This bounding function is defined as follows, for each row k of the schedule:

$$\mathbf{u}_k.\vec{n} + w_k \geq \Theta^S(\vec{x}_S) - \Theta^R(\vec{x}_R) \qquad \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S} \qquad (3)$$
$$\mathbf{u}_k \in \mathbb{N}^p, w_k \in \mathbb{N}$$

Intuitively, the value of $\mathbf{u}_k.\vec{n} + w_k$ is an indicator of an upper bound on the distance between dependent iterations. To minimize the distance between dependent iterations, we minimize the value of $\mathbf{u}_k$ and w_k. If, after minimization, $\mathbf{u}_k.\vec{n} + w_k = 0$, then the dependent iterations are scheduled at exactly the same time for this loop level, i.e., the loop is parallel.

4.4.3 Maximizing Fine-grain Parallelism

Maximizing fine-grain parallelism requires that no dependences in the generated code are carried by the innermost loop(s). Feautrier addressed the problem of maximizing fine-grain parallelism through an aggressive strategy of strongly satisfying all dependences as early as possible [16]. We propose a different form: minimization of the number of dependences to be solved by the schedule dimension $2d$ (e.g., the one corresponding to the inner-most loop), together with the objective of minimizing the dependence distance at that level. We use the following optimization objective:

$$\min \sum_{\mathcal{D}_{R,S}} \delta_{2d}^{\mathcal{D}_{R,S}} \qquad (4)$$

Indeed, if this sum is equal to 0, and if $\mathbf{u}_{2d}.\vec{n} + w_{2d}$ (from the minimization of the dependence distance at that level) is also equal to 0, then the inner-most loop is parallel as no dependence is carried by this loop. Minimizing both objectives at the same time ensures that we discover inner-most parallel loops whenever possible.

4.4.4 Maximizing Stride-0/1 References

We propose a framework to embed directly, as constraints on the coefficients of Θ, the maximization of the number of stride-0/1 references. It is a complex task to model such a constraint as a convex optimization problem, so that we can use standard solvers to find coefficients of Θ. The reader is referred to a technical report [27] for detailed examples.

Access functions after scheduling In a nutshell, we aim for sufficient conditions for an array reference to be stride-0/1, when the innermost loop is parallel. These conditions come from properties of the polyhedral code generation process which builds the expressions connecting the input iterator symbols (e.g., `i`, `j`) with the new loop iterators created (e.g., `t2`, `t4`). For instance, $\Theta^R = (0, i+j, 0, j, 0)$ is a valid schedule for R in Fig. 1. Two loops are created after code generation, and their schedule is given by the even dimensions of Θ: $t2 = i + j$, $t4 = j$. In the access `A[i-1][j]`, after loop transformation, `i` and `j` are replaced by expressions in the form of the new loop iterators `t2`, `t4`. This process is done by constructing a system of equalities between the `tX` variables and the original iterators, based on the schedule, and performing Gaussian elimination. Equalities with one original iterator on the left hand side are then obtained. With the above example, we would get `A[t2-t4-1][t4]`, which is not a stride-1 reference along `t4`, the inner loop.

Our approach to model sufficient conditions for the stride of a reference to be 0 or 1 is to reason about the system of equalities built to create the new references after loop transformations. Specifically, we want to ensure that the innermost loop iterator (to be vectorized) appears only in the right-most index function (the Fastest Varying Dimension, FVD) of an array reference, or does not appear at all in the reference. Otherwise, the reference is not stride-0/1.

As an illustration of a sufficient condition for stride-1: assuming the innermost non-constant schedule dimension p represents a parallel loop, then for a reference to be stride-1 it is enough to have (1) the schedule coefficients for rows less than p corresponding to the iterators present in non-FVD functions are non-zero at least once; (2) for at least one of the iterators present in the FVD function, the schedule coefficients for rows less than p corresponding to this iterator are all zero.

Convex encoding of access stride properties To encode this kind of property, we resort to two complementary structures in the ILP. First, a new set of Boolean variables $\gamma_{i,j}$ is introduced, to help us model when a schedule coefficient $\theta_{i,j}$ is 0 or not. There are as many $\gamma_{i,j}$ variables as there are $\theta_{i,j}$ variables, and for each of them we set $\gamma_{i,j} \leq \theta_{i,j}$. Intuitively, we sum these γ variables to count how many schedule coefficients are non-zero. For $\gamma_{i,j}$ to accurately capture when a schedule coefficient is non-zero, we maximize $\sum \gamma_{i,j}^R$ so that $\gamma_{i,j}^R$ is set to 1 as soon as $\theta_{i,j}^R$ is greater or equal to 1.[1]

Second, we use an auxiliary representation of the access functions, using a normalized matrix G^F for an access function F. Intuitively, it is used to represent which iterators appear in the non-FVD index functions, and which appear in the FVD with a coefficient of 1. G^F has two rows: all iterators appearing in the non-FVD are represented in the first row, and in the second row iterators appearing in the FVD with a coefficient of 1 are represented. Precisely, for the access function matrix F of dimension $l \times d + n + 1$, G^F is constructed as follows. (1) G^F has 2 rows and d columns. (2) $\forall i \in \{1..l-1\}$, $\forall j \in \{1..d\}$, if $F_{i,j} \neq 0$ then $G^F_{1,j} = 1$, $G^F_{1,j} = 0$ otherwise; (3) $\forall j \in \{1..d\}$, if $F_{l,j} = 1$ then $G^F_{2,j} = 1$, $G^F_{2,j} = 0$ otherwise.

This leads to the following definition [27], wherein besides the γ variables we also define new Boolean variables μ_F^j and ν_F^j (one pair per loop surrounding the reference F), and σ_1^F and σ_2^F to model this seemingly non-convex problem in a convex fashion.

DEFINITION 5 (Constraint for Stride-0/1). *Given a statement R surrounded by d loops; a legal schedule Θ^R where all loop-carried dependences have been strongly satisfied before level $2d$; a memory reference F_R^A for an array A of dimension l with stride-0/1 potential; μ_F^j, ν_F^j, σ_1^F and σ_2^F, a collection of Boolean decision variables; and G^F the normalized matrix for F. When given semantics-preserving Θ^R with parallel inner-dimension, the following constraints*

$$\left(\sum_{k=1}^{d-1} g_{1,j}.\gamma_{2k,j} \geq g_{1,j}.\mu_F^j\right) \wedge \left(\sum_{k=1}^{d-1} g_{2,j}.\gamma_{2k,j} \leq (d-1).g_{2,j}.\nu_F^j\right) \quad \forall j \in \{1..d\}$$
$$\left(\sum_{j=1}^{d} g_{1,j}.\mu_F^j \geq \sum_{j=1}^{d} g_{1,j}.\sigma_1^F\right) \wedge \left(\sum_{j=1}^{d} g_{2,j}.\nu_F^j \leq \sum_{j=1}^{d} g_{2,j} - \sigma_2^F\right)$$

are verified if $\sigma_1^F = 1$ and $\sigma_2^F = 1$ (stride-1), or if $\sigma_1^F = 1$ and $\sigma_2^F = 0$ (stride-0). Otherwise it is not stride-1/0.

The optimization objective is formulated as follows.

[1] As we perform lexicographic optimization, we can place this maximization as the very last optimization objective to not disrupt the actual performance-driven optimization objectives.

DEFINITION 6 (Maximization of Stride-0/1 References). *The number of non-stride-0/1 references is minimized by encoding, for each memory reference F in the program, the constraints of Def. 5, and optimizing the following objectives:*

$$\max \sum_F \sigma_1^F,\ \max \sum_F \sigma_2^F$$

Example To illustrate the above definitions, Fig. 4 shows how they are applied on a simple program with a single reference. The stride optimization objectives yield the optimum value of $\sigma_1 = 1$ and $\sigma_2 = 1$. $\sigma_1 = 1$ implies that $\mu^2 = 1$, which in turn makes $\gamma_{2,2} = 1$. Thus, in the final schedule $\theta_{2,2} \geq 1$. $\sigma_2 = 1$ forces $\nu^1 = 0$, which propagates and sets $\gamma_{2,1} = 0$. In the final schedule, $\theta_{2,1} = 0$ and $\theta_{2,2} \geq 1$. This corresponds to making the loop `j` the outer loop.

`for (i = 0; i < M; ++i)`
`  for (j = 0; j < N; ++j)`
`  A[j][i] = 0;`
	$G = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

$0 \cdot \mu^1 + 1 \cdot \mu^2 \geq (0+1) \cdot \sigma_1$	$1 \cdot \nu^1 + 0 \cdot \nu^2 \leq 1 - \sigma_2$
$0 \cdot \gamma_{2,1} \geq 0 \cdot \mu^1$ $1 \cdot \gamma_{2,2} \geq 1 \cdot \mu^2$	$1 \cdot \gamma_{2,1} \leq 1 \cdot \nu^1$ $0 \cdot \gamma_{2,2} \leq 0 \cdot \nu^2$

Figure 4. Example of stride-0/1 constraints

4.4.5 Maximizing Permutability

Our approach to maximize permutability is to add constraints to capture level-wide permutability [11], and maximizing the number of such dimensions. We first recall the definition of permutable dimensions.

DEFINITION 7 (Permutability condition). *Given two statements R,S. Given the conditions for semantics-preservation as stated by Def. 4. Their schedule dimensions are permutable up to dimension k if in addition:*

$$\forall \mathcal{D}_{R,S},\ \forall p \in \{1,\ldots,k\},\ \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S},$$
$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \geq \delta_p^{\mathcal{D}_{R,S}} \tag{5}$$

Enforcing this constraint on all schedule dimensions may lead to an empty solution set, if not all dimensions are permutable. So, we need to relax this criterion in order to capture *when* it is true, while not removing schedules from the solution set. We start by introducing Boolean decision variables, $\rho_p^{\mathcal{D}_{R,S}}$, i.e., one ρ variable per δ variable, and set $\delta_p^{\mathcal{D}_{R,S}} \geq \rho_p^{\mathcal{D}_{R,S}}$. These variables, when set to 1, will make Eq. (2) equal to Eq. (5), and will not impact the solution set when set to 0. This is achieved by replacing Eq. (2) by:

$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \geq -\sum_{k=1}^{p-1} (\delta_k^{\mathcal{D}_{R,S}} - \rho_k^{\mathcal{D}_{R,S}}).(K.\vec{n} + K) + \delta_p^{\mathcal{D}_{R,S}}$$

When a dependence has been previously satisfied ($\delta_p^{\mathcal{D}_{R,S}} = 1$ for some p), instead of nullifying the legality constraint at subsequent levels, we will attempt to maximize the number of cases where Eq. (5) holds, with the following optimization objective for a given dimension p:

$$\max \sum_{\mathcal{D}_{R,S}} \rho_p^{\mathcal{D}_{R,S}}$$

We note that as δ and ρ variables are connected by an inequality, maximizing ρ variables implicitly forces the strong satisfaction of dependences as early as possible.

4.5 Putting It All Together

The above constraints are all embedded into a single optimization problem, containing numerous Boolean decision variables, as well as the schedule coefficients for all statements. We combine all our optimization objectives into a single optimization problem that is solved by Integer Linear Programming, by finding the lexicographically smallest point in the space. The order of the optimization objectives determines which objective will be maximally/minimally solved first, with succeeding objectives being optimally solved given the max/min solution of the previous objectives. Our combined problem P is:

$$P: \quad \left\{ \min \sum_{\mathcal{D}_{R,S}} \delta_{2d}^{\mathcal{D}_{R,S}},\ \min \mathbf{u}_{2d}.\vec{n} + w_{2d},\ \max \sum_F \sigma_1^F,\ \max \sum_F \sigma_2^F, \right.$$
$$\left. \forall k\ \max \sum_{\mathcal{D}_{R,S}} \rho_k^{\mathcal{D}_{R,S}},\ \forall k \neq 2d\ \min \mathbf{u}_k.\vec{n} + w_k,\ \max \sum_{i,j} \gamma_{i,j} \right\}$$

There may still be multiple solutions to this optimization problem, each satisfying the criteria for extraction of maximal codelets as per Def. 2. While technically enumerating all optimal solutions should be performed for best performance, in practice we limit ourselves to the lexicographically smallest solution of the above problem. The resulting schedule is then applied to the program, polyhedral code generation is performed, and candidate codelets are detected on the generated code (all parallel innermost loops with only stride-0/1 references are candidate codelets). Those candidate codelets are the input to the second stage of our optimization framework, which performs retiming+skewing of statements to minimize the number of unaligned load/stores. In addition, permutable loops are detected through dependence analysis on the generated code, and are marked as candidate loops for unroll-and-jam.

4.6 Minimizing Unaligned Stores and Loads

The final stage of our method to extract valid candidate codelets for ISA-specific SIMD synthesis is to minimize the number of unaligned stores and loads in each candidate codelet found by the previous loop transformation step. Despite the fact that hardware alignment constraints are usually dealt with at a lower compilation level [15], this task should more naturally belong to the high-level transformation engine, where multidimensional loop transformations as well as precise array dataflow analysis can be performed.

To achieve this objective, we perform a combination of *statement retiming* at the codelet level, and additional loop skewing for the unroll-and-jammed loops. Intuitively, we 'shift' statements in the codelet such that (when possible) all array elements which are referenced by the vectorizable loop become aligned if the first element being referenced by the loop is aligned. Previous work by Henretty et al. [23] develops a complete framework for statement retiming so as to minimize stream alignment conflict on innermost vectorizable loops. Our candidate codelets fit their definition of vectorizable loops, and to find the relevant retiming factor for our optimization purpose, we use a slightly adapted version of that algorithm. The main difference is that we add an additional scalar quantity to the shifts found by their algorithm so that stores to different arrays need the same iteration peeling quantity to ensure alignment with the technique shown in Sec. 3.1. This is not strictly necessary, but simplifies the code generation and can reduce the number of peeled iterations in the scalar prologue/epilogue codes.

Finally, we need to take care of alignment properties in case of unroll-and-jam of the loops. For this purpose, we resort to skewing the unroll-and-jammable loops by a positive factor, so that the array index expression along the FVD is a multiple of the vector length. Such transformation is always legal, as a positive skew cannot

change the dependence direction. As an illustration, for a Jacobi-2D example the access function after parametric tiling and full tile extraction, but before retiming is of the form `A[-2t+i][-2t+j]`. Assuming $V = 4$, we apply an additional skew by 2 to unroll-and-jam along t, to get `A[-4t+i][-4t+j]`. In this manner, when unrolling by any factor along t and jamming in j, references will still be vector-aligned in j.

In summary, to minimize store/load alignment even in the presence of unroll-and-jam, for each candidate codelet we compute an ad-hoc polyhedral transformation made only of shifts and skews, and apply this transformation using a polyhedral code generator. The resulting codelet is then transformed to abstract vector code using the technique depicted in Sec. 3.1, creating valid inputs to the next optimization stage: ISA-specific SIMD synthesis of vectorizable codelets, as detailed in Sec. 5.

5. ISA-specific SIMD Generation

In this section we discuss the back-end compilation step that translates pre-vectorized code into machine-specific high-performance code. The final high-performance code is structurally similar to the popular auto-tuning package FFTW [18], the auto-tuning BLAS package ATLAS [42], and general-size auto-tuning libraries generated by SPIRAL [35, 41]: the main work is performed in small, highly optimized code blocks called codelets, which are automatically generated. A higher-level approach (the polyhedral framework to parallelization) breaks down the bigger computation into a skeleton that performs all the book-keeping and calls the codelets appropriately. The first important question is how to restructure the computation to enable the utilization of efficient codelets, which we discuss in Sec. 4. The remaining problem to solve is how to compile codelet specifications into highly efficient code, a problem similar to the role of `genfft` [17] in FFTW.

5.1 Overview

Our line codelet generator takes as input a specification of the tile code in a LISP-like intermediate representation (a one-to-one translation is systematically possible from our abstract SIMD form to this LISP-like form) that captures the necessary amount of information needed to generate highly efficient SIMD code. The representation is designed to convey everything that is known by construction (alignments, loop trip count properties, aliasing, etc.) without over-specifying the problem. Our codelet generator then performs various platform independent transformations such as common subexpression elimination (CSE), array scalarization and strength reduction. Furthermore more advances optimizations such as converting unaligned loads into aligned loads or ISA-specific strength reduction and pattern transformation are performed. Loops are unrolled in a reuse-conscious way that depends on the problem type. The output of the codelet generator is highly optimized sequential C code augmented by ISA-specific intrinsics. The code contains well-shaped simple loops, large basic blocks in single static assignment (SSA) form, easily analyzable array index expressions, and explicit SIMD vector instructions. This code is finally compiled with the target vendor compiler.

5.2 Line Codelet Specification

The input to our codelet generator is a program in a LISP-like code representation. The representation enables SIMD architecture-independent meta programs that construct architecture specific instances when instantiated. Only the information needed for performing the operation is captured in the abstraction, details regarding the hardware and the exact calling convention for the SIMD intrinsics are abstracted away using vector primitives that have a type and a length. During the code generation and optimization process the codelet generator will translate the polymorphic vector operations to their exact implementation according to the targeted platform and compiler.

The input language expresses the following operations in an ISA-independent way: 1) loads and stores that are guaranteed to be naturally aligned (e.g, in 128-bit SSE a 128-bit vector load for which the start address is 16-byte aligned), 2) vector splats and compile-time constants, 3) unaligned loads and stores and their displacement loop-carried or constant displacement modulo vector length, 4) in-register transposes across 2, 4,..., V registers (including pack/unpack and interleave/deinterleave), and 5) in-register partial and full reductions (e.g., horizontal adds or a dot-product instruction).

We show an illustrative example that is generated by the pre-vectorization stage below. The SIMD tile meta-program is represented as a tree data structure, using standard C-like commands (`decl` for variable declarations, `loop` for `for` loops, `assign` for assignments, etc.). Multi-dimensional array access is directly supported (`sv_nth2D_addr`), and the meta-program aspect can be seen in expressions like `isa.vload` and `isa.vloadu` (aligned and unaligned load instruction in the current SIMD ISA passed into the meta-program through `isa`). SSA code is supported by constructing "fresh" variables. The input language including meta-programming capabilities is extensible and new primitives needed by new types of kernels or new vector ISAs can be added easily. Our example show an editorially simplified input to the codelet generator specifying a 2D Jacobi SIMD vector line codelet. The full example contains more annotations attached to the variables.

```
Jacobi2D_tile := isa ->
  let(range0 := var.fresh_t(TInt), ...,
    vreg0 := var.fresh_t(isa.vtype),...,
    func(TInt, "jacobi2d_tile",[loop_bound_aligned_1,
      loop_bound_aligned_2, c4, c6, shift_0, shift_1, local_c8, B, A],
      decl([range0,k0,c8,vreg0,vreg1,vreg2,vreg3,...],
        chain(
          ...
          assign(vreg1,isa.vload(sv_nth2D_addr(A,add(mul(V(-2),
            c4),c6),sub(add(mul(V(-2),c4),c8),1),n,1))),
          assign(vreg2,isa.vloadu(sv_nth2D_addr(A,add(mul(V(-2),
            c4),c6),sub(sub(add(mul(V(-2),c4),c8),1),1),n,1))),
          ...
          assign(vreg0,mul(0.200000,add(add(add(add(vreg1,
            vreg2),vreg3),vreg4),vreg5))),
          ...
          assign(deref(tcast(TPtr(datatype),sv_nth2D_addr(A,add(
            add(mul(V(-2),c4),c6),sub(0,1)),add(add(mul(V(-2),c4),
            add(c8,sub(shift_0,shift_1))),sub(0,1)),n,1))),vreg6)
        ))
));
```

5.3 Code Generation and Optimization

Our codelet generator is a full-fledged highly aggressive basic block compiler with very limited support for loops. It instantiates the generic tile specification for a particular SIMD ISA and then optimizes the resulting code, and outputs a C function with compiler and architecture specific SIMD intrinsics.

Loop unrolling We aggressively unroll loops to produce large basic blocks. Modern superscalar out-of-order processors have huge instruction caches, reorder windows, store-to-load forwarding, multiple outstanding loads, and many other architectural features that are fully exercised in steady state only by basic blocks with hundreds of instructions. We convert the massive basic blocks into single static assignment (SSA) form to enable better register allocation through the compiler. In addition, unrolling a time loop by a factor of two allows transparently moving between an input vector and a temporary space, avoiding extra copy operations.

Loop pipelining and blocking The best loop unrolling strategy depends on the type of code that is underlying the tile codelet. Our codelet generator at present supports three patterns:

First, stencil and filter-type data flow graphs require software pipelining and register rotation. This is either supported in hardware

(as on Itanium) or must be done in software through unrolling and multi-versioning of variables. Proper loop pipelining achieves the optimal steady state load/store to computation ratio (e.g., one load, one store and 3 flops per loop iteration for 1D Jacobi). Jacobi or wavelet filters are examples of this pattern.

Second, dense linear algebra-type data flow graphs have high arithmetic intensity. This enables register-level double buffering: on half of the register file a register tile computation is performed while on the other half of the register file in parallel the results from the previous register tile are stored and the data for the new register tile is loaded. The L1 tile of BLAS3 DGEMM is an example of this pattern [42].

Third, $O(N \log N)$-type data flow graphs require depth first source code scheduling [17]. This utilizes the available registers most efficiently and the slow-growing reuse help tolerating a limited amount of register spilling. FFTW's codelets, and SPIRAL-generated fixed-size FFT and DCT kernels as well as, sorting networks are examples of this pattern.

Unaligned vector array scalarization Unaligned load operations following aligned store operations are common in stencil-like codes and introduce two performance penalties. Firstly, unaligned loads are substantially more expensive than aligned loads, even on the latest Intel processors (SandyBridge and Nehalem). Secondly, the read-after-write dependency introduced by an aligned store followed by an unaligned load that touch an overlapping memory area is costly as it introduces hard barriers for the compiler to reorder instructions to hide memory latencies. They also can introduce problems in write buffers and store-to-load forwarding hardware and reduce the number of registers that can be used.

Our codelet generator performs an important optimization to eliminate unaligned load operations. It first unrolls the outer (time) loop of a stencil to collect the store and load operations from neighboring iterations of the outer loop in the same basic block. Next, it replaces unaligned loads with aligned loads and in-register permutations (vector shifts). Then, it replaces the ensuing matching aligned store/load pairs by automatic (register) vector variables, in effect scalarizing the SIMD vector array. Finally, it performs ISA-specific strength reduction and common subexpression elimination on the introduced permutation instructions to minimize their cost. Often, more than one instruction sequence can be used to implement the vector shift and carefully choosing the sequences allows to reuse partial results across multiple iterations of the innermost loop. The result is a large basic block free of unaligned memory accesses and thus free of unaligned read-after-write dependencies that would stall the pipeline. The basic block further can take better advantage of large register files and fully benefits from super-scalar out-of-order cores.

Alignment versioning On many SIMD ISAs the instructions required to shift data within SIMD registers to resolve misalignment take the shift value as immediate and thus require it to be a compile-time constant. On other ISAs different instruction sequences are required for different shift values. The miss-alignment may be introduced through a loop variable or may vary across codelet invocations. Finally, there may be multiple arrays that are independently misaligned.

When necessary our codelet generator performs code specialization with respect to unknown array miss-alignment. The largest scope within the codelet for which the miss-alignment is constant (usually either the whole codelet or the outermost loop body) becomes a `switch` statement that contains one `case` for each combination of miss-alignments. Each `case` contains specialized code that inlines the miss-alignment values and the resulting specific vector shift instruction sequences. The potentially large code size blowup is usually no problem given the big instruction cache size and can be compensated through slightly less aggressive unrolling.

Generic optimizations Finally, the codelet generator applies standard basic block optimizations used by program generators such as FFTW's `genfft`, SPIRAL and ATLAS. This includes copy propagation, constant folding, common subexpression elimination, array scalarization, and source code scheduling. The codelet generator also performs simplification of array indexing expression and expresses them using patterns supported by the ISA, and can generate array-based and pointer-based code.

5.3.1 Quality of the Generated Codelets

Below we show the code generated for an 1D Jacobi tile for Intel SSSE3. The instruction set operates on 4-way 32-bit float vectors and supports the `palignr` instruction. For readability we extract the kernel as inline function. In the actual generated code the function would be inlined. The example code is a partial tile codelet that does not have a time loop inside and thus unaligned store-to-load forwarding through unaligned vector array scalarization is not applied. Nevertheless, all unaligned loads have been replaced by aligned loads and the `palignr` instruction. The codelet generator implements a software pipeline that for every loop iteration of the unrolled loop loads one SIMD vector, computes one SIMD vector result with two vector adds, one vector multiply and two vector alignment operations, and stores the result.

```
__m128 __forceinline kernel(__m128 in0, __m128 in1, __m128 in2) {
  return _mm_mul_ps(_mm_set1_ps(0.333333333),
    _mm_add_ps(_mm_castsi128_ps(_mm_alignr_epi8(_mm_castps_si128(in0),
      _mm_castps_si128(in1), 4)),
    _mm_add_ps(in1,_mm_castsi128_ps(_mm_alignr_epi8(_mm_castps_si128(in1),
      _mm_castps_si128(in2), 12)))));
}

void jacobi1d_tile(__m128 *in, __m128*out, int n) {
  __m128 t0, t1, t2, t3, t4, t5, t6, t7,..., t17, t18, t19;
    t0 = in[0]; t1 = in[1];
        for (i=1; i<n-1; i+=20) {
      t2 = in[i+1]; out[i+0] = kernel(t0, t1, t2);
      t3 = in[i+2]; out[i+1] = kernel(t1, t2, t3);
      t4 = in[i+3]; out[i+2] = kernel(t2, t3, t4);
      ...
      t18 = in[i+17]; out[i+16] = kernel(t16, t17, t18);
      t19 = in[i+18]; out[i+17] = kernel(t17, t18, t19);
      t0 = in[i+19]; out[i+18] = kernel(t18, t19, t0);
      t1 = in[i+20]; out[i+19] = kernel(t19, t0, t1);
  }
}
```

Below we show the assembly generated for this function by the Intel C++ compiler 11.1 in 64-bit mode (EM64T), targeting a SandyBridge processor supporting the VEX encoding and 3-operand instructions. We see the compact addressing encoding and that on this processor the code looks almost perfect.

```
jacobi1d_tile   PROC
; parameter 1(in): rcx
; parameter 2(out): rdx
; parameter 3(n): r8d
        vmovaps   xmm1, XMMWORD PTR [rcx]
        vmovaps   xmm3, XMMWORD PTR [16+rcx]
        mov       r8d, 1
        mov       eax, 16
        vmovaps   xmm0, XMMWORD PTR [_2il0floatpacket.287]
.B2.2:: vmovaps   xmm4, XMMWORD PTR [16+rax+rcx]
        vpalignr  xmm1, xmm1, xmm3, 4
        add       r8, 20
        vpalignr  xmm5, xmm3, xmm4, 12
        vaddps    xmm2, xmm3, xmm5
        vaddps    xmm5, xmm1, xmm2
        vmulps    xmm1, xmm0, xmm5
        vmovaps   XMMWORD PTR [rax+rdx], xmm1
        ... ; about 300 lines of assembly repeating the last 5 lines
        ... ; cycling through the 16 XMM registrer
        vmovaps   xmm3, XMMWORD PTR [320+rax+rcx]
        vpalignr  xmm2, xmm1, xmm3, 12
        vaddps    xmm2, xmm1, xmm2
        vaddps    xmm5, xmm4, xmm2
        vmulps    xmm2, xmm0, xmm5
        vmovaps   XMMWORD PTR [304+rax+rdx], xmm2
        add       rax, 320
        cmp       r8, 401
        jl        .B2.2
        ret

_2il0floatpacket.287    DD      03eaaaaabH,03eaaaaabH,03eaaaaabH,03eaaaaabH
```

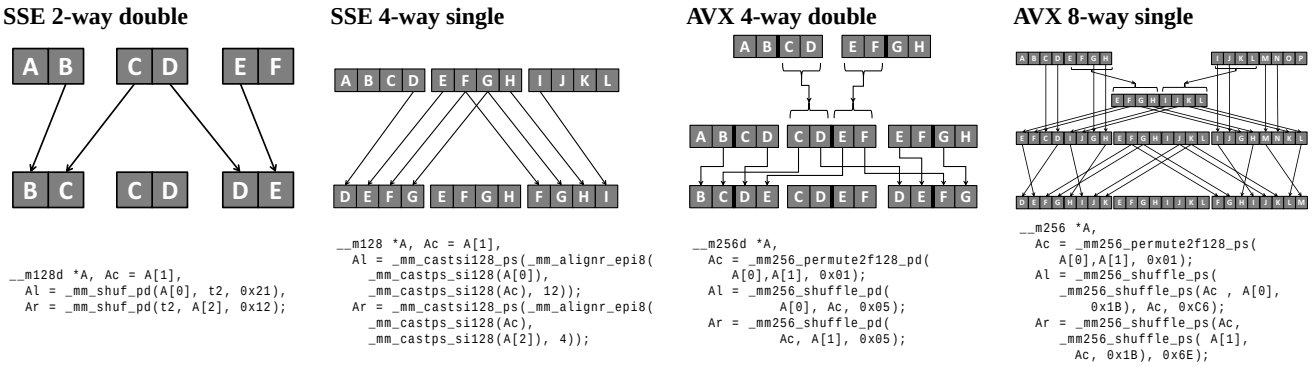

Figure 5. Extraction of left-shifted (denoted by `Al`), centered (`Ac`) and right-shifted (`Ar`) SIMD vectors from three aligned vectors `A[0]`, `A[1]`, and `A[2]` through permutation instructions. We display the data flow and instruction sequence for shifting on SSE 2-way double and 4-way single, and AVX 4-way double and 8-way single, as supported on Intel's SandyBridge processors.

One concern is the large size SSE instructions and decoder limitations, however, post-decoder micro-op caches can alleviate this problem on high performance processors. In Sec. 6 we will analyze the tile performance for a collection of kernels, machines and vector lengths.

5.4 Cost Model

Our codelet generator provides a simple cost model for the generated codelets to aid codelet (pre-)selection without the need for profiling the code. The line codelets are in steady state run cache resident (in hot L1 cache). Thus a weighted instruction count is a good first order metric to predict the relative performance of codelets. The metric captures the fact that generated codelets run close to the machine peak and this places a cost on every instruction, even if executed in parallel. The metric in particular allows to measure the vectorization overhead (number of arithmetic operations versus number of all operations). By counting source-level SIMD operations like `_mm_add_ps`, `_mm_sub_ps`, and `_mm_unpacklo_ps` instead of assembly instructions like `movaps`, `lea` and `mov` the metric is more resilient to effects on aggressive superscalar out-of-order processors like the Intel Core i7. We pick the coefficients for various operations to provide a relative ordering that favors cheaper operations over more expensive operations and models the relative cost of operations in the target micro-architecture. While the metric cannot predict actual execution times it is sufficient to prune the search space and find good candidates and aids auto-tuning in the high-level framework.

6. Experimental Results

6.1 Experimental Protocol

We evaluate our framework on a collection of benchmarks where the core computational part is a SCoP. Specifically, we experiment with 5 stencil computations that arise in typical image processing and physical simulation applications, and 3 linear algebra benchmarks. Problem sizes are selected to be larger than the last level cache (LLC) size of 8 MB. Each benchmark is evaluated using single-precision (`float`) and double-precision (`double`). Experiments are performed on an Intel SandyBridge i7-2600K (3.4GHz, 4 cores, 32KB L1, 256KB L2, 8MB L3), using either the SSE or AVX vector instruction set. The testbed runs a native 64-bit Linux distribution. We use Intel ICC 13.0 with `-fast -parallel -openmp`, except for the sequential baseline where we use `-fast`. For AVX experiments, the flag `-xAVX` is used.

For each benchmark, we apply the following flow: (1) loop transformations for parallel parametric tiling with full-tile separation, and coarse-grain shared-memory parallel execution; (2) for each full-tile, perform loop transformation to expose maximal codelets, and perform abstract SIMD vectorization; (3) for each codelet found, perform ISA-specific SIMD synthesis. This framework is implemented using PolyOpt/C, a polyhedral compiler based on the LLNL ROSE infrastructure [2]. The SIMD codelet generator is implemented in the SPIRAL system's back-end infrastructure. A web page and codelet generation web service with makefile tie-in can be found at [3]. On average, our end-to-end framework (not considering auto-tuning time) takes about 25 seconds to generate a program version for benchmarks like Jacobi-2d, laplacian-2d, or correlation; and about 3 minutes for benchmarks like Jacobi-3D, laplacian-3D, or doitgen.

6.2 Codelet Extraction Statistics

Table 1 reports statistics on the ILP scheduling problem developed in this paper. We report for some representative benchmarks the number of dependence polyhedra for array references, the total number of schedule coefficient variables $\theta_{i,j}$, the total number of additional variables inserted to model our problem, the total number of constraints in the system to be solved (this includes the semantics preserving constraints from dependence polyhedra), and the time to optimally solve the ILP using PIPLib. More details can be found in a technical report [27].

Benchmark	# deps	# refs	# $\theta_{i,j}$	# others	# cst.	Time (s)
jacobi-2d	20	8	140	486	3559	3.0711
laplacian-2d	20	10	140	502	3591	3.1982
poisson-2d	32	12	140	674	5277	5.8132
correlation	5	9	70	177	640	0.0428
covariance	3	4	70	176	593	0.0415
doitgen	3	4	117	269	911	0.1383

Table 1. ILP statistics

6.3 Tile Codelet Performance

Table 2 summarizes the operation count in steady state for the Jacobi kernels. This establishes the performance of the tile codelets in a stand-alone fashion. Here each codelet is made L1-resident, and is the result of unroll-and-jamming multiple loops in the codelet, leading to multi-loop vectorization in practice. We list the theoretical maximum floating-point operations per cycle and the actually measured floating-point operations per cycle for SSE and AVX on the processor described above, and also for a 2.66 GHZ Intel Nehalem processor. The SandyBridge processor supports SSE 4.2 and

Problem					SandyBridge 3.4 GHz								Nehalem 3.3 GHz			
					SSE 2-way		SSE 4-way		AVX 4-way		AVX 8-way		SSE 2-way		SSE 4-way	
Name	+	×	mem	shft	peak	meas	peak	meas	peak	meas	peak	meas	peak	meas	peak	meas
Jacobi 1D 3pt	2	1	2	1-5	3	2.5	6	5.5	4	2.7	8	8.2	3	1.4	6	2.9
Jacobi 2D 5pt	4	1	2	1-5	2.5	2.5	5	4.8	5	4.3	10	7.5	2.5	2.4	5	2.4
Jacobi 3D 7pt	6	1	2	1-5	2.3	1.2	4.6	4.5	4.6	3.8	9.3	8.9	2.3	1.1	4.6	4

Table 2. Cost analysis of the tile codelet specifications and performance (flop/cycle) on two different machines (Intel SandyBridge Core i7 2600K and Intel Nehalem X5680). We provide operation counts in steady state for adds/subtracts, multiply, vector shifts, loads and stores.

AVX and provides the three-operand VEX encoding for SSE instructions, which results in very compact code.

A three-point 1D Jacobi stencil in steady state requires one load, one store, two additions, one multiplication and two shift operations. The SandyBridge processor can transfer 48 bytes/cycle between the L1 cache and the register file, execute one vector (SSE or AVX) addition and vector multiplication each cycle and can perform one shift/permute operation per cycle. The decoder can decode 4 to 5 x86 instructions per cycle and caches decoded micro-ops. The hard limiting factor is the two additions needed per stencil result and thus the maximum performance is 6 flops/cycle for 4-way single-precision SSE (one addition every cycle and a multiplication every other cycle). We achieve 5.5 flops/cycle steady-state L1 performance. In the 2D case the maximum is 5 flop/cycle, which we achieve. This is a testament to the power of the latest generation of Intel processors and shows that our tile codelets are very efficient. The table summarizes further results across kernels and ISAs.

6.4 Full Program Performance

One of the key aspects of our framework is the ability to perform extensive auto-tuning on a collection of parameters with significant impact on performance. The two categories of parameters we empirically tune are (1) the tile sizes, so as to partition the computation in L1-resident blocks; and (2) the unroll-and-jam factor, that affects the number of operations in a codelet. While technically the parameter space is extremely large, it can be greatly pruned based on a number of observations. (1) We test tile sizes whose footprint is L1-cache resident. (2) We set the tile size of the vector dimension to be some multiple of the unrolling factor of this loop by the vector length, so as to minimize the peeling required for alignment. (3) We keep only tile sizes where the percentage of total computation performed by the full tiles is above a given threshold, so as to avoid tile shapes where a significant fraction of the computation is done by partial tiles. For 2D benchmarks we use a threshold of 80%, and a threshold of 60% for 3D benchmarks. (4) We unroll-and-jam all permutable dimensions, using unrolling factors of 1, 2, 4 and 8.

The auto-tuning is broken into two phases. In the first phase, approximately 100 tile sizes are tested and the 10 with the most iteration points in full tiles are selected. This process takes 2-3 minutes. The second phase tests 6-8 codelet optimizations for each of the 10 tile sizes, taking about 10-15 minutes.

Time breakdown over full/partial tiles and tile codelets To gain insights into the effectiveness and potential limitations of our framework, we first present a breakdown of time expended within tile codelets, within full tiles, and in partial tiles. Table 3 shows the performance comparison using 3 metrics: ATC, the time spent in all tile codelets; AFT, the time spent in all full tiles; FP, and the total time of the full program. We make the following observations: Peeling is the main reason for loss of performance when moving from ATC to AFT, and slowdowns can vary between $1.1\times$ and $2\times$. The fraction of performance loss depends on the vector length and unrolling factors used in the FVD. Thus, these two parameters must be tightly coupled with the FVD tile size while remaining tile sizes must be relatively small for minimizing the peeling overhead. Also, partial tiles have a considerable impact on the FP performance. In general partial tiles can cause between $1.3\times$ to $3\times$ slowdown over AFT, and they are particularly detrimental for higher dimensional iteration spaces (e.g. Jacobi-3D and doitgen). In general, bigger tile sizes improve performance but only up to a certain point, as they also gradually push more iterations from full tiles into partial tiles, even reducing the number of tiles that can be executed concurrently. Finally, the accumulated effect of peeling and partial tiles execution can yield between $1.4\times$ and $6\times$ slowdown from ATC to FP.

Benchmark	SIMD ISA	ATC (sec)	AFT (sec)	FP (sec)
jacobi-2d	SSE	0.040	0.046	0.068
jacobi-3d	SSE	0.458	0.514	1.113
jacobi-2d	AVX	0.036	0.054	0.077
jacobi-3d	AVX	0.188	0.354	1.062
laplacian-2d	SSE	0.046	0.056	0.071
laplacian-3d	SSE	0.482	0.538	1.131
laplacian-2d	AVX	0.031	0.042	0.061
laplacian-3d	AVX	0.197	0.373	0.993
poisson-2d	SSE	0.051	0.064	0.090
poisson-2d	AVX	0.029	0.049	0.075
correlation	SSE	0.840	0.915	1.179
correlation	AVX	0.752	0.917	1.139
covariance	SSE	0.846	0.922	1.159
covariance	AVX	0.733	0.883	1.121
doitgen	SSE	0.191	0.353	0.865
doitgen	AVX	0.158	0.308	0.834

Table 3. Time breakdown for kernels: All Tile Codelets (ATC), All Full Tiles (AFT=ATC+Peeling) and Full Program (FP=AFT+Partial Tiles)

Overall performance Finally, we report in Table 4 the full-application performance for the benchmarks. For each benchmark $\times$ vector ISA $\times$ data type, we compare our performance in GF/s to that attained by ICC on the input program (both sequentially and with automatic parallelization flags) and PTile, the performance of PTile [5], a state-of-the-art Parallel Parametric Tiling software (tile sizes have been auto-tuned). The performance achieved using only the front-end of our framework (transformation for codelet extraction) is reported in the Prevect columns, i.e., we restructure the program to expose codelets but simply emit standard C code for the codelet body and use the C compiler's vectorizer. SIMD reports the performance achieved by using our ISA-specific SIMD synthesizer on each codelet after codelet extraction.

We observe very significant performance improvements over ICC and PTile, our two reference compilers. Up to $50\times$ improvement is achieved using AVX for the covariance benchmark, for instance. We systematically outperform ICC (both with and without auto-parallelization enabled) with our complete framework using AVX (the SIMD/AVX column) since ICC fails to effectively vectorize most of the benchmarks. However, we note that due to the polyhedral model's limitation of not allowing explicit pointer management, the stencil codes have an explicit copy-back loop nest instead of a pointer flip. We note that the stencil benchmarks are

Benchmark	PB Size	single							double						
		ICC		PTile	Prevect		SIMD		ICC		PTile	Prevect		SIMD	
		seq	par		SSE	AVX	SSE	AVX	seq	par		SSE	AVX	SSE	AVX
jacobi-2d	20×2000^2	2.96	3.71	6.24	17.15	13.22	25.39	20.96	1.79	1.86	6.25	13.35	11.80	17.86	15.72
laplacian-2d	20×2000^2	4.30	4.44	6.29	18.7	17.85	24.86	26.57	2.15	2.23	6.30	13.19	13.13	16.01	18.53
poisson-2d	20×2000^2	4.23	6.68	17.47	19.56	14.75	31.77	42.23	3.08	3.36	16.86	15.19	11.31	20.67	29.77
jacobi-3d	20×256^3	4.21	4.70	3.82	5.38	5.04	3.84	5.53	2.22	2.06	4.01	2.70	2.87	2.67	3.41
laplacian-3d	20×256^3	4.80	5.44	4.39	6.13	5.67	6.99	6.26	2.55	2.38	4.58	3.34	3.17	4.15	3.98
correlation	2000^3	0.94	0.81	26.21	18.78	19.89	33.35	52.43	0.64	0.63	14.38	9.56	9.66	21.28	26.29
covariance	2000^3	0.97	0.99	26.53	18.70	20.08	34.85	55.92	0.65	2.16	13.52	9.66	9.70	21.88	27.23
doitgen	256^4	8.63	8.56	14.72	31.35	30.76	41.63	52.66	4.85	4.68	9.14	16.85	16.80	26.45	25.96

Table 4. Performance data in GFLOP/s.

severely memory-bandwidth bounded. Although ICC is able to automatically parallelize and vectorize the benchmarks, it is unable to perform time-tiling on the benchmarks and therefore the memory bandwidth limits the speedup achieved from parallelism. We also outperform, usually very significantly, the PTile research compiler in all but two cases (Jacobi-3D and Laplacian-3D, DP). These anomalies are due to our criteria of "good tile sizes" during the auto-tuning phase: tile sizes are selected based on the percentage of points that will be executed in full tiles vs. partial tiles. Therefore, a good configuration for our line codelet is not necessarily the overall best tile configuration. As explained before, increasing tile sizes shift iterations from full to partial tiles (which are slower) and reduce the number of concurrent tiles down to possibly a single one.

We also remark that the Prevect performance does not necessarily follow the vector length (and hence the arithmetic throughput), nor does it necessarily outperform ICC or PTile. One of the main reason is the complexity, in terms of number of basic blocks, of the codelets which are generated. By exploiting large tile sizes and unroll-and-jam, codelets with hundreds of blocks are generated and ICC simply cannot process the loop nest without splitting it, therefore negating the potential benefit of our approach. In general, we have observed that despite being in a form that satisfies the vectorizability criteria, the C code generated by our high-level framework does not lead to effectively exploited SIMD parallelism by the compiler, emphasizing the impact from an explicit coupling with a SIMD synthesizer. In some cases, ICC degrades performance with automatic parallelization, possibly due to inaccurate profitability models. PTile can also be slower than ICC parallel, due to complicated loop bound expressions as seen for single precision Jacobi and Laplacian 3D. The 3D stencils require a careful balance between maximization of the size of full tiles for high performance of the codelets, and keeping the size small enough to ensure that a high percentage of operations are in full tiles. Other tile shapes such as diamond tiling [4] may be needed for better performance.

7. Related Work

Automatic SIMD vectorization has been the subject of intense research in the past decades, i.e. [15, 26, 28, 30, 43]. These work are usually focusing on the back-end part, that is the actual SIMD code generation from a parallel loop [15, 28, 30], or on the high-level loop transformation angle only [12, 26, 38, 40]. To the best of our knowledge, our work is the first to address simultaneously both problems by setting a well-defined interface between a powerful polyhedral high-level transformation engine and a specialized SIMD code generator. Vasilache et al. also integrated SIMD and contiguity constraints in a polyhedral framework, in the R-Stream compiler [40], with similar objectives as ours. However, to the best of our knowledge, they are not considering the coupling of this framework with a powerful back-end SIMD code generator as we do. Other previous work considered inner- and outer-loop vectorization [31], our proposed work makes also a step forward by doing (tiled) *loop nest* vectorization, as codelets embed in their body up to all iterations of the surrounding loops.

Program generation (also called generative programming) has gained considerable interest in recent years [8, 9, 13, 21, 36]. The basic goal is to reduce the development, maintenance, and analysis of software. Among the key tools for achieving these goals, domain-specific languages provide a compact representation that raises the level of abstraction for specific problems and hence enables the manipulation of programs [10, 20, 24, 37]. Our codelet generator is an example of such a program generation system.

Automating the optimization of performance libraries is the goal in recent research efforts on automatic performance tuning, program generation, and adaptive library frameworks that can offer high performance with greatly reduced development time. Examples include ATLAS [42], Bebop/Sparsity [14, 25], and FFTW [18] for FFTs. SPIRAL [35] automatically generates highly efficient fixed-size and general-size libraries for signal processing algorithms across a wide range of platforms. SPIRAL and FFTW provide automatic SIMD vector codelet generation while ATLAS utilizes contributed hand-written SIMD vector kernels. While our transformation+synthesis approach bears some resemblance with those work, we address a much more general problem which requires to combine highly complex program transformations – that can be modeled effectively only by means of the polyhedral framework – with ISA-specific code generation.

8. Conclusion

Automatic short-vector SIMD vectorization is ubiquitous in modern production and research compilers. Nevertheless, the task of automatically generating effective programs — addressing the data locality, coarse-grain and SIMD parallelism challenges — remains only partly solved in the vast majority of cases.

We have made a statement about a viable scheme to achieve this goal, for a class of programs that arises frequently in compute-intensive programs. We have isolated and formalized program optimizations that can be effectively performed by a high-level loop transformation engine, from those optimizations that can be effectively implemented by SIMD code generation. We have used the power and expressiveness of the polyhedral compilation framework to formalize a series of scheduling constraints so as to form maximal *vectorizable codelets*, targeting parallelization, data reuse, alignment, and the stride of memory references in a single combined problem. As a result, we have unleashed the power of a custom ISA-specific SIMD code synthesizer, which translates those codelets into very effective (up to near-peak) SIMD execution. We have demonstrated the power of our approach on a collection of benchmarks, providing very significant performance improvement over an auto-parallelizing production compiler as well as a state-of-the-art research compiler.

Acknowledgments

The authors acknowledge support by DOE through awards DE-SC0005033 and DE-SC0008844, NSF through awards 0926688, 1116802 and 0926127, by the U.S. Army through contract W911NF-10-1-0004, and by Intel ECG.

References

[1] PoCC, the polyhedral compiler collection. http://pocc.sourceforge.net.

[2] PolyOpt/C. http://hpcrl.cse.ohio-state.edu/wiki/index.php/polyopt/c.

[3] www.spiral.net/software/stencilgen.html.

[4] V. Bandishti, I. Pananilath, , and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *ACM/IEEE Conf. on Supercomputing (SC'12)*, 2012.

[5] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, April 2010.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[7] C. Bastoul and P. Feautrier. More legal transformations for locality. In *Euro-Par'10 Intl. Euro-Par conference, LNCS 3149*, pages 272–283, Pisa, august 2004.

[8] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[9] D. Batory, R. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *Proc. Automated Software Engineering Conference (ASE)*, 2002.

[10] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, June 2008.

[12] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science Technical Report, 2008.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[14] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proc. of the IEEE*, 93(2):293–312, 2005.

[15] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, 2004.

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

[17] M. Frigo. A fast Fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[18] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[19] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[20] K. J. Gough. Little language processing, an alternative to courses on compiler construction. *SIGCSE Bulletin*, 13(3):31–34, 1981.

[21] GPCE. ACM conference on generative programming and component engineering.

[22] A. Hartono, M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *ICS*, 2009.

[23] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short simd architectures. In *ETAPS International Conference on Compiler Construction (CC'11)*, pages 225–245, Saarbrcken, Germany, Mar. 2011. Springer Verlag.

[24] P. Hudak. Domain specific languages. Available from author on request, 1997.

[25] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int'l J. High Performance Computing Applications*, 18(1), 2004.

[26] K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann, 2002.

[27] M. Kong, L.-N. Pouchet, and P. Sadayappan. Abstract vector SIMD code generation using the polyhedral model. Technical Report Technical Report 4/13-TR08, Ohio State University, Apr. 2013.

[28] S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[29] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, pages 201–214, 1997.

[30] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI*, 2006.

[31] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, 2008.

[32] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI*, pages 90–100. ACM Press, 2008.

[33] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *ACM Supercomputing Conf. (SC'10)*, New Orleans, Lousiana, Nov. 2010.

[34] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *POPL*, pages 549–562, Austin, TX, Jan. 2011.

[35] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232– 275, 2005.

[36] D. R. Smith. Mechanizing the development of software. In M. Broy, editor, *Calculational System Design, Proc. of the International Summer School Marktoberdorf*. NATO ASI Series, IOS Press, 1999. Kestrel Institute Technical Report KES.U.99.1.

[37] W. Taha. Domain-specific languages. In *Proc. Intl Conf. Computer Engineering and Systems (ICCES)*, 2008.

[38] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT*, Sept. 2009.

[39] N. Vasilache. *Scalable Program Optimization Techniques in the Polyhedra Model*. PhD thesis, University of Paris-Sud 11, 2007.

[40] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *Proc. of IMPACT'12*, Jan. 2012.

[41] Y. Voronenko and M. Püschel. Algebraic signal processing theory: Cooley-tukey type algorithms for real dfts. *IEEE Transactions on Signal Processing*, 57(1), 2009.

[42] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing*, 1998. `math-atlas.sourceforge.net`.

[43] M. J. Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley, 1996.