Throughput Optimization for High-Level Synthesis Using Resource Constraints

Peng Li[1,2] Louis-Noël Pouchet[2,3] Jason Cong[2,1,3]
[1] Peking University peng.li@pku.edu.cn
[2] University of California, Los Angeles {pouchet,cong}@cs.ucla.edu
[3] UCLA/PKU Joint Research Institute in Science and Engineering

Abstract

Programming productivity of FPGA devices remains a significant challenge, despite the emergence of robust high level synthesis tools to automatically transform codes written in high-level languages into RTL implementations. Focusing on a class of programs with regular loop bounds and array accesses, the polyhedral compilation framework provides a convenient environment to automate many of the manual program transformation tasks that are still needed to improve the QoR of the HLS tool.

In this work, we demonstrate that traditional affine loop transformations are not always enough to achieve the best throughput, determined by the Initiation Interval (II) for loop pipelining, and other transformations such as Index-Set Splitting (ISS) can lead to better performance. We develop a customized affine+ISS optimization algorithm that aims at reducing the II of pipelined inner loops to reduce the program latency. We report experimental results on numerous affine computations, showing significant latency and energy improvements.

Categories and Subject Descriptors B.5.2 [*Hardware*]: Design Aids — optimization; D 3.4 [*Programming languages*]: Processor — Compilers; Optimization

Keywords Program Optimization; High-Level Synthesis; Loop Pipelining

1. Introduction

Recently, FPGA devices are utilized as reconfigurable accelerators for a variety of applications with higher performance and energy efficiency compared to traditional general purpose processors. However, programming productivity of FPGA devices remains a significant challenge. Traditional register transfer level (RTL) design is time-consuming, error prone and difficult to debug.

High level synthesis (HLS) tools can automatically transform algorithms written in high level languages (e.g. C, C++, SystemC) to RTL implementations. After three decades of research and development by academia and industry, HLS has become a promising productivity boost for the semiconductor industry especially the FPGA industry [15]. For example, AutoESL [35]'s successor Xilinx Vivado HLS [1] is now part of standard Xilinx Vivado Suite available to all Xilinx FPGA designers. While the state-of-art HLS tools can achieve comparative quality of result (QoR) to manual RTL designs [15, 23] for simple application kernels, there is still a significant performance gap (up to 40X according to [26]). To improve the QoR of the HLS tool, designers often perform a number of explicit source-code modifications to transform the original code to HLS-friendly code addressing several key issues such as on-chip buffer management, attention to loop dependence, avoidance of memory port conflicts and communication/computation overlapping.

Recent research on polyhedral optimization framework has shown great potential to automate some of these manual code transformations for a class of programs with regular loop bounds and array accesses (namely, affine programs [19]). The polyhedral compilation framework is a powerful model to optimize a class of programs whose loop bounds, conditionals and array index functions are affine forms of the surrounding loop iterators. Recent work showed how to fully automate transformations for data reuse and latency optimization [11, 27, 28, 37] using this model and how to customize polyhedral code generation for HLS purpose [36]. While the performance gain over off-the-shelf usage of Vivado HLS can be impressive (10x or more improvement in some cases), these transformations aim first and foremost at reducing and amortizing off-chip communication cost. The final performance of the program requires also to carefully optimize the code implementing the computation on the FPGA.

In this work we consider the problem of optimizing the performance (i.e., throughput) of the computation part of an affine program to be executed on the FPGA. Assuming that a first set of transformations has been applied to maximize data reuse, for instance using PolyOpt/HLS [4, 28], the present work focuses on improving the performance of the code that performs the actual computation, assuming all the data has been brought on-chip by a communication/prefetch function [28]. We make the following contributions.

- We perform a comprehensive study of the impact of standard, parallelism-driven loop transformations on numerous affine kernels with HLS, to characterize when these transformations are not sufficient to achieve good performance.
- We evaluate the use of Index-Set Splitting as a complementary transformation to extract additional loop-level parallelism and further reduce the II of affine programs.
- We introduce a optimization algorithm using Index-Set Splitting for HLS, based on memory port conflict detection, to separate out conflict-free loop iterations leading to further latency improvements.
- We report extensive performance results using our algorithms, achieving up to 4× latency improvement over tiling-driven affine transformations.

The paper is organized as follows. Sec. 2 outlines our work and discusses previous work. Sec. 3 recalls the principles of polyhedral program optimization. Sec. 4 presents our approaches for extracting inner parallelism based on data dependences. Sec. 5 introduces our approach for II optimization through ISS for resolving memory port conflicts. Sec. 6 presents extensive experimental results.

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://impact.gforge.inria.fr/impact2014

2. Overview and Related Work

Loop Pipelining in HLS Loop pipelining [5] is a key performance optimization technique in high-level synthesis. With loop pipelining, parallelism across loop iterations can be exploited by starting the next iteration of the loop before the completion of the current iteration. While the number of cycles spent for each loop iteration stays unchanged, the total execution cycles of the entire loop is decreased by overlapping operations from several iterations through loop pipelining. Loop pipelining features two key parameters:

- The **initiation interval** (or *II* for short), that corresponds to the number of cycles between the execution of two adjacent loop iterations.
- The **depth**, that represents the number of cycles required to completely execute one iteration of the loop.

Figure 1 illustrates a loop with II as 3 and depth as 8. The total execution cycles of the entire loop can be formulated as $Cycle = (TripCount - 1) * II + Depth$. As a result, the performance of a pipelined loop is driven by the *II* (assuming there is more than one loop iteration).

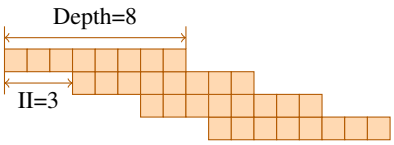

Figure 1. Loop Pipelining

The achieved loop *II* is limited both by loop-carried dependences and resource constraints. The existence of dependences between loop iterations will force a delayed start of the next loop iteration, until the data needed is computed by the previous iteration. With limited hardware resources, the loop iteration need to be delayed until the resource needed is available. In particular, a source for limited II comes from simultaneous access to the same BRAM bank by two (pipelined) iterations. Eliminating such resource constraint has been studied extensively using array partitioning [14, 25, 33], which splits an array into multiple arrays, increasing the number of memory ports available at the cost of increased resource usage.

Polyhedral compilation for HLS Previous work studied the power of the polyhedral transformation framework for FPGA design. For instance, DEFACTO combines a parallelizing compiler technology (SUIF) with early hardware synthesis tools to propose a C-to-FPGA design flow [17, 29], and MMAlpha [22] focused on systolic designs. These works illustrated the benefit of using advanced compiler technologies for memory optimization and parallelization analysis. Recently, Pouchet et al. proposed an advanced loop transformation framework based on the polyhedral model for automatic data reuse optimization using HLS [28], considering a much richer space of program transformations than previous work. Bayliss et al. [11] used the polyhedral model to compute an address generator exploiting data reuse; however they do not consider any loop transformation and therefore do not restructure the program to better exploit its inherent data locality potential. Darte et al. use lattice-based memory allocation to reduce memory usage while still implementing the available reuse [16]. Alias et al. develop a polyhedral compiler framework to optimize off-chip memory traffic and exploit on-chip data reuse for the Altera C2H toolchain [6, 7]. Zuo et al. recently proposed complementary polyhedral transformations for better latency of affine programs using HLS [37], and studied how to customize polyhedral code generation for HLS and FPGA mapping purposes [36].

Our starting point for this work is the set of transformation techniques implemented in the open-source software PolyOpt/HLS [4], which is a powerful automatic source-to-source transformation framework [28] aimed at generating optimized C code that is input to tools such as Vivado HLS. In order to further improve the performance of the generated codes, we study in the present paper techniques (to be complementary with data reuse optimizations in PolyOpt/HLS) aimed at improving the latency of the computation functions. That is, we aim for better performance of the computation modules, assuming all data has already been brought on-chip. This is a valid assumption, as PolyOpt/HLS already takes care through careful generation of communication and prefetch functions of moving all data needed in and out the chip, such that each computation function operates only on on-chip data. Our past experience with PolyOpt/HLS showed that when data reuse is carefully implemented, many computations that were apparently memory bound become compute-bound on a Convey HC-1ex machine [28]. We aim in this work for improvements of the computation module latency wherever possible.

Affine loop transformations PolyOpt/HLS relies on advanced program transformation techniques to partition the computation into atomic blocks wherever possible. Such blocks (or tiles) can often be executed in parallel, leading to exposing coarse-grain parallelism that is exploited on the FPGA by replicating the computation module for one tile, so as to execute multiple tiles in parallel when possible. The algorithm implemented to expose tiling is the Tiling Hyperplane method [13], that roughly takes care of the overall program structure to ensure effective data reuse and off-chip communication schemes can be derived. Inside a tile, loop-level parallelism is exposed using additional loop transformations such as loop interchange, with the objective of exposing inner parallel loops in a manner similar to the requirements of CPU SIMD vectorization [28]. A significant advantage of exposing inner loops which are parallel is that we mark them for pipelining and add pragmas to inform there is no loop-carried dependence, usually leading to much smaller II. This transformation approach is described in Sec. 4.1.

While there could be inner-loop parallelism potentially available after using particular affine loop transformations such as skewing, the algorithms we use in PolyOpt/HLS are driven by tiling constraints and do not systematically lead to exposing inner-loop parallelism inside a tile: only inter-tile parallelism is exposed on tilable programs. We are in need of another loop transformation method that takes the generated tile code as input, and restructure it to expose inner parallelism when possible. In this paper we develop an approach based on ISS [21] that focuses particularly on non-uniform dependences, as uniform dependences are usually well handled by the existing algorithms in PolyOpt/HLS.

ISS for non-uniform data dependences PolyOpt/HLS excels at optimizing programs with uniform dependences (that is, those where the distance between dependent iterations is constant [9]). For affine programs with non-uniform dependences (that is, those where the dependence distance is not constant for the loop [9]) it may not even be possible to find a parallel inner loop using only traditional affine transformations such as skewing, interchange, fusion, distribution, and shifting. This observation served as the motivation for Index-Set Splitting [21]. ISS splits iteration domains into pieces (that is, splits one loop into multiple loops) and is complementary to traditional affine loop transformations. After ISS, traditional affine loop transformations can be applied on the resulting program, leading to using piecewise-affine loop transformations instead of affine only transformations without ISS, thereby broadening the program transformation space.

The idea of ISS can be applied in HLS to improve the throughput of loop nests by exposing inner parallel loops. In such case, the initiation interval of the loop is determined only by resource constraints, and we can safely mark the loops as dependence-free to let HLS tools perform more aggressive optimizations. Our ISS-based technique for exposing more parallelism is presented in Sec. 4.2. We remark that in our framework, we are not executing parallel inner loops in a parallel fashion, we execute them in a pipelined fashion

only. Parallelism is exposed and exploited at a coarse-grain level in PolyOpt/HLS, by design choice. But our techniques to expose inner parallel loops that are free of memory port conflict can be seamlessly used by other frameworks, including those implementing parallel execution of inner (parallel) loops.

Index Set Splitting for resource constraints Typical on-chip memory modules in FPGA devices have only 2 ports. With 3 accesses on array B in the innermost loop, the loop II is at least 2. An array can be partitioned into multiple banks to support more simultaneous accesses [14, 33]. Nonetheless, not all inner loop iterations are always exposing a memory port conflict. The ideas behind ISS can be applied to resource constraints: loop iterations can be split in two sets according to the resource conflicts, to obtain one conflict-free loop and one loop where conflicts systematically occur. Our ISS-based technique for resource optimization is shown in Sec. 5.

We observe that currently Vivado HLS (2013.3) cannot always determine the lack of memory port conflict on such codes, and without further treatment fails to meet the target II of 1 for the conflict-free loop. Indeed, contrary to data dependence, there is no pragma available to instruct Vivado HLS about the lack of memory port conflict. To solve this limitation, we generate a separate memory module and connect it with the computation modules synthesized by Vivado HLS using a stream interface [25, 37]. In the generated memory module, arrays are cyclically partitioned into two banks. We used this technique to bypass memory port conflict detection in Vivado HLS. This aspect is orthogonal to the present work and does not influence the algorithms we develop in this paper.

3. Polyhedral Compilation

We first review the key components of a polyhedral compilation framework, including the various mathematical objects used to represent programs and their transformations.

3.1 Background and Program Representation

The *polyhedral model* is a flexible and expressive representation for loop nests with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [19, 20], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved 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 can be found in image processing filters, linear algebra computations, etc. [20]. Program optimization in this framework is a three-stage process. First, the program is analyzed to extract its polyhedral representation, including dependence information and access pattern.

Iteration Domains For all textual statements in the program the set of its run-time instances is captured with a set of affine inequalities intersected with an integer lattice [10]. When the statement is enclosed by loop(s), all iterations of the loop(s) are captured in the iteration domain of the statement S, noted $\mathcal{D}_S$. $\mathcal{D}_S$ contains only integer vectors (or, integer points if only one loop encloses the statement S). The *iteration vector* $\vec{x}_S$ is the vector of the surrounding loop iterators. Each vector in $\mathcal{D}_S$ corresponds to a specific set of values taken by the surrounding loop iterators (starting from the outermost to the innermost enclosing loop iterator) when S is executed.

Access functions They represent the location of the data accessed by the statement. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of a read reference `A[i][j]` surrounded by 3 loops i, j and k is simply $f_B(i,j,k) = (i,j)$.

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*. This is defined at the granularity of the array cell. If two instances $\vec{x}_R$ and $\vec{x}_S$ refer to the same array cell and at least one of these references is a write, then they are said to be in dependence. Therefore to respect the program semantics, the transformed program must ensure $\vec{x}_R$ and $\vec{x}_S$ are executed in the same order as in the original program. Given two statements R and S and a data dependence $R \rightarrow S$, a dependence polyhedron, written $\mathcal{D}_{R,S}$, contains all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$. This modeling represents seamlessly uniform and non-uniform dependences.

Multiple dependence polyhedra may be required to capture all dependent instances, at least one for each pair of array references accessing the same array cell (scalars being a particular case of array). It is possible to have several dependence polyhedra per pair of textual statements, as some may contain multiple array references. In our work, all dependence polyhedra are automatically extracted from the program polyhedral representation, using the Candl tool [2] that is integrated in PolyOpt/HLS.

Program Transformations The second step in polyhedral program optimization is to compute a transformation for the program. Such a transformation captures, in a single step, what may typically correspond to a sequence of several tens of textbook loop transformations [20]. It takes the form of a carefully crafted affine multidimensional schedule (that is, a matrix), together with (optional) iteration domain or array subscript transformations [13,28]. A schedule is a function which associates a logical execution date (a timestamp) to each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the target program, statement instances will be executed according to the increasing lexicographic order of their timestamp. To construct a full program optimization, we build a collection of schedules $\Theta = \{\Theta^{S1}, \ldots, \Theta^{Sn}\}$, that is a list of the statement scheduling function for each statement in the program, such that for all dependent instances the producer instance is scheduled before the consumer one.

Polyhedral Code Generation Finally, the last step of polyhedral loop transformation is to generate a transformed program according to the optimization we previously computed. Syntactically correct transformed code is generated back from the polyhedral representation, and this code scans the iteration spaces according to the schedule we have computed with the Tiling Hyperplane method. We use the CLooG, a state-of-the-art code generator [10] to perform this task, that has been optimized for HLS purpose [36].

4. Dealing with Data Dependences

One effective way to improve throughput with loop pipelining is to eliminate loop-carried dependence. While this is not a requirement for loop pipelining, optimizing affine programs can usually expose one parallel inner loop, possibly through aggressive program transformations. Then, the innermost parallel loops are marked with pragmas to inform HLS tools that there is no loop-carried dependence, and that the loop should be pipelined with the smallest possible II value subject to resource constraints.

In this section, we focus on program transformations needed to expose at least one level of inner parallelism, for the purpose of using loop pipelining on these loops. Indeed, if such loops are pipelined, only resource constraints can lead to an II larger than 1. We first present in Sec. 4.1 our approach using tiling-driven affine transformations, that can extract at least one level of parallelism for all affine programs with only uniform dependences. We then present in Sec. 4.2 a customized approach based on Index-Set Splitting to expose further inner parallelism for programs with partial / non-uniform dependences.

4.1 Tiling-driven Affine Transformations

In order to expose coarse-grain parallelization as well as data locality optimizations, a proven approach is to compute a polyhedral transformation which is geared towards minimizing dependence distance while exposing outer-loop parallelism when possible. This optimization is implemented via a composition of multi-dimensional tiling, fusion, skewing, interchange, shifting, and peeling. It is known as the Tiling Hyperplanes method [12, 13], which is implemented in PolyOpt/HLS [28].

The tiling hyperplane method has proved to be very effective in integrating loop tiling into polyhedral transformation sequences [13, 24]. Bondhugula et al. proposed an integrated optimization scheme that seeks to maximally fuse a group of statements, while making the outer loops permutable (i.e., tilable) [12, 13] when possible. A schedule is computed such that parallel loops are brought to the outer levels, if possible. When coarse-grain parallelism is exposed (such as through pipelined tile parallelism), it can be automatically exploited to support concurrent execution on the FPGA.

Previous work on SIMD vectorization for affine programs has proposed effective solutions to expose inner-loop-level parallelism [13, 30]. In PolyOpt/HLS a similar technique is implemented, which aims at exposing inner parallel loops in the code of a (computation) tile. It sinks a parallel outer loop (if any) in the tile code to the innermost level through a sequence of loop interchanges — it is always legal to sink an outer parallel loop inside a loop nest. As a result, we mark all innermost loops with a specific `#pragma HLS pipeline`, and let the HLS tool find the best II it can for this loop.

Results We have applied the above affine loop transformations method to a collection of affine benchmarks from PolyBench/C [3]. We observed that for the majority of benchmarks (11 out of 14) we tested, an II of 1 can be achieved by tiling-driven affine transformations alone (that is, the technique implemented in PolyOpt/HLS), as shown in later Sec. 6. Very interestingly, we also observe that II in some benchmarks (for instance Floyd-Warshall and Trmm) is not improved after affine transformations. A careful analysis showed two root causes. First, for these two benchmarks, non-uniform dependences and/or dependences that only affect a subset of the iteration domain prevented the affine transformations to expose a fully parallel inner loop. Second, for the rest of the benchmarks, our analysis showed that resource conflicts (e.g., memory port conflict) was one of the key reason for failing to meet the target II of 1.

This first series of experiments motivated the development of another program transformation scheme, aimed at properly dealing with non-uniform/partial dependences, as shown below. We integrate the support of memory port conflict elimination in later Sec. 5.

4.2 More Parallelism With Index-Set Splitting

Index-Set Splitting ideas take roots in seminal work on tiling such as from Kennedy [8] and Wolfe [34], and was generalized for the polyhedral compilation model by Griebl and Feautrier [21]. ISS in essence amounts to splitting the iteration domain of (some) statements into multiple sub-domains. That is, intuitively, a loop surrounding a statement and executing N iterations is for instance split into two loops, each of which doing $N/2$ iterations. The benefit for HLS is straightforward: it will allow independent and different treatment of each sub-loop created, something not always possible with the original, non-split code.

Let us take a program which has an inner loop to be pipelined where a data dependence exists between only *some* of the iterations, while others are independent. Current HLS tools, such as Vivado HLS, assume a uniform loop II across all loop iterations to simplify the control logic. Therefore, once there is a single dependence between two iterations, this dependence will be conservatively used to compute II for the entire loop. By carefully breaking the loop into multiple loops, one may expose parallelism in some loop(s) created, and the synthesis tool can generate significantly better code for the loop containing only independent iterations. Our objective is to design an algorithm aimed at exposing inner parallel loops that uses ISS together with complementary techniques to sink the parallel loop(s) obtained at the inner-most level through sequences of interchange.

Customized ISS for Increased Parallelism Our customized ISS-based algorithm to expose parallel inner loops is shown in Fig. 2, we refer to this algorithm as ISS-Dep. It assumes the input code fragment is the result of the application of tiling-driven loop transformations for data reuse (e.g., using PolyOpt/HLS) and that all data is already on-chip, this strongly relaxes data locality objectives for our algorithm. In other words, we can safely increase the distance between iterations accessing the same data inside the tile, it will not change the program off-chip communication scheme implemented. Our algorithm differs from [21] on several aspects. First, we build it to focus on exposing parallel inner loops, and combine index-set splitting with loop permutation to expose inner parallel loops. Second, by only considering a subset of the possible splits, we ensure by construction that the split we consider is always legal, without having to consider all program data dependences. Third, we iteratively process dependences one by one, without resorting to complex incoming dependence path computation as in [21]. This simplification was sufficient to expose parallelism in the benchmarks we considered in conjunction with the fact that we applied as a pre-pass tiling-driven transformations to expose parallelism when possible.

The objective of this algorithm is to expose inner parallel loops in a more effective way than the tiling-driven transformation system depicted above. In particular we resort to recursive index-set splitting (from the inner-most loop to the outer-most loop) combined with affine transformations to expose more parallel inner loops when possible. Our algorithm is initially called on each inner-most loop which is not parallel.

```
DependenceSplit:
Input:
    l: Polyhedral loop nest (SCoP)
Output:
    l: in-place modification of l

1   D ← getAllDepsBetweenStatementsInLoop(l)
2   D ← removeAllLoopIndependentDeps(D, l)
3   parts ← {}
4   foreach dependence polyhedron D_{x,y} ∈ D do
5       D_y ← getTargetIterSet(D_{x,y}) ∩ D_l
6       if |D_y| < |D_l| then
7           parts ← parts ⋃ {D_y}
8       else
9           continue
10      end if
11  end do
12  l' ← split(l, parts)
13  if sinkParallelLoops(l') ≠ true
      .or. parentLoop(l) = null then
14      l ← l'
15      return
16  else
17      DependenceSplit(parentLoop(l))
18  end if
```

Figure 2. ISS-Dep: Customized Splitting for Parallelism

Function `getAllDepsBetweenStatementsInLoop` from algorithm `DependenceSplit` in Fig. 2 collects all dependence polyhedra between statements inside the loop (that is, all dependences between statements outside that loop and statements inside it are discarded). Function `removeAllLoopIndependentDeps` removes all dependence polyhedra from the set which are describing loop-independent dependences, therefore the output set contains only loop-carried dependences for loop l. We note $\mathcal{D}_l$ the iteration do-

main of the "loop", which is a polyhedron made of the affine inequalities of the loop bounds of l. Function `getTargetIterSet` computes the set of target iterations in the polyhedron $\mathcal{D}_{x,y}$ so as to obtain the set of iterations that are targets in the dependence. It projects out the dimensions associated to x (the source), as illustrated in the previous section. The result is intersected with $\mathcal{D}_l$ to obtain only the set of loop iterations of l which are target to a dependence. $|\mathcal{D}_l|$ denotes the number of points in the polyhedron $\mathcal{D}_l$, we do not perform any split if the loop-carried dependence touches all iterations (that is, the loop is purely sequential).

`split` splits a loop into multiple loops, according to the set of sub-domains (*parts*) which has been computed. To generate the new loop structure we create the code scanning each polyhedron in *part* using CLooG [10]. It returns an AST l' with one loop (nest) per element in *part*. To form the input to CLooG that preserves the program semantics we proceed in three steps. First, for each polyhedron $\mathcal{D}_y$ in *parts* we compute $\mathcal{D}'_y = convHull(\mathcal{D}_y) \cap \mathcal{D}_l$, the convex hull of $\mathcal{D}_y$. The result is a set *part'* of polyhedra, containing all $\mathcal{D}'_y$ computed. We do this to ensure there is no hole in the sets $\mathcal{D}_y$, as projecting out dimensions of an integer polyhedron can lead to a non-convex set. Computing the convex hull ensures $\mathcal{D}'_y$ is a polyhedron. Second we compute a union of polyhedra I from $\mathcal{D}_l$ and *part'*. The objective is to create a set of disjoint subsets (one per sub-loop to be generated) such that $\cup_i I_i = \mathcal{D}_l$ and $\cap_i I_i = \emptyset$. We take $\mathcal{D}_l$ into consideration as there may be iterations in the original loop which are never target of a dependence, and therefore not in any of the $\mathcal{D}_y$. I is obtained by (1) updating each element in *part'* to ensure that in the resulting set all elements are disjoint, using intersection and difference between elements to compute the points to remove in each $\mathcal{D}'_y$, if any, to ensure disjointess; and (2) creating the difference $rem = \mathcal{D}_l \setminus \cup_i part'_i$ which is itself a union of polyhedra, and set $I = part' \cup \{rem\}$. We note that all these operations are seamlessly supported in the Integer Set Library [31]. The third and final stage is to order the elements in I by increasing lexicographic order of their first iteration (point), to reflect the original order of the loop iterations and ensure the generated code will follow the original execution order for the loop that is split. As a result, our ISS algorithm by construction preserves the program semantics. More advanced splitting techniques allowing for non-convex splits are left for future work, our experiments showed that this approach is sufficient for the benchmarks we have tested.

Finally, function `sinkParallelLoops` takes a sequence of loop (nests), and for each of them detects parallel loops using polyhedral dependence analysis and sinks them using loop permutation to the inner-most loop level when applicable. The function returns `true` if all inner-most loops are parallel in the generated loop (nest) $l\prime$. If there are inner-most loops which are still not parallel, the full algorithm is called again but on the parent loop, attempting to split at this dimension instead. For instance for Trmm, this is needed as in the original code loop `k` is purely sequential, but the surrounding loop `j` can be split, leading to two loop nests after splitting, and the newly created `j` loops are sink inwards. Function `parentLoop` returns the surrounding loop, or `null` if the loop is already the outer-most loop inside the tile (that is, the outer-most intra-tile loop).

We remark two aspects of our algorithm. First, by construction of the `split` function, we preserve the order of each loop iteration. That is, when splitting the loop, the semantics is necessarily preserved. Second, one can see that aggressive split leads to numerous loop nests being generated. This can pose a problem in terms of resource usage on FPGA, as resource sharing typically occurs between statements under the same inner loop. In practice, with our approach, this did not prove to be a problem. We apply the splitting as a post-processing, once affine transformations have been used to minimize the number of loop-carried dependences at the inner-most loop level. We have observed that there is only very few dependences left (usually 1 in our benchmark suite), leading to a very small number of split (usually 1, up to 3).

Results We used algorithm `DependenceSplit` on each benchmark for which the target II of 1 was not met because of remaining loop-carried data dependences. This is shown in Table 1, with rows ISS-dep. The results for the two benchmarks on which tiling-driven affine transformations failed to improve the II over the original code show that `DependenceSplit` can further reduce the II by a significant factor, from 5-8 to 2. It however can come at the cost of increased resource usage, such as with Trmm for instance which uses about 1.5x more flip-flops. This is explained by the increase in the number of statements in the program, leading to lower resource sharing.

The II found by Vivado is still not 1 for all pipelined inner loops. After careful analysis, the memory access pattern requires more read/write ports than those available on chip, leading to multiple cycles (2 here) needed simply to load/store the data from BRAMs for Trisolv. While one could address this problem possibly using aggressive array partitioning [33], we propose an alternative approach based on ISS that aims to not increase the BRAM consumption (contrary to aggressive array partitioning), but however can increase other resources such as LUTs and flip-flops.

5. Dealing with Resources

After applying standard affine loop transformations to expose inner parallel loops when possible, and ISS-Dep on the remaining (inner) loops which were still not fully parallel, we focus in this step exclusively on inner-most loops which are parallel (according to data dependences), but for which memory port conflicts prevent the loop to be fully pipelined.

To address this last performance issue, we propose to rely on ISS to split the problematic loops into sub-loops with different memory port conflict properties. The rationale is that memory port conflicts usually do not occur between each loop iteration, but only between a subset of them (those which lead to accessing the same banks, such as in `A[i]` and `A[i+4]`, `A[i+8]`, ... if we have four banks).

Bank conflict set First, let us define what is a memory port conflict for banked access, for the case of a single port.

DEFINITION 1 (Bank conflict). *Given two memory add-resses x and y (assuming cyclic mapping of addresses to banks using the % function). They access the same bank iff:*

$$x \,\%\, B \quad = \quad y \,\%\, B \tag{1}$$

with B the number of banks.

We also remark that Eq. (1) can be equivalently written:

$$\exists k \in \mathbb{Z}, \qquad x - y = B * k$$

We note that while cyclic mapping is used here, a wide range of mapping including block mapping and block-cyclic partitioning could be supported with similar techniques [11, 32]. In this work, we are interested only in possibly splitting the inner-most loop into multiple loops, so that different II for each loop generated could possibly be achieved. In the motivating example we saw that a conflict systematically occurs for some subset of the iterations in the inner-most loop (the even ones), and never occurs for the rest of the iterations (the odd ones). We now define a bank conflict set, as the set of inner-most loop iterations for which a conflict necessarily occurs between two references in the same iteration.

DEFINITION 2 (Bank conflict set). *Given an inner-most loop l, whose iteration domain is $\mathcal{D}_l$, and two references F_A^1 and F_A^2 accessing the same array A. The bank conflict set $\mathcal{C}_{F_A^1,F_A^2} \subseteq \mathcal{D}_l$ is:*

$$\mathcal{C}_{F_A^1,F_A^2} : \left\{ \vec{x}_l \in \mathcal{D}_l \mid \exists k \in \mathbb{Z},\ lin\left(F_A^1\right) - lin\left(F_A^2\right) = k * B \right\}$$

With $lin(x)$ the linearized form of x.

Definition 2 relies on linearizing the access functions, that is an access $(i+3, j+2)$ for an array of size 10×10 is represented as $(i+3)*10+j+2$. While this may pose a problem with arrays whose size is unknown, we use high-level synthesis only when each array size is fully known at compile-time, so as to make the synthesis possible. That is, the array size in each dimension is a known constant, and therefore the linearized form of a multi-dimensional affine array access is always an affine expression.

The algorithm We now present our algorithm for splitting inner-most loops according to their bank conflicts in Fig. 3.

It essentially splits the original iteration domain of inner-most loops into tuple of sets, containing either conflict-free iterations or conflict-guaranteed iterations. We note that as we only address the case of inner-most parallel loop, any reordering/splitting of the domain is necessarily legal. The Integer Set Library [31] is integrated in PolyOpt/HLS and is used to manipulate these sets and generate the C code that scans them.

```
ResourceSplit:
Input:
    l: inner-most parallel affine loop
    sz: size of arrays in l
    B: number of banks available
Output:
    l: in-place modification of l

1   lst ← {}
2   all ← ∅
3   foreach array A ∈ l do
4       foreach distinct pair of references F_A^i, F_A^j ∈ l do
5           C_{F_A^i,F_A^j} ← buildConflictSet(B, sizes(A), F_A^1, F_A^2, D_l)
6           lst ← lst ⋃ {C_{F_A^1,F_A^2}}
7           all ← all ∪ C_{F_A^1,F_A^2}
8       end do
9   end do
10  rem ← D_l \ all
11  lst ← { lst, rem}
12  l' ← codegen(lst)
13  l ← finalize(l, l')
```

Figure 3. Customized Splitting Algorithm for HLS

Function `buildConflictSet` builds the conflict set as shown in Def. 2, taking as input the access functions and the array size in each dimension so as to build a linearized (but affine) conflict expression. Function `codegen` is traditional polyhedral code generation in CLooG [10]: we input a list of domains (*lst*), and rely on CLooG to perform adequate polyhedral separation. Precisely, when there are multiple conflicts for a particular iteration, it will be put in a separate loop nest than iterations with a lower number of conflict. This meets our goal, as intuitively the higher the number of conflict per iteration, the larger the II. This procedure will automatically create loops where the II can be one (no conflict), loops where it can be two (one conflict), etc. taking into account the surrounding loop iterators value that make the conflict occur (j in our example). Function `finalize` uses the loop structure generated by CLooG and replaces the body of each loop generated by the body of the original loop. Technically, for better performance, this function selectively merges certain loops in a single loop. Leaving out the loops whose unique statement inside its body corresponds to the domain *rem* in Fig. 3, we merge together all consecutive loops with a single statement in their body. That is, if there are two different conflicts occurring at consecutive (but non-intersecting) iteration ranges, then these loops are merged into a single loop before being ultimately replaced by the original loop body.

Putting it all together Our flow to optimize computation functions on FPGAs is as follows.

1. For each SCoP, apply the Tiling Hyperplane method, customized for SIMD parallelism exposure [13, 28]. This is a requirement for effective data reuse and off-chip communication generation in our framework.
2. For each inner loop which still has loop-carried dependence, apply ISS-Dep until all inner loops are parallel, or no useful split is found.
3. Mark all inner loops for pipelining, and synthesize the code using HLS. For each inner loop with II > 1, apply the algorithm for bank conflict elimination through splitting, and use this modified code as input to HLS tools.

This approach, while significantly outperforming in some cases previous work on PolyOpt/HLS [28] that was limited to only step (1) above, still has potential room for improvement in terms of performance. For instance we did not explore the impact of tile size in the final computation performance, nor different trade-offs between more aggressive array partitioning (increased BRAM) versus more aggressive loop splitting (increased LUT/FF/DSP). Numerous simplifications we did to ensure always-legal splitting can also lead to missing parallelism opportunities. On the other hand, our approach often achieves an II of 1 for at least some of inner loops that are pipelined, as shown in ISS-Res rows in Sec. 6, where an II of 1.5 is reported when at least half of the inner-most loop iterations are pipelined with an II of 1, the rest of the iterations being put in a separate loop with II of 2.

6. Experimental Results

In this section, we present our experimental results using a set of computation kernels and applications. We first discuss the experiments setup and evaluated benchmarks. Then, we show the performance improvements of our proposed optimization technique.

6.1 Experimental Setup

Benchmark description We use 15 linear algebra kernels and applications from PolyBench/C 3.2 [3]. Double precision floating point is used as the default data type in computations as in the original code. Computations (tiles) operate on small datasets (array sizes are typically 500 for single dimensional arrays and 128×128 for two-dimensional arrays) so that data fits in on-chip memory. A description of each benchmark can be found in Table 1.

Program variants For each benchmark, we compare the performance of four different program variants. The first variant is the original source code without any code transformation, serving as a baseline for further improvements. The second variant is generated by the polyhedral compiler infrastructure PolyOpt/HLS 0.2 (see [4] for a similar framework), which itself is based on PoCC 1.2 [2] to support affine loop transformations and loop tiling. The running time of the loop transformation computation in PolyOpt/HLS includes solving several Parametric Integer Linear Programs [18], which is done in no more than a few seconds for the tested benchmarks. Our proposed algorithms for ISS are quasi-linear in the number of polyhedra to process, and computes the result in a second or less. Techniques in [36] are used to optimize the code generation for HLS. Index set splitting techniques introduced in this paper are used to generate the last two variants, with ISS for loop dependence used in the third variant and ISS for both loop dependence and resource conflict used in the last variant. Methods in [25] [37] are used to take advantage of array partitioning opportunities that current Vivado HLS fails to exploit. In all versions, the innermost loops are marked for loop pipelining, and we also insert compilation pragmas for all variables which are written in the parallel loop iteration, to prevent conservative dependence analysis.

Circuit generation We use the Xilinx Kintex-7 FPGA device as the target hardware platform. All program variants are fed into Xilinx Vivado 2013.3 high-level synthesis, logic synthesis and physical

Table 1. Experimental results

Bmk.	Description	Version	II	Cycles	CP(ns)	LUT	FF	DSP	Sta. Pwr	Dyn. Pwr	Egy/mJ
2mm	Matrix-multiply D=α*A*B*C+β*D	Orig	5	21512194	7.981	1612	1410	14	0.156	0.013	2.23
		Affine	1	8335874	7.612	1782	1510	14	0.156	0.016	1.02
3mm	Matrix-multiply G=(A*B)*(C*D)	Orig	5	31948803	8.174	1600	1552	14	0.156	0.006	1.57
		Affine	1	636371	8.908	2580	2371	25	0.156	0.005	0.28
atax	Matrix Transpose and Vector Mult	Orig	5	1511502	8.257	1385	1093	14	0.156	0.019	0.24
		Affine	1	531852	7.726	1488	1174	17	0.156	0.018	0.07
bicg	Kernel of BiCGStab Linear Solver	Orig	5	1255502	8.176	1438	1158	14	0.156	0.012	0.12
		Affine	1	53185	7.763	1606	1428	17	0.156	0.011	0.05
doitgen	Multiresolution Analysis Kernel	Orig	5	5607425	7.828	1126	1024	14	0.156	0.009	0.22
		Affine	1	1114331	7.659	1769	1776	23	0.156	0.006	0.10
gemm	Matrix-multiply C = α.A.B + β.C	Orig	6	12582925	7.701	1225	1089	14	0.156	0.008	0.77
		Affine	1	2124418	8.062	1783	1753	29	0.156	0.015	0.26
gemver	Vector Mult. and Matrix Addition	Orig	5	3250551	7.902	2778	2427	30	0.156	0.007	0.18
		Affine	1	555991	7.791	3733	3656	57	0.156	0.014	0.06
gesummv	Scalar, Vector and Matrix Mult	Orig	5	1260501	7.705	1652	1541	14	0.156	0.014	0.17
		Affine	1	532737	7.705	1652	1541	29	0.156	0.026	0.11
mvt	Matrix Vector Product and Transpose	Orig	6	3000016	7.496	1371	1108	15	0.156	0.018	0.40
		Affine	1	265361	7.573	1897	1890	31	0.157	0.022	0.04
syrk	Symmetric rank-k operations	Orig	6	12599316	7.808	1397	1217	14	0.156	0.024	2.36
		Affine	1	2124418	8.028	1784	1793	29	0.156	0.034	0.58
syr2k	Symmetric rank-2k operations	Orig	10	20987924	8.123	1675	1415	14	0.156	0.019	3.24
		Affine	1	2126978	7.982	3055	3069	54	0.156	0.025	0.43
floyd-warshall	Finding Shortest Paths in a Graph	Orig	8	16777218	5.827	1085	791	3	0.156	0.009	1.08
		Affine	8	16980993	5.889	1182	852	3	0.156	0.012	1.20
		ISS-Dep	2	4407041	5.645	1435	1481	3	0.158	0.017	0.44
trmm	Triangular matrix-multiply	Orig	5	5642753	7.398	1387	1229	14	0.156	0.009	0.72
		Affine	5	3913057	7.418	2160	1964	14	0.156	0.008	0.86
		ISS-Dep	2	2101106	7.696	1374	1500	25	0.156	0.017	0.32
trisolv	Triangular Solver	Orig	5	637001	9.091	4418	2962	14	0.157	0.083	0.48
		Affine	2	266002	9.035	4445	2992	18	0.156	0.101	0.24
		ISS-Res	1.5	219002	8.799	5360	3575	18	0.157	0.107	0.21

implementation tools to generate bitstreams for FPGA. 10ns is used as the timing constraints for all circuit generation steps.

Optimization metrics We use FPGA-specific metrics to quantify the quality of each circuit generated by different program variants. The number of LUTs, FFs and DSPs are used to reflect the resource utilization of a design. All the resource utilization data are reported by Xilinx Vivado tool after the place-and-route step. Critical path delay and execution cycle are used to capture the performance of a design. Critical path delay is extracted from the post place-and-route Xilinx Vivado tool report while the execution cycle is reported by a cycle-accurate SystemC simulator with the target design and the testbench as the input. Switching activities of each net are traced by the simulator using value change dump (VCD) files for more accurate power estimation. Power data is reported by the Xilinx Vivado tool with the place-and-routed circuits and the circuit simulation traces as input.

6.2 Results

Table 1 describes all the raw data reported by the tool-chain including the various resource usage, critical path, execution cycles and power consumption. These results have been discussed in previous sections of the paper.

In addition to the raw metrics, we also to measure the total latency and energy consumed. Since that for the benchmark selected in this paper, a large fraction of power consumed is static power, we used normalized energy [36] as the metrics to total energy consumption, where only a portion of the static power of the entire FPGA device are taking into account. These latency and normalized energy metrics are computed as follows.

$$
\begin{aligned}
Latency &= Critical\_Path * Execution\_Cycles \\
Area_{ratio} &= \max\left(\frac{LUT_{used}}{LUT_{Avail.}}, \frac{FF_{used}}{FF_{Avail.}}, \frac{DSP_{used}}{DSP_{Avail.}}\right) \\
Energy_{norm} &= (Area_{ratio} * Pwr_{Static} + Pwr_{Dyn.}) * Lat.
\end{aligned}
$$

Interestingly, affine loop transformations may decrease the execution latency for some benchmarks (e.g. floyd-warshall and trmm) due to the overhead of loop tiling. For other benchmarks, affine transformation can reduce the execution latency significantly. Power consumption and resource utilization will increase after aggressive loop pipelining but not proportional to the throughput boost since the resources are typically underutilized in the unpipelined implementations.

7. Conclusions and Future Work

Designers often have to perform a number of explicit source-code modifications to transform the original code to HLS-friendly code to improve the QoR of the HLS tool. Recent research on polyhedral optimization framework has shown great potential to automate these manual code transformations. In this work, we investigated the particular problem of improving the performance of affine programs that have been already optimized for data reuse and communication/computation overlapping, focusing on the computation part exclusively.

We presented detailed performance analysis of a tiling-driven transformation framework, highlighting its limitations in only some cases. We have developed and evaluated a customized method tailored for HLS purpose to cope with these limitations, including the optimization of memory port conflicts.

As future work, we will investigate the trade-off between decreasing latency versus increasing resource usage such as LUTs and flip-flops. Indeed, our method can generate significant resource usage overhead, which may not be compensated by the latency improvement in a maximal performance scenario where we aim to cover the entire area with accelerator replications. We will also investigate the comparison of our work with various array partitioning techniques, both in terms of resource usage and latency benefit. One current limitation of our work is the generation of multiple loops from a single loop, which limits the ability of Vivado to implement effective resource sharing.

Acknowledgment This work was supported in part by the National High Technology Research and Development Program of China 2012AA010902, RFDP 20110001110099 and 20110001120132, and NSFC 61103028; the US National Science Foundation through awards 0926127 and 1321147; and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] http://www.xilinx.com/products/design-tools/vivado/index.htm.

[2] Pocc 1.2. http://pocc.sourceforge.net.

[3] Polybench 3.2. http://polybench.sourceforge.net.

[4] PolyOpt/HLS 0.1. http://www.cs.ucla.edu/ pouchet/software/polyopthls/.

[5] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *ESOP*, volume 300, pages 221–235. Springer, 1988.

[6] C. Alias, A. Darte, and A. Plesco. Optimizing ddr-sdram communications at c-level for automatically-generated hardware accelerators an experience with the altera c2h hls tool. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 329 –332, july 2010.

[7] C. Alias, A. Darte, and A. Plesco. Optimizing remote accesses for offloaded kernels: application to high-level synthesis for fpga. *SIGPLAN Not.*, 47(8):285–286, Feb. 2012.

[8] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[9] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[10] 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, Sept. 2004.

[11] S. Bayliss and G. A. Constantinides. Optimizing sdram bandwidth for custom fpga loop accelerators. In *ACM/SIGDA Intl. symp. on Field Programmable Gate Arrays*, pages 195–204, New York, NY, USA, 2012. ACM.

[12] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *ETAPS CC*, Apr. 2008.

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

[14] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 16(2):15:1–15:25, Apr. 2011.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473 –491, april 2011.

[16] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Comput.*, 54:1242–1257, October 2005.

[17] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the defacto system. In *LCPC'03*, pages 52–70. 2003.

[18] P. Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988.

[19] 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.

[20] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.

[21] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *Int. J. of Parallel Programming*, 28(6):607–631, 2000.

[22] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware design methodology with the Alpha language. In *FDL'01*, Lyon, France, Sept. 2001.

[23] B. D. T. Inc. An independent evaluation of: High-level synthesis tools for xilinx fpgas, 2010.

[24] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[25] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, pages 488–495, 2012.

[26] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012, Jan. 2012.

[27] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung. Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: A geometric programming framework. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(3):305–315, Mar. 2009.

[28] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays*, Monterey, California, Feb. 2013. ACM Press.

[29] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in fpga-based systems. In *Programming Language Design and Implementation*, 2002.

[30] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *IEEE PACT*, pages 327–337, 2009.

[31] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, pages 299–302, 2010.

[32] Y. Wang, P. Li, and J. Cong. Theory and algorithm for generalized multidimensional memory partitioning in high-level synthesis. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (to appear)*, FPGA'14, 2014.

[33] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 12:1–12:8, 2013.

[34] M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.

[35] Z. Zhang, Y. Fan, W. Jiang, G. han, and J. Cong. Autopilot: a platform-based esl synthesis system. In *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[36] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving Polyhedral Code Generation for High-Level Synthesis. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2013.

[37] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving High Level Synthesis Optimization Opportunity Through Polyhedral Transformations. In *Proc. of the ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays (FPGA'13)*, 2013.