Improving Compiler Scalability:
Optimizing Large Programs at Small Price

Sanyam Mehta ∗ Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota Twin Cities, MN, USA
sanyam.mehta@gmail.com yew@cs.umn.edu

Abstract
Compiler scalability is a well known problem: reasoning about the application of useful optimizations over large program scopes consumes too much time and memory during compilation. This problem is exacerbated in polyhedral compilers that use powerful yet costly integer programming algorithms to compose loop optimizations. As a result, the benefits that a polyhedral compiler has to offer to programs such as real scientific applications that contain sequences of loop nests, remain impractical for the common users.

In this work, we address this scalability problem in polyhedral compilers. We identify three causes of unscalability, each of which stems from large number of statements and dependences in the program scope. We propose a one-shot solution to the problem by reducing the effective number of statements and dependences as seen by the compiler. We achieve this by representing a sequence of statements in a program by a single super-statement. This set of super-statements exposes the minimum sufficient constraints to the Integer Linear Programming (ILP) solver for finding correct optimizations. We implement our approach in the PLaTo polyhedral compiler and find that it condenses the program statements and program dependences by factors of 4.7x and 6.4x, respectively, averaged over 9 hot regions (ranging from 48 to 121 statements) in 5 real applications. As a result, the improvements in time and memory requirement for compilation are 268x and 20x, respectively, over the latest version of the PLaTo compiler. The final compile times are comparable to the Intel compiler while the performance is 1.92x better on average due to the latter’s conservative approach to loop optimization.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors - Compilers; Optimization

Keywords Compiler scalability; optimization; polyhedral model; statement condensation; O-molecule;

∗ This author is currently affiliated with the University of Illinois at Urbana Champaign.

1. Introduction
Compiler scalability is a well known problem: precise analysis of large program scopes to reason about the application of optimizations leads to a significant increase in the compile time and memory requirement. As a result, production compilers choose to limit optimization to small scopes. In other words, they trade performance for compile time (or, equivalently, programmer productivity [25]). However, with the era of multiple (many) cores on chip and corresponding accentuation of the memory wall and bandwidth wall, there is a renewed focus on high-level compiler optimizations for not only extracting parallelism but also for improving memory performance (such as through the loop fusion optimization that merges multiple loop nests to achieve data reuse). As a result, analyzing large program scopes to exploit data reuse opportunities spanning multiple nests of loops, or in other words, performing global program transformations, is necessitated. In fact, our experiments (Section 6) confirm the immense potential for improving the memory (and hence, parallel) performance of applications through such global program transformations. For example, the applu application from the SPEC OMP2012 Suite achieves an overall parallel performance speedup of 2.17x after loop fusion optimization over the Intel compiler, when run on an 8-core Intel Xeon processor.

Having evolved over the last two decades, polyhedral compilers, armed with exact dependence analysis and seamless transformation composition, have proven adept at performing global program transformation. This is especially useful because their traditional counterparts (including production compilers) are plagued by some of the artificial limitations such as non-conformable loop bounds, limitations of phase ordering and difficulty in composing transformations. This has restricted the traditional compilers to optimizing individual loops, or merely fusing consecutive loops with same loop bounds and loop order [8, 21]. However, while the polyhedral compilers have shown promise in effectively optimizing large scopes [19], the compile time and memory requirement is prohibitively expensive. The unscalability in polyhedral compilers stems from the massive (fifth degree polynomial) increase in compile time with the number of statements [26]. This unscalability of the employed techniques in polyhedral compilers has thus been recognized as a key problem, particularly as polyhedral compilers are becoming increasingly useful [3, 13, 23, 28].

Traditional work on compiler scalability has focused upon either reducing the cost of analyzing each dependence [12, 24] or the number of dependences to be analyzed [14], or both [17]. The compilers perform such an analysis to find the Program Dependence Graph (PDG) [11]. The PDG can then be used further to reason the application of program optimizations. However, it is this step of applying optimizations, especially when it involves large scopes such as in loop fusion, that is much more time (and memory) con-
sioning than merely analyzing dependences. For example, optimizing the computationally-intensive subroutine, \textit{rhs} (containing 106 statements within 3 large loop nests), in the \textit{applu} benchmark application from the SPEC OMP2012 Suite consumes nearly 3 hours using the PLuTo [4] polyhedral compiler whereas the instance-wise dependence analysis takes less than a second. Similar is observed in multiple scientific applications that contain significant opportunity for data reuse through global program transformation. Thus, we argue that the problem of compiler (un)scalability is better defined as large compile times observed when performing the actual global transformation as opposed to dependence analysis as is commonly believed; clearly, this unscalability has not yet been addressed by the compiler community.

Through this work, we make the following contributions,

1. We identify three causes of unscalability within state-of-the-art (polyhedral) compilers.
2. We address all those causes of unscalability that basically stem from large program size in the number of statements through our one-shot solution. In our solution, we perform optimization over statement-blocks (which we call \textit{Optimization Molecules} or \textit{O-molecules}) instead of individual statements as in existing state-of-the-art polyhedral compilers. In other words, the program is represented through these \textit{O-molecules} that are much fewer in number as compared to statements, allowing for faster optimizations.
3. We evaluate our proposed strategy on 9 hot regions in 5 real applications from the SPEC and NAS Parallel Benchmark Suites, and achieve an average improvement of 268x and 20x in the compile time and memory requirement, respectively, over the state-of-the-art PLuTo polyhedral compiler. The PLuTo compiler by itself achieves a performance improvement of 1.92x on (geometric) average over the Intel compiler (\textit{ifort}) for the 9 hot regions. But, all this performance is at the cost of very long compile time. As a result of implementing our work in PLuTo, the compile time of such programs is now comparable to \textit{ifort} (i.e. a few seconds as compared to hours originally), while achieving the same optimization (and performance) as the original version of PLuTo. Thus, we believe that our work will make the use of polyhedral compilers practical for the common user.

The rest of the paper is organized as follows. The following section (Section 2) explains the key insight that led to the development of our proposed strategy of condensing a program into O-molecules for faster compilation. Section 3 provides a relevant introduction to the polyhedral framework. Section 4 then discusses the precise causes of the unscalability in polyhedral compilers while optimizing large programs with many statements. This is followed by a detailed discussion of the implementation of our technique within the polyhedral framework in Section 5. Section 6 compares the performance of our technique with regards to compile time and memory requirement against a state-of-the-art polyhedral compiler (PLuTo), and also the Intel production compiler. The related work is discussed in Section 7, and Section 8 concludes this work.

2. Key Insight

Consider the example program shown in Figure 1a. The program is constructed to represent some of the characteristic features in real application programs. These features include frequent updates to arrays such as \textit{rsd1} and \textit{rsd2}, and also frequent use of temporary variables such as \textit{flux1} and \textit{flux2} as shown in the figure.

Conventionally, loop optimizations in a (polyhedral) compiler are applied at the granularity of individual statements. In theory, this allows to achieve the optimal transformed program since each statement has the maximum degree of freedom of transformation in such an approach. However, in this work, we deviate from this conventional approach to favor compiler scalability. The essential insight is that we coarsen the granularity of optimization structures from individual statements to statement-blocks, which we call O-molecules. We define an O-molecule as a set of successive statements in the original program that have the same iteration domain (i.e. same enclosing loops). For example, statements S1-S3 have the same iteration domain, and therefore comprise an O-molecule. Similarly, S4-S5, S6-S8, and S9-10 are the other 3 O-molecules in the example, while statements S1-S5 do not comprise an O-molecule since statements S1-S3 and statements S4-S5 have different iteration domains (in loop \textit{k}). As a result, the compiler now sees the program as a collection of O-molecules rather than statements, where each O-molecule is represented by a single \textit{super-statement}, \textit{Smol}. Consequently, all statements in an O-molecule are forced to undergo the same transformation as \textit{Smol}. In summary, since O-molecules are much fewer than statements, program compilation is very fast. We refer to this technique as \textit{statement condensation} in the paper.

Statement condensation further allows to condense the set of dependences to be analyzed. This is because each dependence enforces constraints to limit the degree of freedom of transformation of its source and destination statement, but the use of O-molecules requires any dependence to only impose those constraints on its source and destination \textit{O-molecule} instead. Thus, dependences that have the same source and destination O-molecules, and the same dependence distance (on all loops) are grouped into a single class with a single representative dependence, \textit{Drep}. The dependence graph can thus be seen as a graph of super-statements (\textit{Smol}s) as vertices and \textit{Drep}s as directed edges representing dependences between \textit{Smol}s. Since many dependences in real applications have small dependence distances and are therefore similar [12], there is significant opportunity to reduce the effective number of dependences through such dependence condensation.

Ensuring correctness. Since any statement in an O-molecule undergoes the same transformation as the molecule’s \textit{Smol}, the correctness of the transformed program is ensured by imposing dependence constraints collectively incurred by all statements on \textit{Smol}. As a result, some statements in the O-molecule are more constrained than they need to be. However, in no case shall a statement be less constrained than it should be, and hence a correct transformation is ensured.

Preserving optimization opportunities. Statement condensation may result in loss of optimization opportunities for reasons noted above. We analyze that there are two possible scenarios of such a happening. They are, (1) when local loop optimizations (such as partial loop distribution to enable vectorization) could improve performance, and (2) when a finer granularity movement of statements could allow better data reuse in the transformed program. However, in the former case, final program performance is not hurt since backend compilers such as the Intel compiler are adept at performing local optimizations (that involve easy and cheap analysis) such as vectorization, and can do so on the transformed source program. Moreover, the scenario in the latter case only results when selective statements in a loop provide better data reuse upon being fused into another loop’s body and it is also legal to do so. We find that this is rarely the case. Among 5 applications that we use for our experiments, we found that such a scenario occurs in one application (\textit{swim}), and using O-molecules only results in minimal performance degradation. The benefits in compile time, however, would much more than compensate in such cases.

Choice of O-molecules as optimization structures. When targeting scalability while preserving global optimization opportunities, there could be other choices of optimization structures than our
O-molecules, which are either finer or coarser than O-molecules. Note that current polyhedral compilers use the finest possible optimization structures, individual statements, and therefore have poor scalability.

When considering finer structures, one attractive alternative is to choose Strongly Connected Components in the innermost loop as optimization structures. An O-molecule can contain multiple SCCs. Each SCC can thus be considered as an O-atom. An O-atom is attractive because forcing only the statements in an O-atom to have the same transformation, will not miss out on local optimizations such as vectorization. However, we use O-molecules in this work for they offer much more opportunity for statement condensation. An O-molecule also allows for more pronounced dependence condensation because we observe that in most programs, statements from different SCCs tend to exhibit similar dependence patterns.

Coarser optimization structures would imply that each structure may include statements that have different iteration domains in the original program. Since the technique of statement condensation forces all statements in each optimization structure to have the same transformation, this strategy assumes that it would be feasible to fuse statements that are in different loops originally (such as Statements S1-S5 in Figure 1a). Clearly, it is not always possible to determine this easily before the actual transformation. We choose statements that have the same iteration domain to form an O-molecule since we know apriori that it is feasible to obtain a correct schedule (e.g., as it exists in the original program) where all statements in an O-molecule will have the same transformation.

**Practical utility of O-molecules.** This notion of ‘O-molecule’ proves very effective in reducing the complexity of analysis without losing optimization opportunities. As a result of breaking our example program into O-molecules, the effective number of statements analyzed reduce to 4 (Super-statements S1,S2 in the 4 O-molecules in Figure 1c) from 10 in the original program. This reduction leads to a corresponding reduction in the number of variables in the ILP (Integer Linear Programming) formulation for finding the program transformation (explained in more detail in Section 4), leading to a significant reduction in analysis complexity. However, to ensure program correctness of resulting transformation with such a reduced set of (super-)statements, the chosen each Smol in each O-molecule must accommodate the transformation restricting constraints that are imposed by dependences involving all statements in the O-molecule. Thus, a super-statement may effectively contain multiple reads/writes as shown in Figure 1c.

Figures 1b and 1d explain how the use of O-molecules helps to reduce the number of dependences for program analysis. Figure 1b shows the dependence vectors between statements from loops 1 and 2, and those between statements from loops 2 and 4. Using O-molecules, the granularity of optimization is coarsened to individual loops, and thus each loop can be considered to be represented by its super-statement (with possibly multiple reads/writes) as shown in Figure 1d. Thus, for correct program transformation, it is sufficient to only consider those dependences that expose unique

---

1 In a loop, the cyclically dependent statements form a Strongly Connected Component or an SCC [20]. Statements in an SCC cannot therefore be separated (distributed) in different loops.
The dependence polyhedron not only captures the iteration domain of a loop but also expresses algebraic dependencies as a set of linear inequalities. To model a composition of complex transformations as a single affine function ($T$) to represent a composition of loop transformations for the entire SCoP, each dimension or level of this multi-dimensional affine function is represented by $\phi_1$ and is defined as follows:

$$\phi_1(s) = (c_1^s c_2^s ... c_m^s) (i s) + c_0^s$$  \hspace{1cm} (1)

where $i s$ is the loop iteration vector for statement $S$, and $c_0^s ... c_m^s$ are constants, better known as coefficients of the hyperplane. Figure 2d shows the multi-dimensional affine functions ($T_S$ through $T_{S_4}$) for each of the four statements of the gemver benchmark, where each function has 3 dimensions or levels represented by $\phi^1$, $\phi^2$, and $\phi^3$, respectively.

The one-dimensional affine transform ($\phi$) for each statement specifies a loop hyperplane. A loop hyperplane is an $n - 1$ dimensional sub-space of an $n$ dimensional space represented by the normal $(c_1^s c_2^s ... c_m^s) \neq 0$. A legal loop hyperplane corresponds to a loop in the transformed program. For example, for gemver, $\phi_{S_2} = (1 0 0) (i j)^T = i$, and it represents a hyperplane that corresponds to the outermost loop (i-loop) for statement $S_2$ as shown in Figure 2d. A legal hyperplane does not violate any unsatisfied dependence, $e_{S_i \rightarrow S_j}$, at that loop level, i.e.

$$\phi_{S_i}(\vec{s}) - \phi_{S_j}(\vec{s}) \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_{e_{S_i \rightarrow S_j}}$$  \hspace{1cm} (2)

where $\vec{s}$ (source) and $\vec{t}$ (target) are instances of statements $S_i$ and $S_j$, respectively. The above condition implies that the loop hyperplane preserves the direction of dependences between two instances of statements $S_i$ and $S_j$.

The above conditions (Equation 2), when expanded becomes,

$$\left( c_1^s c_2^s ... c_m^s \right) \vec{t} - \left( c_1^s c_2^s ... c_m^s \right) \vec{s} \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_{e_{S_i \rightarrow S_j}}$$  \hspace{1cm} (3)

This is, however, non-linear in the unknown co-efficients of the phis and loop index variables. Thus, the affine form of the Farkas lemma is used for linearizing this legality condition.

Lemma 1 (Affine form of the Farkas lemma). If a non-empty polyhedron is defined by $p$ inequalities or faces,

$$a_k \vec{x} + b_k \geq 0, \quad k = 1, p$$

then, an affine form $\psi$ is non-negative everywhere in that polyhedron iff it is a non-negative linear combination of the faces:

Figure 2: Overview of the Polyhedral Framework.
\[ \psi(\vec{x}) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k \vec{x} + b_k), \quad \lambda_0, \lambda_1, ..., \lambda_p \geq 0 \quad (5) \]

where the \( \lambda \)s in Equation 5 are called the Farkas multipliers.

The legality condition in Equation 3 thus takes the following form - the non-linear form in terms of the loop variables (Equation 3) is now expressed as a non-negative linear combination of the faces of the dependence polyhedron through the use of Farkas lemma.

\[ \left( c S_j \ c S_i \ ... \ m S_i \right) \vec{f} - \left( c S_j \ c S_i \ ... \ m S_i \right) \vec{s} \equiv \lambda_0 + \sum_{k=1}^{m} \lambda_k P_{E_k^k}, \quad \lambda_0 \geq 0 \quad (6) \]

Now, the coefficients (denoting a hyperplane) on the LHS and RHS can be equated to get rid of the loop variables. It is important to note that the legality condition has to be satisfied for every dependence in the SCoP, and thus Farkas lemma is also applied for every dependence. The resulting constraints (linear in the coefficients of the phis) are aggregated. For the purpose of eliminating the Farkas multipliers, Fourier-Motzkin Elimination (FME) is employed.

In the polyhedral model, the loop hyperplanes are found such that the maximum possible number of statements can be fused. To support fusion, individual statements may selectively undergo loop interchange, shifting, peeling, etc. For example, statement S1 undergos loop interchange for successful fusion with statement S2 in the gemver code as shown in Figure 2d. Further, if some fusion-preventing dependences exist, then the involved SCCs are selectively distributed. For example, the existence of a fusion-preventing dependence between S2 and S3 leads to their distribution to different loops. In other words, the step of finding loop hyperplanes implicitly leads to all high-level optimizations including fusion, interchange, shifting, peeling. Loop tiling too, is performed after all loop hyperplanes are found, and relies on information such as perm utability of the loops found. This work adds scalability to the key transformation-compute step within polyhedral compilers, and therefore to all high-level loop optimizations.

4. Causes of Unscalability

The bulk of the time spent in compilation through a polyhedral compiler is in 3 distinct phases. These are, (1) constructing the set of constraints in the coefficients of the statement-wise loop hyperplanes from the dependence polyhedra, (2) solving these linear constraints using an Integer Linear Programming (ILP) solver to obtain the transformation matrix (with individual loop hyperplanes) for each program statement, and (3) incorporating further constraints to ensure linear independence of the next hyperplane solution with those already found. In this section, we describe how each of these phases proves to be a culprit causing the unscalability of polyhedral compilers.

4.1 First Culprit: Phase I - Constructing legality constraints

As noted in Section 3, the linearized legality condition in Equation 6 must be satisfied for every dependence in the SCoP. Thus, the number of constraints increase linearly with number of dependences, which themselves increase quadratically with the number of statements. This leads to quadratic increase in the number of constraints with program statements. Consequently, both the compile time and memory requirement increase significantly.

Increase in compile time. As discussed in Section 3, the FME method is used to eliminate the Farkas multipliers when linearizing the legality condition. Each of the multipliers are eliminated one at a time. Running an elimination step over \( n \) inequalities results in an increase in the number of inequalities (to a maximum of \( n^2/4 \) inequalities [7]) for subsequent steps. The number of these steps are proportional to the number of statements, and thus in a large program, a considerable time is spent to eliminate all Farkas multipliers. Furthermore, this step of eliminating Farkas multipliers for linearizing legality condition is performed for every dependence and thus the contribution of Phase-I to the overall compile time is notable.

Increase in memory requirement. Phase-I is the only memory consuming phase in the entire compilation. The memory is used to hold the constraints contributed by each dependence edge. These constraints are input to the ILP solver to find hyperplane solutions. Since a dependence edge can enforce constraints on the transformation coefficients of any two program statements, the total number of variables used to express all the constraints (from all dependences) in a matrix (we call it the Constraint Matrix, \( C \)) is \( \text{dim} \cdot |V| \), where \( \text{dim} \) denotes statement dimensionality. Thus, the width of the Constraint Matrix, \( C \), is \( \text{dim} \cdot |V| \). The height of the matrix is given by \( k \cdot |E| \) since each dependence in \( E \) contributes a fixed number of constraints, \( k \). Thanks to elimination of all Farkas multipliers through Fourier Motzkin Elimination, the size of \( k \) is small - it is proportional to the sum of dimensionalities of source and destination statements. However still, the overall memory requirement becomes \( \text{dim}^2 \cdot |V| \cdot |E| \), or \( |V|^3 \). We find that for programs that contain around 100 statements, the overall memory requirement is already of the order of a few gigabytes.

4.2 Second Culprit: Phase II - Using an ILP solver to find legal hyperplanes

As mentioned above, the number of constraints (inequalities) input to the ILP solver equal \( k \cdot |E| \), and the number of variables involved are \( \text{dim} \cdot |V| \). We thus estimate the (time) complexity of the ILP solver as follows. Let \( Z(m, n) \) be the complexity of ILP using the Simplex algorithm with \( m \) constraints and \( n \) variables. Then, on a typical input, \( Z(m, n) = O,(m+n)mn \) on average [26]. In our ILP, this amounts to a complexity of \( |E| |V| |E| |V| \), or \( |V|^2 \), i.e. a quintic complexity in the number of statements. This can lead to large compilation times for SCoPs with many statements.

4.3 Third Culprit: Phase III - Finding linearly independent hyperplanes

Using an ILP solver, we find solutions to statement-wise loop hyperplanes one at a time, starting with the hyperplane corresponding to the outermost loop. However, it is required that we find as many independent hyperplanes as the dimensionality of the polytope (i.e. the maximum depth of any statement within the SCoP). Thus, before a hyperplane for the next loop-level is found, it is essential to augment the ILP formulation with additional constraints that ensure that the next hyperplane found is linearly independent to those previously found. For this purpose, a sub-space (\( H_S^S \)) orthogonal to \( H_S \) (where \( H_S \) represents the hyperplane solutions found so far), is constructed as follows [5]:

\[ H_S^S = I - H_S^T (H_S H_S^T)^{-1} H_S \quad (7) \]

Clearly, \( H_S^S H_S = I \). For linear independence of the next hyperplane \( h^* \) with all hyperplanes in \( H_S \), the following must hold:

\[ H_S^S h^* \neq 0 \quad (8) \]

Furthermore, these additional constraints need to be constructed in such a way that when combined with the existing legality con-
strains, a solution of the coefficients of the next hyperplane is still possible. This involves computing the intersection of the integer sets given by Equation 8 and the constraints in the coefficients of hyperplanes obtained by eliminating all Farkas multipliers in Equation 6. This set intersection computed using the Integer Set Library (ISL) [29] further involves simplification of the constraints over many steps until the actual intersection is computed. The number of these simplification steps are proportional to the number of dependences, $|E|$. Further, each step has a complexity of $O(|E|^2)$.

Since the hyperplanes found are statement-wise, the linear independence constraints are also statement-wise, and thus, the complexity of this step becomes, $O(|E|^2|V|)$, or $O(V^3)$, i.e. quintic in the number of statements as for Phase II. However, unlike Phase II, this quintic complexity of Phase III is more realistic (i.e. does not depend on other factors such as implementation details, etc.), and we actually find this step to be most time consuming for large programs.

Thus, we find that among all three culprits, the common motive behind the unscalability problem is the large number of statements within a single SCoP, which also amounts to large numbers of dependences. One solution to the problem could be to break a single large SCoP into multiple smaller SCoPs, but that breaks the very purpose of achieving global program transformations such as fusing multiple loop nests for reuse. Thus, we propose our solution that solves unscalability without losing important global optimization opportunities. We attack the problem at the root, i.e. we effectively cut down the number of statements, and also the number of dependences in a program through our Statement Condensation and Dependence Condensation algorithms. The program is then represented with just super-statements and representative dependences, which still allows to correctly reason about the application of any particular loop optimization. This cut in the number of statements (and dependences) serves as a one-shot solution to all the above causes of unscalability, and allows to compile SCoPs with many statements in affordable time and memory budget. The following section details the two algorithms used to implement our solution.

5. Implementation

As discussed in the earlier sections, program condensation for the purpose of speeding up high-level loop optimizations, involves two steps, (1) Dependence condensation - considering only unique dependences (D_{rep}) between O-molecules, and (2) Statement condensation - considering only the super-statement (S_{mol}) per O-molecule for computing transformation. The two steps are closely related. In the polyhedral framework, each dependence contributes a subset of (transformation-restricting) constraints on the coefficients of statement-wise affine functions (representing hyperplanes). To ensure correctness of transformation performed using the condensed program, each $S_{mol}$ in the program must absorb all the (unique) transformation-restricting constraints imposed on any statement in the respective O-molecule, or in other words, those constraints that are imposed by all D_{rep} associated with that O-molecule. The following example elucidates the idea.

Consider the example program in Figure 3a. From the definition of an O-molecule (in Section 2), we find that statements S1-S4 and S5-S7 are the two O-molecules in the program. Further, we construct the two super-statements (say, $S_{M1}$ and $S_{M2}$, respectively) for these O-molecules by bedecking them with those reads and writes that become the source or destination of any of the D_{rep} associated with the O-molecule (similar to that in Figure 1c). Figure 3b shows a correctly transformed program that uses super-statements to represent all transformation-restricting constraints incurred by the O-molecules. Note that the two O-molecules are fused in the transformed program. Essentially, this fusion is rendered feasible after shifting statements S5-S7 by 1 iteration in loop $k$.

On the other hand, Figure 3c shows a case of an incorrectly transformed program that results when $S_{M1}$ and $S_{M2}$ fails to account for constraints incurred by one of the D_{rep} associated with its O-molecule. Precisely, the transformed program in Figure 3c results when the D_{rep} denoting the dependence class with dependences between statements S3-S6 (on flux1 in S4) and S4-S7 (on flux2 in S4) is ignored in the dependence graph. Thus, loop fusion is incorrectly deemed feasible without shifting in loop $k$, resulting in incorrect transformation.

Algorithm 1 groups dependences involving the same source and destination O-molecules, and with the same dependence distance along all loops or dimensions. A single dependence (D_{rep}) from each such group is chosen to contribute to the dependence-wise l-egality constraints given by Equation 6 and redundant dependences (whose dependence distances along all loops are the same as one of the chosen dependences) are ignored for computing transformation. An important point to note here is that since the algorithm only checks for the equality of dependence distances of two depen-
After the set of representative dependences, $D_{rep}$, is obtained from Algorithm 1, all constraints from $D_{rep}$ involving an O-molecule are transferred to its super-statement, $S_{mol}$. This effectively amounts to multiple reads and writes in $S_{mol}$ as shown earlier in Figure 1c. As a result, the program semantics for the purpose of loop optimizations can be captured by just the set of super-statements.

Algorithm 2 details how this is actually effected by showing a code excerpt. In the code excerpt, the matrix $cst$ accumulates the dependence-wise legality constraints. Columns of $cst$ correspond to the variables input to the ILP, i.e., the coefficients of the hyperplane of each statement in the original program, or each O-molecule in the condensed program. Rows of $cst$ represent an individual constraint or inequality, and is a function of the number of dependences. Originally, each program dependence would add constraints in $cst$, but we cut down those dependences to the set $D_{rep}$ using Algorithm 1. Further, each dependence in $D_{rep}$ adds constraints to only the variables representing coefficients of $S_{mol}$. This can be seen in lines 5 and 6 where the offsets in each row of the constraint matrix can now vary to a maximum of statement dimensionality (denoted as, $dim = $ loop depth) times the number of O-molecules. In other words, the number of variables input to the ILP are reduced by a factor of $\frac{\text{Number of statements}}{\text{Number of O-molecules}}$.

6. Experimental Evaluation

In this section, we compare the time and memory consumed for compilation by the state-of-the-art PLuTo polyhedral compiler, the Intel compiler, and our work on statement and dependence condensation (which we refer to as scalefuse). The Intel compiler, although very efficient in terms of time and memory consumption for compilation, is very conservative in analyzing large scopes for global program transformation. It cannot therefore capitalize upon most optimization opportunities. We also compare the performance results of the Intel compiler and the PLuTo polyhedral compiler to reveal the importance of making polyhedral techniques scalable.

6.1 Setup

The test programs were compiled (and run) on an Intel Xeon processor (E5-2650) with 8 Sandy Bridge-EP cores, operating at 2.0GHz. The processor has private L1 (32KB per core) and L2 (256KB per core) caches and a 20MB shared L3 cache, and 8GB memory. We implement our work in PolyOpt [22], a source-to-source optimizer that can automatically detect SCoPs in programs and uses PLuTo for loop optimizations on the identified SCoPs. We use the Intel compiler v14 as the backend compiler with ‘-O3’ and ‘-parallel-openmp’ as the compiler options.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Benchmark Suite</th>
<th>Category</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>apps</td>
<td>NPB/OMP2012</td>
<td>Computational Fluid Dynamics (CFD)</td>
<td>N=102, CLASS B</td>
</tr>
<tr>
<td>sp</td>
<td>NPB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zeusmp</td>
<td>CPU2006</td>
<td>Simulation of astrophysical phenomena</td>
<td>Reference input</td>
</tr>
<tr>
<td>twomas</td>
<td>OpenMP2012</td>
<td>Weather prediction</td>
<td>Reference input</td>
</tr>
</tbody>
</table>

6.2 Benchmarks

The benchmarks used in these experiments include 5 real application programs used in the scientific community, and in other published research. A brief description of these benchmarks is given in Table 1. In these 5 applications, we identify 9 hot regions spanning 6 subroutines. While our proposed technique is more generally applicable, we choose the given set of applications on 2 considerations, (1) they are part of a popular benchmark suite that is representative of real applications, and (2) the hot regions in them constitute a SCoP, and thus be amenable to optimizations by a polyhedral compiler. It is important to note that each of these 9 SCoPs contain multiple large loop-nests with the number of statements in each SCoP ranging from 48 to 121. Such large sequences of statements are known to be hard for the compilers to optimize, both due to complex source codes and large compile times. Particularly, polyhedral compilers have not been shown to be effective on such large programs.

6.3 Results and Discussion

Table 2 compares the time and memory consumed for compilation by different compilers. It is important to note that the maximum memory required for any benchmark is around 6GB while the memory on the test processor is 8GB. This ensures that the compile times shown in our results are not influenced by frequent page faults that result when the memory used reaches the memory...
Table 2: Compile time, memory requirement of different compilers; Speedup (in compile time) of scalefuse wrt smartfuse

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Subroutine</th>
<th># statements</th>
<th># deps</th>
<th>dim</th>
<th>Compile time (s)/Memory req. (MB)</th>
<th>Speedup wrt smartfuse</th>
<th># S&lt;sub&gt;mol&lt;/sub&gt;</th>
<th># D&lt;sub&gt;rep&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>rhs</td>
<td>48</td>
<td>1419</td>
<td>4</td>
<td>4/116 303/703 4.11/61.88</td>
<td>74x</td>
<td>15</td>
<td>391</td>
</tr>
<tr>
<td>sp</td>
<td>rhs</td>
<td>50</td>
<td>1428</td>
<td>4</td>
<td>4/116 360/766 4.132/77.93</td>
<td>87x</td>
<td>15</td>
<td>391</td>
</tr>
<tr>
<td>lu</td>
<td>rhs</td>
<td>106</td>
<td>3033</td>
<td>4</td>
<td>1.97/1.7 11302/5542 15.39/92.22</td>
<td>734x</td>
<td>23</td>
<td>849</td>
</tr>
<tr>
<td>zeusmp</td>
<td>hsmoc.1</td>
<td>121</td>
<td>2660</td>
<td>3</td>
<td>5/141 4903.72/1207.8 11.76/40.12</td>
<td>417x</td>
<td>21</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>hsmoc.2</td>
<td>118</td>
<td>2493</td>
<td>3</td>
<td>5.9/151 6784.99/1256.26 11.7/44.21</td>
<td>578x</td>
<td>18</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>hsmoc.3</td>
<td>120</td>
<td>2296</td>
<td>3</td>
<td>5.7/143 8121.38/1643.15 13.87/55.59</td>
<td>585x</td>
<td>18</td>
<td>234</td>
</tr>
<tr>
<td></td>
<td>lorentz.1</td>
<td>98</td>
<td>2227</td>
<td>3</td>
<td>3.1/95 2.6550.48/2548.09 6.5/62.44</td>
<td>1006x</td>
<td>21</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>lorentz.2</td>
<td>92</td>
<td>2149</td>
<td>3</td>
<td>2.49/86 7759.11/3389.42 8.7/67.61</td>
<td>892x</td>
<td>18</td>
<td>247</td>
</tr>
<tr>
<td>swim</td>
<td>main</td>
<td>52</td>
<td>472</td>
<td>2</td>
<td>1.82/56.1 7.4/65.28 0.637/56.56</td>
<td>12.4x</td>
<td>12</td>
<td>152</td>
</tr>
</tbody>
</table>

The last two columns in Table 2 show the number of statements (# S<sub>mol</sub>) and representative dependences (# D<sub>rep</sub>) actually analyzed by scalefuse. On average, the number of statements and dependences analyzed reduce by factors of 4.7x and 6.4x, respectively. Since the time and memory complexity of the three key phases of compilation is polynomial with a large power in the number of statements, the gains from our proposed statement condensation (and dependence condensation) are large. In particular, the improvement in compile time in more pronounced since the compile time varies as $|V|^3$ while the memory requirement varies as $|V|^2$. Another interesting point to note is that the number of representative dependences (D<sub>rep</sub>) are larger in case of bt, sp, and applu benchmarks than the others because these benchmarks involve a lot more variables, while the subroutines in the zeusmp benchmark repeatedly reuse multiple temporary variables with the same name, revealing more opportunity for dependence condensation. In general, however, the trend is that the benchmarks with more number of statements (and dependences) witness a larger improvement from scalefuse.

The swim benchmark, although has almost the same number of statements as bt and sp, takes much less time to compile using PLuTo for it has much fewer dependences. This each statement is an SCC by itself (ignoring the outermost serial time loop from the transformation) due to the absence of circular dependences that appear with temporary variables. It is in fact for the same reason (small SCCs) that a finer granularity movement becomes possible in swim using PLuTo, giving slightly better performance than scalefuse. Thus, in such cases where the program contains very small SCCs as in swim, the user may as well choose to not use scalefuse since the compile times are anyway manageable. For larger programs that can take hours to yield a better transformation, scalefuse definitely proves useful.

Table 3 shows the time taken by each of the three phases that we identify as culprits for long compile times in Section 4. Note that the time taken for dependence analysis (not included in the table) is less than a second for all benchmarks, as stated earlier in...
Section 1. Clearly, `scalefuse` outperforms PLuTo considerably in all three phases. As discussed in Section 4, Phase III is the most time consuming step in compilation. Consequently, the most notable reduction through `scalefuse` is also for Phase III, i.e. the phase where PLuTo adds constraints to guarantee linear independence of future hyperplanes with those already found. Although Phase III in the latest versions of PLuTo is more conservative than earlier versions, it is necessary to do so to obtain feasible solutions. With `scalefuse`, it is now rendered scalable. In addition, the achieved scalability will allow `scalefuse` to model a much larger space of transformations within the polyhedral frameworks in the available compile time window since existing frameworks such as PLuTo only model a sub-space of affine transformations. Loop reversal, for example, is not modeled, but has been recently shown to be quite effective in enabling other transformations [6].

7. Related Work

The problem of unscalability of compilers has been well recognized in the literature. Much of the past work has been devoted to speeding up either the dependency testing through newer (faster, and less conservative) dependence tests, or the construction of the Program Dependence Graph (PDG) itself to be used later in program optimization. However, as discussed in Section 1, the bottleneck as far as compile time is concerned, comes from the transformation-compute phase that starts after the PDG is constructed. This is a more recent problem that has become particularly relevant in the context of polyhedral compilers that are capable of composing high-level global program optimizations.

**Dependence Testing.** Dependence testing answers the question whether two statements (or statement instances) refer to the same memory location across iterations. Most dependence tests are approximate, yet conservative, i.e. if a dependence exists, it is reported; however, false dependences may also be reported leading to (unnecessarily) constrained optimization. These include the fast GCD-test [2], the Banerjee test [31], the Power test [32], etc. There are also exact tests that give an exact solution, but at a higher computational cost, such as the Omega-test [24], and the PIP-test [9].

**Instance-wise dependence analysis** [27] subsumes previous work in dependence testing for it precisely tells which instances of the involved statements are in a dependence. Thus, this instance-wise analysis is more powerful dependence abstraction than its less precise counterparts used in previously proposed testing algorithms, such as (1) dependence levels [1] that tells which loops in the loop-nest carry a dependence, and (2) distance vectors [16, 33] that shows the difference of the loop counters of dependent instances. As a result, it aids the application of finer program transformations such as those in the polyhedral compilers.

**Scalable PDG construction.** In order to speed up the construction of PDG while not sacrificing precision, Maydan et al. [17] and Goff et al. [12] propose to use a collection of different dependence tests. They show that faster tests such as the GCD-test suffice for most dependences, while expensive tests can be used for remaining dependences. Maydan et al. also use memoization to avoid initiating a test on those dependences that are similar to those seen already. Early work on fusion [15, 18] in traditional compilers proposes to use individual loops as fusion candidates to cut down the analysis complexity, but they do not consider loop fusion in conjunction with other important optimizations such as interchange, and are subjected to other aforementioned limitations which makes fusion impractical in real applications.

Recently, [14] propose a method to enable fast and precise construction of the Directed Acyclic Graph of Strongly Connected Components (DAGSCC, also called condensation of the PDG). The authors recognize that the DAGSCC is actually useful in program optimization rather than the PDG, and thus propose to directly compute the former by analyzing fewer dependences. This approach is more closely tied to program transformation, but does not provide a way to speed (scale) up the step of applying global program transformations from the computed DAGSCC. In addition, the authors analyze individual hot loops and not the entire hot region or the program. Thus, their framework does not perform global program transformation.

**Scalability within Polytope model.** The problem of unscalability of polyhedral techniques (i.e. the simplex algorithm for solving the linear programs for scheduling) when applied to large scopes was recognized by Feautrier [10]. He also proposed to avert the problem by calling the simplex algorithm on smaller scopes or modules that are comparable to functions. Each of these modules could be optimized in much less time than all of them together. However, this misses the essential global optimizations such as fusing those modules and then parallelizing the fused structure. Among existing polyhedral implementations, we found that Polly [13] (high-level optimizer for LLVM) uses a basic block as the optimization structure which is similar to our choice of O-molecules in `scalefuse` (except when there is presence of conditional statements in the loop body). Polly makes this choice because it derives polyhedral representation from the LLVM IR. This equips Polly also to achieve scalability as `scalefuse`, but the optimizer currently suffers from the same limitation (as above) of constructing separate schedules for basic blocks. Upadrashta et al. [26] have identified this problem and have proposed a sub-polyhedral technique using (Unit-)Two-Variable-Per-Inequality or (U)TVPI Polyhedra. Their technique is an under-approximation of a general polyhedron, and thus does not guarantee to find best solutions. If the under-approximation results in an empty polyhedra (or a non-solution), the authors propose to distribute loops to achieve a solution. This will again introduce sub-optimal solutions. However, the authors show the scalability of their technique over the simplex algorithm used in PLuTo for highly unrolled matrix kernel. The matrix kernel although unrolled to introduce dependences on the order of thousands, still takes less than 5 seconds to compile using PLuTo, given the very simple nature of dependences. We believe that achieving a feasible solution by such an approach may be a challenge when using real application programs that contain imperfectly nested loops in multiple nests with different loop order. Our work in this paper does not use any under-approximations, and uses knowledge of program semantics to cut-down the size of input (in terms of variables and constraints) to the simplex algorithm. We also demonstrate its scalability on real-world applications. Lastly, a recent work [30] proposes integer set coalescing that discovers opportunities for coalescing the different constraints from dependences and thus simplifies the input to the scheduler. This work is complementary to ours and can help to further bring down the overall compile time.

8. Conclusion

In this work, we address a key problem in the polyhedral (parallelizing) compilers, namely, unscalability of the employed algo-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Subroutine</th>
<th>Compile time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Phase - I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>smartfuse</td>
</tr>
<tr>
<td>it</td>
<td>dos</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>rms</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>rhs</td>
<td>81.72</td>
</tr>
<tr>
<td>outperfs</td>
<td></td>
<td>170.18</td>
</tr>
<tr>
<td>seccom</td>
<td></td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>81.72</td>
</tr>
</tbody>
</table>
rithms. We identify three causes of this unscalability, each of which stems from large number of program statements (and program dependencies) in the source program. We therefore propose a one-shot solution, scalefuse, to this problem - represent the entire hot region with a smaller set of statements (and dependences) that is sufficient for the compiler to perform loop optimizations. Our algorithms for the purpose are based on the notion of an 'O-molecule', where statements in an O-molecule are represented by a single superstatement. This allows to achieve the needed statement (and dependence) condensation. As a result, we can significantly reduce the time and memory requirement for compilation of real applications as compared to the state-of-the-art polyhedral compilers, and match those of the Intel compiler. We show that scalefuse does not significantly constrain program transformation for our test applications. The application of our proposed approach to other applications and the choice of the granularity of O-molecules in such cases is certainly interesting future research. However, we believe that through this work, the strengths of program analysis and transformation offered by a polyhedral compiler can be effectively and practically employed to optimize real application programs in addition to kernels.

Acknowledgments

We would like to acknowledge NSF grants CNS-0834599 and CCF-0708822 for supporting this work. We would like to particularly acknowledge Professor Albert Cohen and the anonymous reviewers who helped us improve the quality of our paper in its final version.

References