Proceedings of IMPACT 2013

IMPACT 2013

Proceedings of the

3rd International Workshop on Polyhedral Compilation Techniques

Berlin, Germany
January 21, 2013

in conjunction with HiPEAC 2013

Workshop organizers and proceedings editors:

Armin Größlinger
Louis-Noël Pouchet

Each article in these proceedings is copyrighted by its respective authors.

Acknowledgements

The program chairs are grateful to the members of the program committee for their incredible work, time commitment and dedication during the review process for IMPACT 2013. We are also grateful to the authors who submitted to IMPACT, Albert Cohen for his keynote speech, and all participants and attendees of IMPACT 2013 in Berlin.

IMPACT 2013 in Berlin, Germany (in conjuction with HiPEAC 2013) is the third workshop in a series of international workshops on polyhedral compilation techniques. The previous workshops were held in Chamonix, France (2011) in conjuction with CGO 2011 and Paris, France (2012) in conjuction with HiPEAC 2012.

Contents

Committees **1**

Tiling & Dependence Analysis

David G. Wonnacott, Michelle Mills Strout
On the Scalability of Loop Tiling Techniques . . . 3

Tomofumi Yuki, Sanjay Rajopadhye
Memory Allocations for Tiled Uniform Dependence Programs . . . 13

Sven Verdoolaege, Hristo Nikolov, Todor Stefanov
On Demand Parametric Array Dataflow Analysis . . . 23

Parallelism Constructs & Speculation

Imèn Fassi, Philippe Clauss, Matthieu Kuhn, Yosr Slama
Multifor for Multicore . . . 37

Dustin Feld, Thomas Soddemann, Michael Jünger, Sven Mallach
Facilitate SIMD-Code-Generation in the Polyhedral Model by
Hardware-aware Automatic Code-Transformation . . . 45

Johannes Doerfert, Clemens Hammacher, Kevin Streit, Sebastian Hack
SPolly: Speculative Optimizations in the Polyhedral Model . . . 55

Organizers and Program Chairs

Armin Größlinger (University of Passau, Germany)
Louis-Noël Pouchet (University of California Los Angeles, USA)

Program Committee

Christophe Alias (ENS Lyon, France)
Cédric Bastoul (INRIA, France)
Uday Bondhugula (IISc, India)
Philippe Clauss (University of Strasbourg, France)
Albert Cohen (INRIA, France)
Alain Darte (ENS Lyon, France)
Paul Feautrier (ENS Lyon, France)
Martin Griebl (University of Passau, Germany)
Sebastian Hack (Saarland University, Germany)
François Irigoin (MINES ParisTech, France)
Paul Kelly (Imperial College London, UK)
Ronan Keryell (Wild Systems/Silkan, USA)
Vincent Loechner (University of Strasbourg, France)
Benoît Meister (Reservoir Labs, Inc., USA)
Sanjay Rajopadhye (Colorado State University, USA)
P. Sadayappan (Ohio State University, USA)
Michelle Mills Strout (Colorado State University, USA)
Nicolas Vasilache (Reservoir Labs, Inc., USA)
Sven Verdoolaege (KU Leuven/ENS, France)

Additional Reviewers

Somashekar Bhaskaracharya
Roshan Dathathri
Alexandra Jimborean
Athanasios Konstantinidis
Andreas Simbürger

On the Scalability of Loop Tiling Techniques

David G. Wonnacott
Haverford College
Haverford, PA, U.S.A. 19041
davew@cs.haverford.edu

Michelle Mills Strout
Colorado State University
Fort Collins, CO, U.S.A. 80523
mstrout@cs.colostate.edu

ABSTRACT

The Polyhedral model has proven to be a valuable tool for improving memory locality and exploiting parallelism for optimizing dense array codes. This model is expressive enough to describe transformations of imperfectly nested loops, and to capture a variety of program transformations, including many approaches to loop tiling. Tools such as the highly successful PLuTo automatic parallelizer have provided empirical confirmation of the success of polyhedral-based optimization, through experiments in which a number of benchmarks have been executed on machines with small- to medium-scale parallelism.

In anticipation of ever higher degrees of parallelism, we have explored the impact of various loop tiling strategies on the asymptotic degree of available parallelism. In our analysis, we consider "weak scaling" as described by Gustafson, i.e., in which the data set size grows linearly with the number of processors available. Some, but not all, of the approaches to tiling provide weak scaling. In particular, the tiling currently performed by PLuTo does not scale in this sense.

In this article, we review approaches to loop tiling in the published literature, focusing on both scalability and implementation status. We find that fully scalable tilings are not available in general-purpose tools, and call upon the polyhedral compilation community to focus on questions of asymptotic scalability. Finally, we identify ongoing work that may resolve this issue.

1. INTRODUCTION

The Polyhedral model has proven to be a valuable tool for improving memory locality and exploiting parallelism for optimizing dense array codes. This model is expressive enough to express a variety of program transformations, including many forms of *loop tiling*, which can improve cache line utilization and avoid false sharing [16, 37, 36], as well as increase the granularity of concurrency.

For many codes, the most dramatic locality improvements occur with *time tiling*, i.e., tiling that spans multiple iterations of an outer time-step loop. In some cases, the degree of locality can increase with the number of time steps in a tile, providing *scalable locality* [39]. For non-trivial examples, time tiling often requires *loop skewing* with respect to the time step loop [27, 39], often referred to as *time skewing* [39, 38]. This transformation typically involves imperfectly nested loops, and was thus not widely implemented before the adoption of the polyhedral approach. However, the PLuTo automatic parallelizer [19, 6] has demonstrated considerable success in obtaining high performance on machines with moderate degrees of parallelism by using this technique to automatically produce OpenMP parallel code.

Unfortunately, the specific tiling transformations that have been implemented and released in tools like PLuTo involve pipelined execution of tiles, which prevents full concurrency from the start. The lack of immediate full concurrency is sometimes dismissed as a start-up cost that will be trivially small for realistic problem sizes. While this may be true for the degrees of parallelism provided by current multicore processors, this choice of tiling can impact the asymptotic degree of concurrency available if we try to scale up data set size and machine size together, as suggested by Gustafson [15]. Furthermore, Van der Wijngaart et al. [35] have modeled and experimentally demonstrated the load imbalance that occurs on distributed memory machines when using the pipelined approach.

In this paper, we review the status of implemented and proposed techniques for tiling dense array codes (including the important sub-case of stencil codes) in an attempt to determine whether or not the techniques that are currently being implemented are well suited to machines with higher demands for parallelism and control of memory traffic and communication. The published literature on tiling for automatic parallelization seems to be divided into two disjoint categories: "practical" papers describing implemented but unscalable techniques for automatic parallelizers for dense array codes, and "theoretical" papers describing techniques that scale well but are either not implemented or not integrated into a general automatic parallelizer.

In Section 2 of this paper, we discuss that the approach currently used by PLuTo does not allow full scaling as described by Gustafson [15]. In Section 4, we survey other tilings that have been suggested in the literature, classify each approach as fully scalable or not, and discuss its implementation status in current automatic parallelization tools. We also address recent work by Bondhugula et al. [3] on a tiling technique that we believe will be scalable, though asymptotic scaling is not addressed in [3]. Section 5 presents our conclusions: we believe the scalable/implemented dichotomy is an artifact of current design choices, not a fundamental limitation of the polyhedral model, and can thus be addressed via a shift in emphasis by the research community.

```
    // update N pseudo-random seeds T times
    // assumes R[ ] is initialized
       for t = 1 to T
          for i = 0 to N-1
S0:          R[i] = (a*R[i]+c) % m
```

Figure 1: "Embarrassingly Parallel" Loop Nest.

2. TILING AND SCALABILITY

In his 1988 article "Reevaluating Amdahl's Law" [15], Gustafson observed that, in actual practice, "One does not take a fixed size problem and run it on various numbers of processors", but rather "expands [the problem] to make use of the increased facilities". In particular, in the successful parallelizations he described, "as a first approximation, the amount of work that can be done in parallel *varies linearly with the number of processors*", and it is "most realistic to assume *run time*, not *problem size*, is constant". This form of scaling is typically referred to as *weak scaling* or *scalable parallelism*, as opposed to the *strong scaling* needed to give speed-up proportional to the number of processors for a fixed-size problem.

Weak scaling can be found in many *data parallel* codes, in which many elements of a large array can be updated simultaneously. Figure 1 shows a trivial example that we will use to introduce our diagrammatic conventions (following [38]). In Figure 2 each statement execution/loop iteration is drawn as an individual node, with sample values given to symbolic parameters. The time axis, or outer loop, moves from left to right across the page. The grouping of nodes into tiles is illustrated with variously shaped boxes around sets of nodes. Arrows in the figure denote direction of flow of information among iterations or tiles. Line-less arrowheads indicate values that are live-in to the space being illustrated. (When comparing our figures to other work, note that presentation style may vary in several ways: some authors use a time axis that moves up or down the page; some draw data dependence arcs from a use to a definition, thus pointing into the data-flow like a weather vane; some illustrate tiles as rectangular grids on a visually transformed iteration space, rather than with varied shapes on the original iteration space.)

Figure 2 makes clear the possibility of both (weak) scalable parallelism and scalable locality. In the execution of a tile of size $(\tau \times \sigma)$, i.e., τ iterations of the t (time) loop and σ iterations of the i (data) loop, data-flow does not prevent P processors from concurrently executing P such tiles. Each tile performs $O(\sigma \times \tau)$ computations and has $O(\sigma)$ live-in and live-out values; if each processor performs all updates of one data element before moving to the next, $O(\tau)$ operations can be performed with $O(1)$ accesses to main memory.

Inter-iteration data-flow can constrain, or even prevent, scalable parallelism or scalable locality. For the code in Figure 1, we can scale up parallelism by increasing N and P, or we can scale up locality by increasing T with the machine balance. However, beyond a certain point (i.e., $\sigma = 1$), we can no longer use additional processors to explore ever increasing values of T for a given N. (An increase in CPU clock speed might help in this situation, though beyond a certain point it would likely not help performance for increasing N for a given T.)

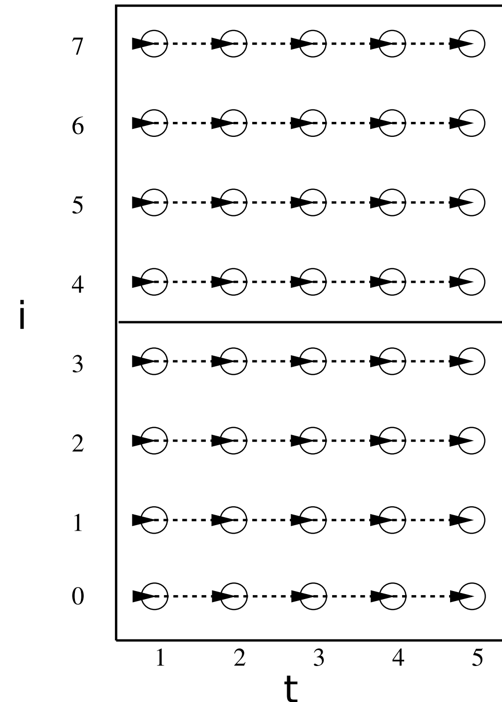


Figure 2: Iteration Space of Figure 1 with T=5, N=8, tiled with $\tau = 5, \sigma = 4$.

As we show in the next two sections, scalable parallelism may be constrained not only by the fundamental data-flow, but also by the approach to parallelization.

3. PIPELINED PARALLELISM

In a Jacobi stencil computation, each array element is updated as a function of its value and its neighbors' values, as shown (for a one-dimensional array) in Figure 3. Thus, we must perform time skewing to tile the iteration space (except in the degenerate case of $\tau = 1$, which prevents scalable locality). Figure 4 illustrates the usual tiling performed by automatic parallelizers such as PLuTo, though for readability our figure shows far fewer loop iterations per tile. Nodes represent executions of statement `S1`; for simplicity, executions of `S2` are not shown. (The same data-flow also arises from a doubly-nested execution of the single statement `A[t%2,i] = (A[(t-1)%2,i-1]+2*A[(t-1)%2,i]+A[(t-1)%2,i+1])/4`, but some tools may not recognize the program in this form.)

Array data-flow analysis [11, 22, 23] is well understood for programs that fit the polyhedral model, and can be used to deduce the data-flow arcs from the original imperative code. The data-flow arcs crossing a tile boundary describe the communication between tiles; in most approaches to tiling for distributed systems, inter-processor communication is aggregated and takes place between executions of tiles, rather than in the middle of any tile. The topology of the inter-tile data-flow thus gives the constraints on possible concurrent execution of tiles. For Figure 4, concurrent ex-

```
      for t = 1 to T
        for i = 1 to N-2
S1:       new[i] = (A[i-1]+2*A[i]+A[i+1])/4
        for i = 1 to N-2
S2:       A[i] = new[i]
```

Figure 3: Three Point Jacobi Stencil.

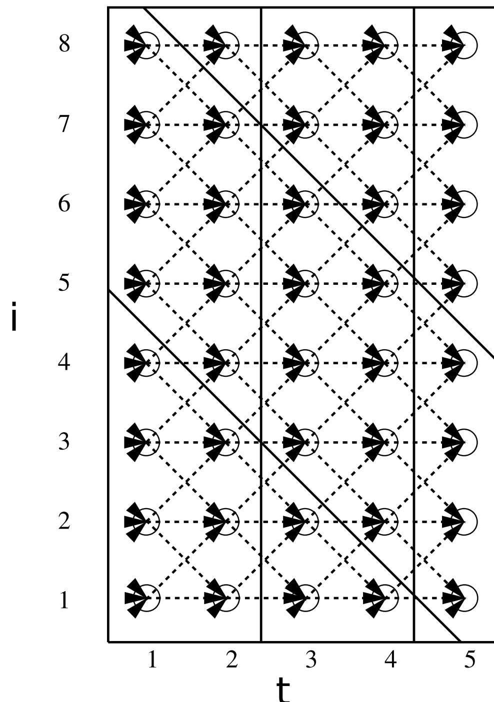


Figure 4: Iteration Space of Figure 3 with T=5, N=10, tiled with $\tau = 2, \sigma = 4$.

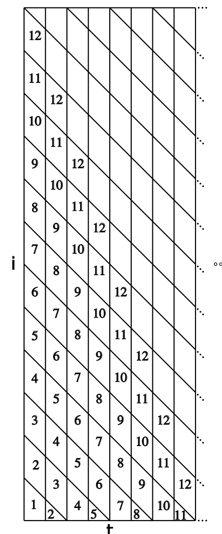


Figure 5: Wavefronts of Pipelined Tile Execution.

ecution is possible in pipelined fashion, in which execution of tiles progresses as a *wavefront* that begins with the lower left tile, then simultaneously executes the two tiles bordering it (above and to the right), and continues to each wave of tiles adjacent to the just-completed wave.

As has been noted in the literature, this is not the only way to tile this set of iterations; however, other tilings are not (currently) selected by fully-automatic loop parallelizers such as PLuTo [19]. Even the semi-automatic AlphaZ system [40], which is designed to allow programmers to experiment with different optimization strategies, cannot express many of these tilings. If such tools are to be considered for extreme scale computing, we must consider whether or not the tiling strategies they support provide the necessary scaling characteristics.

3.1 Scalability

To support our claim that this pipelined tiling does not always provide scalable parallelism, we need only show that it fails to scale on one of the classic examples for which it has shown dramatic success for low-degree parallelism, such as the easily-visualized one-dimensional Jacobi stencil of Figure 3. We will first do so, and then discuss the issue in higher dimensions.

For some problem sizes, the tiling of Figure 4 can come close to realizing scalable parallelism: if $P = \frac{N}{\sigma+\tau}$ and $\frac{T}{\tau}$ is much larger than P, most of the execution is done with P processors. Figure 5 illustrates the first 8τ time steps of such an example, with $P = 8$, $\sigma = 2\tau$, and $N = P(\sigma + \tau)$ (the ellipsis on the right indicates a large number of additional time steps). The tiles executed in the first 12 waves are numbered 1 to 12 for reference, and individual iterations and data-flow are omitted for clarity. For all iterations after 11, this tiling provides enough parallelism to keep eight processors busy for this problem size, and for large T the running time approaches $\frac{1}{8}$ of the sequential execution time (plus communication/synchronization time, which we will discuss later). If we double both N and P, a similar argument shows the running time approaches $\frac{1}{16}$ of the now twice-as-large sequential execution time, i.e., the same parallel execution time, as described by Gustafson.

However, as N and P continue to grow, the assumption that $\frac{T}{\tau} \gg P$ eventually fails, and scalability is lost. Consider what happens in Figure 5 if $T = 8\tau$, i.e., the ellipsis corresponds to 0 additional time steps. At this point, doubling N and P produces a figure that is twice as tall, but no wider; parallelism is limited to degree 8, and execution with 16 processors requires 35 steps rather than the 23 needed for Figure 5 when $T = 8\tau$ (note that the upper-right region is symmetric with the lower-left). Thus, communication-free execution time has increased rather than remaining constant. Increasing T (rather than N) with P is no better, and in fact no combination of N and T increase can allow 16 processors to execute twice the work of 8 in the same 23 wavefronts: adding even one full row or one full column of tiles means 24 wavefronts are needed.

Figure 5 was, of course, constructed to illustrate a lack of scalability. But even if we start with a more realistic problem

size, i.e., with $N \gg P$ and $T \gg P$, the pipelined tiling still limits scalability. Consider what happens we apply the tiling of Figure 5 with parameters $N = 10000\sigma, T = 1000\tau, P = 100$, in which over 99% of the tiles can be run with full 100-fold parallelism, and then scale up N and P together by successive factors of ten. Our first jump in size gives $N = 100000\sigma, T = 1000\tau, P = 1000$, which still has full (1,000-fold) parallelism in over 98% of the tiles.

After the next jump, to $N = 1000000\sigma, T = 1000\tau, P = 10000$ there is no 10,000-fold concurrency in the problem. Even if the application programmer is willing to scale up T rather than just N, in an attempt to reach full machine utilization, the execution for $N = 38730\sigma, T = 25820\tau, P = 10000$ still achieves 10,000-fold parallelism in only 85% of the 10^9 tiles. No combination of N and T allows *any* 100,000-fold parallelism on 10^{10} tiles with this tiling... to maintain a given degree of parallelism asymptotically, we must scale *both* N and T with P, contrary to Gustafson's original definition. While this may be acceptable for some application domains, we do not presume it to be universally appropriate, and thus see pipelined tiling as a potential restriction on the applicability of time tiling.

It is not always realistic to scale the number of time steps T. Bassetti et al. [4] introduce an optimization they call sliding block temporal tiling. They indicate that in these relaxation algorithms such as Jacobi "[g]enerally several sweeps are made". In their experiments they use up to 8 sweeps. Zhou et al. [41] use 16,384 in their modified benchmarks. Experiments with the Pochoir compiler [34] used 200 time steps because their cache oblivious performance optimization improves temporal locality. In summary, the number of time iterations in stencil computation performance optimization research varies dramatically. Work from the BeBOP group at Berkeley [8] discusses how often multiple sweeps over the grid within one loop occur and indicate that it may not be as common as those of us working on time skewing imagine. This makes it even more important for tiling strategies to provide scalable parallelism that does not require the number of time steps to be on par with the spatial domain.

3.2 Tile Size Choice and Communication Cost

The above argument presumes a fixed tile size, ignoring the possibility of reducing the tile size to increase the number of tiles. However, communication costs (either among processors or between processors and RAM) dictate a minimal tile size below which performance will be negatively impacted (see [38, 19] for further discussion).

Note that per-processor communication cost is not likely to shrink as the data set size and number of processors is scaled up: Each processor will need to send the same number of tiles per iteration, producing per-processor communication cost that remains roughly constant (e.g., if processors are connected to nearest neighbors via a network of the same dimensionality as the data set space) or rising (e.g., if processors are connected via a shared bus).

Even if we ignore communication costs entirely (i.e. in the notoriously unscalable PRAM abstraction), tile size cannot shrink below a single-iteration (or single-instruction) tile, and eventually our argument of Section 3.1 comes into play in asymptotic analysis.

3.3 Other Factors Influencing Scalability and Performance

Our argument focuses on tile *shape*, but a number of other factors will influence the degree of parallelism actually achieved by a given tiling. As noted above, reductions in tile size could, up to a point, provide additional parallelism (possibly at the cost of performance on the individual nodes).

The use of global barriers or synchronization is common in the original code generators, but note that MPI code generators are under development for both Pluto and AlphaZ. While combining different tilings and code generation schemes raises practical challenges in implementation, we do not see any reason why any of the tilings discussed in the next section could not, in principle, be executed without global barriers within the tiled iteration space.

High performance on each node also requires attention to a number of other code generation issues, such as code complexity and impact on vectorization and prefetching. Furthermore, these issues could be exacerbated by changes in tile shape [3, Section III.B]. Thus, different tiling *shapes* may be optimal for different hardware platforms or even different problem sizes, depending on the relative costs of limiting parallelism vs. per-node performance. Both [3, Section III.B] and [33] discuss these issues and the possibility of choosing a tiling that is scalable in some, but not all, data dimensions.

3.4 Higher-Dimensional Codes

While the two-dimensional iteration space of the three-point Jacobi stencil is easy to visualize on paper, many of the subtleties of tiling techniques are only evident in problems with at least two dimensions of data and one of time. For pipelined tiling, the conflict between scalability is essentially the same in higher dimensions: for a pipelined tiling of a hyper-rectangular iteration space of dimension d, eventually the amount of work must grow by $O(k^d)$ to achieve parallelism $O(k^{d-1})$.

Conversely, in higher dimensions, the existence of a wavefront that is perpendicular to the time dimension (or any other face of a hyper-rectangular iteration space) is frequently the sign of a parallelization that admits some form of weak scalability. However, as we will see, the parallelism of some tilings scales with only some of the spatial dimensions.

4. VARIATIONS ON THE TILING THEME

The published literature describes many approaches to loop tiling. In this section, we survey these approaches, grouping together those that produce similar (or identical) tilings. Our descriptions focus primarily on the tiling that would be used for the code of Figure 3, which is used as an introductory example in many of the descriptions. We illustrate the tilings of this code with figures that are analogous to our Figure 5, with gray shading highlighting a single tile

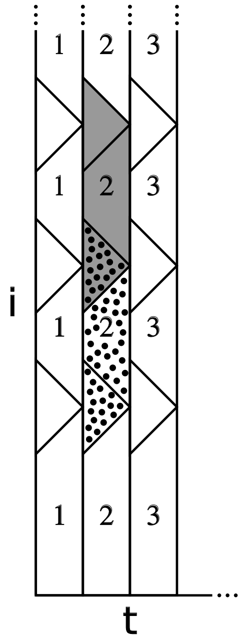


Figure 6: Overlapped Tiling.

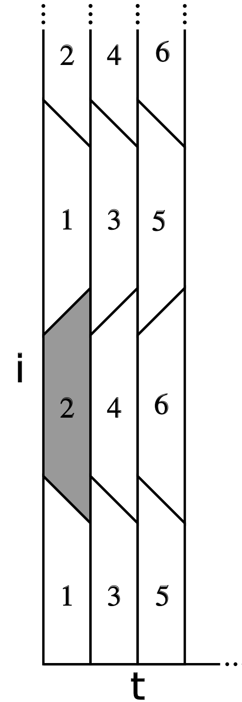


Figure 7: Trapezoidal Tiling.

from time step two. We delve into the complexities of more complex codes only as necessary to make our point.

For iterative codes with intra-time-step data-flow, the tile shapes discussed below are not legal (or cannot be legally executed in an order that permits scalable parallelism). For example, consider an in-place update of a single copy of a data set, e.g. the single statement `A[i] = (A[i-1]+2*A[i]+A[i+1])/4` nested inside `t` and `i` loops. Since each tile must wait for data from tiles of with lower values of `i` and the same value of `t`, pipelined startup is necessary. While such code provides additional asymptotic concurrency when both T and N increase, we see no way to allow concurrency to grow linearly with the total work to be done in parallel. Thus, pipelined tiling does not produce a scalability disadvantage for such codes.

Note that our discussion below focuses on distinct tile shapes, rather than distinctions among algorithms used to deduce tile shape or size or the manner in which individual tiles are scheduled or assigned to processors. For example we do not specifically discuss the "CORALS" approach [32], in which an iteration space is recursively subdivided into parallelograms, avoiding the need to choose a tile size in advance of starting the computation. Regardless of size, variation in size, and algorithmic provenance, the information flow among atomic parallelogram tiles still forces execution to proceed along the diagonal wavefront, and thus still limits asymptotic scalability.

4.1 Overlapped Tiling

A number of projects have experimented with what is commonly called overlapped tiling [26, 4, 2, 25, 10, 24, 19, 7, 20, 41]. In overlapped tiling for stencil computations, a larger halo is maintained so that each processor can execute more than one time step before needing to communicate with other processors. Figure 6 illustrates this tiling. Two individual tiles from the second wavefront have been indicated with shading, one with gray and one with polkadots; the triangular polkadotted and gray region is in both tiles, and thus represents redundant computation. This overlap means that all tiles along each vertical wavefront can be executed in parallel while still improving temporal data locality.

In terms of parallelism scalability, overlapped tiling does scale because all of the tiles can be executed in parallel. If a two-dimensional tiling in a two-dimensional spatial part of a stencil is used as the seed partition, then two dimensions of parallelism will be available with no need to fill a pipeline. This means that as the data scale, so will the parallelism.

The problem with overlapped tiling is that redundant computation is performed. This leads to a trade-off between parallel scalability and execution time. Tile size selection must also consider the effect of the expanded memory footprint caused by overlapped tiling.

Auto-tuning between overlapped sparse tiling and non-overlapped sparse tiling [29, 30] for irregular iteration spaces has also been investigated by Demmel et al. [9] in the context of iterative sparse matrix computations where the tiling is a run-time reordering transformation [28].

4.2 Trapezoidal Tiling

Frigo and Strumpen [12, 13, 14] propose an algorithm for limiting the asymptotic cache miss rate of "an idealized parallel machine" while providing scalable parallelism. Figure 7 illustrates that even a simplified version of their approach

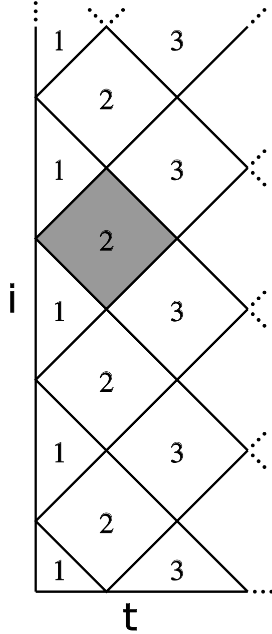


Figure 8: One data dimension and the time dimension in a diamond tiling [33]. The diamond extends into a diamond tube in the second data dimension.

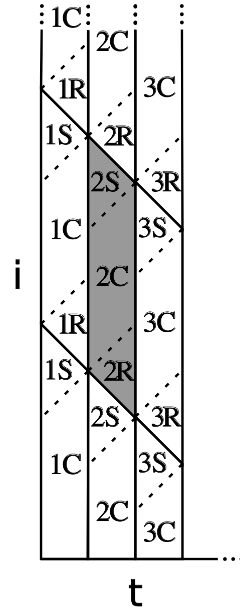


Figure 9: Molecular tiling.

can enable weak scaling for the examples we discuss here (their full algorithm involves a variety of possible decomposition steps; our figure is based on Figure 4 of [14]). In our Figure 7, the collection of trapezoids marked "1" can start simultaneously; after these tiles complete, the mirror-image trapezoids that fill the spaces between them, marked "2", can all be executed; after these steps, a similar pair of sets of tiles "3" and "4" complete another τ time steps of computation, etc. For discussion of the actual transformation used by Frigo and Strumpen, and its asymptotic behavior, see [14].

The limitation of this approach is not its scalability, but rather the challenge of implementing it in a general-purpose compiler. Tang et al. [34] have developed the Pochoir compiler, based on a variant of Frigo and Strumpen's techniques with a higher degree of asymptotic concurrency [34]. However, Pochoir handles a specialized language that allows only stencil computations. Tools like PLuTo handle a larger domain of dense array codes; it may be possible to generalize trapezoidal tiling to PLuTo's domain, but we know of no such work.

4.3 Diamond Tiling

Strzodka et al. [33, 31] present the CATS algorithm for creating diamond "tube" tiles in a 3-d iteration space. The diamonds occur in the time dimension and one data dimension, as in Figure 8. The tube aspect occurs because there is no tiling in the other space dimension. Each diamond tube can be executed in parallel with all other diamond tubes within a temporal row of diamond tubes. For example, in Figure 8 all diamonds labeled "1" can be executed in parallel, after which all diamonds labeled "2" can be executed in parallel, etc. Within each diamond tube, the CATS approach schedules another level of wavefront parallelism at the granularity of iteration points.

Although Strzodka et al. [33] do not use diamond tiles for 1-d data/2-d iteration space, diamond tiles are parallel scalable within that context. They actually focus on the 2-d data/3-d iteration space, where asymptotically, diamond tiling only scales for one data dimension. The outermost level of parallelism over diamond tubes only scales with one dimension of data since the diamond tiling occurs across time and one data dimension. On page 2 of [33], Strzodka et al. explicitly state that their results are somewhat surprising in that asymptotically their behavior should not be as good as previously presented tiling approaches, but the performance they observe is excellent probably due to the concurrent parallel startup that the diamond tiles provide.

Diamond tiling is a practical approach that can perform better than pipelined tiling approaches because it avoids the pipeline fill and drain issue. The diamond tube also has advantages in terms of intra-tile performance: fine-grained wavefront parallelism and leveraging pre-fetchers. The disadvantages of the diamond tiling approach are that it has not been expressed within a framework such as the polyhedral model (although it would be possible, just not with rectangular tiles); that the approach does not cleanly extend to higher dimensions of data (only one dimension of diamond tiles are possible with other dimensions doing some form of pipelined or split tiling); and that the outermost level of parallelism can only scale with one data dimension.

4.4 Molecular Tiling

Wonnacott [38] described a tiling for stencils that allows true weak scaling for higher-dimensional stencils, performs no re-

dundant work, and contains tiles that are all the same shape. However, Wonnacott's *molecular tiles* required mid-tile communication steps, as per Pugh and Rosser's *iteration space slicing* [21] as illustrated in Figure 9. Each tile first executes its *send slice* (labeled "S"), the set of iterations that produce values that will be needed by another currently-executing tile, and then sends those values; it then goes on to execute its *compute slice* ("C"), the set of iterations that require no information from any other currently-executing tile; finally, each tile receives incoming values and executes its *receive slice* ("R"), the set of iterations that require these data. In higher dimensions, Wonnacott discussed the possibility of extending the parallelograms into prisms (as diamonds are extended into diamond tubes in the diamond tiling), but also presented a multi-stage sequence of send and receive slices to provide full scalability.

Once again, a transformation with potential for true weak scaling remains unrealized due to implementation challenges. No implementation was ever released for iteration space slicing [21]. For the restricted case of stencil computations, these molecular tiles can be described without reference to iteration space slicing, but they make extensive use of modulo constraints supported by the Omega Library's code generation algorithms [18], and Omega has no direct facility for generating the required communication primitives.

The developers of PLuTo explored a similar *split tiling* [19] approach, and demonstrated improved performance over the pipelined tiling, but this approach was not used for the released implementation of PLuTo.

4.5 A New Hope

Recent work on the PLuTo system [3] has produced a tiling that we believe will address the issue of true scalability with data set size, though the authors frame their approach primarily in terms of "enabling concurrent start-up" rather than improving asymptotic scalability. For a one-dimensional data set, this approach is essentially the same as the "diamond tiling" of Section 4.3, but for higher-dimensional stencils it allows scalable parallelism in all data dimensions.

Although a Jacobi stencil on a two-dimensional data set has an "obvious" four-sided pyramid of dependences, a collection of four-sided pyramids (or a collection of octahedrons made from pairs of such pyramids) cannot cover all points, and thus does not make a regular tiling. The algorithm of [3] produces, instead, a collection of six-faced tiles that appear to be balanced on one corner (these figures are shown with the time dimension moving up the page). Various sculptures of balanced cubes may be helpful in visualizing this tiling; those with limited travel budgets may want to search the internet for images of the "Zabeel park cube sculpture". The approach of [3] manages to construct these corner-balanced solids in such as way that the three faces at the bottom of the tile enclose the data-flow.

Experiments with this approach [3] demonstrate improved results (vs. pipelined tiling) for current shared-memory systems up to 16 cores. The practicality of this approach on such a low degree of parallelism suggests that the per-node penalties discussed in our Section 3.3 are not prohibitively expensive.

While the algorithm is described in terms of stencils, and the authors only claim concurrent startup for stencils, it is implemented in a general automatic parallelizer (PLuTo). We belive it would be interesting to explore the full domain over which this tiling algorithm provides concurrent startup.

The authors of [3] do not discuss asymptotic complexity, but we hope that future collaborations could lead to a detailed theoretical and larger-scale empirical study of the scalability of this technique, using the distributed tile execution techniques of [1] or [5].

4.6 A Note on Implementation Challenges

The pipelined tile execution shown in Figures 4 and 5 is often chosen for ease of implementation in compilers based on the polyhedral model. Such compilers typically combine all iterations of all statements into one large iteration space; the pipelined tiling can then be seen as a simple linear transformation of this space, followed by a tiling with rectangular solids. This approach works well regardless of choice of software infrastructure within the polyhedral model.

The other transformations may be more sensitive to choice of software infrastructure, or the subtle use thereof. At this time, we do not have an exact list of which transformations can be expressed with which transformation and code generation libraries. We are working with tools that allow the direct control of polyhedral transformations from a text input, such as AlphaZ [40] and the Omega Calculator [17], in hopes of better understanding the expressiveness of these tools and the polyhedral libraries that underlie them.

5. CONCLUSIONS

Current work on general-purpose loop tiling exhibits a dichotomy between largely unimplemented explorations of asymptotically high degrees of parallelism and carefully tuned implementations that restrict or inhibit scalable parallelism. This appears to result from the challenge of general implementation of scalable approaches. The pipelined approach requires only a linear transformation of the iteration space followed by rectangular tiling, but does not provide true scalable parallelism. Diamond tiling scales with only one data dimension. Overlapped, trapezoidal, and molecular tiling each pose implementation challenges (due to redundant work, non-uniform tile shape/orientation, or non-atomic tiles, respectively).

We believe automatic parallelization for extreme scale computing will require a tuned implementation of a general technique that does not inhibit or restrict scalability; thus future work in this area must address scalability, generality, and quality of implementation. We are optimistic that ongoing work by Bondhugula et al. [3] may already provide an answer to this to dilemma.

6. ACKNOWLEDGMENTS

This work was supported by NSF Grant CCF-0943455, by a Department of Energy Early Career Grant DE-SC0003956, and the CACHE Institute grant DE-SC04030.

7. REFERENCES

[1] Abdalkader, M., Burnette, I., Douglas, T., and Wonnacott, D. G. Distributed shared memory and compiler-induced scalable locality for scalable cluster performance. *Cluster Computing and the Grid, IEEE International Symposium on 0* (2012), 688–689.

[2] Allen, G., Dramlitsch, T., Foster, I., Goodale, T., Karonis, N., Ripeanu, M., Seidel, E., and Toonen, B. The cactus code: A problem solving environment for the grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)* (Pittsburg, PA, USA, 2000).

[3] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling stencil computations to maximize parallelism. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (November 2012), SC '12, ACM Press.

[4] Bassetti, F., Davis, K., and Quinlan, D. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science 1505* (1998).

[5] Bondhugula, U. Compiling affine loop nests for distributed-memory parallel architectures. Preprint, 2012.

[6] Bondhugula, U., Hartono, A., and Ramanujam, J. A practical automatic polyhedral parallelizer and locality optimizer. In *In PLDI âĂŹ08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation* (2008).

[7] Christen, M., Schenk, O., Neufeld, E., Paulides, M., and Burkhart, H. Manycore Stencil Computations in Hyperthermia Applications. In *Scientific Computing with Multicore and Accelerators*, J. Dongarra, D. Bader, and J. Kurzak, Eds. CRC Press, 2010, pp. 255–277.

[8] Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., and Yelick, K. Optimizations and performance modeling of stencil computations on modern microprocessors. *SIAM Review 51*, 1 (2009), 129–159.

[9] Demmel, J., Hoemmen, M., Mohiyuddin, M., and Yelick, K. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)* (Los Alamitos, CA, USA, 2008), IEEE Computer Society.

[10] Ding, C., and He, Y. A ghost cell expansion method for reducing communications in solving pde problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), Supercomputing '01, ACM, pp. 50–50.

[11] Feautrier, P. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming 20*, 1 (Feb. 1991), 23–53.

[12] Frigo, M., and Strumpen, V. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 361–366.

[13] Frigo, M., and Strumpen, V. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2006), SPAA '06, ACM, pp. 271–280.

[14] Frigo, M., and Strumpen, V. The cache complexity of multithreaded cache oblivious algorithms. *Theor. Comp. Sys. 45*, 2 (June 2009), 203–233.

[15] Gustfson, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (May 1988), 532–533.

[16] Irigoin, F., and Triolet, R. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (1988), pp. 319–329.

[17] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. The Omega Calculator and Library. Tech. rep., Dept. of Computer Science, University of Maryland, College Park, Apr. 1996.

[18] Kelly, W., Pugh, W., and Rosser, E. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, Virginia, Feb. 1995), pp. 332–341.

[19] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective automatic parallelization of stencil computations. In *Proceedings of Programming Languages Design and Implementation (PLDI)* (New York, NY, USA, 2007), vol. 42, ACM, pp. 235–244.

[20] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–13.

[21] Pugh, W., and Rosser, E. Iteration slicing for locality. In *12th International Workshop on Languages and Compilers for Parallel Computing* (Aug. 1999).

[22] Pugh, W., and Wonnacott, D. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), pp. 140–151.

[23] Pugh, W., and Wonnacott, D. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems 20*, 3 (May 1998), 635–678.

[24] Rastello, F., and Dauxois, T. Efficient tiling for an ode discrete integration program: Redundant tasks instead of trapezoidal shaped-tiles. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IPDPS '02, IEEE Computer Society, pp. 138–.

[25] Ripeanu, M., Iamnitchi, A., and Foster, I. T. Cactus application: Performance predictions in grid environments. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing* (London, UK, UK, 2001), Euro-Par '01, Springer-Verlag, pp. 807–816.

[26] Sawdey, A., and O'Keefe, M. T. Program analysis of overlap area usage in self-similar parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1998), LCPC '97, Springer-Verlag, pp. 79–93.

[27] Song, Y., and Li, Z. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices (PLDI) 34*, 5 (May 1999), 215–228.

[28] Strout, M. M., Carter, L., and Ferrante, J. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, June 2003), ACM.

[29] Strout, M. M., Carter, L., Ferrante, J., Freeman, J., and Kreaseck, B. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)* (Berlin / Heidelberg, July 2002), Springer.

[30] Strout, M. M., Carter, L., Ferrante, J., and Kreaseck, B. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications 18*, 1 (February 2004), 95–114.

[31] Strzodka, R., Shaheen, M., and Pajak, D. Time skewing made simple. In *PPOPP* (2011), C. Cascaval and P.-C. Yew, Eds., ACM, pp. 295–296.

[32] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing* (June 2010), ACM, pp. 49–59.

[33] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP)* (Taipei, Taiwan, September 2011), IEEE Computer Society, pp. 517–581.

[34] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 117–128.

[35] Van der Wijngaart, R. F., Sarukkai, S. R., Mehra, and P. The effect of interrupts on software pipeline execution on message-passing architectures. In *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, Pennsylvania, USA, May 25–28, 1996* (New York, NY 10036, USA, 1996), ACM, Ed., ACM Press, pp. 189–196.

[36] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *Programming Language Design and Implementation* (New York, NY, USA, 1991), ACM.

[37] Wolfe, M. J. More iteration space tiling. In *Proceedings, Supercomputing '89, Reno, Nevada* (Reno, Nevada, November 1989), ACM, Ed., ACM Press, pp. 655–664.

[38] Wonnacott, D. Using Time Skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium* (May 2000), IEEE.

[39] Wonnacott, D. Achieving scalable locality with Time Skewing. *Internation Journal of Parallel Programming 30*, 3 (June 2002), 181–221.

[40] Yuki, T., Basupalli, V., Gupta, G., Iooss, G., Kim, D., Pathan, T., Srinivasa, P., Zou, Y., and Rajopadhye, S. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Tech. rep., Technical Report CS-12-101, Colorado State University, 2012.

[41] Zhou, X., Giacalone, J.-P., Garzarán, M. J., Kuhn, R. H., Ni, Y., and Padua, D. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 207–218.

Memory Allocations for Tiled Uniform Dependence Programs *

Tomofumi Yuki
Colorado State University
Fort Collins
Colorado, U.S.A.
yuki@cs.colostate.edu

Sanjay Rajopadhye
Colorado State University
Fort Collins
Colorado, U.S.A.
Sanjay.Rajopadhye@colostate.edu

ABSTRACT

In this paper, we develop a series of extensions to schedule-independent storage mapping using Quasi-Universal Occupancy Vectors (QUOVs) targeting tiled execution of polyhedral programs. By quasi-universality, we mean that we restrict the "universe" of the schedule to those that correspond to tiling. This provides the following benefits: (i) the shortest QUOVs may be shorter than the fully universal ones, (ii) the shortest QUOVs can be found without any search, and (iii) multi-statement programs can be handled. The resulting storage mapping is valid for tiled execution by any tile size.

1. INTRODUCTION

In this paper, we discuss storage mappings for tiled programs, especially for the case when tile sizes are not known at compile time. When the tile sizes are parameterized, most techniques for storage mappings [6, 13, 15, 20] cannot be used due to the non-affine nature of parameterized tiling. However, we cannot combine parametric tiling with memory re-allocation if we cannot find a legal allocation for all legal tile sizes.

One approach that can find storage mappings for parametrically tiled programs is the Schedule-Independent Storage Mapping proposed by Strout et al. [19]. For programs with uniform dependences, schedule-independent memory allocation finds storage mappings that are valid for any legal execution of the program, including tiling by any tile size. We present a series of extensions to schedule-independent mapping for finding legal and compact storage mappings for polyhedral programs with uniform dependences.

Schedule-Independent Storage Mapping is based on what are called Universal Occupancy Vectors (UOVs), that characterize when a value produced can safely be overwritten. As originally defined by Strout et al, UOVs are *fully universal*, where the resulting allocations are valid for *any* legal schedule. In this paper, we restrict the UOVs to smaller universes and exploit their properties to efficiently find good UOVs. In the remainder of this paper, we call such UOVs that are not fully universal *Quasi-UOVs* (QUOVs) to distinguish them from fully universal ones. Using QUOVs, we can find valid mappings for tiled execution by any tile size, but not necessarily valid for other legal schedules. This leads to more compact storage mappings for cases when fully universal allocation is an overkill.

*This work was funded in part by the National Science Foundation, Award Numbers: 1240991 and 0917319

The restriction on the universality leads to the following:

- QUOVs may be shorter than the shortest UOV. Since the universe of possible schedules is restricted, valid storage mappings may be more compact. We use Manhattan distance as the length of UOVs as a cost measure when we describe optimality of a projective memory allocation.
- The shortest QUOV for tiled loop programs can be analytically found, and the dynamic programming algorithm presented by Strout et al. is no longer necessary.
- Imperfectly nested loops can be handled. The original method assumed single statement programs (and hence perfectly nested loop programs.) We extend the method by taking statement orderings, often expressed in the polyhedral model as constant dimensions, into account. This is possible because we focus on tiled execution, where tiling applies the same schedule (except for ordering dimensions) to all statements.

The input to our analysis is a program in polyhedral representation, which can be obtained from loop programs through array data-flow analysis [7, 14]. Our methods can be used for loop programs in single assignment form; an alternative view used in some of the prior work [19, 20].

In addition, we present a method for isolating boundary cases based on index-set splitting [9]. UOV-based allocation assumes that every a dependence is active at all points in the iteration space. In practice, programs have boundary cases where certain dependences are only valid at iteration space boundaries. We take advantage of the properties of QUOVs we develop to guide the splitting.

2. BACKGROUND

In this section, we present the background necessary for this paper. We first introduce the terminology used, and then present an overview of Universal Occupancy Vectors [19].

2.1 Polyhedral Representations

Polyhedral representations of programs primarily consist of statement domains and dependences. Statement domains represent the set of iteration points where a statement is executed. In polyhedral programs, such sets are described by a finite union of polyhedra.

In this paper, we focus on programs with uniform dependences, where the producer and the consumer differ by a constant shift. We characterize these dependences using a

vector, which we call *data-flow vector*, drawn the producer to the consumer. For example, if a value produced by an iteration $[i, j]$ is used by another iteration $[i + 1, j + 2]$, the corresponding data-flow vector is $[1, 2]$.

2.2 Schedules and Storage Mappings

The schedules are represented by affine mappings that map statement domains to a common dimensional space, where the lexicographic order denotes the order of execution. In the polyhedral literature, statement orderings are commonly expressed as constant dimensions in the schedule. Furthermore, one can represent arbitrary orderings of loops and statements by adding $d + 1$ additional dimensions [8], where d is the dimensionality of statement domains in the programs[1]. To make the presentation consistent, we assume such schedules are always used, resulting in a $2d + 1$ dimensional schedule.

The storage mappings are often represented as a combination of affine mappings and dimension-wise modulo factors [13, 16, 19].

2.3 Tiling

Tiling is a well known loop transformation that was originally proposed as a locality optimization [12, 17, 18, 22]. It can also be used to extract coarser grained parallelism, by partitioning the iteration space to tiles (blocks) of computation, some of which may run in parallel [12, 17].

Legality of tiling is a well established concept defined over contiguous subsets of the schedule dimensions (in the range of the scheduling function; scheduled space), also called *bands* [3]. These dimensions of the schedules are tilable, and are also known to be fully permutable [12].

The range space of the schedules given to statements in a program all refers to the common space, and thus have the same number of dimensions. Among these dimensions, a dimension is tilable if all dependences are not violated (i.e., the producer is not scheduled after the consumer, but possibly be scheduled to the same time stamp,) with a one-dimensional schedule using only the dimension in question. Then any contiguous subset of such dimensions forms a legal tilable band.

We call a subset of dimensions in an iteration space to be tilable, if the identity schedule is tilable for the corresponding subset. The iteration space is *fully-tilable* if all dimensions are tilable.

2.4 Universal Occupancy Vectors

A Universal Occupancy Vector (UOV) [19] is a vector that denotes the "distance after which" when a value may safely be overwritten in the following sense. When an iteration z is executed, its value is stored in some memory location. Another iteration z' can reuse this same memory location if all iterations that use the value produced by z have been executed. When a storage mapping is such that z and z' are mapped to the same memory location, the difference of these points, $z' - z$, is called the occupancy vector.

Universal Occupancy Vector is a specialization of such vectors, where all iterations k that use z are guaranteed to be executed before z' in any legal schedule. Since most machine models assume that, within a single assignment statement, reads happen before writes in a given time step, z' may also use the value produced by z.

Additionally, we introduce a notion of scoping to UOVs. We call a vector v to be an UOV with respect to a set of dependences $\mathcal{I}$, if the vector v satisfies the necessary property to be an UOV for a subset of all points k that use the value produced by z with one of the dependences in $\mathcal{I}$.

Once the UOV is computed, the mapping that corresponds to the projection of the statement domain along the UOV is a legal storage mapping. If the UOV crosses more than one integer points, then an array that corresponds to a single projection is not sufficient. Instead, multiple arrays are used in turn, implemented using modulo factors. The necessary modulo factor is the GCD of elements of the UOV.

The trivial UOV; a valid, but possibly suboptimal UOV is computed as follows.

1. Construct the set of data-flow vectors corresponding to all the dependences that use the result of a statement.
2. Compute the sum of all data-flow vectors in the constructed set.

The above follows from a simple proposition shown below, and an observation that the data-flow vector of a dependence is a legal UOV with respect to that dependence.

PROPOSITION 1 (SUM OF UOVS). *Let u and v be, respectively, the UOVs for two sets of dependences $\mathcal{U}$ and $\mathcal{V}$. Then $u + v$ is a legal UOV for $\mathcal{U} \cup \mathcal{V}$.*

PROOF. The value produced at z is dead when $z + u$ can legally be executed with respect to the dependences in $\mathcal{U}$, and similarly for $\mathcal{V}$ at $z + v$. Since there is a path from z to $z + u + v$ by following the edges $z + u$ and $z + v$ (in either order), the value produced at z is guaranteed to be used by all uses, $z + u$ and $z + v$, when $z + u + v$ can legally be executed. □

The optimality of UOVs without any knowledge of size or shape of the iteration space is captured by the length of the UOV. However, the length that should be compared is not the Euclidean length, but the Manhattan distance. We discuss the optimality of UOV-based allocations and other projective allocations in Section 7.

3. OVERVIEW OF OUR APPROACH

UOV-based allocation give legal mappings even for schedules that cannot be implemented as loops. For example, even a run-time work stealing scheduler can use UOV-based allocation. However, this is obviously an overkill if we only consider schedules that can be implemented as loops.

The important change in perspective is that we are not interested in schedule-independent storage mappings, although the concept of UOV is used. We are only interested in using UOV-based allocation in conjunction with tiling. Thus, our allocation is partially *schedule-dependent*. The overview of our storage mapping strategy is as follows:

1. Extract polyhedral representations of programs (array expansion.)
2. Perform schduling and apply the schedules as transformations to the iteration space[2]. After the transformation, lexicographic scan of the resulting iteration space

[1]Note that for uniform dependence programs, all statement domains have the same number of dimensions.

[2]This can be viewed as pre-processing to code generation [2].

respects the schedules. The resulting space should be (partially) tilable to take advantage of our approach.

3. Apply UOV-guided index-set splitting (Section 6.) This step attempts to isolate boundaries of statement domains that negatively influence storage mappings.

4. Apply QUOV-based allocation (Section 4.) Our proposed storage mapping based on extensions to the UOVs are applied to each statement after the splitting. Although inter-statement sharing of arrays may be possible, such optimization is beyond the scope of this paper.

The order of presentation does not follow the above for two reasons. One is that the UOV-guided splitting is an optional step that can further optimize memory usage. In addition, splitting introduces multiple statements to the program, and requires our extension to handle multiple statements presented in Section 5.

4. QUOV-BASED ALLOCATION FOR TILED PROGRAMS

In this section, we present a series of formalism to analytically find the shortest QUOV. We first develop a lemma that can eliminate dependences while constructing UOVs. We then apply the lemma to find the shortest QUOVs in different contexts.

4.1 Relevant Set of Dependences for UOV Construction

The trivial UOV, which also serves as the starting point for finding the optimal UOV, is found by taking the sum of all dependences. However, this formulation may lead to significantly inefficient starting points. For example, if two dependences with data-flow vectors $[1,0]$ and $[2,0]$ exist, the former dependence may be ignored during UOV construction since a legal UOV-based allocation using only the latter dependence is also guaranteed to be legal for the former dependence.

We may refine both the construction of the trivial UOV and the optimality algorithm by reducing the set of dependences considered during UOV construction. The optimality algorithm presented by Strout et al. [19] searches a space bounded by the length of trivial UOV using dynamic programming. Therefore, reducing the number of dependences to consider will improve both the trivial UOV and the dynamic programming algorithm.

The main intuition is that if a dependence can be transitively expressed by another set of dependences, then it is the only dependence that needs to be considered. This is formalized in the following lemma.

LEMMA 1 (DEPENDENCE SUBSUMPTION). *If a dependence f can be expressed as compositions of dependences in a set $\mathcal{G}$, where all dependences in $\mathcal{G}$ are used at least once in the composition, then a legal UOV with respect to f is also a legal UOV with respect to all elements of $\mathcal{G}$.*

PROOF. Given a legal UOV with respect to a single dependence f, the value produced at z is preserved at least until z' defined by $f(z') = z$, can be executed. Let the set of dependences in $\mathcal{G}$ be denoted as g^x, $1 \leq x \leq |\mathcal{G}|$. Since composition of uniform functions is associative and commutative, there is always a function g^* obtained by composing dependences in $\mathcal{G}$, such that $f = g^* \circ g^x$ for each x. Thus, all points z'', $g^x(z'') = z$, are executed before z' for all x. Therefore, a legal UOV with respect to f is guaranteed to preserve the value produced at z until all points that directly depend on z by a dependence in set $\mathcal{G}$ have been executed. □

Finding a composition in the above can be implemented as an integer linear programming problem. The problem may also be viewed as determining if a set of vectors are linearly dependent when restricted to positive combinations. The union of all sets $\mathcal{G}$, called subsumed dependences, found in the initial set of dependences can be ignored when constructing the UOV.

Applying Lemma 1 may significantly reduce the number of dependences to be considered. However, the trivial UOV of the remaining dependences may still not be the shortest UOV. For example, consider data-flow vectors $[1,1], [1,-1], [1,0]$. Although the vectors are independent by positive combinations, the trivial UOV $[3,0]$ is clearly longer than another UOV $[2,0]$. Further reducing the set of dependences to consider requires a variation of Lemma 1 that allows f also to be a composition of dependences. This leads to complex operations, and the dynamic programming algorithm for finding optimal UOV by Strout et al. [19] may be a better alternative. Instead of finding the shortest UOV in the general case, we show that such UOV can be found very efficiently for a specific context, namely tiling.

4.2 Finding the Shortest QUOV for Tiled Programs

When UOV-based allocation is used in the specific context of tiling, the shortest QUOV can be analytically found. If we know that the program is to be tiled, we can add *dummy* dependences to restrict the universality of the storage mapping, while maintaining tilability. In addition, we may assume that the dependences are all non-positive (for the tilable dimensions) as a result of pre-scheduling step to ensure the legality of tiling. For the remainder of this section, the "universe" of UOVs is one of the following restricted universes: fully tilable, fully sequential, and mixed sequential and tilable.

Note that the expectation is that "tilable" iteration spaces are tiled in a later phase. We analyze iteration spaces that will be tiled using QUOV, and then apply tiling. We also assume that the iteration points are scanned in the lexicographic order within a tile.

THEOREM 1 (SHORTEST QUOV, FULLY TILABLE). *Given a set of dependences $\mathcal{I}$ in a fully tilable space, the shortest QUOV u for tiled execution is the element-wise maxima of data-flow vectors of all dependences in $\mathcal{I}$.*

PROOF. Let the element-wise maxima of all data-flow vectors be the vector m, and f_m be a dependence with data-flow vector m. For unit vectors u^d in each of the d dimensions, we introduce *dummy* dependences f^d with data-flow vector u^d. Because these dependences have non-negative components, the resulting space is still tilable. For all dependences in $\mathcal{I}$ there exists a sequence of compositions with the dummy dependences to transitively express f_m. Using Lemma 1, the only dependence to be considered in UOV construction can therefore be reduced to f_m, which has the trivial UOV of m.

It remains to show that no QUOV shorter than m exists for the set of dependences $\mathcal{I}$. The shortest QUOV is defined

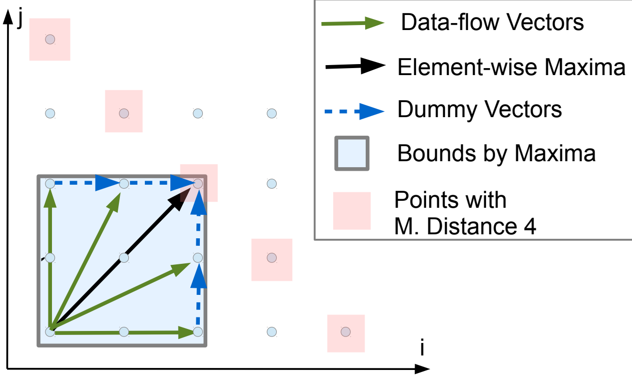


Figure 1: Illustration of Theorem 1 for the set of dependences with data-flow vectors $[2,0]$, $[2,1]$, $[1,2]$, and $[0,2]$. The element-wise maxima of the data-flow vectors correspond to the shortest UOV. The value produced by the bottom left iteration is used by the destination of the data-flow vectors. The data-flows induced by dummy dependences guarantees that the iteration pointed by the element-wise maxima is only executed after all iterations that depend on the bottom left. None of the other iterations with the same Manhattan distance (4) can be reached, since it requires backward data-flow along at least one of the axes.

by the closest[3] point from z that can be reached from all uses of z by following the dependences. Since the choice of z does not matter, let us use the origin, $\vec{0}$, to simplify our presentation. This allows us to use the data-flow vectors interchangeably with coordinate vectors.

Then, the hyper-rectangle with diagonal m includes all $\mathcal{I}$, and all bounds of the hyper-rectangle are touched by at least one dependence. Since all dependences in a fully tilable space are restricted to have non-negative data-flow vectors, no points within the hyper-rectangle can be reached by following dependences. Thus, it is clear that m is the closest common point that can be reached by those that touch the bounds. □

The theorem is illustrated in Figure 1, and is contrasted with the trivial UOV used by Strout et al. [19] in Figure 2.

The basic idea of inserting dummy dependences to restrict the possible schedule can be used beyond tilable schedules. One important corollary for sequential execution is the following.

Corollary 1 (Sequential Execution). *Given a set of dependences $\mathcal{I}$ in an n-dimensional space where lexicographic scan of the space is a legal schedule. Let m be the lexicographic maximum of the data-flow vectors of all dependences in $\mathcal{I}$. Then the shortest QUOV u for lexicographic execution is either m or the vector $[m_1+1, 0, \cdots, 0]$ where m_1 is the first element of m.*

Proof. For sequential execution, we may introduce dummy dependences to any lexicographically preceding point. Then the dependence, with a data-flow vector whose first element is m_1 can can subsume other dependences with lower values in the first element according to Lemma 1 by introducing appropriate dummy dependences.

[3] Shortest and closest are both in terms of Manhattan distance.

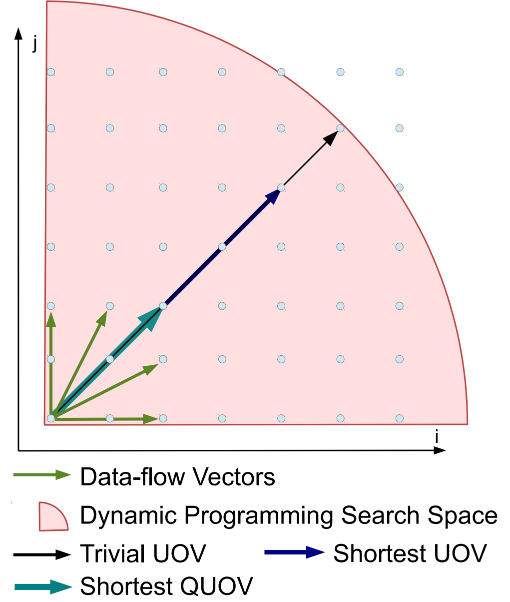


Figure 2: Comparison against the trivial UOV computed as proposed by Strout et al. [19] for the same set of dependences as in Figure 1. The trivial UOV is $[5,5]$ and it becomes the radius on the bounds of the search space for the dynamic programming algorithm proposed by Strout et al. [19]. In contrast, Theorem 1 gives the shortest QUOV by simply computing the element-wise maxima of the data-flow vectors. Furthermore, the shortest fully universal UOV is twice as long as the shortest QUOV, since the unit length dummy dependences cannot be assumed. The search space is bounded by a sphere since Euclidean distance is used by Strout et al. [19], but can be adapted to Manhattan distance.

For the remaining dependences there are two possibilities:

- We may use dummy dependences of the form $[1, *, \cdots, *]$ to let a dependence with data-flow vector $[m_1+1, 0, \cdots, 0]$ subsume all remaining dependences.
- We may use m as the QUOV following Theorem 1.

It is obvious that when the former options is used, $[m_1+1, 0, \cdots, 0]$ is the shortest. The optimality of the latter case follows from Theorem 1. Thus, the shortest UOV is the shortest among these two options. □

Note that m can be shorter than $[m_1+1, 0, \cdots, 0]$ only when $m = [m_1, 0, \cdots, 0]$. In addition, although the above corollary can be applied to tilable iteration spaces, the tilability may be lost due to memory-based dependences introduced by the allocation.

The allocation given by the above corollary is not schedule-independent at all. It is an analytical solution to the storage mapping of uniform dependence programs, where the schedule is the lexicographic scan of the iteration space.

The following corollary can trivially be established by the combination of the above.

Corollary 2 (Sequence of Tilable Spaces). *Given a set of dependences $\mathcal{I}$ in a space where a subset of*

```
for (i=0:N)
    S1[i] = foo();
for (j=0:N)
    S2[j] = bar(S1[j]);
```

(a) When $\theta_{S1} = (i \rightarrow 0, i, 0)$ and $\theta_{S2} = (j \rightarrow 1, j, 0)$.

```
for (i=0:N)
    S1[i] = foo();
    S2[i] = bar(S1[i]);
```

(b) When $\theta_{S1} = (i \rightarrow 0, i, 0)$ and $\theta_{S2} = (j \rightarrow 0, j, 1)$.

Figure 3: Two possible schedules for statements `S1` and `S2`. Note that statement `S1` in Figure 3a requires $O(N)$ memory whereas it only requires a scalar in Figure 3b, although the code shown is still in single assignment.

the dimensions are tilable, and lexicographic scan is legal for other dimensions, the shortest QUOV u for tiled execution of the tilable space, and sequential execution of the rest is the combination of vectors computed for each contiguous subset of either sequential or tilable spaces.

Note that the above corollary only takes effect when there are sequential subsets with at least two contiguous dimensions. When a single sequential dimension is surrounded by tilable dimensions, its element-wise maxima and lexicographic maxima are equivalent.

Using the above, the shortest QUOV for sequential, tiled, or a hybrid combination, can be computed very efficiently.

5. HANDLING OF PROGRAMS WITH MULTIPLE STATEMENTS

In many programs, there are multiple statements depending on each other. The original formulation of UOVs are for single statement programs [19]. In this section, we show that the concept can be adapted to multi-statement programs, knowing that we restrict the universe to tiled execution of the iteration space.

5.1 Limitations of UOV-based Allocation

Allocations based on UOVs have a strong property that they are valid for any legal schedule. Here, the schedule is not limited to affine schedules in the polyhedral model, and time stamps to each operation can be assigned arbitrarily, as long as they respect the dependences. The concept of UOV applies to reuse among writes to a single common space, and relies on the fact that every iteration writes to the same space (or array.) Different statements may write to different arrays, in programs with multiple statements, and hence UOV cannot be directly used.

For example, consider a program with two statements `S1` and `S2`:

- $\mathcal{D}_{S1} = \{i | 0 \leq i \leq N\}$
- $\mathcal{D}_{S2} = \{j | 0 \leq j \leq N\}$

where the dependence is such that iteration x of `S1` must executed before x of `S2`.

Figure 3 illustrates two possible schedules and its implications on memory usage. Note that we do not discuss storage mapping for $S2$, since it is not used within the code fragment above. A *fully universal* UOV-based allocation would have to take account for such variations of a schedule, but such extension may not even make sense. When two statements are scheduled differently, the dependence between two statements in the scheduled space may no longer be uniform.

When we apply the concept of UOV for tiling, we are no longer interested in arbitrary schedules. We first apply all non-tiling scheduling decisions before performing storage mapping. Therefore, the only change in the execution order comes from a tiling transformation viewed as a post-processing, so the same "schedule" applied to all statements involved. This allows us to extend the concept of UOVs to imperfectly nested loops.

5.2 Handling of Statement Ordering

When the ordering dimensions are represented as constant dimensions, the elements in the UOV require special handling. In the original formulation there is only one statement, and thus every iteration point writes to the same array. When multiple statements exist in a program, the iteration space of a statement is a subset of the combined space, and are made disjoint by statement ordering dimensions. Thus, not all points in the common space correspond to a write, and this affects how the UOV is interpreted.

Consider the program in Figure 3b. The only dependence involving `S1` has data-flow $[0, 0, 1]$, and since it is the only dependence, it is the shortest UOV. Literal interpretation of this vector as UOV means that the iteration $z + [0, 0, 1]$ can safely overwrite the value produced by z. However, this interpretation does not make sense in the context of by-statement allocation, since statement `S2` at $z+[0, 0, 1]$ writes to a different array.

We handle multi-statement programs by removing d out $d + 1$ constant dimensions for the purpose of UOV-based analysis. We apply the following rule to remove constant dimensions by transferring the information carried by these dimensions to others. Let v be a data-flow vector of a dependence in the scheduled space. We first apply the following rule to the vector v:

- For each constant dimension $x > 0$, where $v_x > 0$, set $v_{x-1} = max(v_{x-1}, 1)$

Once the above rule is applied, all the constant dimensions, except for the first one, are removed to obtain $d + 1$ dimensional data-flow vector. We repeatedly apply the above to all dependences in the program, resulting with a set of data-flow vectors with $d + 1$ dimensions.

We justify the above rule in the following. When the constant dimension of the data-flow is greater than 0, it means that some textually later statement uses the produced value. With respect to this particular dependence, the memory location may be reused once this textually later statement has been executed. However, there is always exactly one iteration of a specific statement in a constant dimension, since it is an artificial dimension for statement ordering. Therefore, the earliest possible iteration that can overwrite the memory location in question is in the next iteration of the immediate surrounding loop dimension.

For example, the only dependence of `S1` in Figure 3b is $[0, 0, 1]$. The value produced by `S1` at i is used by `S2` at i, but only overwritten by another iteration of `S1` at $i + 1$. Therefore, we transfer the dependence information to a

```
for (t=0:T)
   for (i=0:N)
      A[i] = foo(A[i]);              //S1
   for (i=1:N)
      A[i] = bar(A[i-1], A[i]);    //S2
```

Figure 4: Example to illustrate influences of boundary cases. Note that the value produced by S1 is last used by S2 of the same outer loop iteration, except for when $i = 0$, in which case it is used again by S1 at $[t + 1, 0]$.

preceding dimension.

Projecting a $d+1$ dimensional domain along a vector does indeed produce a domain with d dimensions. This is because in the presence of imperfect nests, d dimensional storage may be required for d dimensional statements even with uniform dependences. Consider the code in Figure 3a. The only use of S1 is by S2 with data-flow $[1, 0, 0]$ in the scheduled space. Applying the rule described above removes the last dimension, yielding $[1, 0]$ as the QUOV. Projecting the statement domain of S1 (in the scheduled space with d constant dimensions removed) along the vector $[1, 0]$ gives a two-dimensional domain: $\{i, x|0 \leq i \leq N \wedge x = 0\}$. Although the domain is two-dimensional, it is effectively one-dimensional because of the equality in the constant dimension.

It is also important to note a special case when the UOV takes the form: $[0, \cdots, 0, 1]$. When the last constant dimension is the only non-zero entry, it is obvious that the statement requires only a scalar, since its immediately consumed.

6. UOV GUIDED INDEX SET SPLITTING

In the polyhedral representation of programs there are usually boundary cases that behave differently from the rest. For instance, the first iteration of a loop may read from inputs, whereas successive iterations use values computed by previous iterations.

In the polyhedral model, storage mapping is usually computed for each statement. With pseudo-projective allocations, the same allocation must be used for all points in the statement domain. Thus, dependences that only exist at the boundaries influence the entire allocation.

For example, consider the code in Figure 4. The value produced by S1 at $[t, i]$ is last used by S2 at $[t, i+1]$ for $i > 0$. However, the value produced by S1 at $[t, 0]$ is last used by S1 at $[t + 1, 0]$. Thus, the storage mapping for S1 must ensure that a value produced at $[t, i]$ is live until $[t + 1, i]$ for all instances of S1. This clearly leads to wasteful allocations, and our goal is to avoid them.

One solution to the problem is to apply a form of *index set splitting* [9] such that the boundary cases and common cases have different mappings. In the example above, we wish to use piece-wise mappings for S1 at $[t, i]$ where the mapping is different for two disjoint sets $i = 0$ and $i > 0$. This reduces the memory usage from an array of size $N + 1$ to 2 scalars.

Once, the *pieces* are computed, application of piece-wise mappings can done through splitting the statements (nodes) as defined by the pieces, and then applying a separate mapping to each of the statements after split. Thus, the only interesting problem that remain is finding meaningful splits.

In this section, we present an algorithm for finding the split with the goal of minimizing memory usage. The algorithm is guided by Universal Occupancy Vectors and works best with UOV-based allocations. The goal of our index set splitting is to isolate boundaries that require longer lifetime than the main body. Thus, we are interested in a subdomain of a statement with a different dependence pattern than that in the rest of the statement's domain. We focus on boundary domains that contain at least one equality. The approach may be generalized to boundary planes of constant thickness using Thick Face Lattices [11].

The original index-set splitting [9] aimed at finding better schedules. The quality of a split is measured by its influence on possible schedules: whether different scheduling functions for each piece yields a better schedule.

In our case, the goal is different. Our starting point is the program after affine scheduling, and we are now interested in finding storage mappings. When the dependence pattern is the same at all points in the domain, splitting cannot improve the quality of the storage mapping. Since the dependence pattern is the same, the same storage mapping will be used for all pieces (with a possible exception of the cases when the split introduces equalities or other properties related to shape of the domains). Because two points that may have been in the nullspace of the projection may now be split into different pieces, the number of points that can share the same memory location may be reduced as the result of splitting.

Thus, as a general measure of quality, we seek to ensure that a split influences the choice of storage mapping for each piece. The obvious case when splitting is useless is when a dependence function at a boundary is also in the main part. We present Algorithm 1 based on this intuition to reduce the number of splits.

The intuition of the algorithm is that we start with all dependences with equalities in their domain as candidate pieces. Then we remove some of the dependences where splitting does not improve the allocation from candidate pieces. The obvious case is when the same dependence function exists in the non-boundary cases (i.e., dependences with no equalities in their domain). In addition, more sophisticated exclusion is performed using Theorem 1.

The algorithm can also be easily adapted for non-uniform programs. It may also be specialized/generalized by adding more dependence elimination rules to Step 2. This requires a method similar to Lemma 1 for other memory allocation methods.

Example

Let us describe the algorithm in more detail with an example. The statement $S1$ in Figure 4 has three data-flow dependences:

- $\mathcal{I}_1 = S1[t, i] \rightarrow S2[t, i]$ when $0 \leq i \leq N$
- $\mathcal{I}_2 = S1[t, i] \rightarrow S2[t, i + 1]$ when $i < N$
- $\mathcal{I}_3 = S1[t, i] \rightarrow S1[t + 1, i]$ when $i = 0$

The need for index-set splitting does not arise until some prior scheduling fuses the two inner loops. Let the scheduling functions be:

- $\theta_{S1} = (t, i \rightarrow 0, t, 0, i, 0)$
- $\theta_{S2} = (t, i \rightarrow 0, t, 0, i, 1)$

Algorithm 1 UOV-Guided Split

Input:

$\mathcal{I}$: Set of uniform dependences that depend on a statement S. A dependence is a pair $\langle f, D \rangle$ where f is the dependence function and D is a domain. The domain is the constraints on the producer statement.

Output:

$\mathcal{P}$: A partition of D_S, the domain of S, where each element defines a piece of the split.

Algorithm:

We first inspect the domain of dependences in $\mathcal{I}$ to detect equalities.

Let

$\mathcal{I}_b$ be the set of dependences with equalities, and

$\mathcal{I}_m$ be the set of those without equalities.

Then,

1. **foreach** $\langle f, D \rangle \in \mathcal{I}_b$,

   if $\exists \langle g, E \rangle \in \mathcal{I}_m; f = g$ **then** remove $\langle f, D \rangle$ from $\mathcal{I}_b$

2. Further remove dependences from $\mathcal{I}_b$ using the following *if applicable*:

   (a) Theorem 1 and its corollaries. The following steps are only for Theorem 1.

   Let m be the element-wise maxima of dataflow vectors in $\mathcal{I}_m$.
   foreach $\langle f, D \rangle \in \mathcal{I}_b$
   – Let v be the dataflow vector of f.
   – **if** $\forall_i : v_i \leq m_i$ **then** remove $\langle f, D \rangle$ from $\mathcal{I}_b$.

3. Group the remaining dependences in $\mathcal{I}_b$ into groups $\mathcal{G}_i$, $0 \leq i \leq n$, where $\forall X, Y \in \mathcal{G}_i; \mathcal{D}_X \cap \mathcal{D}_Y \neq \emptyset$. In other words, group the dependences with overlapping domains in the producer space.

4. **foreach** $i \in 0 \leq i \leq n$, $\mathcal{P}_i = \bigcup_{\forall X \in \mathcal{G}_i} \mathcal{D}_X$

5. **if** $n \geq 0$ **then** $\mathcal{P}_{n+1} = \mathcal{D}_S \setminus \bigcup_{i=0}^{n} \mathcal{P}_i$ **else** $\mathcal{P}_0 = \mathcal{D}_S$

The data-flow vectors in the scheduled space, after removing constant dimensions (we also remove the outer-most one since there is only one loop at the outer-most level) are:

- $\mathcal{I}_1^{'} = [0, 1]$ when $0 \leq i \leq N$
- $\mathcal{I}_2^{'} = [0, 1]$ when $i < N$
- $\mathcal{I}_3^{'} = [1, 0]$ when $i = 0$

As the pre-processing step, we separate the dependences into two sets based on the equalities in the domain:

- $\mathcal{I}_b = \{\mathcal{I}_3^{'}\}$; those with equalities, and
- $\mathcal{I}_m = \{\mathcal{I}_1^{'}, \mathcal{I}_2^{'}\}$; those without equalities.

Then we use Step 1 to eliminate identical dependences. If the same dependence function is both in the boundary and the main body, separating the boundary does not reduce the number of distinct dependences to be considered. Therefore, splitting such domain does not positively influence the storage mapping, and hence is removed from further consideration. Since $\mathcal{I}_3^{'}$ is different from the other two, this step does not change the set of dependences for this example.

Step 2a is the core of this algorithm. The general idea is the same as Step 1, but we use additional properties of UOV-based allocation to identify more cases where splitting does not positively influence the storage mapping.

Theorem 1 states that if all elements of a data-flow vector are less than the corresponding element of the element-wise maxima of all data-flow vectors under consideration, the dependence does not influence shortest QUOV. Therefore, if the candidate dependence to split does not contribute to the element-wise maxima, the split is useless in terms of further shortening the QUOV. As an illustration, consider the rectangle in Figure 1 defined by the element-wise maxima. If separating a dependence does not shrink the size of the rectangle, the length of QUOV cannot be shortened.

In this example, the element-wise maxima of dependences in $\mathcal{I}_m$ is $[0, 1]$. However, the boundary dependence has data-flow vector $[1, 0]$, and when combined, the element-wise maxima becomes $[1, 1]$. Therefore, the boundary dependence does contribute to the element-wise maxima, and is not removed from the candidate set in Step 2a. The dependences that are left in the set $\mathcal{I}_b$ after this step are the set of dependences that will be split from the main body.

Step 3 is a grouping step, where dependences to be split are grouped into those that have overlap in their domains.

Two sub-domains of a statement cannot be split into separate statements, unless the computation is duplicated. Although computing the values redundantly may be an option, we enforce the two statements to be jointly split as one additional statement. This step is irrelevant for our example, since our example only has one dependence in set $\mathcal{I}_b$.

The last step is a cleanup step, which adds the remainder of splits to the set of partitions.

Let `S1b` be the statement after splitting the domain of $\mathcal{I}_3$ from `S1`, and `S1a` be the remainder of the main body of `S1`. The domain of statements in the program are now:

- $\mathcal{D}_{S1a} = \{t, i | 0 \leq t \leq T \wedge 1 \leq i \leq N\}$
- $\mathcal{D}_{S1b} = \{t, i | 0 \leq t \leq T \wedge i = 0\}$
- $\mathcal{D}_{S2} = \{t, i | 0 \leq t \leq T \wedge 1 \leq i \leq N\}$

and the dependences are:

- $\mathcal{I}_1^* = S1a[t, i] \rightarrow S2[t, i]$ when $1 \leq i \leq N$
- $\mathcal{I}_2^* = S1a[t, i] \rightarrow S2[t, i + 1]$ when $i < N$
- $\mathcal{I}_3^* = S1b[t, i] \rightarrow S1a[t + 1, i]$ when $i = 0$

The QUOV for `S1a` is $[0, 0, 0, 0, 1]$ in the scheduled space with constant dimensions, which is the aforementioned special case, and only a scalar is required for `S1a`. The QUOV for `S1b` is $[1, 0]$ after removing the constant dimensions. The projection of its domain along this vector is also a constant, due to the equality in its domain. Thus, the storage requirement for `S1` in the original program becomes two scalars, which is much smaller than what is required without the splitting.

7. RELATED WORK

There is a lot of prior work on storage mappings for polyhedral programs [1, 4, 5, 13, 15, 19, 20, 21]. Most approaches focus on the case when the schedule for statements are given.

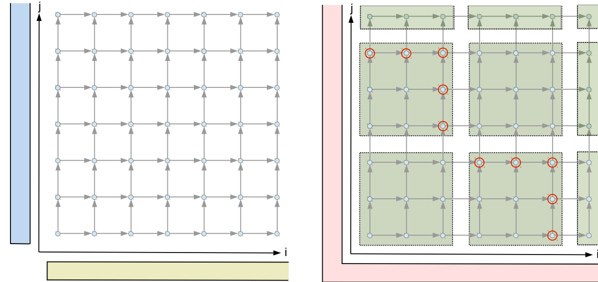
7.1 Efficiency of UOV-based Allocation

By the nature of its strategy, UOV-based allocation, including those using QUOVs, cannot yield more compact storage mappings compared to alternative strategies for a specific schedule. However, UOV-based allocation may not be as inefficient as one might think for programs that require $d - 1$ dimensional storage. The misconception is (at least partially) due to the trade-off between memory usage and parallelism that is often overlooked. Consider the following code fragment with Smith-Waterman(-like) dependences.

```
for (i=1:N)
   for (j=1:M)
       H[i,j] = foo(H[i-1,j], H[i,j-1]);
```

As illustrated in Figure 5, UOV-based allocation, even with QUOVs, gives a storage mapping that use $O(N + M)$ memory for this program. However, the program cannot be tiled if $O(N)$ or $O(M)$ storage mappings are used due to memory-based dependences. One approach to still accomplish tiled execution of this program is to transform the program such that the iteration space is tilable even when the memory-based dependences are under consideration.

For the above program, this requires a skewing as depicted in Figure 6. Once the skewing is applied so that $O(M)$ storage mapping can be tiled, the UOV-based allocation for the same program will also yield a storage mapping with $O(M)$ memory. Note that the skewed iteration space has less parallelism (longer critical path length) when compared to the original rectangular iteration space. The parallelism is effectively traded off with decreased memory usage.



(a) Iteration space and possible storage mappings (b) Set of live values in tiled execution

Figure 5: Iteration space of Smith-Waterman(-like) code and possible storage mappings. Figure 5a shows two possible storage mappings, either horizontal or vertical projection of the iteration space. The size of memory is $O(M)$ or $O(N)$, depending of the direction of the projection. Figure 5b shows the iteration space after tiling, and the values that are live after executing two tiles on the diagonal. Observe that neither projection (horizontal or vertical) can store all the live values. One example of a valid projective storage mapping is the projection along $[1, 1]$ that use $O(N + M)$ memory.

For this example, when the other condition (amount of parallelism) is equal, the allocation using UOVs is no worse than what is considered a more efficient allocation. This observation can be generalized to other instances of uniform dependence programs, such as Jacobi/Gauss-Seidel stencils. How to jointly find schedule and storage mappings to explore such trade-off is still an open problem.

7.2 Optimality of Projective Allocations

There is also an upper bound on dimension-wise optimality. Quilleré and Rajopadhye [15] show that the number of linearly independent projection vectors can be viewed as the primary criterion for optimality of storage mappings.

UOV-based allocation, as originally defined, was limited to allocations with one projection vector by its nature, and therefore, is limited to finding $d - 1$ dimensional storage for d dimensional iteration space. The additional optimizations we describe in Section 5 allow us to overcome this limitation, but for many, if not most, uniform dependence programs, the lower bound on the number of memory dimensions is $d - 1$. Therefore, UOV-based allocations are no more than a constant fold more expensive.

Now, the constant factor can become important, but Quilleré and Rajopadhye [15] give a number of examples to illustrate that the problem is subtle when the size of the iteration domain is parameterized. For some programs Lefebvre-Feautrier [13] gives a better memory footprint than the Quilleré-Rajopadhye method, while for others, it is worse.

It is easy to show that any multiple, by some integer greater than one, of a legal UOV uses more memory. Two

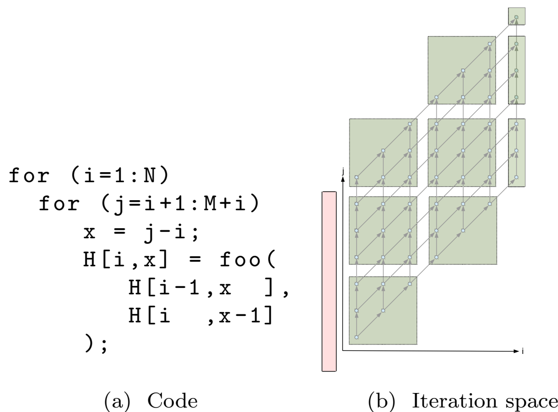


Figure 6: The code and iteration space after skewing the original program. It is easy to see that $[1, 1]$ is the shortest UOV for this program. With this UOV, the amount of memory used is $O(M)$.

UOVs that are not constant multiples of each other are often difficult to compare. For example, memory usage of two allocations based on UOVs $[1, 1]$ and $[2, 0]$ are only parametrically comparable. With $N \times M$ iteration space, the former use $N + M$ and the latter use $2N$. The optimal allocation in such case depends on the values of N and M that are not known until run-time.

Informally, increasing the Manhattan distance will always increase memory usage by either increasing the GCD, and hence increasing the mod factor, or by increasing the angle of the projection, and hence increasing the size of the projected space.

7.3 Parametric Tile Sizes

Our main motivation for QUOVs is to find storage mappings that are valid for tiled execution by any (legal) tile sizes. Schedule-dependent approaches cannot provide storage mappings for parametric tile sizes due to its non-affine nature.

It is possible to provide tile coordinates and tile sizes as additional parameters to the polyhedral representation, and then apply polyhedral storage mappings for each tile individually. Although this approach makes sense in certain contexts (e.g., [10]), it is not suitable for others (e.g., shared memory parallelization.) As shown in Figure 6, UOV-based allocation maps iterations from different tiles to a single memory location, allowing inter-tile reuse of storage. There is no need to transfer data from one tile to another in UOV-based allocation.

7.4 Affine Occupancy Vectors

Thies et al. [20] present an extension to the concept of UOV to affine schedules, named Affine Occupancy Vectors. They restrict the universality to affine scheduling, rather than the full universe. Although the idea of restricting the universality has some similarities with our work, the restricted universe is still the entire affine scheduling space. In addition, they only handle one-dimensional affine schedules.

8. CONCLUSIONS

We have presented a series of extensions to the Schedule-Independent Storage Mapping. Although UOVs were originally used for schedule-independent mappings, our extensions restrict the universality of the occupancy vectors to analyze a specific class of schedules; tiling.

For such a restricted universe, Quasi-UOVs can be shorter than fully universal ones, leading to more compact memory. We can also take advantage of its properties to directly find the shortest QUOV.

Although UOV-based allocations are limited to uniform dependence programs, storage mappings that are legal for a class of schedules is an interesting alternative to most memory allocation methods that require schedules to be given.

Our extensions aim to make UOV-based allocations more practical by providing efficient method for finding the shortest UOV for a smaller, but an important universe, tilable programs.

9. REFERENCES

[1] C. Alias, F. Baray, and A. Darte. Bee+Cl@k: an implementation of lattice-based array contraction in the source-to-source translator rose. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Language, Compiler and Tool Support for Embedded Systems*, volume 13, pages 73–82, 2007.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th IEEE International Conference on Parallel Architecture and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[4] Y. Bouchebaba and F. Coelho. Tiling and memory reuse for sequences of nested loops. In *Proceedings of the 8th International Euro-Par Conference*, volume 2400, page 255, 2002.

[5] A. Cohen. Parallelization via constrained storage mapping optimization. In *Proceedings of the International Symposium on High Performance Computing*, pages 83–94, 1999.

[6] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[7] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

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

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

[10] S. Guelton, A. Guinet, and R. Keryell. Building retargetable and efficient compilers for multimedia instruction sets. In *2011 International Conference on*

Parallel Architectures and Compilation Techniques, pages 169–170, 2011.

[11] G. Gupta and S. Rajopadhye. Simplifying reductions. In *Proceedings of the 33rd ACM Conference on Principles of Programming Languages*, PoPL '06, pages 30–41, New York, NY, USA, Dec 2006. ACM.

[12] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '88, pages 319–329. ACM, 1988.

[13] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[14] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[15] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[16] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[17] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2 of *ICPP '90*, pages 179–186, 1990.

[18] R. Schreiber and J. J. Dongarra. Automatic blocking of nested loops. Technical report, 1990.

[19] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. *ACM SIGOPS Operating Systems Review*, 32(5):24–33, 1998.

[20] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation*, PLDI '01, pages 232–242. ACM, 2001.

[21] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Proceedings of the 2nd International Euro-Par Conference*, pages 389–397, 1996.

[22] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.

On Demand Parametric Array Dataflow Analysis

Sven Verdoolaege
Consultant for LIACS, Leiden
INRIA/ENS, Paris
Sven.Verdoolaege@ens.fr

Hristo Nikolov
LIACS, Leiden
nikolov@liacs.nl

Todor Stefanov
LIACS, Leiden
stefanov@liacs.nl

ABSTRACT

We present a novel approach for exact array dataflow analysis in the presence of constructs that are not static affine. The approach is similar to that of fuzzy array dataflow analysis in that it also introduces parameters that represent information that is only available at run-time, but the parameters have a different meaning and are analyzed before they are introduced. The approach was motivated by our work on process networks, but should be generally useful since fewer parameters are introduced on larger inputs. We include some preliminary experimental results.

1. INTRODUCTION AND MOTIVATION

Array dataflow analysis [12] (also known as value-based dependence analysis [20]) is of crucial importance in the polyhedral framework with applications in array expansion [2, 10], scheduling [13], equivalence checking [5, 27] optimizing computation/communication overlap in MPI programs [18] and the derivation of process networks [24, 28] to name but a few. For program fragments that are static affine, dataflow analysis can be performed exactly [12, 15, 20]. For programs that contain certain dynamic and/or non-affine constructs, both exact [4] and approximate [21] approaches have been proposed. In particular, fuzzy array dataflow analysis (FADA) [4] introduces additional parameters, whose values depend on run-time information, that allow the dependences to be represented exactly. After all parameters have been introduced, certain properties on the parameters are derived that allow for a simplification of the result. In the end, the parameters may also be projected out, resulting in approximate but static dependences.

The initial motivation for our new approach stems from our work on the derivation of process networks. A process network derived from a sequential program is essentially a refinement of the dataflow graph of the program, where nodes in the graph correspond to processes and edges to communication channels, and is mainly used to represent the task-level parallelism available in the program. These process networks may then be mapped to various hardware implementations [17]. The parameters introduced by FADA can be used to construct control channels between the different processes [22]. Unfortunately, preliminary experiments with the only known publicly available implementation of FADA [6] have shown that this approach tends to introduce too many parameters to be practically useful, whence the need for an alternative approach. For our application, we are mainly interested in dynamic and/or non-affine conditions and this is currently the only extension beyond static affine programs that we support.

Our approach shares many similarities with FADA and is therefore also useful for other applications of this approach [3] (a description of these other applications is however beyond the scope of the present paper). In particular, we also introduce parameters whose values depend on run-time information. However, our parameters have a *different meaning*, essentially representing the last iteration of a potential source that was executed, and we analyze their effect *before* they are introduced. This allows us to (typically) introduce *fewer* parameters, resulting in simpler dependence relations that can be computed more efficiently. Unlike the FADA approach, our approach does not rely on a resolution engine, but instead performs operations on affine sets and relations to determine which parameters to add and which constraints they should satisfy.

We start with a description of our representation of static affine programs in Section 2 and an overview of standard dataflow analysis in Section 3. These sections also introduce notation that is used in later sections. The representation of dynamic conditions in the input is explained in Section 4, while the representation of dynamic dependence relations is explained in Section 5. Section 6 describes the computation of dynamic dependence relations. In Section 7, we discuss some extensions. We conclude with preliminary experimental results in Section 8 and a comparison to related work in Section 9.

2. PROGRAM REPRESENTATION

Each input program is represented using a polyhedral model [14], consisting of *iteration domains*, *access relations*, *dependence relations* and a *schedule*. Each statement has an associated iteration domain containing the values of the iterators of the enclosing loops for which the statement is executed. For example, the iteration domain of the statement in Line 14 of Listing 1 is

$$\{\, \mathtt{H}(k, i, j) \mid 0 \le k, j \le 99 \wedge 0 \le i < 99 \,\}. \tag{1}$$

Note that `n` is modified inside the program and can therefore not be treated as a parameter. This iteration domain is therefore an overapproximation of the set of executed iterations, based on the bounds on `n` specified by the user in Line 3. The mechanism for filtering out the iterations that have not actually been executed is explained in Section 4. An access relation maps elements of an iteration domain to the element(s) of an array domain accessed by that iteration of the associated statement through some array reference. For example, the access to `a` in the same statement is

```
1        int n, m;
2        int a[100][100];
3        #pragma value_bounds n 0 99
4        #pragma value_bounds m 0 100
5
6   N1:  n = f();
7        for (int k = 0; k < 100; ++k) {
8   M:       m = g();
9            for (int i = 0; i < m; ++i)
10               for (int j = 0; j <= n; ++j)
11  A:               a[j][i] = g();
12           for (int i = 0; i < n; ++i)
13               for (int j = 0; j < m; ++j)
14  H:               h(i, j, a[i + 1][j]);
15  N2:      n = f();
16       }
```

Listing 1: Code with locally static conditions

represented as $\{\,\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)\,\}$, simplified with respect to the iteration domain. In this paper, a dependence relation maps a (nested) read access relation to the write access relation that wrote the value being read by the read access. For example, the dependence relation for the above read access is $\{\,(\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \to (\mathtt{A}(k,j,i+1) \to \mathtt{a}(i+1,j)) \mid 0 \le k,j \le 99 \land 0 \le i < 99\,\}$, while the dependence relation for the read of `n` in the bound of the enclosing loop (simplified with respect to the iteration domain) is

$$\begin{aligned}&\{\,(\mathtt{H}(0,i,j) \to \mathtt{n}()) \to (\mathtt{N1}() \to \mathtt{n}())\,\} \cup \\ &\{\,(\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k \ge 1\,\}.\end{aligned} \tag{2}$$

Note that the above access and dependence relations are all single-valued functions. We will, however, also use the "$\to$" notations for other relations that may not represent single-valued functions. The schedule maps the union of all iteration domains to a common space where the execution order of the corresponding statement iterations is determined by the lexicographical order.

Each of the above sets and relations is represented in `isl` [25]. The iteration domains, the access relations and the initial schedule are extracted using `pet` [26], while the construction of the dependence relations is the topic of the present paper. In the base case, the input program consists of expression statements, `if` conditions and `for` loops with only static quasi-affine index expressions, conditions and bounds. The representation of dynamic (or non-affine) conditions, the main focus of the present paper, is explained in Section 4. Dynamic loop conditions and dynamic index expressions are briefly discussed in Section 7. The initial schedule describes an ordering that corresponds to the original execution order. To simplify the exposition, we will not take arbitrary initial schedules as input, but instead exploit the positions of the statements inside the abstract syntax tree, in particular the number of shared outer loops and the relative order of pairs of statements.

3. STANDARD DATAFLOW ANALYSIS

Several algorithms have been proposed in the literature for performing dataflow analysis in the case of static affine programs [12, 15, 20]. Our implementation in `isl` is a variation of these algorithms. We do not claim any novelty in this implementation and we will not describe the implementation in detail here. Instead, we only highlight those aspects that are important for understanding the remainder of this paper.

Dataflow analysis is performed separately for each read access (a.k.a., "sink" or "consumer") C. For each such read access we consider all the write accesses (a.k.a., "potential source" or "producer") P that access the same array, ordering them such that the "closest" are considered first. Let A_C and A_P be the corresponding access relations. Furthermore, let B represent the relative ordering of the statements. That is, if $\mathcal{I}$ is the union of all iteration domains, then B is a binary relation on $\mathcal{I}$ with $\mathbf{i} \to \mathbf{j} \in B$ iff $\mathbf{j}$ is executed before $\mathbf{i}$. For example, $\{\,\mathtt{H}(k,i,j) \to \mathtt{N2}(k') \mid 0 \le k,k',i,j \le 99 \land k' < k\,\}$ is a subset of B. Note that bold variable names are used to denote (named) integer tuples. Furthermore, if S is a set and if R, R_1 and R_2 are relations, then let 1_S be the identity relation on S, i.e,

$$1_S = \{\,\mathbf{s} \to \mathbf{s} \mid \mathbf{s} \in S\,\},$$

let R^{-1} be the inverse of R, i.e.,

$$R^{-1} = \{\,\mathbf{t} \to \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R\,\},$$

let $R_2 \circ R_1$ be the composition of R_1 and R_2, i.e.,

$$R_2 \circ R_1 = \{\,\mathbf{s} \to \mathbf{u} \mid \exists \mathbf{t} : \mathbf{s} \to \mathbf{t} \in R_1 \land \mathbf{t} \to \mathbf{u} \in R_2\,\},$$

let $R_1 \times R_2$ denote the cross product of R_1 and R_2 with

$$R_1 \times R_2 = \{\,(\mathbf{s} \to \mathbf{t}) \to (\mathbf{u} \to \mathbf{v}) \mid \mathbf{s} \to \mathbf{u} \in R_1 \land \mathbf{t} \to \mathbf{v} \in R_2\,\}$$

and let $\underrightarrow{\operatorname{ran}} R$ map a nested copy of R to its range, i.e.,

$$\underrightarrow{\operatorname{ran}} R = \{\,(\mathbf{s} \to \mathbf{t}) \to \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R\,\}.$$

If the value read by an element in the domain of A_C was written inside the program fragment under consideration, then it was written by one of the domain elements of one of those write access relations A_P. In particular, it is an element of the image of the relation $\left(A_P^{-1} \circ A_C\right) \cap B$ for some P. Since we want to keep track of the array elements being accessed, we can extend the above computation to

$$D_{C,P}^{\text{mem}} = \left((\underrightarrow{\operatorname{ran}} A_P)^{-1} \circ (\underrightarrow{\operatorname{ran}} A_C)\right) \cap (B \times 1_{\mathcal{A}}), \tag{3}$$

where $D_{C,P}^{\text{mem}}$ refers to the "memory based" dependences of C on P and $\mathcal{A}$ is the union of all array domains. For example, if $A_P = \{\mathtt{N2}(k) \to \mathtt{n}() \mid 0 \le k \le 99\}$ then $\underrightarrow{\operatorname{ran}} A_P = \{\,(\mathtt{N2}(k) \to \mathtt{n}()) \to \mathtt{n}() \mid 0 \le k \le 99\,\}$. Combining this with $A_C = \{\,\mathtt{H}(k,i,j) \to \mathtt{n}() \mid 0 \le k,i,j \le 99\,\}$, we obtain $D_{C,P}^{\text{mem}} = \{\,(\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k') \to \mathtt{n}()) \mid 0 \le k,k',i,j \le 99 \land k' < k\,\}$.

If the iteration domain of the statement containing C has dimension d, then we first consider elements of the D_C^{mem} relations such that the first d iterators in domain and range have the same value, then $d-1$ iterators, continuing until we consider the case where 0 iterators have the same value. Let ℓ denote this number of equal iterators. For each value of ℓ, we consider the potential sources P that share at least ℓ outer loops, compute the maximum image element of $D_{C,P}^{\text{mem}} \cap (E_=^\ell \times 1_{\mathcal{A}})$, with $E_=^\ell$ expressing that exactly ℓ outer iterators have the same value, adding constraints that ensure this maximal element is executed after any previously computed maximum at this level. When moving from ℓ to $\ell-1$, we only consider those elements from the sink

access relation for which no source has been found so far. The main operation in this computation is then that of the partial lexicographical maximum of a relation M on a domain U, which returns a relation mapping elements u of U to the lexicographically greatest element associated to u by M, along with a set of elements in U that do not have any images in M. Let us define some more operations on sets and relations. Specifically, if S_1 and S_2 are sets and if R is a relation, then the universal relation from S_1 to S_2 is denoted

$$S_1 \to S_2 = \{ \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \in S_1 \wedge \mathbf{t} \in S_2 \},$$

while the domain and range of R are denoted

$$\operatorname{dom} R = \{ \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R \}.$$

and

$$\operatorname{ran} R = \{ \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R \}.$$

The lexicographical maximum of a relation R is then denoted $\operatorname{lexmax} R$ and is equal to

$$\{ \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R \wedge (\forall \mathbf{t}' : \mathbf{s} \to \mathbf{t}' \in R \Rightarrow \mathbf{t}' \preccurlyeq \mathbf{t}) \},$$

"$\preccurlyeq$" representing the lexicographical order, Finally, the partial lexicographical maximum of M on U is

$$\underset{U}{\operatorname{lexmax}} M = (\operatorname{lexmax}(M \cap (U \to \operatorname{ran} M)), U \setminus \operatorname{dom} M) \tag{4}$$

The partial lexicographical maximum can be computed using parametric integer programming [11]. In the example, the read of `n` in Line 12 can only have 0 equal loop iterators with the write in Line 15. We therefore compute $\operatorname{lexmax}_U D^{\mathrm{mem}}_{C,P}$ with U the (nested) sink access relation. The result consists of the (simplified) relation $\{ (\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k > 0 \}$ and the set $\{ (\mathtt{H}(0,i,j) \to \mathtt{n}()) \}$. Sources for this latter part of the sink relation can be found in Line 6. The final result is shown in (2).

4. FILTERS

As explained in Section 2, if there are any dynamic conditions in the program, then the iteration domain may be an affine overapproximation of the set of executed iterations. To mark those iterations that are actually executed, we apply one or more *filters* to the iteration domain. These filters encode the dynamic conditions that determine whether an iteration in the iteration domain is actually executed. For example, as shown before, the iteration domain (1) is an overapproximation of the executed iterations of the statement in Line 14 of Listing 1. The filter on this iteration domain then expresses the dynamic conditions `i < n` and `j < m`. Each filter consists of a sequence of *filter access relations*, accessing variables that may be updated during the execution of the program, and a *filter value relation*, mapping statement iterations to the possible values of the dynamic variables for which the iteration is executed. For simplicity of exposition, we will assume that all the filter access relations are functions and that there is only one filter. The more general case is discussed in Appendix A. Non-affine conditions are treated as dynamic conditions.

More formally, let S be a statement and I_S its iteration domain. Furthermore, let the filter on S consist of a filter value relation V^S and n^S filter access relations F_i^S. We have $F_i^S \subseteq I_S \to (I_S \to \mathcal{A})$ and $V^S \subseteq I_S \to \mathbb{Z}^{n^S}$. That is, each filter access relation maps an iteration to an access of an array element and the filter value relation maps an iteration to a tuple of values. Moreover, each F_i^S is a function on the iteration domain. The application of the relation R to the set S is defined as

$$R(S) = \{ \mathbf{u} \mid \exists \mathbf{t} : \mathbf{t} \to \mathbf{u} \in R \wedge \mathbf{t} \in S \}.$$

Furthermore, let

$$(a_j)_{j=1}^{n}$$

be the n-tuple with elements a_j and let $\mathcal{V}(\{ \mathbf{j} \to \mathbf{b} \})$ be the value of $\mathbf{b}$ at $\mathbf{j}$. Iteration $\mathbf{k} \in I_S$ is then executed iff $\mathit{executed}^S(\mathbf{k})$ holds, with

$$\mathit{executed}^S(\mathbf{k}) = \left(\mathcal{V}\left(F_j^S(\mathbf{k}) \right) \right)_{j=1}^{n^S} \in V^S(\mathbf{k}), \tag{5}$$

where $F_j^S(\mathbf{k})$ is short for $F_j^S(\{ \mathbf{k} \})$. An access $\mathbf{k} \to \mathbf{a}$ from an iteration $\mathbf{k}$ is executed iff the iteration is executed, so that $\mathit{executed}^S((\mathbf{k} \to \mathbf{a})) = \mathit{executed}^S(\mathbf{k})$. Let $\underrightarrow{\operatorname{dom}} R$ map a nested copy of R to its domain, i.e.,

$$\underrightarrow{\operatorname{dom}} R = \{ (\mathbf{s} \to \mathbf{t}) \to \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R \},$$

then, initially, each filter access relation is a subset of the relation $\left(\underrightarrow{\operatorname{dom}}(\mathcal{I} \to \mathcal{A}) \right)^{-1}$. That is, each iteration is mapped to an access that takes place at the same iteration.

We consider two types of filters, one for the general case and one for the special case of "locally static affine" conditions. A condition is considered to be locally static affine if it is an affine expression in variables that are definitely not modified between the point where they are evaluated and all the statements that are guarded by the condition. Such variables are called "locally static". In this case, the accesses to the locally static variables themselves are used as filter accesses. Otherwise, a new statement is created that evaluates the condition and writes the resulting boolean value (0 or 1) to a virtual array, as in [26, Section 4.3]. This virtual array is then used as a filter access in a condition that simply evaluates the element of the virtual array. The first type of filter is allowed in both `if`-conditions and loop conditions, while the second type is in the current context only allowed in `if`-conditions. In Section 7.1, we explain how to also handle the second type in loop conditions.

Consider, for example, the code in Listing 1. The condition `i < n` in Line 12 references the variable `n` which is not a parameter because it is assigned in Line 6 and in Line 15. However, the value of `n` is clearly not changed between the condition in Line 12 and the statement in Line 14 governed by this condition. Similarly, `m` is also locally static for the statement. The filter for statement `H` therefore has filter access relations

$$F_1^{\mathtt{H}} = \{ \mathtt{H}(k,i,j) \to (\mathtt{H}(k,i,j) \to \mathtt{n}()) \} \tag{6}$$

and

$$F_2^{\mathtt{H}} = \{ \mathtt{H}(k,i,j) \to (\mathtt{H}(k,i,j) \to \mathtt{m}()) \}. \tag{7}$$

The filter value relation $V^{\mathtt{H}}$ is

$$\{ \mathtt{H}(k,i,j) \to (n,m) \mid 0 \le k \le 99 \wedge 0 \le i < n \wedge 0 \le j < m \}. \tag{8}$$

For convenience to the reader, the dimensions that correspond to the filters have been named after the arrays (in this case scalars) that are being accessed by the corresponding filter access relations. However, the reader should keep

```
1    state = 0;
2    while (1) {
3        sample = radioFrontend();
4        if (t(state)) {
5 D:         state = detect(sample);
6        } else {
7 C:         decode(sample, &state, &value0);
8            value1 = processSample0(value0);
9            processSample1(value1);
10       }
11   }
```

Listing 2: Code with dynamic conditions, adapted from [7, Figure 8.1]

in mind that these names are completely arbitrary. Note that the condition of the loop in Line 12 is allowed since it is locally static affine.

The code in Listing 2 has a generic dynamic condition in Line 4. `pet` introduces a separate virtual array, say $\mathtt{t}_0$, for storing the result of the condition and a separate statement, say $\mathtt{S}_0$, for computing this result. The access relation that writes to the filter is of the form

$$\{\, \mathtt{S}_0(i) \to \mathtt{t}_0(i) \,\}.$$

Note that `pet` introduces an implicit iterator (called i in this relation) with non-negative values [26, Section 3.3] for the loop `while (1)` in Line 2. The filter value relation of the statement in Line 5 is

$$V^{\mathtt{D}} = \{\, \mathtt{D}(i) \to (1) \mid i \geq 0 \,\}$$

with filter access relation

$$F^{\mathtt{D}} = \{\, \mathtt{D}(i) \to (\mathtt{D}(i) \to \mathtt{t}_0(i)) \,\}.$$

That is, the statement is executed for values of i greater than or equal to 0 such that $\mathtt{t}_0(i)$ at $\mathtt{D}(i)$ is equal to 1. (Recall that $\mathtt{t}_0(i)$ is a boolean variable that only attains values 0 and 1.) The filter value relation of the statement in Line 7 is

$$V^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (0) \mid i \geq 0 \,\} \tag{9}$$

with filter access relation

$$F^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (\mathtt{C}(i) \to \mathtt{t}_0(i)) \,\}. \tag{10}$$

Most of the analysis in Section 6 is based on filter values being equal or different in different iterations. We therefore need to be able to identify that filter values accessed in different iterations are actually the same. This information can be obtained by applying dataflow analysis on the arrays accessed by the filters. As explained below, the result of this analysis may or may not depend on any parameters as defined in Section 5. If any such parameters are involved, then we keep the original filter access relations. If, on the other hand, the resulting dependence relations do *not* depend on any such parameters, then the original filter access relations can be replaced by a composition with these dependence relations. For the code in Listing 1, the dependence relation for `n` in Line 12 is shown in (2). Applying this dependence relation to the filter access relation in (6) yields

$$\begin{aligned} F_1^{\mathtt{H}} = \{\, &\mathtt{H}(0,i,j) \to (\mathtt{N1}() \to \mathtt{n}()) \,\} \cup \\ &\{\, \mathtt{H}(k,i,j) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k \geq 1 \,\}. \end{aligned} \tag{11}$$

For the filter access relation in (10), no dataflow analysis needs to be performed since we know each element of the virtual array $\mathtt{t}_0$ is written by exactly one iteration of the statement $\mathtt{S}_0$. The filter access relation can therefore be replaced by

$$F^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (\mathtt{S}_0(i) \to \mathtt{t}_0(i)) \,\}. \tag{12}$$

In principle, the meaning of $\mathcal{V}$ depends on whether sources have been found for the filter array or not, i.e., whether we have been able to compose the original filter access relations with dependence relations or not. In particular, if sources have been found, then $\mathcal{V}(\{\, \mathbf{j} \to \mathbf{b} \,\})$ is the value of $\mathbf{b}$ *after* the write in iteration $\mathbf{j}$, while if sources have not been found, then $\mathcal{V}(\{\, \mathbf{j} \to \mathbf{b} \,\})$ is the value of $\mathbf{b}$ *before* the read in iteration $\mathbf{j}$. In practice, however, this difference is not important since filter accesses for which no sources have been found will never be matched to any other filter accesses.

5. PARAMETER REPRESENTATION

If the iteration domain of a potential source P is affected by any filters, then we cannot simply compute the lexicographical maximum in (4) since the maximal element in the range of M may not actually be executed, even if some of the other elements are. We will therefore introduce parameters that represent the "last executed" source iteration. By equating the iterators in the range of M to these parameters, the lexicographical maximization operation will simply return these parameters, which are guaranteed to represent an executed iteration.

The exact form and meaning of the parameters depends on whether we are considering the final result of the dataflow analysis or intermediate results. Let us first consider the final result. Let $D_{C,P}$ be the final dependence relation, the description of which may involve some parameters $\beta_C^Q(\mathbf{k})$ and $\boldsymbol{\lambda}_C^Q(\mathbf{k})$, where Q may be either P or some other potential source and $\mathbf{k}$ is an element of the sink access relation A_C. Note that in principle the parameters depend on the sink access $\mathbf{k}$, but as we will explain below, we do not need to make this dependence explicit in our representation. Let $D'_{C,P}$ be the result of projecting out all these parameters. The parameters then have the following meaning. The parameter $\beta_C^Q(\mathbf{k})$ is a boolean variable that expresses whether any of the elements in $D'_{C,P}(\mathbf{k})$ is executed. If $\beta_C^Q(\mathbf{k}) = 1$, then $\boldsymbol{\lambda}_C^Q(\mathbf{k})$ represents the last element in $D'_{C,P}(\mathbf{k})$ that is executed. (If $\beta_C^Q(\mathbf{k}) = 0$, then $\boldsymbol{\lambda}_C^Q(\mathbf{k})$ is undefined.) In other words, we have

$$\begin{aligned} \beta_C^P(\mathbf{k}) = 0 \Rightarrow &\forall \mathbf{j} \in D'_{C,P}(\mathbf{k}) : \neg \mathit{executed}^{S_P}(\mathbf{j}) \\ \beta_C^P(\mathbf{k}) = 1 \Rightarrow &\mathit{executed}^{S_P}(\boldsymbol{\lambda}_C^P(\mathbf{k})) \wedge \\ &\forall \mathbf{j} \in D'_{C,P}(\mathbf{k}) : \mathbf{j} \succ \boldsymbol{\lambda}_C^P(\mathbf{k}) \Rightarrow \neg \mathit{executed}^{S_P}(\mathbf{j}), \end{aligned} \tag{13}$$

with S_P the statement containing access P and *executed* as defined in (5). For example, let C by the access to `state` in the statement $\mathtt{S}_0$ evaluating the condition in Line 4 of Listing 2. Let P by the access to the same variable (`state`) from statement `C`. The dependence relation for the dependence of C on P is of the form $(\beta_C^P, \lambda_C^P) \to \{\, (\mathtt{S}_0(i) \to \mathtt{state}()) \to (\mathtt{C}(\lambda_C^P(i)) \to \mathtt{state}()) \mid \beta_C^P(i) = 1 \wedge i = \lambda_C^P(i) + 1 \geq 1 \,\}$. That is, there is only a dependence if access P (from statement `C`) was ever executed (before $\mathtt{S}_0(i)$) and if the last execution was in the previous iteration. Note that if the last execution

was in some earlier iteration, then C would not depend on the access in statement `C`, but on that in statement `D`. This result is obtained in Section 6.4.

During the computation, the resulting dependence relation is obviously not known, but we do know that it is a subset of $D^{\text{mem}}_{C,P}$ (3). The relation M in the current maximization problem (4) is also a subset of $D^{\text{mem}}_{C,P}$. We can therefore temporarily treat $\boldsymbol{\lambda}^P_C(\mathbf{k})$ as the last element of $D^{\text{mem}}_{C,P}(\mathbf{k})$ that is executed. Note that because we still assume static affine index expressions, the array element accessed by this last executed element of $D^{\text{mem}}_{C,P}(\mathbf{k})$ is necessarily the same as that accessed by $\mathbf{k}$. We can also exploit additional information to reduce the number of elements in the $\boldsymbol{\lambda}^P_C(\mathbf{k})$ vector that need to be explicitly represented. In particular, we know that the first ℓ iterators in the domain and range of M (with ℓ as in Section 3) are pairwise equal and that the same property holds for the previously considered maximization problems within the current dataflow problem, i.e., for the same C. This allows us to avoid introducing elements of the $\boldsymbol{\lambda}^P_C(\mathbf{k})$ vector until we really need them. In particular, we keep track in $\sigma^P_C(\mathbf{k})$ of the number of initial elements in the domain of $\mathbf{k}$ and in $\boldsymbol{\lambda}^P_C(\mathbf{k})$ that are (implicitly) equal to each other. Values of $\sigma^P_C(\mathbf{k})$ smaller than ℓ then mean that there is no element in $D^{\text{mem}}_{C,P}(\mathbf{k})$ that is executed and that shares the values of the first ℓ iterators with $\mathbf{k}$. As a special case, $\sigma^P_C(\mathbf{k}) < 0$ means that no element in $D^{\text{mem}}_{C,P}(\mathbf{k})$ is executed. Summarizing, (13) is replaced by

$$\begin{aligned} \sigma^P_C(\mathbf{k}) < \ell \Rightarrow & \forall \mathbf{j} \in D''_{C,P}(\mathbf{k}) : \neg \mathit{executed}^{S_P}(\mathbf{j}) \\ \sigma^P_C(\mathbf{k}) \geq \ell \Rightarrow & \mathit{executed}^{S_P}(\boldsymbol{\lambda}^P_C(\mathbf{k})) \wedge \\ & \forall \mathbf{j} \in D''_{C,P}(\mathbf{k}) : \mathbf{j} \succ \boldsymbol{\lambda}^P_C(\mathbf{k}) \Rightarrow \neg \mathit{executed}^{S_P}(\mathbf{j}), \end{aligned} \tag{14}$$

where $D''_{C,P} = D^{\text{mem}}_{C,P} \cap E^{\ell}_{\geq}$, with $E^{\ell}_{\geq}$ expressing that at least ℓ outer iterators have the same value.

After the dataflow computation has finished, we need to convert the intermediate representation (14) to the final representation (13). If $\ell^P_{\leq}$ is the smallest value of ℓ for which we had to apply parametrization for a given potential source P, then we do not need to introduce dimensions of $\boldsymbol{\lambda}^P_C$ before $\ell^P_{\leq}$. Instead, we need to make the equalities implied by σ^P explicit and set $\beta^P_C(\mathbf{k}) = 1$ when $\sigma^P \geq \ell^P_{\leq}$ and $\beta^P_C(\mathbf{k}) = 0$ when $\sigma^P < \ell^P_{\leq}$. The parameter σ^P can then be projected out. Besides dimensions before $\ell^P_{\leq}$, we also do not need to introduce dimensions of $\boldsymbol{\lambda}^P_C$ for which $D^{\text{mem}}_{C,P}(\mathbf{k})$ attains a single value (for any given value of $\mathbf{k}$) or that correspond to loops inside the innermost condition that is not static affine.

Since $\boldsymbol{\lambda}^P_C$ and σ^P_C depend on the sink iteration, it may appear that we would need to treat them as uninterpreted functions. During the entire computation, there is however no interaction between different sink iterations. That is, any of the intermediate relations during the computation only refers to a single sink iteration and the parameters $\boldsymbol{\lambda}^P_C$ and σ^P_C (if present) always refer to that single sink iteration. This means that we can keep the relation between $\boldsymbol{\lambda}^P_C$ and σ^P_C on one hand and $\mathbf{k}$ on the other hand entirely implicit and simply treat $\boldsymbol{\lambda}^P_C$ and σ^P_C as parameters. The intended meaning of those parameters is then the value of the corresponding functions at the particular value of $\mathbf{k}$ involved in the given access or dependence relation. This reasoning is essentially the same as that of [1] for showing that his $\boldsymbol{\alpha}$ vectors can be treated as parameters.

Algorithm 1: Parametric partial lexicographical maximum

```
(type, S1, S2) = parametrization(M, U)
if type = Input then
    M := intersect_range(M, S1)
    U := intersect(U, S2)
else if type = Empty then
    M := empty(M)
end
(R,E) = partial_lexmax(M, U)
if type = Output then
    R := intersect_range(R, S1)
end
```

6. PARAMETRIZATION

Recall that during dataflow analysis (Section 3), we frequently compute a partial lexicographical maximum of the form (4). The inputs to this operation are based on the static affine iteration domains and so we may need to introduce parameters representing the last executed source iteration (Section 5) to take into account the filters (Section 4) on the iteration domains. In particular, we replace a call "(R,E) = partial_lexmax(M, U)", corresponding to (4), by the pseudocode in Algorithm 1. The different types of parametrization in this code are explained in Section 6.1, while the determination of which of these parametrizations should be applied is explained in Section 6.3. The sets S_1 and S_2 used during the parametrization are constructed in Section 6.2 and Section 6.4.

6.1 Types of Parametrization

We are presented with a maximization problem of the form (4), with U a set of iterations of a sink C and M a relation between sink iterations and iterations of a potential source P and we want to decide if we need to introduce parameters. In principle, the result is that either parametrization is required (type = Input) or it is not required (type = None). However, we also consider a couple of other special cases (type = Empty and type = Output). In particular, we may find that it is impossible for any of the source iterations to execute given that the corresponding sink is executed. In such cases we want to avoid introducing parameters since there is no dependence and we therefore do not need any extra parameters to represent the dependence. However, we cannot indicate to the dataflow analysis that no parametrization is required (type = None) as then it would treat the source iterations as definitely being executed. Instead, we communicate to the dataflow analysis that M should be replaced by an empty relation (type = Empty). Another special case occurs when we are able to determine that no parametrization is required, but that this detection depends on information available about *other* potential sources. For reasons beyond the scope of the present paper, the construction of process networks can in this case be facilitated by introducing parameters anyway, but only on the *result* of the maximization problem rather than on the input of the maximization problem (type = Output). A schematic overview of the determination of the type of parametrization to apply is shown in Algorithm 2. The process is explained in more detail in Section 6.3.

6.2 Application

If parametrization is required, then we need to equate the source iteration to the parameters λ^P representing the last executed iteration of the potential source P (14). That is, the range of M (or, more precisely, the domain of the nested relation in this range) is intersected with

$$(\boldsymbol{\lambda}_I^P, \sigma^P) \to \{\, \mathtt{S}_P(\mathbf{j}) \mid \mathbf{j}_I = \boldsymbol{\lambda}_I^P \wedge \sigma^P \geq \ell \,\}, \qquad (15)$$

where I selects the elements of $\boldsymbol{\lambda}^P$ that need to be explicitly represented as explained in Section 5 and ℓ the number of equal outer iterators in the domain and range of M as in Section 3. These constraints determine the set S_1 in Algorithm 1. For example, let C be the read of `A` in Line 14 of Listing 1 and let P be the write in Line 11. The corresponding statements share at most one loop iterator. When $\ell = 1$, then we have that M is equal to

$$\{\, (\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \to (\mathtt{A}(k,j,i+1) \to \mathtt{a}(i+1,j)) \,\}, \qquad (16)$$

where the outer dimensions k are equal because $\ell = 1$ and the inner dimensions i' and j' are equal to j and $i+1$ because the same array element is accessed. As we will see in Section 6.3, no parametrization is required for this problem, but let us for illustrative purposes assume that we do want to apply parametrization. Dimension 0 of $\boldsymbol{\lambda}$ does not need to be introduced (yet) because $0 < \ell$. Since $D^{\mathrm{mem}}_{C,P}$ is equal to

$$\{\, (\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \\ \to (\mathtt{A}(k',j,i+1) \to \mathtt{a}(i+1,j)) \mid k' \leq k \,\},$$

the remaining dimensions of $\boldsymbol{\lambda}$ do not need to be introduced either because they are fully determined by $D^{\mathrm{mem}}_{C,P}$ (and therefore also by $D'_{C,P} \subseteq D^{\mathrm{mem}}_{C,P}$). In effect, we would only need to introduce σ^P with constraint $\sigma^P \geq 1$.

The parametrization of the source domain expresses that the source is (essentially) the last of the potential source iterations associated to the sink through $D^{\mathrm{mem}}_{C,P}$. However, it only does so through the introduction of parameters that implicitly depend on the sink iteration $\mathbf{k}$. The parametrization of the sink takes care of expressing that this last iteration, if it exists, actually belongs to $D^{\mathrm{mem}}_{C,P}(\mathbf{k})$. In particular, we first split the set of sink iterations U into two parts, one with associated potential source iterations and one without, i.e.,

$$U_1 = U \cap (\operatorname{dom} M) \qquad \text{and} \qquad U_2 = U \setminus (\operatorname{dom} M). \qquad (17)$$

The parametrization is only applied to U_1 and expresses that either none of the potential source iterations in $D^{\mathrm{mem}}_{C,P}(\mathbf{k})$ that share the first ℓ iterators is executed or that there is such an executed potential source iteration and that it belongs to $D^{\mathrm{mem}}_{C,P}(\mathbf{k})$. That is, the sink C is intersected with

$$(\boldsymbol{\lambda}^P, \sigma^P) \to \left\{\, \mathbf{k} \mid \sigma^P < \ell \vee \left(\boldsymbol{\lambda}^P \in D^{\mathrm{mem}}_{C,P}(\mathbf{k}) \wedge \sigma^P \geq \ell\right) \,\right\}. \qquad (18)$$

These constraints, together with those of Section 6.4 below, determine the set S_2 in Algorithm 1. In practice, we only introduce the same set of dimensions of the $\boldsymbol{\lambda}^P$ vector as introduced by (15). In particular, the second disjunct is obtained by applying the parametrization of (15) to the range of $D^{\mathrm{mem}}_{C,P}$ and computing the domain of the result.

6.3 Detection

Let us now explain in more detail the steps in the determination of which parametrization to apply as sketched in Algorithm 2. If M and/or U are empty or if P is not affected by any filter, then no parametrization is required (Line 2). Otherwise, we compute the possible values for the filter elements at the potential source, given that the sink is executed. Clearly, we can only do this if the sink is affected by a filter and if source and sink have some filter accesses in common. If the computed possible values are disjoint from the filter value relation on the source, then no source iteration is executed when the sink is executed and we indicate that M should be replaced by an empty relation (Line 6). Otherwise, we check if U references any parameters that were introduced by previous calls to the parametrization. If so, we use the constraints on those parameters and the filters of the associated (other) potential sources to derive extra information about the filters at the sink (Line 8). This derivation is explained in Section 6.3.2. The resulting information is then propagated again to potential source P and a new relation of possible values is computed. If this relation is disjoint from the filter value relation on the source, then we again indicate that M should be replaced by an empty relation (Line 10). Otherwise, if the computed relation is a subset of the filter value relation on the source, then we know the source is always executed and we indicate that either no parametrization is required (Line 13) or that parametrization is only required on the output of the maximization (Line 16), depending on whether the relation computed based on only information on the sink was already a subset of the filter value relation. Finally, if none of these cases apply, then parametrization is required (on the input of the maximization problem, Line 18).

Algorithm 2: Type of parametrization

1 **if** *M is empty, U is empty or there are no filters on the source* **then**
2 | **return** None
3 **end**
4 F := filters on the sink
5 **if** *filters on the source contradict F* **then**
6 | **return** Empty
7 **end**
8 F' := update(F, filters on other sources)
9 **if** *filters on the source contradict F'* **then**
10 | **return** Empty
11 **end**
12 **if** *filters on the source imply F* **then**
13 | **return** None
14 **end**
15 **if** *filters on the source imply F'* **then**
16 | **return** Output
17 **end**
18 **return** Input

6.3.1 Filter Values implied by the Sink

Let us illustrate the derivation of filter value constraints on the potential source from those on the sink based on an example. The general case is explained in Section A.2. In particular, let us reconsider the example from Section 6.2. Let M_1 be the result of projecting out the array elements from M, i.e.,

$$M_1 = \{\, \mathtt{H}(k,i,j) \to \mathtt{A}(k,j,i+1) \mid 0 \leq i \leq 98 \,\},$$

where we omit the constraints on j and k for brevity. The filters on statement A are similar to those on H in (6) and (7). The first of these was updated in (11) to take into account the filter source. The second can be updated to

$$F_2^{\mathtt{H}} = \{\, \mathtt{H}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le i \le 98 \,\}.$$

For statement A, we have, say,

$$F_1^{\mathtt{A}} = \{\, \mathtt{A}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le j \le 99 \,\}.$$

We want to check whether the fact that the sink is executed tells us anything about the values of these filter variables at the source. Note that if the sink is not executed, then it does not need any values and so there is no need to compute dataflow dependences for sink iterations that are not executed.

The first step is to check whether any of the source filter arrays are also accessed by the corresponding sink iterations. To do so, we pull back the filter access relations over M_1, resulting in

$$F_1^{\mathtt{A}} \circ M_1 = \{\, \mathtt{H}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le i \le 98 \,\},$$

where the constraints $0 \le i \le 98$ derive from $0 \le i+1 \le 99$ and the domain constraints of M_1. Since this relation is a subset of $F_2^{\mathtt{H}}$ i.e., $\forall \mathbf{j} \in M_1(\mathbf{k}) : F_1^{\mathtt{A}}(\mathbf{j}) \subseteq F_2^{\mathtt{H}}(\mathbf{k})$, and similarly for $F_2^{\mathtt{A}} \circ M_1 \subseteq F_1^{\mathtt{H}}$, we know that (5) also holds for the source filter accesses for all $\mathbf{j} \in M_1(\mathbf{k})$, i.e.,

$$\left(\mathcal{V}\left(F_i^{\mathtt{A}}(\mathbf{j})\right)\right)_{i=1}^2 \in V_1(\mathbf{k})$$

with V_1 derived from $V^{\mathtt{H}}$ in (8) by changing the order of the range dimensions to match the matching of the filters, i.e.,

$$\{\, \mathtt{H}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le i < n \wedge 0 \le j < m \,\}.$$

Note that in this particular example, we have $F_1^{\mathtt{A}} \circ M_1 = F_2^{\mathtt{H}}$ and $F_2^{\mathtt{A}} \circ M_1 = F_1^{\mathtt{H}}$ but equality is not required in the general case. There is also no need for every filter access relation on the source to match a filter access relation on the sink and vice versa. Besides reordering dimensions of $V^{\mathtt{H}}$, we may in the general case also have to project out dimensions and/or introduce unconstrained dimensions.

Precomposing V_1 with M_1^{-1}, we obtain a mapping from source iterations to filter values that allow one or more corresponding sink iterations to be executed. In the example, we obtain

$$\{\mathtt{A}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le j-1 < n \wedge 0 \le i < m\}.$$

Since this relation is a subset of the filter value relation on A, i.e.,

$$\{\, \mathtt{A}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le i < m \wedge 0 \le j \le n \,\},$$

we know that for each sink iteration in the domain of M' that is executed, the corresponding source iterations are also executed and therefore no parametrization is required.

6.3.2 Filter Values implied by Other Sources

The derivation of extra information from other potential sources is again illustrated based on an example. The general case is explained in Section A.3. In particular, let us consider the determination of the sources for the access to `state` in Line 4 of Listing 2. We first consider the write P in Line 7 as a potential source for $\ell = 0$. Parametrization is required and in particular (18) specializes to

$$(\lambda_0^P, \sigma^P) \to \left\{\, \mathtt{S_0}(i) \mid \sigma^P < 0 \vee \left(0 \le \lambda_0^P < i \wedge \sigma^P \ge 0\right) \,\right\}.$$

The dataflow analysis then continues looking for sources from the write Q in Line 5. Let us now consider the case where $\sigma^P < 0$. Since statement $\mathtt{S_0}$ is not affected by any filters, we need to turn to the other potential sources, in particular P, for any constraints that could help us determine whether parametrization is required.

We first construct a relation N mapping sink iterations to iterations of P that have definitely *not* been executed (according to (14) and given $\sigma^P < 0$), i.e.,

$$N = \{\, \mathtt{S_0}(i) \to \mathtt{C}(i') \mid 0 \le i' < i \,\}.$$

This relation can be constructed by subtracting iterations that may have executed from $D_{C,P}^{\text{mem}}$ (with the array elements projected out). In the example, there are no iterations that may have executed (since $\sigma^P < 0$) and so in this case $N = D_{C,P}^{\text{mem}}$. From (5), we know that the values of the corresponding filter elements (12) do *not* satisfy the filter access relation (9), i.e., for all $\mathtt{S_0}(i) \to \mathtt{C}(i') \in N$ we have

$$\mathcal{V}(F^{\mathtt{C}}(\mathtt{C}(i'))) \notin V^{\mathtt{C}}(\mathtt{C}(i')).$$

In other words,

$$\mathcal{V}(F^{\mathtt{C}}(\mathtt{C}(i'))) \in V_1(\mathtt{C}(i')),$$

with

$$V_1 = \{\, \mathtt{C}(i) \to (1) \mid i \ge 0 \,\}.$$

Note that `t_0` is a boolean variable, so if its value is not 0, it must be 1. Combining N and $F^{\mathtt{C}}$ (12) into a single relation, we obtain

$$N_1 = \{\, (\mathtt{S_0}(i) \to (\mathtt{S_0}(i') \to \mathtt{t_0}(i')) \to \mathtt{C}(i') \mid 0 \le i' < i \,\}.$$

Let $R_1 \rtimes R_2$ be the domain product of two relations, mapping nested pairs of domain elements from R_1 and R_2 to their shared images, i.e.,

$$R_1 \rtimes R_2 = \{\, (\mathbf{s} \to \mathbf{t}) \to \mathbf{u} \mid \mathbf{s} \to \mathbf{u} \in R_1 \wedge \mathbf{t} \to \mathbf{u} \in R_2 \,\}.$$

In general, we then have $N_1 = N \rtimes \left(F^{\mathtt{C}}\right)^{-1}$. The relation N_1 maps a pair of sink iteration and filter access to source iterations that have definitely not been executed and that perform the filter access, with value in V_1. The same filter element may be accessed by multiple source iterations, each of them imposing the V_1 constraints. We therefore compute the intersection of the V_1 images over all associated source iterations. In the example, only a single source iteration is associated to a given filter element and we obtain

$$V_2 = \{\, (\mathtt{S_0}(i) \to (\mathtt{S_0}(i') \to \mathtt{t_0}(i')) \to (1) \mid 0 \le i' < i \,\}.$$

Projecting out the filter access, we obtain the filter value relation

$$V_3 = \{\, \mathtt{S_0}(i) \to (1) \mid 1 \le i \,\},$$

which is valid for any element of $F^{\mathtt{C}}(N(\mathtt{S_0}(i)))$, with $i \ge 1$. This information can then be propagated to the potential source Q using the technique of Section 6.3.1, from which it can be concluded that Q is always executed (in the current case where $\sigma^P < 0$) and that therefore no parametrization is required. Note that the combined filter access relation $F^{\mathtt{C}} \circ N$ is no longer a function, but, as explained in Section A.2, the computations are also valid for multi-valued access relations. We just need to be careful about the domains of the access relations.

6.4 Additional Constraints

If we determine that we need to apply parametrization, then we may in some cases wish to impose additional constraints on the introduced parameters beyond those of Section 6.2. In particular, we may find that not all potential source iterations are executed (otherwise no parametrization would be required), but that *some* potential source iterations are definitely executed. If so, we add constraints that impose that there is a last execution ($\sigma^P \geq \ell$) and that this last execution is no earlier than any of the definitely executed iterations.

Additionally, we check for conflicts between the filters associated to the source of the newly added parameters and those of previously added parameters, introducing extra constraints that avoid the conflicts. As usual, we only show an example, while the details are explained in Section A.4. Continuing from the example in Section 6.3.2, let us consider the case where we are looking for an instance of the write Q in Line 5 of Listing 2 that is executed after the last execution of the write P in Line 7. Parametrization is required in this case, but the sink already contains parameters that refer to P, so we look for possible conflicts between the parameters of P and Q.

Based on dataflow analysis on the filter arrays, e.g., (12), we know that identical iterations of `D` and `C` access the same filter element, written in the same iteration. These filter elements therefore need to have the same value. Let us try to derive conflicts from iterations of both statements that are not executed. To ease the notation, we will only consider the case $\sigma^P, \sigma^Q \geq 0$ and omit these variables and constraints. The iterations in $D^{\mathrm{mem}}_{C,P}$ that are not executed are given by

$$\lambda_0^P \to \{\, \mathtt{S}_0(i) \to \mathtt{C}(i') \mid 0 \leq \lambda_0^P < i' < i \,\}$$

with filter value $\{\, \mathtt{C}(i') \to (1) \,\}$ and similarly for Q. Pairing up the non-executed iterations of `C` and `D` and restricting them to the pairs that should have the same value, we obtain

$$(\lambda_0^P, \lambda_0^Q) \to \{\, \mathtt{S}_0(i) \to (\mathtt{C}(i') \to \mathtt{D}(i')) \mid 0 \leq \lambda_0^P, \lambda_0^Q < i' < i \,\}.$$

Considering now the pairs of iterations from the two statements that map to values that allow the iterations to not be executed and such that these values are the same, we find that there are no such pairs. Subtracting this set of pairs of iterations that have the same value from the range of the relation above, we obtain a mapping from sink iterations to pairs of source iterations that should have the same values for their filters but in fact do not. In this example, an empty set is subtracted so that the relation remains the same. The domain of this relation, i.e.,

$$(\lambda_0^P, \lambda_0^Q) \to \{\, \mathtt{S}_0(i) \mid 0 \leq \lambda_0^P, \lambda_0^Q \leq i - 2 \,\}.$$

represents conflicting values of the parameters. In particular, it is impossible for the last executed iterations of P and Q to both be before the previous iteration of the loop. These impossible cases are removed from the sink parametrization S_2.

7. EXTENSIONS

In this section, we briefly discuss two extensions, dynamic loop bounds, which are supported by our dataflow analysis implementation, and dynamic index expressions, which are currently not supported.

```
1     for (int x1 = 0; x1 < n; ++x1) {
2   S1:   s = f();
3         for (int x2 = 0; P(x1, x2); ++x2) {
4   S2:       s = g(s);
5         }
6   R:    h(s);
7     }
```

Listing 3: C version of example E1 from [1, Section 3.2.2]

7.1 Dynamic Loop Conditions

In principle, dynamic loop conditions can be handled by introducing a filter that represents the last executed iteration of the loop. Consider, for example, the loop in Line 3 of Listing 3. The loop condition depends on some unknown function `P` applied to the loop iterator and is therefore not (locally) static affine. We could introduce a virtual scalar, say `m`, that represents the final iteration of the loop and that implicitly depends on `x1`. The filter value relation would then be of the form $(n) \to \{\, \mathtt{S2}(x_1, x_2) \to (m) \mid 0 \leq x_1 < n \wedge 0 \leq x_2 \leq m \,\}$. Unfortunately, the resulting dataflow dependence relations would not be practically useful for the construction of process networks since this last executed iteration is not known in advance.

Instead, we record the result of the dynamic loop condition in a virtual array and make the body of the loop depend on the value of the current *and all previous* iterations of the loop being 1. That is, the statement on Line 4 has filter value relation

$$V^{\mathtt{S2}} = (n) \to \{\, \mathtt{S2}(x_1, x_2) \to (1) \mid 0 \leq x_1 < n \wedge x_2 \geq 0 \,\}$$

with filter access relation

$$F^{\mathtt{S2}} = n \to \{\, \mathtt{S2}(x_1, x_2) \to \mathrm{t}_0(x_1, a) \mid 0 \leq a \leq x_2 \,\}.$$

Note that this filter access relation is not a function, but a multi-valued filter access relation and therefore requires the treatment of Appendix A. The statement evaluating the condition is made to depend on all previous iterations. Note that we currently only handle dynamic loop conditions for the purpose of dataflow analysis and not for the actual construction of process networks.

7.2 Dynamic Index Expressions

Our implementation currently does not support dynamic index expressions. We would have to allow the access relations to be approximate as well and the parameters would change to mean that the iteration is executed *and* that the element being accessed is the same as that accessed by the sink. This change in meaning would limit the conclusions we could draw from the new parameters.

8. EXPERIMENTS

Our approach has been implemented in the `da` and `pn` tools of the `isa` prototype tool suite (`git://repo.or.cz/isa.git`). The `da` tool performs dataflow analysis and supports dynamic loop conditions, while the `pn` tool additionally constructs a process network and currently does not support dynamic loop conditions. Table 1 shows the results of a preliminary experimental comparison of our `da` tool against that of [6], which is an implementation of the

input	da			fadatool			fadatool -s		
	time	p	d	time	p	l	time	p	l
Lst 1	0.01s	0	5	0.01s	6	6	0.01s	6	6
Lst 2	0.01s	4	9	0.01s	6	16	incorrect		
fuzz4	0.06s	3	9	0.02s	4	9	0.01s	0	9
for1	0.02s	2	3	0.01s	4	46	0.02s	2	3
for2	0.03s	2	3	0.09s	12	5k	0.04s	4	3
for3	0.04s	2	3	42s	24	1M	0.08s	6	3
for4	0.06s	2	3				0.16s	8	3
for5	0.08s	2	3				0.25s	10	3
for6	0.14s	2	3				0.42s	12	3
c_if1	0.02s	2	3	0.01s	2	4	0.01s	2	4
c_if2	0.02s	2	10	0.02s	4	52	0.02s	2	8
c_if3	0.03s	2	22	0.03	6	723	0.36s	3	16
c_if4	0.02s	2	10	0.17s	8	9k	1m	4	28
whil1	0.01s	0	4	0.00s	1	4	0.01s	0	4
whil2	0.03s	3	4	0.01s	5	6	incorrect		
if_var	0.03s	4	3	0.01s	2	8	0.01s	2	4
if_wh	0.04s	2	14	0.01s	5	58	0.02s	4	58
if2	0.02s	2	2	0.46s	12	29k	0.04s	4	2

Table 1: Experimental Results

approach that most closely resembles our own. In particular, we use version isa-0.11-319-gead5e27 of `da` and version fda6009 of `fadatool`. We use `fadatool` both with and without the `-s` option, since the results can be wildly different. It should be noted that both tested tools are prototypes. We should therefore be careful about drawing conclusions from these results, especially since `fadatool -s` sometimes produces incorrect results. Since the inputs in the table are relatively simple, correctness was determined through visual inspection.

For each tool and for each test case, we report the time taken by the analysis, the number of parameters introduced and the number of disjuncts in the dependence relations (for `da`) or the number of leaves in the quasts (for `fadatool`). Note that as explained in Section 9 below, the internal representation of dependence relations inside `fadatool` includes an additional parameter similar to our β. Since these internal parameters are not explicitly printed in the output of the tool, they are not included in the parameter count for `fadatool`. By contrast, the β parameters *are* included in the parameter count for `da`. The first two inputs are those of Listing 1 and Listing 2, modified to pass through the default `fadatool` parser. In particular, the parser does not support multiple writes in a single statement. It was therefore difficult to convert our more extensive test cases. The remaining cases (with some of the names abbreviated) come from the `fadalib` distribution. We omit those test cases that contain index expressions that are not static affine since we cannot handle them. Of note is that `fadatool` fails to recognize that no parameters need to be introduced on Listing 1 and that without the `-s` option, it quickly runs out of control on the `for` test cases. The `cascade_if` results may be somewhat misleading since `pet` recognizes that the filter variables used in the `if` conditions are the same and that some of the inner tests are implied by the outer test, greatly simplifying the input to `da`.

9. RELATED WORK

Despite its name, FADA is to the best of our knowledge the only alternative approach that allows for an exact (but run-time dependent) dataflow analysis in the presence of dynamic and/or non-affine conditions or index expressions. The main differences are that FADA introduces different parameters (with a different meaning), that they are only analyzed *after* all parameters have been introduced and that the analysis is performed using resolution on general first order logic formulas. A new vector of parameters $\boldsymbol{\alpha}$ is introduced for every maximization problem similar to (4), meaning that potentially many more parameters are introduced. The absence of a solution (similar to our $\beta = 0$) is represented as $\perp$ on paper and is reported to be represented by a scalar variable similar to our β inside the implementation of [6]. Note that it is sometimes suggested [8, 22] to use a value outside the iteration domain, but this may not always be easy to determine and is impossible for 0D iteration domains. Our dataflow analysis on filter arrays is a special case of the iterative approach of [1].

Other approaches to dataflow analysis [16, 21] produce approximate results in the presence of constructs that are not static affine. The approach of [16] in particular propagates values to discover static affine constraints in constructs that do not at first appear to be static affine. The authors of [9] propose an algorithm for computing reaching definitions for arrays that applies to both structured and unstructured programs. However, they only focus on how to collect constraints and do not explain how to solve them. Instead, they introduce uninterpreted functions and rely on `Omega` [19] for solving formulas containing such uninterpreted functions. Unfortunately, the support in `Omega` for uninterpreted functions is very limited and cannot handle the constraints they collect. The `iegenlib` library [23] has more extensive support for uninterpreted functions, but does not support a difference operation and can therefore not be used to perform value-based dependence analysis.

10. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach for exact array dataflow analysis in the presence of constructs that are not static affine. Dynamic behavior in the input program is represented using filters. An analysis of these filters determines if the dependences are also run-time dependent. If so, parameters are introduced to represent this run-time dependence, where we are careful to introduce as few parameters as possible. This is made possible by a judicious definition of these parameters. We plan on working on a closer integration with `pet` so that we can perform the dependence analysis incrementally, allowing us to locally treat some variables in the input as symbolic constants (as advocated by [15]) and more easily detect some cases (such as that of Listing 1) where no parameters need to be introduced. Note that the techniques developed in this paper would still be useful on more complicated inputs.

11. ACKNOWLEDGMENTS

This work was partially funded by a gift received by LIACS from Intel Corporation and by the European Commission through the FP7 project CARP id. 287767.

12. REFERENCES

[1] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* PhD thesis, PRiSM -

Laboratoire de recherche en informatique, Feb. 1998.

[2] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. J. Parallel Program.*, 28(3):213–243, 2000.

[3] D. Barthou, J.-F. Collard, and P. Feautrier. Applications of fuzzy array dataflow analysis. In *Euro-Par Conference*, volume 1123 of *Lect. Notes in Computer Science*, pages 424–427, Lyon, Aug. 1996. Springer-Verlag.

[4] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput.*, 40(2):210–226, 1997.

[5] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Euro-Par Conference*, volume 2400 of *Lect. Notes in Computer Science*, pages 309–313, Paderborn, Aug. 2002. Springer-Verlag.

[6] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati. FADAlib: an open source C++ library for fuzzy array dataflow analysis. In *Intl. Workshop on Practical Aspects of High-Level Parallel Programming*, volume 1, pages 2075—2084, Amsterdam, The Netherlands, May 2010.

[7] T. Bijlsma. *Automatic parallelization of nested loop programs for non-manifest real-time stream processing applications*. PhD thesis, University of Twente, 2011.

[8] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and practice of Parallel Programming*, July 1995.

[9] J.-F. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 286–302, London, UK, 2000. Springer-Verlag.

[10] P. Feautrier. Array expansion. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 429–441. ACM Press, 1988.

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

[12] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, Oct. 1992.

[14] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996.

[15] V. Maslov. Lazy array data-flow dependence analysis. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*, pages 311–325. ACM Press, 1994.

[16] V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 265–269, New York, NY, USA, 1995. ACM.

[17] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM.

[18] S. Pellegrini, T. Hoefler, and T. Fahringer. Exact dependence analysis for increased communication overlap. *Recent Advances in the Message Passing Interface*, pages 89–99, 2012.

[19] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. Technical Report Technical Report CS-TR-3191, Department of Computer Science, University of Maryland, College Park, Maryland, Dec. 1992. An earlier version of this paper appeared at the ACM SIGPLAN '92 Conference on PLDI.

[20] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag, 1994.

[21] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical report, College Park, MD, USA, 1994.

[22] T. Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden University, Leiden, The Netherlands, Sept. 2004.

[23] M. M. Strout, G. George, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Sept. 2012.

[24] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[25] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[26] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[27] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pages 599–613. Springer, June 2009.

[28] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, special issue on Embedded Digital Signal Processing Systems*, 2007, 2007.

APPENDIX

A. MULTI-VALUED FILTER ACCESS RELATIONS

In this appendix, we describe how to extend the representation and manipulation of filters to handle multi-valued filter access relations.

A.1 Filters

In Section 4, we assumed that all filter access relations access a single element in each iteration. Here, we describe how to extend the definitions to handle multi-valued filter access relations. The trickiest part is not so much handling filter access relations that access more than one data element, but filter access relations that may access zero data elements for some iterations. We therefore allow several filters on the same iteration domain, with disjoint domains, and impose that in each filter the domain of the filter value relation is a subset of the domains of all the filter access relations. In particular, each statement S has μ^S filters on its iteration domain, with $\mu^S \geq 0$. Each of the filters $\mathcal{F}_i$, with $1 \leq i \leq \mu^S$, is represented by a sequence of filter access relations $F_{i,j}^S$ with $1 \leq j \leq n_i^S$ and n_i^S the number of filter access relations, and a filter value relation V_i^S. As in Section 4, we have $F_{i,j}^S \subseteq I_S \to (I_S \to \mathcal{A})$ and $V_i^S \subseteq I_S \to \mathbb{Z}^{n_i^S}$. Furthermore, we impose

$$\operatorname{dom} F_{i,j}^S \supseteq \operatorname{dom} V_i^S,$$

for all $1 \leq i \leq \mu^S$ and $1 \leq j \leq n_i^S$, to ensure that all the filter access relations are total on the domains of the corresponding filter value relations, and

$$\operatorname{dom} V_{i_1}^L \cap \operatorname{dom} V_{i_2}^L = \emptyset$$

for all $1 \leq i_1, i_2 \leq \mu^S$ such that $i_1 \neq i_2$, to ensure that the domains of the filter value relations are disjoint.

The definition of $\mathit{executed}^S(\mathbf{k})$ (5) is then replaced by

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_{n_i^S}) \in \prod_{j=1}^{n_i^S} F_{i,j}^S(\mathbf{k}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n_i^S} \in V_i^S(\mathbf{k}) \qquad (19)$$

if $\mathbf{k} \in \operatorname{dom} V_i^S$ and is simply true on those parts of the iteration domain I_S that do not intersect any of the $\operatorname{dom} V_i^S$. That is, an element of the iteration domain I_S is only executed if the tuples of values of all the tuples of accessed filter array elements satisfy the active filter value relation.

A.2 Filter Values implied by the Sink

In this section, we describe the derivation illustrated in Section 6.3.1. We will assume that only a single sink filter and a single potential source filter apply to the domain and range of M. That is, we will assume that $\operatorname{dom} M_1 \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} M_1 \subseteq \operatorname{dom} V^P$, with M_1 the result of projecting out the array elements from M as in Section 6.3.1. In general, M needs to be split up according to the filters.

Let the sink filter consist of n filter access relations F_i and the potential source filter of m filter access relations G_j. Since we are going to compare the filters, we replace the filter access relations by their sources, as explained at the end of Section 4. In particular, $F_i \subseteq I_C \to (I \to A)$ and $G_i \subseteq I_P \to (I \to A)$. We construct a relation H between their possible values, initialized as

$$H = \mathbb{Z}^n \to \mathbb{Z}^m.$$

Let B_j represents the bounds on the values of the array elements accessed by G_j as specified by the user through a `pragma value_bounds` [26, Section 2], or $\mathbb{Z}$ if no such bounds have been specified. Let B be the Cartesian product of these bounds, i.e.,

$$B = \prod_{j=1}^{m} B_j. \qquad (20)$$

The relation can then be refined to

$$H = \mathbb{Z}^n \to B.$$

In the next step, we iterate over the potential source filter access relations and check if we have any information about them in the sink filter value. In particular, we check if any of the sink filter access relations accesses "the same" value(s). If so, we equate the corresponding dimensions in the mapping between filter values. To check if "the same" value(s) are accessed, we pull back the filter access relation over M_1. This results in a relation between sink domain iterations and filter sources such that there is at least one potential source corresponding to the sink domain iteration that accesses that filter source. If this relation is a subset of the filter access relation at the sink, then we know that everything we know about the values of the filter accesses at the sink also applies to the values of the corresponding filter accesses at the corresponding potential source iterations. Note that there may be more than one potential source iteration associated to a given sink iteration and that each of these potential source iteration may access a different element from the filter array. The above process ensures that source and sink values are only equated if *all* of these filter array elements are covered by the sink filter access relation.

In particular, for any $\mathbf{k} \in \operatorname{dom} V^C$ that is executed, we know from (19),

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_n) \in \prod_{j=1}^{n} F_j^C(\mathbf{k}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n} \in V^C(\mathbf{k}).$$

If we have $G_j^P \circ M_1 \subseteq F_i^C$, i.e., $\forall \mathbf{t} \in M_1(\mathbf{k}) : G_j^P(\mathbf{t}) \subseteq F_i^C(\mathbf{k})$, then we know

$$\forall \mathbf{t} \in M_1(\mathbf{k}) : \forall (\mathbf{f}_1, \ldots, \mathbf{f}_m) \in \prod_{j=1}^{n} G_j^P(\mathbf{t}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n} \in V_1(\mathbf{k}), \qquad (21)$$

with $V_1 = H \circ V^C$. The relation H takes care of projecting out those dimensions in V^C for which we were unable to find a corresponding filter access G_j^P, reordering those for which we did find a correspondence and introducing dimensions for those filter accesses G_j^P for which we were unable to find a corresponding filter access F_i^C. The relation $V_1 \subseteq I_C \to \mathbb{Z}^m$ represents what we know about the filter values at the potential sources associated to a given sink domain iteration, given that the sink domain iteration is executed. A further composition with (the inverse of) $M_1 \subseteq I_C \to I_P$, yields a subset of $I_P \to \mathbb{Z}^m$. This relation maps potential source iterations to filter values that allow for one or more corresponding sink iterations to be executed. In particular, if there is an element $\mathbf{k} \in M_1^{-1}(\mathbf{t})$ that is executed, then $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is an element of $V_1(\mathbf{k})$. Given that there is such a $\mathbf{k}$, the tuple $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is therefore an element of the union of $V_1(\mathbf{k}')$ over all $\mathbf{k}' \in M_1^{-1}(\mathbf{t})$. That is, $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is an element of $(V_1 \circ M_1^{-1})(\mathbf{t})$. In other words, for values of the filters outside the relation $V_1 \circ M_1^{-1}$, no corresponding (according to

M_1) sink domain iterations are executed. Note that we cannot take the intersection of $V_1(\mathbf{k}')$ over all $\mathbf{k}'$ because (21) only applies to those $\mathbf{k}$ that are executed and we only know that at least one of the $\mathbf{k}' \in M_1^{-1}(\mathbf{t})$ is executed, not that all of them are executed.

A.3 Filter Values implied by Other Sources

In this section, we describe the derivation illustrated in Section 6.3.2. In particular, we describe the "update" function in Line 8 of Algorithm 2. In particular, during the computation of the partial lexicographical maximum of M on U (4), the set U may already involve parameters that refer to the last iterations of (other) potential sources. We describe how we can exploit the constraints on these parameters to derive extra information about the filter values at the sink. The procedure of Section A.2 then needs to be applied to the updated sink filter to actually derive information about the filter values at the original potential source.

Specifically, we will derive information from the fact that some potential source iterations have *not* been executed. The conditions on the filter values at the potential source that allow the sink to be executed, but not the potential source are mapped back to the sink, first by taking the intersection over all potential source iterations associated to a given sink iteration and filter element and then by taking the union over all accessed filter elements. We take the intersection over the potential source iterations since we know that all those iterations are not executed and so all of the corresponding constraints apply. We take the union over the accessed filter elements, since we need to obtain constraints that are valid for all of those elements. As in Section A.2, we assume that only a single sink filter and a single potential source filter applies to the domain and range of M, i.e., that $\operatorname{dom} M_1 \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} M_1 \subseteq \operatorname{dom} V^P$. As before, M_1 is the result of projecting out the array elements from M. Let us similarly define a U_1 that is the result of projecting out the access array element from U. In the remainder of this section, we will take $D^{\text{mem}}_{C,Q}$ to have the array elements projected out, i.e., $D^{\text{mem}}_{C,Q} \subseteq I_C \to I_Q$.

Let us now look at the construction in more detail. We start with the construction of a mapping from sink iterations to iterations of some access Q that have *not* been executed (according to information in U). We first apply the parametrization of (15) to the iteration domain of Q and construct a relation $N_0 \subseteq U_1 \to I'_Q$ (with I'_Q the result of the parametrization) that is universal, except that the first ℓ dimensions in domain and range are equated. Note that some of the extra parameters in I'_Q also appear in U_1 (otherwise we would not consider this potential source). This means that the relation N_0 relates sink iterations to potential source iterations that include the last potential source iteration executed before the sink iteration. That is, $\{\, \mathbf{k} \to \lambda^Q_C(\mathbf{k}) \mid \mathbf{k} \in U_1 \wedge \sigma^Q_C(\mathbf{k}) \ge \ell \,\}$ is a subset of N_0. In particular, according to (14), the last element of $D^{\text{mem}}_{C,Q}(\mathbf{k})$, with $\mathbf{k} \in U_1$, that shares the first ℓ iterators and where the filter values satisfy the filtered iteration domain of Q is included in this relation. The relation may also contain additional elements since we may not have introduced a parameter for each dimension as explained in Section 5. Projecting out all parameters introduced in (15) (for any iteration domain), we obtain a relation between sink iterations and potential source iterations that include the last potential source iteration for any value of the other parameters. Further combining this relation with a relation mapping potential source iterations to earlier potential source iterations $\{\, \mathrm{S}_Q(\mathbf{i}) \to \mathrm{S}_Q(\mathbf{i}') \mid \mathbf{i} \succcurlyeq \mathbf{i}' \,\}$ results in a relation between sink iterations and potential source iterations that may have executed. In particular, the potential source iterations that are *not* related to a given sink iteration (and that share the first ℓ iterators) are *definitely not* executed. To obtain this relation between sink iterations and potential source iterations that are definitely not executed we subtract the relation computed above from the corresponding memory based dependences $D^{\text{mem}}_{C,Q}$ (with the first ℓ dimensions equated). Let us call the resulting relation $N \subseteq I_C \to I_Q$. Due to the construction, we have for each $\mathbf{k} \to \mathbf{j} \in N$ that $\neg \mathit{executed}^{S_Q}(\mathbf{j})$.

If N is empty, then we cannot use it to derive any information and the computation stops. Otherwise, the first step in our derivation is to apply the computation of Section A.2 to N. This assumes that $\operatorname{dom} N \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} N \subseteq \operatorname{dom} V^Q$, for some filter of Q. The first condition can be enforced by intersecting N with $\operatorname{dom} M \to I_Q$, since we are only interested in sink iterations that belong to $\operatorname{dom} M$ in any case. If we cannot find a filter on Q such that the second condition holds, then the computation stops. During the course of the computation, we will remove filter access relations G^Q_j that are not single-valued. We therefore check if the filter on Q has any single-valued filter access relations. If not, the computation stops.

Applying the computation in Section A.2 (with M replaced by N and the potential source P by the other potential source Q), we obtain a relation $V_0 \subseteq I_Q \to \mathbb{Z}^{m'}$ such that for each $\mathbf{k} \to \mathbf{j} \in N$, we have

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \in V_0(\mathbf{j}).$$

For the same $\mathbf{j}$, since $\neg \mathit{executed}^{S_Q}(\mathbf{j})$, we also know

$$\exists (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \notin V^Q(\mathbf{j}).$$

Combining these results, we have

$$\exists (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \in \left(V_0 \setminus V^Q\right)(\mathbf{j}).$$

Unfortunately, knowing that there is *some* sequence of filter elements $\mathbf{f}_i$ will not allow us to derive any further information. We will therefore assume that that all filter access relations G^Q_i are single-valued. Effectively, this means that we remove those filter access relations that are not single-valued and project out the corresponding dimensions from $V_0 \setminus V^Q$. Let V_1 be the result of this projection. That is, for each $\mathbf{k} \to \mathbf{j} \in N$,

$$\left(\mathcal{V}(G^Q_i(\mathbf{j}))\right)_{i=1}^{m''} \in V_1(\mathbf{j}).$$

Let G be the range product of the single-valued filter access relations G^Q_i.

At this point we have a relation $N \subseteq I_C \to I_Q$ mapping sink iterations to corresponding non-executed potential source iterations, a relation $G \subseteq I_Q \to (I \to A)^{m''}$, mapping potential source iterations to (single) filter elements, and a relation $V_1 \subseteq I_Q \to \mathbb{Z}^{m''}$, mapping potential source

iterations to corresponding filter values that do not allow the potential source iteration to be executed, but do allow the corresponding sink iteration to be executed. We first combine N and G into a single relation

$$N_1 = N \rtimes G^{-1}$$

The result is a subset of $(I_C \to (I \to A)^{m''}) \to I_Q$. We have, for each $(\mathbf{k} \to (\mathbf{f})_i) \to \mathbf{j} \in N_1$,

$$(\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_1(\mathbf{j}).$$

Note that because they represent (part of) a filter, we have $\operatorname{dom} G \supseteq \operatorname{dom} V^Q$. Combined with our assumption that $\operatorname{ran} N \subseteq \operatorname{dom} V^Q$, this ensures that we do not remove any elements from the range of N. We now connect the pairs of sink iterations and filter elements to the possible values of those filter elements at the corresponding potential source iterations by computing

$$V_2 : \operatorname{dom} N_1 \to \mathbb{Z}^{m''} : V_2(\mathbf{k} \to \mathbf{f}) = \bigcap_{\mathbf{j} \in N_1(\mathbf{k} \to \mathbf{f})} V_1(\mathbf{j}). \quad (22)$$

That is, we consider the potential source iterations $\mathbf{j}$ that have definitely not been executed before a certain sink iteration $\mathbf{k}$ and compute the intersection of the possible values of the filter elements $\mathbf{f}$ over all those definitely not executed source iterations. We have, for each $\mathbf{k} \to (f)_i \in \operatorname{dom} V_2$,

$$(\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_2(\mathbf{k} \to (\mathbf{f})_i).$$

Note that different potential source iterations associated to the same sink iteration may access different elements from the filter arrays. We therefore need to be careful to only combine constraints on values associated to the same elements. If N_1 is single-valued, the intersection in (22) is computed over a single element and so we can simply compute V_2 as

$$V_2 = V_1 \circ N_1.$$

Otherwise, it is computed as

$$V_2 = N_1 \sqsubseteq V_1^{-1},$$

with $\sqsubseteq$ the non-empty subset operation on two relations, which constructs a relation between the domain elements of the two relations such that the image of the first domain element is a subset of the image of the second domain element. That is, $R_1 \sqsubseteq R_2$ is equal to

$$\{\, \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \in \operatorname{dom} R_1 \wedge \mathbf{t} \in \operatorname{dom} R_2 \wedge R_1(\mathbf{s}) \subseteq R_2(\mathbf{t}) \,\}$$

The constraint $R_1(\mathbf{s}) \subseteq R_2(\mathbf{t})$ can be expressed as

$$\forall \mathbf{u} : \mathbf{s} \to \mathbf{u} \in R_1 \Rightarrow \mathbf{t} \to \mathbf{u} \in R_2$$

or

$$\neg \exists \mathbf{u} : \mathbf{s} \to \mathbf{u} \in R_1 \wedge \mathbf{t} \to \mathbf{u} \notin R_2$$

where $\mathbf{u}$ may be restricted to $\operatorname{ran} R_1$. The operation can therefore be computed as

$$R_1 \sqsubseteq R_2 = R_2^{-1} \circ R_1 \setminus \left(((\operatorname{dom} R_2 \to \operatorname{ran} R_1) \setminus R_2)^{-1} \circ R_1\right).$$

Finally, we project out filter elements and compute

$$V_3 : I_C \to \mathbb{Z}^{m''} : V_3(\mathbf{k}) = \bigcup_{\mathbf{f} \in (\operatorname{dom} V_2)(\mathbf{k})} V_2(\mathbf{k} \to \mathbf{f}).$$

The resulting relation contains all values of all filter elements read by any non-executed iteration associated to a certain sink iteration. That is, for every $\mathbf{k} \in \operatorname{dom} V_3 = \operatorname{dom} N$,

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_{m''}) \in \prod_{i=1}^{m''} G_i^Q(N(\mathbf{k}))) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_3(\mathbf{k}).$$

V_3 can be computed as

$$V_3 = V_2 \circ (\underrightarrow{\operatorname{dom}}(\mathcal{W}^{-1}(\operatorname{dom} V_2)))^{-1},$$

with $\mathcal{W}^{-1} S$ extracting the nested relation from the set S, i.e.,

$$\mathcal{W}^{-1} S = \{\, \mathbf{s} \to \mathbf{t} \mid (\mathbf{s} \to \mathbf{t}) \in S \,\}.$$

The relation V_3 may only apply to a subset of the domain of M_1. Since we do not have any information about elements outside $\operatorname{dom} V_3 = \operatorname{dom} N$, we extend V_3 to the entire domain of M_1 as

$$V_4 = V_3 \cup \left((\operatorname{dom} M_1 \setminus \operatorname{dom} N) \to \mathbb{Z}^{m''}\right).$$

We now want want to intersect V^C with V_4, but as in Section A.2 we first need to align the filter access relations, by constructing a relation H mapping the filter values of V_4 to those of V^C. In this case, however, we also allow extra filter access relations to be added, both to the original set of filter access relations of the sink and to the filter access relations associated to V_4. We do this to be able to collect as much information as possible at the sink. In particular, some of the sources may involve the same filter access relations even if these filter access relations do not originally appear among those of the sink and we still want to combine the information from different sources at the sink.

More specifically, we look for filter access relations F_i^C that are identical to some $G_j^Q \circ N$. If we cannot find such an F_i^C, we add $G_j^Q \circ N$ to the sink filter access relations (adjusting V^C). In both cases, we express the correspondence in H. Additionally, we look for filter access relations F_i^C that form a (strict) subset of some $G_j^Q \circ N$. If so, we add F_i^C to the filter access relations associated to V_4, adjusting V_4 by duplicating the constraints on the corresponding dimension to the new dimension that corresponds to the extra filter access relation. Again, we express the correspondence in H. At the end, we apply H to V_4 and intersect V^C with the result.

A.4 Avoid Inconsistencies

As explained in Section 6.4, some values of the parameters expressing the last iteration of potential source P, introduced in Section 6.2 and constrained by (18), may not be consistent with the fact that the sink C is executed or with the values of parameters expressing the last iteration of other potential sources introduced before. In this section, we describe how we can remove some of these inconsistencies. Note that leaving in these inconsistencies does not lead to incorrect results, but only to less accurate results. We therefore do not need to remove every possible inconsistency, but instead try to remove those that we can easily discover.

Let us start with inconsistencies that arise from the fact that two potential sources, the current one (P) and one that was considered before (Q), are executed. In particular, assume that iteration $\mathbf{i}$ of P is executed and iteration $\mathbf{j}$ of Q

as well. According to (19), we then have

$$\forall(\mathbf{f}_1,\ldots,\mathbf{f}_m) \in \prod_{j=1}^{m} F_j^P(\mathbf{i}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^m \in V^P(\mathbf{i}) \tag{23}$$

for some filter of P and

$$\forall(\mathbf{f}_1,\ldots,\mathbf{f}_{m'}) \in \prod_{j=1}^{m'} F_1^Q(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{m'} \in V^Q(\mathbf{j}) \tag{24}$$

for some filter of Q. If we can find a pair of subsequences of filter access relations that access some sequence of filter array elements in common, then the corresponding dimensions of $V^P(\mathbf{i})$ and $V^Q(\mathbf{j})$ need to have some value in common. Let $K \subseteq I_P \times I_Q$ contain those pairs of iteration that access the same filter array elements and let $H \subseteq \mathbb{Z}^m \times \mathbb{Z}^{m'}$ express the correspondence of values. That is, H is expressed as one or more equalities equating pairs of dimensions, one from $V^P(\mathbf{i})$ and one from $V^Q(\mathbf{j})$. The relation

$$T = \mathcal{W}^{-1}\left(\left(V^P \times V^Q\right)^{-1}(\mathcal{W}H)\right), \tag{25}$$

where

$$\mathcal{W}R = \{\, (\mathbf{s} \to \mathbf{t}) \mid \mathbf{s} \to \mathbf{t} \in R \,\},$$

then contains those pairs of iterations that allow for a common value. Removing those from K, we obtain the relation

$$K' = K \setminus T$$

of pairs of iterations that *should* allow for a common value, but in fact do not. To map these inconsistencies to the parameters we construct a relation D_1 from $D^{\text{mem}}_{C,P}$ (with array elements projected out). In particular, we intersect the domain of $D^{\text{mem}}_{C,P}$ with the parametrization of (15) and equate the first ℓ dimensions of domain and range. We similarly construct a relation D_2 from $D^{\text{mem}}_{C,Q}$. The inconsistent values of the parameters are then obtained as

$$(D_1 \rtimes D_2)(\mathcal{W}K') \tag{26}$$

and the result is removed from the sink parametrization.

The above procedure may be repeated for any appropriate pair of K and H. In our implementation, we currently only construct a single such pair in a greedy way. In particular, we start out with a universal K and H and consider pairs of filter access relations $F_i^P \subseteq I_P \to (I \to A)$ and $F_j^Q \subseteq I_Q \to (I \to A)$ that access the same array. For each such pair, we compute

$$K_{i,j} = \left(F_j^Q\right)^{-1} \circ F_i^P \subseteq I_P \to I_Q \tag{27}$$

and check if $K_{i,j}$ has a non-empty intersection with the current value of K. If so, we replace K with this intersection and adjust H accordingly. Otherwise, we skip this pair of filter access relations.

We have only considered inconsistencies based on pairs of potential sources that *are* executed. It is also possible to derive inconsistencies based on one or both of the potential sources *not* being executed. Let us first consider the case where P is not executed and Q is executed. In this case, (23) is replaced by

$$\exists(\mathbf{f}_1,\ldots,\mathbf{f}_m) \in \prod_{j=1}^{m} F_j^P(\mathbf{i}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^m \notin V^P(\mathbf{i}).$$

Whereas in the case of (23) and (24) a conflict occurs as soon as there is *any* sequence of accessed elements for which no matching value can be found, in this case we only know something about *some* sequence of accessed elements at P and we can therefore only arrive at a conflict if *all* of the accessed elements are also accessed by Q. In particular, (27) needs to be replaced by

$$K_{i,j} = F_i^{L_P} \sqsubseteq F_j^{L_Q} \subseteq I_{L_P} \to I_{L_Q}.$$

Additionally, V^P in (25) should be replaced by

$$(I_{L_P} \to B) \setminus V^{L_P}$$

with B the bounds on the possible values on the array elements, as computed in (20), while D_1 in (26) should refer to iterations that are *not* executed. That is, rather than intersecting $D^{\text{mem}}_{C,P}$ with the parametrization of (15), we first map this parametrization to lexicographically later elements and take the union with the set

$$(\sigma^P) \to \{\, \mathrm{S}_{S_P}(\mathbf{j}) \mid \sigma^P < \ell \,\}.$$

This second part accounts for the fact that when $\sigma^P < \ell$, none of the iterations that share the first ℓ iterators have executed.

The case where P is executed and Q is not executed is handled similarly. For the case where both P and Q are not executed, we can only draw any conclusion for those iterations that access a single filter element. That is, $K_{i,j}$ of (27) is replaced by

$$\{\, \mathbf{i} \to \mathbf{j} \in (\operatorname{dom} F_i^P) \to (\operatorname{dom} F_j^Q) \mid \forall \mathbf{f}_1, \mathbf{f}_2 \in F_i^P(\mathbf{i}), \mathbf{f}_3, \mathbf{f}_4 \in F_j^Q(\mathbf{j}) : \mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3 = \mathbf{f}_4 \,\}.$$

This relation can be computed by removing

$$\operatorname{dom}\left(\left(F_i^P \ltimes F_i^P\right) \setminus (I_P \to 1_{I \to A})\right)$$

from the domain of $\left(F_j^Q\right)^{-1} \circ F_i^P$ and

$$\operatorname{dom}\left(\left(F_j^Q \ltimes F_j^Q\right) \setminus (I_Q \to 1_{I \to A})\right)$$

from its range, with

$$R_1 \ltimes R_2 = \{\, \mathbf{s} \to (\mathbf{t} \to \mathbf{u}) \mid \mathbf{s} \to \mathbf{t} \in R_1 \wedge \mathbf{s} \to \mathbf{u} \in R_2 \,\}.$$

That is, we consider pairs of image elements associated to the same domain element of F_i^P or F_j^Q and remove pairs of identical image elements. That only leaves pairs of different image elements and the domain of the relation represents those domain elements that have multiple image elements associated to them.

Inconsistencies between the potential source P and the sink C are removed in a similar way, except that C is always executed so that we only need to consider two cases, one where P is executed and one where P is not executed. Furthermore, the relation D_2 in (26) is replaced by the identity relation on the iteration domain of the sink, i.e., 1_{I_C}.

Multifor for Multicore

Imèn Fassi
Dpt of Computer Science
Faculty of Sciences of Tunis
University El Manar
1060 Tunis, Tunisia
fassi.imen@gmail.com

Philippe Clauss
Team CAMUS, INRIA
University of Strasbourg
boulevard S. Brant
67400 Illkirch, France
philippe.clauss@inria.fr

Matthieu Kuhn
Team ICPS, LSIIT lab.
University of Strasbourg
boulevard S. Brant
67400 Illkirch, France
kuhn@unistra.fr

Yosr Slama
Dpt of Computer Science
Faculty of Sciences of Tunis
University El Manar
1060 Tunis, Tunisia
yosr.slama@gmail.com

ABSTRACT

We propose a new programming control structure called "multifor", allowing to take advantage of parallelization models that were not naturally attainable with the polytope model before. In a multifor-loop, several loops whose bodies are run simultaneously can be defined. Respective iteration domains are mapped onto each other according to a run frequency – the grain – and a relative position – the offset –. Execution models like dataflow, stencil computations or MapReduce can be represented onto one referential iteration domain, while still exhibiting traditional polyhedral code analysis and transformation opportunities. Moreover, this construct provides ways to naturally exploit hybrid parallelization models, thus significantly improving parallelization opportunities in the multicore era. Traditional polyhedral software tools are used to generate the corresponding code. Additionally, a promising perspective related to non-linear mapping of iteration spaces is also presented, yielding to run a loop nest inside any other one by solving the problem of inverting "ranking Ehrhart polynomials".

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Performance

Keywords

programming control structure, parallel programming, polytope model

1. INTRODUCTION

We have definitely entered a new era in programming. Parallelism is everywhere, from the many-core processor architectures that are from now on fitting mainstream computers, to the software applications that are now mixing intensive computations, specialized routines, network communication and multi-threading. Many researches focus in proposing new languages supposed to facilitate programming inside this complex environment [8, 10, 11, 13], or in proposing hardware or software support like transactional memory systems [6, 14, 15], supposed to prevent incorrect computations while still providing good performance. However, all these proposals face intractable issues. Most proposed languages imply to change drastically programmers habits and have weak chances to be adopted by the software industry [4]. Moreover, even if they offer interesting constructions to express parallel tasks, they are not solving the fundamental problem of correct and efficient parallelism extraction, which induces dependency analysis, data locality optimization, task grain adjustment, etc. Performance is also strongly dependent of their implementation, i.e. of their compilers or runtime systems. On the other hand, hardware or software support does not either result in hiding parallelization complexity to the user. And overall, these mechanisms are of high complexity by themselves, which mostly often make them unrealistic for a real usage [3].

Nevertheless, parallel programming has already a long history, where gradual extensions have been proposed. Some of them were pretty successful and are still current. For example, directive-based languages, as OpenMP [2], are extensions to mainstream languages. The use of their instructions is not mandatory when inserted in a source code, and they can be discovered and adopted progressively by any developer. They are not breaking the programming habits, while offering efficient parallelization opportunities. Although they are not solving either the fundamental complexity of parallelization, they nicely open the door of high performance computing to anyone.

At the same time, a lot of relevant transformation techniques have been discovered in order to exhibit parallelism or to optimize code, particularly on loops, as software pipelining, loop unrolling, tiling, skewing, etc [1, 16, 12]. These are applied either by experienced programmers, or automatically by compilers or optimization tools. In the parallelism era, compilers and runtime systems have to accelerate their progresses in automatic parallelization, but at the same time, programmers have to be brought to become, at least, who we called "experienced programmers" ten or twenty years ago.

We argue that a good way to achieve such an emancipation to parallel programming is to gradually extend mainstream languages with new programming control structures

that are derived from already existing ones. Our idea is that many well-known optimizing and parallelizing code transformations should now be applied naturally by developers, but only in their minds, while using a control structure translating their enriched algorithmic reasonings. The existence of such control structures will condition them to enlarge their way of reasoning while programming. In the same way that it is currently natural to express the repetition of code blocks by using loops, or to abstract parametrized code by using functions, it should now be natural to bring closer instructions that are using the same operands, or to arrange code snippets in vertical and horizontal directions to express simultaneity, sequencing and overlapping.

Following this idea, we propose a new control structure called *Multifor*, which can be seen as an extension of for-loops to multiple for-loops running at the same time, whose respective instructions are run sequentially inside one loop body as a traditional for-loop, but run in any interleaved order, or in parallel, between the bodies. Additionally to traditional parameters as indices, bounds and steps, we propose to introduce a grain and an offset, allowing to mix loops of different execution frequencies and of different starting positions. Such programming construction translates naturally to code transformations as loop fusion, loop pipelining or loop partitioning. Moreover, it facilitates many code optimizations as data locality improvement or parallelization. It can be seen as an extension of for-loops from "vertical" to "horizontal" programming.

The second motivation of this proposal is related to the parallel programming models that are covered by the polytope model. Traditional application of this model does not allow to naturally express parallel programming models as task parallelism, dataflow or MapReduce. We show that the multifor construct allows to schedule loop nest processes by mapping together their respective iteration domains. A multifor code can be represented geometrically by a particular union of polyhedra, each being previously dilated, compressed or translated, either entirely or partially, according to transformation factors defined by constants or affine functions.

This new programming structure implies interesting implementation challenges for a compiler, from its front-end to its backend. We show that, as it is already the case with for-loops, the polytope model is quite well adapted to analyze, optimize and translate multifor constructs into efficient code.

Finally, as a promising perspective, we also propose a non-linear mapping of the iteration domains guided by the ranks of the iterations. This approach opens the possibility of mapping any iteration domain onto any other domain without being constrained by their shapes. It leads to solve the general problem of executing any loop nest by any other loop nest of the same trip count. Mathematically speaking, the general solution to this problem is based on inverting "ranking Ehrhart polynomials" [5, 9].

The paper is organized as follows. In the next section, syntax and semantics of multifor loops are described, illustrated with a few examples of multifor headers and graphical representations. We also highlight the code parallelization and transformation schemes that are possible with multifor-loops. In Section 3, we discuss the main issues related to the implementation of multifor constructs and their corresponding code generation. Several real and representative code examples are presented in Section 4, highlighting the interesting contributions of this new control structure. The promising perspective of non-linear iteration space mapping is the topic of Section 5, where a solution for inverting ranking Ehrhart polynomials is proposed. Finally, conclusions and further perspectives are given in Section 6.

2. SYNTAX AND SEMANTICS

In this paper, we describe the initial syntax and semantics for the multifor construct. However, they can be extended in many ways in the future. We first present the case of one unique multifor construct, as the case of nested multifor-loops present some specificities which are presented afterwards.

2.1 Non-nested multifor-loops

The multifor syntax is defined by:

$$
\begin{array}{ll}
\textbf{multifor} \; (& index_1 = expr, [index_2 = expr, ...] \; ; \\
& index_1 < expr, [index_2 < expr, ...] \; ; \\
& index_1 += cst, [index_2 += cst, ...] \; ; \\
& grain_1, [grain_2, ...] \; ; \\
& offset_1, [offset_2, ...] \;) \; \{ \\
& prefix : \{statements\} \\
\} &
\end{array}
$$

where [...] denotes optional arguments, $index_i$ denotes the indices of the loops composing the multifor, $expr$ denotes affine arithmetic expressions on enclosing loop indices, or constants, cst denotes an integer constant, $grain$ and $offset$ are positive integers, $grain \geq 1, offset \geq 0$, and $prefix$ is a positive integer associating each statement to a given for-loop composing the multifor-loop, according to the order in which they are defined (0 for the first loop, 1 for the second loop, etc.). Without loss of generality, we consider in the following that the index steps, cst, always equal one, since the general case can be easily deduced.

Each for-loop composing the multifor behaves as a traditional for-loop, but all are mapped on a same global "virtual referential" domain, which can also be seen as a template. The way iterations of the for-loops are mapped is defined by their respective offset and grain. The grain defines the frequency in which the associated loop has to run, relatively to the referential. For instance, if the grain equals 2, then one iteration of the associated loop will run over 2 iterations of the referential. The offset defines the gap between the first iteration of the referential and the first iteration of the associated loop. For instance, if the offset equals 3, then the first iteration of the associated loop will run at the fourth iteration of the referential loop.

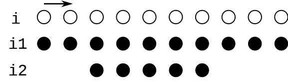
The size and shape of the referential is deduced from the for-loops composing the multifor-loop. Geometrically, it is defined as the disjoint union of the for-loop domains, where each domain has been previously shifted according to its offset and dilated according to its grain. The disjoint union is the union of adjacent convex domains, each being scanned by a referential for-loop. The relative positions of the iterations of the for-loops composing the multifor-loop inside the referential depends of the overlapping of their respective domains. It means that on domains where only one for-loop iterations are run, the grain becomes a compression factor. In general, the greatest common divisor of the grains of all the for-loops overlapping on a same referential domain

is used as the factor for compressing the points of this referential domain, according to the lexicographic order. On domains where several for-loops iterations are run, these are run in interleaved fashion, or simultaneously.

Let us illustrate this definition with a few examples. Consider the following multifor-loop header:

multifor $(i_1 = 0, i_2 = 10; i_1 < 10, i_2 < 15; i_1{+}{+}, i_2{+}{+}; 1, 1; 0, 2)$

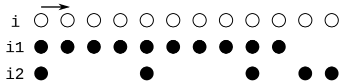
In this example, the offset of index i_1 is zero, and the one of index i_2 is 2. Thus, the first iteration of the i_1-loop will run immediatly, while the first iteration of the i_2-loop will run at the 3rd iteration of the multifor, but with $i_2 = 0$. This behavior is illustrated by the figure below:



Notice that the index values have no effect on the relative positions of the for-loops bodies, which are uniquely determined by the grain and the offset. Another example is:

multifor $(i_1 = 0, i_2 = 10; i_1 < 10, i_2 < 15; i_1{+}{+}, i_2{+}{+}; 1, 4; 0, 0)$

Now, the i_1-grain is 1 and the i_2-grain is 4. In such a case, for one iteration of the i_2-loop, four iterations of the i_1-loop will be run on the domain on which they overlap. The second domain is compressed by a factor of 4, since only the i_2-loop is run, as it is illustrated below:



2.2 Nested multifor-loops

Nested multifor-loops present some particularities and specific semantics has to be described. Without loss of generality, let us consider two nested multifor-loops composed of two for-loop nests:

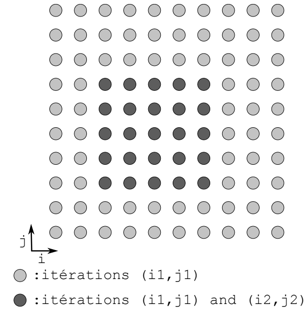
```
multifor ( index1 = expr, index2 = expr;
           index1 < expr, index2 < expr;
           index1+ = cst, index2+ = cst;
           grain1, grain2;
           offset1, offset2 ) {
           prefix : {statements}
   multifor ( index3 = expr, index4 = expr;
              index3 < expr, index4 < expr;
              index3+ = cst, index4+ = cst;
              grain3, grain4;
              offset3, offset4 ) {
              prefix : {statements}
   }
           prefix : {statements}
}
```

Such a nest behaves as two for-loop nests, $(index_1, index_3)$ and $(index_2, index_4)$ respectively, running simultaneously in the same way as it is for one unique multifor-loop. The grain of the inner multifor-loop introduces a delay for the associated for-loop, since the same reasoning as with the non-nested case is applied at each loop depth. The lower and upper bounds are affine functions of the enclosing loop indices of the same for-loop[1]. Let us consider some examples of nested multifor headers.

[1]Notice that this restriction could be evicted for some amazing extensions.

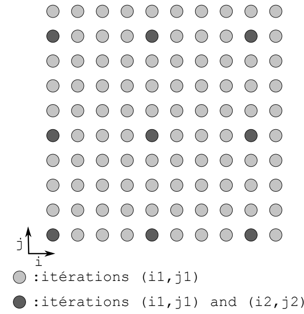
multifor $(i_1 = 0, i_2 = 0; i_1 < 10, i_2 < 5; i_1 + +, i_2 + +; 1, 1; 0, 2)$
multifor $(j_1 = 0, j_2 = 0; j_1 < 10, j_2 < 5; j_1 + +, j_2 + +; 1, 1; 0, 2)$

The second for-loop nest has a 2-offset at each loop depth. Hence it is delayed in each dimension of the referential domain:



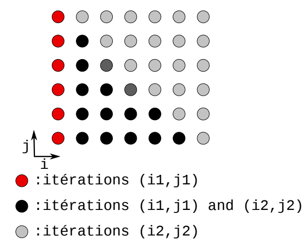
multifor $(i_1 = 0, i_2 = 0; i_1 < 10, i_2 < 3; i_1 + +, i_2 + +; 1, 4; 0, 0)$
multifor $(j_1 = 0, j_2 = 0; j_1 < 10, j_2 < 3; j_1 + +, j_2 + +; 1, 4; 0, 0)$

The second for-loop nest has a 4-grain at each loop depth. Hence its iterations are spaced by 4 in each dimension of the referential domain:



multifor $(i_1 = 0, i_2 = 0; i_1 < 6, i_2 < 6; i_1 + +, i_2 + +; 1, 1; 0, 1)$
multifor $(j_1 = 0, j_2 = 0; j_1 < 6 - i_1, j_2 < 6; j_1 + +, j_2 + +; 1, 1; 0, 0)$

In this example, the upper bound of the inner loop of the first loop nest is an affine function.



2.3 Multifor-loop parallelization and code transformations

The multifor construct exhibits a straightforward parallelization strategy which is to run, at each iteration, the loop bodies of the defined for-loops in parallel. This model of parallelization provides new opportunities to the polytope model, since it enables the expression of parallel programming models that were unattainable before, as dataflow computing or fixed-depth MapReduce, as it will be shown on code examples in Section 4.

Nevertheless, the multifor construct still allows OpenMP-like loop parallelization for each for-loop of the multifor, thus providing hybrid parallelization strategies.

Moreover, each for-loop, or for-loop nest, of a multifor, can be transformed using any well-known polyhedral transformation. However, in the context of a multifor, these transformations may be guided by the interactions between the for-loops, in order to achieve better performance or better data locality for instance. Another opportunity is the transformation of imperfect loop nests into multifor-loop nests of perfectly nested loops.

3. IMPLEMENTATION ISSUES

3.1 Reference domain

Consider one multifor-loop level. The referential for-loops cadencing the multifor execution have the constraint of scanning a sufficient number of iterations. Let us denote by f the number of for-loops. By computing the disjoint union of all for-loops iteration domains, we obtain a set of adjacent domains D_i on which some of the f loops overlap. Let us denote by $lb_i, ub_i, grain_i$ and $offset_i, i = 1..f$ the parameters characterizing each for-loop in the multifor header. Let us set $nlb_i = offset_i$ and $nub_i = (ub_i - lb_i + 1) \times grain_i + offset_i$, which define the lower and upper bounds of each loop in the referential domain, since the computation of nlb_i consists in translating the domain and the computation of nub_i in dilating the domain by a factor which equals the grain. The disjoint union of D_i's is computed using these latter bounds. The initial index value of the referential domain is $MIN_{i=1..f}(nlb_i)$. Hence, the total number of iterations in the referential domain, before compression, is:

$$MAX_{i=1..f}(nub_i) - MIN_{i=1..f}(nlb_i) + 1$$

In order to generate the referential for-loops for each domain D_i, the last step consists in compressing each D_i by a factor defined by $lcm(grain_j)$, for all loops j overlapping on D_i.

More generally for any multifor-loop nest, the computation of the referential domain is performed in three steps. First, each iteration domain associated to one for-loop nest composing a multifor-loop nest is translated from the origin according to its offsets, and dilated according to its grains in every dimension. Notice that values actually taken by the indices of the for-loop nests are not defining their positions in the referential domain. Second, a disjoint union is computed and resulting in a union of adjacent convex domains D_i. Third, each D_i may be compressed according to the greatest common divisor of the grains of the associated for-loop nests, and according to the lexicographic order.

3.2 Code generation

When considering sequential code, there are two dual ways to generate the code corresponding to a multifor-loop nest. A first way is to generate loop nests scanning the referential domains through a minimal set of convex domains, and to insert guards in their bodies in order to execute the convenient instructions at each iteration. The number of these guards can be optimized by computing their common sub-domains. The second way is to scan each D_i using a dedicated loop nest with a constant loop body, without guards.

Both solutions can be generated automatically using polytope model tools like PolyLib or CLooG, and by inserting phases to compute the translated and dilated domains, or to compress parts of the resulting referential domains.

If for-loops of given depth of a multifor-loop nest have to be run in parallel, each for-loop has to be run in a separated thread and all threads have to be synchronized at the multifor-loop completion. Notice that this could be enriched by providing OpenMP-like options as NOWAIT.

Original indices of the multifor (i_1, i_2, j_1, ...) have to be retrieved at the beginning of each loop body, by being computed from the referential loop indices. These computations consists in subtracting offsets, or in adding modulos of the referential loop indices relatively to grains, or in multiplying by grains in case of compressed domains.

```
for (i = 0; i < K; i++)
 for (j = 0; j < N; j++)
  a[i][j] = ReadImage();
for (i = 1; i < K - 1; i++)
 for (j = 1; j < N - 1; j++){
  Sbl[i][j] = Sobel(a[i-1][j-1], a[i][j-1], a[i+1][j-1],
                    a[i-1][j], a[i][j], a[i+1][j],
                    a[i-1][j+1], a[i][j+1], a[i+1][j+1]);
  WriteImage(Sbl[i][j]);
}
```

Figure 1: Sobel edge detection code

```
multifor (i1 = 0, i2 = 1; i1 < K, i2 < K - 1;
          i1++, i2++; 1, 1; 0, 3)
 multifor (j1 = 0, j2 = 1; j1 < N, j2 < N - 1;
           j1++, j2++; 1, 1; 0, 3) {
  0 : a[i1][j1] = ReadImage();
  1 : {Sbl[i2][j2] = Sobel(a[i2-1][j2-1], a[i2][j2-1],
                           a[i2+1][j2-1], a[i2-1][j2],
                           a[i2][j2], a[i2+1][j2],
                           a[i2-1][j2+1], a[i2][j2+1],
                           a[i2+1][j2+1]);
       WriteImage(Sbl[i2][j2]); }
 }
```

Figure 2: Sobel edge detection multifor code

4. EXAMPLES

Sobel edge detection: We first consider the code for performing Sobel edge detection of an image shown in Figure 1. The first loop nest of this program reads the input image, while the second loop nest performs the actual edge detection and writes out the output image.

Note that nine neighboring elements have to be read before its resulting pixel can be computed and written. Hence both loop nests can be naturally overlapped by writing the code using the multifor construct exhibiting a data-flow model of computation, shown in Figure 2. The associated referential domain is shown in Figure 3.

Red-Black Gauss-Seidel: The second example is the Red-Black Gauss-Seidel algorithm composed of two phases. The first phase consists in updating the red elements of a grid, which are one point over two in the i and j directions of the grid, starting from the first bottom left corner, using their North-South-East-West (NSEW) neighbors, which are black elements. The second phase consists obviously in updating the black elements from the red ones. For a 2D $N \times N$ problem, the usual code, is of the form shown in Figure 4 (the border elements initialization has been omitted).

On the iteration domain and at each phase, a different

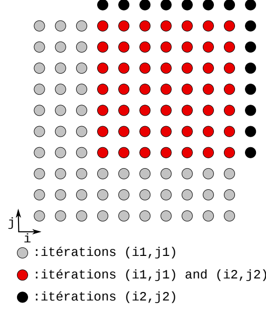


Figure 3: Sobel edge detection referential domain

```
// Red phase
for (i = 1; i < N − 1; i++)
 for (j = 1; j < N − 1; j++)
  if ((i + j) % 2 == 0)
   u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
// Black phase
for (i = 1; i < N − 1; i++)
 for (j = 1; j < N − 1; j++)
  if ((i + j) % 2 == 1)
   u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
```

Figure 4: Red-Black Gauss-Seidel code

lattice of iterations is active. Moreover, the NSEW dependencies prevent any linear parallel schedule. This code can be translated into a multifor-loop nest where the red and black phases each yield two for-loop nests with convenient grains and offsets as shown in Figure 5.

The referential initial domain of the multifor code is represented in Figure 6 on the left, also showing the transformed dependency vectors which are now allowing a linear schedule. On the right in Figure 6, the final iteration domains, after compression, are represented.

This example shows that the multifor construct allows to exploit different kind of parallelism – the so-called wavefront parallelism in this case – since the traditional parallelization consists in parallelizing each of the phases, and to execute each phase one after the other. The multifor strategy can be preferable to improve data locality and thus improve the

```
multifor (i0 = 1, i1 = 2, i2 = 1, i3 = 2; i0 < N − 1, i1 < N − 1,
   i2 < N − 1, i3 < N − 1; i0+ = 2, i1+ = 2, i2+ = 2,
   i3+ = 2; 2, 2, 2, 2; 0, 1, 0, 1)
 multifor (j0 = 1, j1 = 2, j2 = 2, j3 = 1; j0 < N − 1, j1 < N − 1,
   j2 < N − 1, j3 < N − 1; j0+ = 2, j1+ = 2, j2+ = 2,
   j3+ = 2; 2, 2, 2, 2; 0, 1, 1, 0) {
 0 : u[i0][j0] =
     f(u[i0][j0 + 1], u[i0][j0 − 1], u[i0 − 1][j0], u[i0 + 1][j0]);
 1 : u[i1][j1] =
     f(u[i1][j1 + 1], u[i1][j1 − 1], u[i1 − 1][j1], u[i1 + 1][j1]);
 2 : u[i2][j2] =
     f(u[i2][j2 + 1], u[i2][j2 − 1], u[i2 − 1][j2], u[i2 + 1][j2]);
 3 : u[i3][j3] =
     f(u[i3][j3 + 1], u[i3][j3 − 1], u[i3 − 1][j3], u[i3 + 1][j3]);
}
```

Figure 5: Red-Black Gauss-Seidel multifor code

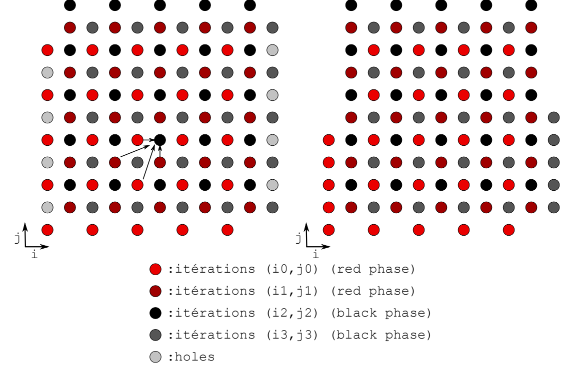


Figure 6: Red-Black Gauss-Seidel referential domain

```
for (j = 1; y < N − 1; j+ = 2)
 u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
for (i = 2; x < N − 2; i+ = 2) {
 for (j = 2; j < N − 1; j+ = 2) {
  u[i][j] = f(u[i][j + 1], u[i][j − 1], u[i − 1][j], u[i + 1][j]);
  u[i][j + 1] = f(u[i][j + 2], u[i][j],
            u[i − 1][j + 1], u[i + 1][j + 1]); }
 for (j = 1; j < N − 1; j+ = 2) {
  u[i + 1][j] = f(u[i + 1][j + 1], u[i + 1][j − 1],
            u[i][j], u[i + 2][j]);
  u[i + 1][j + 1] = f(u[i + 1][j + 2], u[i + 1][j],
                 u[i][j + 1], u[i + 2][j + 1]); } }
for (j = 2; y < N − 1; j+ = 2) {
 u[N − 2][j] = f(u[N − 2][j + 1], u[N − 2][j − 1],
            u[N − 3][j], u[N − 1][j]); }
```

Figure 7: Red-Black Gauss-Seidel generated code

resulting execution time. Here, some parallelism within the red points and within the black points can still be exploited, but also parallelism between red and black points. As an example, we show as a source code the sequential code that could be generated from this multifor-loop nest in Figure 7.

Notice also that more generally, when dependencies allow it, such a decomposition of a loop-nest computation into separated lattices, and expressed as a multifor-loop nest, provides another parallelization strategy that may be often quite interesting due to data locality issues.

Matrix product by blocks: The third example is an algorithm to compute the product of two matrices $n \times n$, $(A \times B = C)$, by partitioning the matrices into uniform blocks. The matrix product is then carried out block by block. We split the two matrices A and B as follows:

- matrix A is divided into two matrices A_1 and A_2 whose dimension is $n/2 \times n$.
- matrix B is divided into two matrices B_1 and B_2 whose dimension is $n \times n/2$.

The product $A \times B = C$ translates to four products: $A_1 * B_1 = C_1, A_1 * B_2 = C_2, A_2 * B_1 = C_3$ and $A_2 * B_2 = C_4$, i.e.,

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix}$$

```
multifor (i_1 = 0, i_2 = 0, i_3 = n/2, i_4 = n/2;
          i_1 < n/2, i_2 < n/2, i_3 < n, i_4 < n;
          i_1 ++, i_2 ++, i_3 ++, i_4 ++;
          1,1,1,1 ; 0,0,0,0 ) {
  multifor (j_1 = 0, j_2 = n/2, j_3 = 0, j_4 = n/2;
            j_1 < n/2, j_2 < n, j_3 < n/2, j_4 < n;
            j_1 ++, j_2 ++, j_3 ++, j_4 ++;
            1,1,1,1 ; 0,0,0,0 ) {
    0 : c[i_1][j_1] = 0;
    1 : c[i_2][j_2] = 0;
    2 : c[i_3][j_3] = 0;
    3 : c[i_4][j_4] = 0;
      multifor (k_1 = 0, k_2 = 0, k_3 = 0, k_4 = 0;
                k_1 < n, k_2 < n, k_3 < n, k_4 < n;
                k_1 ++, k_2 ++, k_3 ++, k_4 ++;
                1,1,1,1 ; 0,0,0,0 ) {
        0 : c[i_1][j_1] = c[i_1][j_1] + a[i_1][k_1] × b[k_1][j_1];
        1 : c[i_2][j_2] = c[i_2][j_2] + a[i_2][k_2] × b[k_2][j_2];
        2 : c[i_3][j_3] = c[i_3][j_3] + a[i_3][k_3] × b[k_3][j_3];
        3 : c[i_4][j_4] = c[i_4][j_4] + a[i_4][k_4] × b[k_4][j_4];
        }
    }
}
```

Figure 8: Multifor matrix product code

The dimension of matrices C_1, C_2, C_3 and C_4 is $(n/2 \times n/2)$. These four products might be performed simultaneously and can be naturally expressed using the multifor structure as shown in figure 8.

Geometrically, the multifor iteration domain is a $(n/2 \times n/2 \times n)$ rectangle parallelepiped where each point is associated to four iterations of the four included loop-nests. This execution scheme corresponds to the MapReduce strategy since each for-loop nest computes a $n/2 \times n/2$ sub-block (map step), and the combination of all sub-blocks forms the resulting matrix C (reduce step).

Steganography: The fourth example is the decoding phase of a steganography code where an hidden image is extracted from an enclosing one. It is assumed that the upper left pixel of the hidden image is hidden within the upper left pixel of the enclosing image; $HWidth$ and $HHeight$ are the width and the height of the hidden image ; $EWidth$ and $EHeight$ are the width and the height of the enclosing image ; $EImage$ is the image hiding another image ; $HImage$ is the extracted output image that was hidden ; $MImage$ is the output enclosing image hiding no more the image that was hidden in $EImage$. The proposed multifor code version is composed of four simultaneous for-loop nests, the first being dedicated to the extraction of the hidden image, the second to the extraction of the part of the enclosing image which is hiding the hidden image, the third and the fourth being dedicated to copy the pixels directly to the retrieved enclosing image. Since the union of the third and fourth domain is not convex, two loop-nests are necessary to scan it. Notice that we introduce a shortcut in the syntax such that similar loop bodies can be instantiated differently depending on their associated loop-nest. The multifor code is shown in Figure 9 and the referential iteration domain in Figure 10. Notice that full parallelism is exhibited with this code.

```
RGBApixel decode_hidden(i, j)
{
 RGBApixel Pixel1 = *EImage(i, j);
 RGBApixel Pixel2;
 Pixel2.Red = Pixel1.Red%2;
 Pixel2.Green = Pixel1.Green%2;
 Pixel2.Blue = Pixel1.Blue%2;
 Pixel2.Alpha = Pixel1.Alpha%2;
 returnPixel2;
}

RGBApixel decode_main(i, j)
{
 RGBApixel Pixel1 = *EImage(i, j);
 RGBApixel Pixel2;
 Pixel2.Red = Pixel1.Red − Pixel1.Red%2;
 Pixel2.Green = Pixel1.Green − Pixel1.Green%2;
 Pixel2.Blue = Pixel1.Blue − Pixel1.Blue%2;
 Pixel2.Alpha = Pixel1.Alpha − Pixel1.Alpha%2;
 returnPixel2;
}

multifor (i_1 = 0, i_2 = 0; i_3 = 0, i_4 = HWidth; i_1 < HWidth,
          i_2 < HWidth, i_3 < HWidth, i_4 < EWidth;
          i_1 ++, i_2 ++, i_3 ++, i_4 ++; 1,1,1,1; 0,0,0,0)
 multifor (j_1 = 0, j_2 = 0, j_3 = HHeight, j_4 = 0; j_1 < HHeight,
          j_2 < HHeight, j_3 < EHeight, j_4 < EHeight;
          j_1 ++, j_2 ++, j_3 ++, j_4 ++; 1,1,1,1; 0,0,0,0)
 {
  0 : // Retrieve the hidden image
     * HImage(i_1, j_1) = decode_hidden(i_1, j_1);
  1 : // Retrieve the enclosing image
     * MImage(i_2, j_2) = decode_main(i_2, j_2);
  [2, 3] : // Retrieve the enclosing image
     * MImage([i_3, i_4], [j_3, j_4]) = *EImage([i_3, i_4], [j_3, j_4]);
 }
```

Figure 9: Multifor steganography code for the decoding phase

Secret key cryptosystem: The fifth example is a classic secret key cryptosystem that manipulates binary words. It proceeds by splitting a message m into blocks of constant size. These cryptosystems are characterized by the length of each block, the operating mode and the encryption system of each block. Each cipher mode comprises:

1. Cutting in many blocks $m_1, ..., m_k$ the plain text message m;
2. Encrypting the blocks m_i resulting in the encrypted blocks $c_1, ..., c_k$;
3. Concatenating the blocks $c_1, ..., c_k$ to construct the encrypted message c.

Each block is encrypted through the product of two cryptosystems T_1 and T_2. It is classically computed using a loop of the form:

```
for (i = 0; i < k; i ++) {
    c[i] = Encrypt(m[i], T_1);
    c[i] = Encrypt(c[i], T_2); }
```

Suppose the encryption of each block by a given cryptosytem consumes one unit of time. The time required to encrypt the entire message using this loop is $2 \times k$. Let us write this code using a multifor structure:

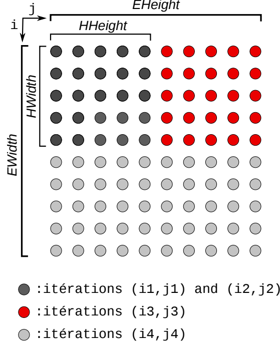


Figure 10: Referential domain for the multifor steganography code

$$\begin{aligned}&\textbf{multifor } (i_1 = 0, i_2 = 0; i_1 < k, i_2 < k;\\&\quad i_1++, i_2++; 1, 1; 0, 1)\ \{\\&\quad 0 : c[i_1] = Encrypt(m[i_1], T_1);\\&\quad 1 : c[i_2] = Encrypt(c[i_2], T_2); \}\end{aligned}$$

This form allows to take advantage of a pipeline scheme where two successive blocks are encrypted in parallel respectively by the cryptosystems T_1 and T_2. Thus, the time required to encrypt the message is $k + 1$.

5. A PROMISING PERSPECTIVE: NON-LINEAR MAPPING

Among the numerous possible extensions, an important one is to map iteration spaces together following a non-linear fashion, such that their shapes has no influence in the mapping. In general, this would allow to execute iterations of any loop nest by any other one of the same trip count, and thus to enlarge significantly the way iterations of different loops can be mapped together. Hence a multifor construct could express quite different computations in a concise way, and augment the number of optimization and parallelization opportunities. Notice that loop nests with different trip counts can also be handled by splitting the largest nest such that one of the resulting nest has the convenient trip count.

Our idea is based on ranking Ehrhart polynomials. It has been shown in previous works dealing with spatial data locality optimization, that it is possible to compute an Ehrhart polynomial associated to a loop nest giving the rank of an iteration [5, 9]. These polynomials have specific properties as being necessarily monotonically increasing according to the lexicographic order of the loop indices, and also defining a bijection between iteration points and the interval of strictly positive integers between one and the total iteration count of the loop nest.

Since ranking Ehrhart polynomials define such bijections, the ability of inverting them would provide a way to retrieve the loop indices corresponding to the rank of an iteration. Hence at any iteration of a loop nest, it would be possible to compute, from the rank, the values of loop indices that would have been reached while running another nest. Thus, any nest could be run by another one, and any iteration space could be mapped onto another one, following the rank of the iterations. Hence this Section deals with the problem of inverting ranking Ehrhart polynomials.

Before proposing a general resolution, we first present a 2-dimensional example.

5.1 2-dimensional example

Consider the two loop nests in listings (1) and (2), where $instruction_k(i_1, i_2)$ denotes the instruction block computed at iteration (i_1, i_2). These loops have as respective ranking Ehrhart polynomials:

$$P_1(i,j) = \frac{i(i-1)}{2} + j \text{ and } P_2(i',j') = i'M_2 + j'$$

$$\begin{aligned}&\textbf{for } (i = 1, i < N, i++)\\&\quad \textbf{for } (j = 0, j < i, j++)\\&\quad\quad instructions_1(i,j);\end{aligned} \qquad (1)$$

$$\begin{aligned}&\textbf{for } (i' = 0, i' < M_1, i'++)\\&\quad \textbf{for } (j' = 0, j' < M_2, j'++)\\&\quad\quad instructions_2(i',j');\end{aligned} \qquad (2)$$

Taking the assumptions that both nests have the same iteration count and that there is no dependency between $instructions_1$ and $instructions_2$, we could merge the two former loop nests and write (3).

$$\begin{aligned}&\textbf{for } (i' = 0, i' < M_1, i'++)\\&\quad \textbf{for } (j' = 0, j' < M_2, j'++)\ \{\\&\quad\quad instructions_1(\mathbf{i},\mathbf{j});\\&\quad\quad instructions_2(i',j');\ \}\end{aligned} \qquad (3)$$

However, we need to express indices (i, j) as a function of (i', j') in order to preserve the execution order of the block $instructions_1$. More precisely, for each iteration number K in loop nest (3), we want to execute the Kth iteration of loop nest (1). This is why we must invert the ranking Ehrhart polynomial P_1, to compute $(i, j) = P_1^{-1}(P_2(i', j'))$.

For any rank K, we have to find a couple of indices (i_0, j_0) such that $P_1(i_0, j_0) = K$. The main idea is to cut the 2-dimensional problem into two one-dimensional problems.

Let us define $Q_1(i) = P_1(i, 0) = \frac{i(i-1)}{2}$. As ranking Ehrhart polynomials are (strictly) increasing, the following relation holds:

$$Q_1(i_0) = P_1(i_0, 0) \leq P_1(i_0, j_0) = K \leq P_1(i_0 + 1, 0) = Q_1(i_0 + 1)$$

And, for the same reason, we know that index i_0 is unique on $\mathbb{N}_+$. Let us now consider polynomial Q_1 as a polynomial over $\mathbb{R}$. By continuity of Q_1 over $\mathbb{R}$, there exists $\alpha \in [0, 1[$ such that $Q_1(i_0 + \alpha) = K$. This shows that the equation $Q_1(x) = K$ has at least one real solution. So we have to find x such that:

$$Q_1(x) = K \Leftrightarrow Q_1(x) - K = 0 \Leftrightarrow \frac{x(x-1)}{2} - K = 0$$

Obviously, this last equation has two real roots:

$$x_1 = \frac{1}{2} - \sqrt{\frac{1+8K}{4}}, x_2 = \frac{1}{2} + \sqrt{\frac{1+8K}{4}}$$

To select the convenient root, we notice that for all $K > 0$, $x_1 \leq 0$. As $i_0 \geq 1$ (according to the loop bounds in (1)), $i_0 + \alpha = x_2$, and thus: $i_0 = \lfloor x_2 \rfloor$. We can now replace i_0 by its value in $P_1(i_0, j_0)$:

$$P(i_0, j_0) = \frac{1}{2}\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1+8K}{4}} \right\rfloor\right)\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1+8K}{4}} \right\rfloor - 1\right) + j_0$$

and finally deduce j_0:

$$j_0 = K - \frac{1}{2}\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1+8K}{4}} \right\rfloor\right)\left(\left\lfloor \frac{1}{2} + \sqrt{\frac{1+8K}{4}} \right\rfloor - 1\right)$$

The resulting code is shown in listing 4.

$$
\begin{array}{l}
K = 0; \\
\textbf{for } (i' = 0, i' < M_1, i' + +) \\
\quad \textbf{for } (j' = 0, j' < M_2, j' + +) \; \{ \\
\quad\quad K + +; \\
\quad\quad i = floor(sqrt((1 + 8 * K)/4) + 1/2); \\
\quad\quad j = K - i * (i - 1)/2; \\
\quad\quad instructions_1(i, j); \\
\quad\quad instructions_2(i', j'); \; \}
\end{array}
\tag{4}
$$

We now present the general case, which can be easily deduced from the 2-dimensional case.

5.2 General case

Without any loss of generality, we assume all loop indices lower bounds equal 0. We consider the N-dimensional ranking Ehrhart polynomial $P(i, j, k, ...)$, and for each K, we seek the tuple $(i_0, j_0, k_0, ...)$ such that $P(i_0, j_0, k_0, ...) = K$.

Similarly to the previous example, we start with the outermost loop index i_0. We define $Q_i(i) = P(i, 0, 0, ..., 0)$ and solve $Q_i(x) - K = 0$. Here is the only issue that differs from the 2D case: as we can't state that Q_i is monotonically increasing on $\mathbb{R}_+$, we have to find a criterion to select the root giving the sought index.

First, we obviously eliminate complex and negative solutions, since $i_0 \in \mathbb{N}_+$, and consider $n \leq N$ positive real roots $\{x_1, .., x_n\}$. As we know that i_0 is unique, a way to select the convenient root is to check which $x \in \{x_1, .., x_n\}$ satisfies :

$$Q_i(\lfloor x \rfloor = i_0) \leq Q_i(x) \leq Q_i(\lceil x \rceil = i_0 + 1)$$

However, this strategy is only applicable at runtime, and may add non negligible time overhead. A compile-time solution is to check if Q_i is monotonically increasing on $\mathbb{R}_+$ by examining its derivative. If so, any root in $\{x_1, .., x_n\}$ is suitable.

Once i_0 has been found, the process starts again with $Q_j(j) = P(i_0, j, 0, ..., 0)$, and so on until all indices have been computed.

6. CONCLUSION

We have proposed a new programming control structure called "multifor" and showed that it allows the polytope model to handle programming models that were not attainable directly before. Important related theoretical studies have still to be conducted, as dependency analysis between the for-loops composing a multifor-loop, or optimizing code transformations that considers interactions between the for-loops.

Many interesting extensions can also be studied as making header parameters, or instructions, dependent of several for-loop indices composing the same multifor-loop level, or defining non-invariant grains and offsets, or introducing conditionals on the effective run of the for-loops, etc. In this paper, we showed that it may be possible to handle non-linear mapping of iteration spaces using inverted ranking Ehrhart polynomials.

The multifor structure can also be used as a representation model for some interacting mechanisms, as concurrent memory accesses, as it is done for sequential codes in [7].

We are planning to implement multifor structures in the Clang/LLVM compiler as an extension to C/C++.

7. REFERENCES

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations.* Kluwer Academic Publishers, 1993. ISBN 0-7923-9318-X.

[2] O. A. R. Board. Openmp application program interface, version 3.1, 2011.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, Sept. 2008.

[4] I. Christadler, G. Erbacci, and A. D. Simpson. Facing the multicore-challenge ii. chapter Performance and productivity of new programming languages, pages 24–35. Springer-Verlag, Berlin, Heidelberg, 2012.

[5] P. Clauss and B. Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News*, 28(1):11–19, Mar. 2000.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. of the 22nd annual symp. on Principles of distributed computing*, PODC '03, pages 92–101. ACM, 2003.

[7] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *6th annual IEEE/ACM int. symp. on Code generation and optimization*, pages 94–103, Boston, United States, Apr. 2008. ACM.

[8] C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[9] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. Supercomput.*, 21(1):37–76, Jan. 2002.

[10] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. Trinder. Seq no more: Better strategies for parallel haskell. In *Proceedings of the 3rd ACM SIGPLAN symposium on Haskell*, pages 91–102, Baltimore, MD, United States, Sept. 2010. ACM Press.

[11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima Series. Artima Press, 2011.

[12] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *Proc. of the 38th annual ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

[13] K. F. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In *FLOPS*, pages 13–18, 2010.

[14] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. of the 11th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[15] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th annual ACM symp. on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[16] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Transformation

Dustin Feld and Thomas Soddemann
Fraunhofer SCAI, Schloss Birlinghoven, 53754
Sankt Augustin, Germany
[dustin.feld|thomas.soddemann]
@scai.fraunhofer.de

Michael Jünger and Sven Mallach
Institut für Informatik, Universität zu Köln,
Weyertal 121, 50931 Köln, Germany
[mjuenger|mallach]
@informatik.uni-koeln.de

ABSTRACT

Although Single Instruction Multiple Data (SIMD) units are available in general purpose processors already since the 1990s, state-of-the-art compilers are often still not capable to fully exploit them, i.e., they may miss to achieve the best possible performance.

We present a new hardware-aware and adaptive loop tiling approach that is based on polyhedral transformations and explicitly dedicated to improve on auto-vectorization. It is an extension to the tiling algorithm implemented within the PluTo framework [4, 5]. In its default setting, PluTo uses static tile sizes and is already capable to enable the use of SIMD units but not primarily targeted to optimize it. We experimented with different tile sizes and found a strong relationship between their choice, cache size parameters and performance. Based on this, we designed an adaptive procedure that specifically tiles vectorizable loops with dynamically calculated sizes. The blocking is automatically fitted to the amount of data read in loop iterations, the available SIMD units and the cache sizes. The adaptive parts are built upon straightforward calculations that are experimentally verified and evaluated. Our results show significant improvements in the number of instructions vectorized, cache miss rates and, finally, running times.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*; B.3.2 [**Memory Architectures**]: Design Styles—*Cache Memories*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Algorithms, Performance, Experimentation

Keywords

SIMD, SSE, AVX, TSS, polyhedral model, polyhedron model, tiling, code generation, automatic parallelization, vectorization, loop optimization, loop transformation

1. INTRODUCTION

Single Instruction Multiple Data (*SIMD*) units offer a speedup-potential brought to a wide range of users. In order to exploit the available concurrency of modern CPUs, one must achieve shared memory parallelism by multithreading and, at the same time, vectorization by effectively applying instructions to multiple data. Both tasks impose a difficult challenge to experienced programmers as well as state-of-the-art compilers. In this paper, we focus on the effective automatic exploitation of SIMD units. We found severe limitations when we made some experiments with the vectorizer of the GNU C Compiler (gcc, version 4.6). To our surprise, even for loops that can be vectorized in a straightforward manner, SSE-instructions were only set if the range specified by the loop bounds was a multiple of the SIMD register width divided by the size of the data type.

Vectorization is difficult since it usually requires an analysis of the dependence structure of the code to be optimized. It demands for the right ordering of instructions and fast accesses to data in order to leverage its full speedup potential. Unfortunately, in many cases even the existence of a legal order of instructions cannot be easily recognized by humans or compiler procedures. Here, polyhedral code optimization is a powerful tool to detect and exploit parallelism in loop structures. It provides a formal characterization of affine loop nests, their iteration spaces, the dependencies between statements and the iteration points in which they occur [2, 7, 9, 12, 13]. It can therefore be used to generate valid transformations of a given source code. Further, *tiling* (or *blocking*) of nested loops is a well-known technique to improve data locality and, if concurrency concerning outer loop dimensions is possible, to perform automatic parallelization [21]. In this manner, transformations such as loop fusion, splitting, skewing, or interchange may enable a coarse-grain (tile-wise) or a fine-grain (loop-internal) concurrency (or both) even where this is not the case for the original source code [2].

There is extensive literature dealing with the optimization of tilings. However, to the best of our knowledge, there is yet no implemented approach that integrates loop transformations which broadly enable automatic vectorization with tilings and an adaptive hardware-aware tile size selection (TSS). Trifunovic et al. [20] analyze the impact of loop transformations on the resulting possibilities to apply auto-vectorization and performance by means of a cost model that can be seamlessly integrated into the polyhedral model. Based on this, they propose a framework to choose the best-suited loop for vectorization within a nest. Unfortunately, it does not comprise a TSS model. As opposed to that, there exist several TSS models which are, however, not explicitly geared towards an improved vectorization. Coleman and McKinley [8] present an iterative technique to calculate cache-fitting tile sizes for all loop dimensions. This is interesting in conjunction with the approach presented

in this paper, especially for cases where there is no vectorizable loop available. This is also true for Sarkar's and Megiddo's [17] approach to generate tile sizes via a memory-oriented cost model of the given loop nest and an analytical model to finally perform the TSS. However, it is restricted to loop nests of depth two or three. Shirako et al. [18] present a method to analytically bound the search space for tile sizes that lead to a good performance. They potentially leave loop dimensions unblocked and propose to tile a vectorizable loop into large blocks. While these general ideas are similar to ours, their approach is merely capable to perform a one-level tiling and based on an empirical search method while ours uses a straightforward polyhedral and analytical basis. Ghosh et al. [11] propose to use cache miss equations [3] for TSS and in order to detect poor cache performance. They show how loop transformations (including tiling) can help to improve on this. Abella et al. [1] use the equations to determine optimal tile sizes by means of a genetic algorithm. However its running time on some inputs is not applicable for a common compilation process.

In this paper, we present a new hardware-oriented TSS approach explicitly targeted to vectorization. It is an adaptive procedure that specifically tiles vectorizable loops with dynamically calculated sizes. We implemented our approach as an extension to PluTo [4, 5] which is an academic source-to-source compiler framework that performs tilings based on polyhedral optimization. The resulting code can be compiled by any *C* compiler. In particular, we use PluTo to obtain valid transformations and tilings of loops as well as to gather information about those loops that can actually be vectorized. However, in contrast to PluTo, we orient the tiling towards an effective use of SIMD units. Instead of partitioning the iteration space with respect to all loop dimensions and into tiles of static size, we restrict the tiling to those loops that are relevant for the data to be processed in a vectorized manner. Further, we dynamically adapt the size of tiles to the SIMD register width and the cache sizes of the underlying hardware. Ideally, our approach leads to a software pipeline of blocks to be processed by the SIMD units. We show for two example source codes that we obtain improved running times by a combination of a measurably well-performing stream of data through the cache hierarchy and an increased rate of issued SIMD instructions.

2. POLYHEDRAL TRANSFORMATIONS, VECTORIZABLE LOOPS AND TILINGS

A loop qualifies for vectorization if it is innermost with respect to its nest and parallelizable, i.e., there are no data dependencies between its iterations.

In the polyhedral model, one considers the iteration points of a loop nest and the dependencies between them using $\mathbb{Z}$-polyhedra [9, 12, 13], as depicted in Fig. 1. In this representation, the index variable of each loop relates to one dimension of the associated polyhedron. A valid transformation corresponds to a change in the order of execution of the iteration points that preserves compliance with the dependencies. This may include the manipulation of loop dimensions (index variables) leading to a deformation of the polyhedron such that computations can be processed in parallel with respect to one or more of the dimensions. By applying integer programming techniques, PluTo is capable to perform transformations such that the necessary communication across the dimensions of the resulting loops is minimized [6]. This is beneficial for parallelization. If communication is not necessary with respect to an outer loop dimension, then a parallelization of the inner loops' iterations is possible. If communication is not necessary with respect to the innermost loop dimension, then the iterations of this loop can be processed in parallel, e.g., by vectorization. Fortunately, PluTo is able to mark loops that qualify for vectorization, possibly by interchanging it to become the innermost one (which we assume from now on to be the case). Furthermore, PluTo can compute a partitioning of the iteration space into tiles. Consider Fig. 1 for an example where a legal (rectangular) tiling is possible only after a transformation of the loop nest. The left tiling is illegal, because iteration points between blocks have reciprocal dependencies. After manipulating the loop dimensions, it is possible to apply the tiling depicted in the right image since it now allows for an order of execution that respects all dependencies. We use the polyhedral representations within PluTo to obtain such legal loop tilings.

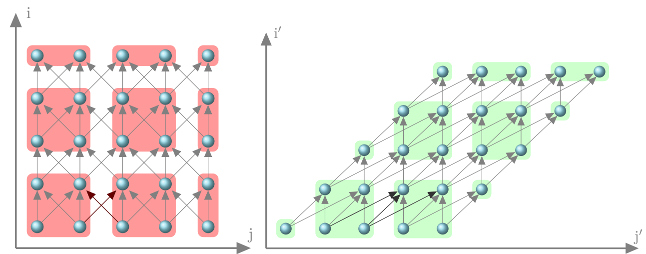


Figure 1: A polyhedral representation of a nested loop with its dependence structure and an invalid tiling (left) as well as a valid one (right).

3. CENTRAL IDEAS

In order to fully exploit the speedup potential of SIMD units, it is advantageous to have access to data that is accordingly prefetched into the available cache hierarchy. How can we achieve this by a smart tiling?

As already stated, a loop to be vectorized must be (made) innermost. Its index variable imposes a constant offset/stride between the memory addresses of data that is to be successively packed into (SIMD) registers while the indices of all other loops remain constant. It can thus be expected that it is beneficial for the successful prefetching of operands, if the innermost loop is executed for a relatively large number of iterations before the control flow leaves it and manipulates the other loops' index variables. Then, the prefetcher should be able to better pre-load future operands while current calculations are served from the cache hierarchy. The prefetching causes cache misses that do not significantly influence the running time but impose cache hits when the operands are really needed. On the other hand, a large number of iterations in the innermost loop may also harm spatial locality in comparison to fewer ones. This is particularly true if the index variable of the innermost loop does not correspond to the minor dimension of all accessed arrays (and therefore not to one-strides in memory).

Clearly, there must be a trade-off between the advantages and disadvantages of a large blocking of the innermost loop. However, we believe that the blocking of (a) *all* loops into

(b) *constantly* sized blocks (like, e.g., the default tiling into blocks of 32 iterations performed by PluTo) is unlikely to perform well for every application and every system. This is especially true for deeply nested loops with many statements within the innermost loop. As an example, if d is the depth of a loop nest, then the innermost statements of a tile using PluTo's default tiling are executed 32^d times and the data accessed by these statements is unlikely to fit into a cache with increasing d. Nonetheless, this fits quite well for 'typical' loop nests with depth two or three and today's usual cache sizes. PluTo allows to also enforce a second-level tiling of the resulting loops into blocks of 8 which already addresses the cache memory hierarchies of modern processors. Alternatively, the user may specify tile sizes manually. We used this fact to elaborate on our ideas and made experiments with different sizes for a specific tiling of one or two loops only.

3.1 Experiments with manual tile sizes

We consider two test cases both of which are written in `C` and have been taken from PluTo's example suite:

- A standard *matrix multiplication* (see Fig. 18)
- A *correlation matrix algorithm* (see Fig. 21)

The environment for the tests and the benchmarks in the subsequent section comprises the following Intel Xeon processor and is running Scientific Linux 6.

CPU:	Intel® Xeon® CPU X5650 (2.67 GHz)
L1 / L2 / L3 cache:	32 KB (data) / 256 KB / 12288 KB
SSE version:	4.2 (128 bit registers)

All runs were performed single-threaded using gcc 4.6 or icc 13.0, both with optimization level `-O3` on single precision floating point data.

3.1.1 Manual one-level tiling

To start, we consider a pure tiling of the vectorized loop only with manually set tile sizes $q^{L1} = 4$, 64 and 256. We compare it to the original source code and PluTo, configured to also generate a one-level tiling of all loops using its default sizes (see Fig. 23).

First, we evaluate the resulting matrix multiplication codes for various choices of (symmetric) matrix sizes $N = M = K$ within a small range for a fine-grain analysis.

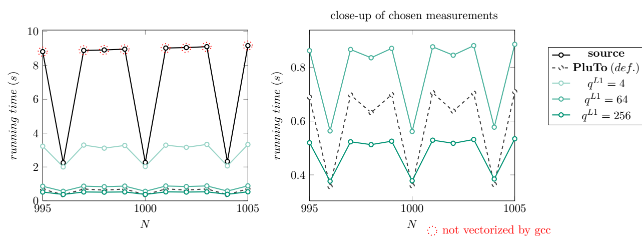


Figure 2: Running times for the one-level-tiled matrix multiplication (fine grain) with gcc

As is depicted in Fig. 2, the performance of the original (source) version highly depends on the size of the matrix. For sizes that are multiples of four, the code generated by gcc performs about four times faster. Application of gcc's verbose mode confirmed the hypothesis that it is only able to apply auto-vectorization in these cases. Likewise, already a very fine-grain tiling with $q^{L1} = 4$ suffices in order to enable auto-vectorization by gcc for any matrix size. Increasing q^{L1} to 256 leads to running times that are comparable to those achieved by the default tiling of PluTo or even faster. The deviant behavior for sizes that are not multiples of four can be explained with the 'remainders' that result from the tiling in these cases. We experimentally determined a perfect correlation between the size of the last tiles and the rate of vectorization, i.e., again gcc appears to not vectorize these remaining loop iterations, especially if their number is odd. We further elaborate on different parameters that influence the sustained performance in Sect. 4.2.

In Table 1, the relative performance from the fine-grained setting is confirmed for a larger interval of register-fitting (multiples of four) and non-register-fitting matrix sizes.

N	256	512	1024	1536	2048
source	0.0096	0.0905	2.3191	7.9956	20.2370
PluTo (def.)	0.0063	0.0536	0.4645	1.4801	3.7902
$q^{L1} = 4$	0.0202	0.1829	2.1804	8.2643	26.1112
$q^{L1} = 64$	0.0071	0.0672	0.6092	2.4307	10.0529
$q^{L1} = 256$	0.0048	0.0454	0.3887	1.6499	5.2734

N	257	513	1025	1537	2049
source	0.0223	0.1956	9.9101	34.0321	82.7667
PluTo (def.)	0.0121	0.1076	0.8839	2.9326	7.5070
$q^{L1} = 4$	0.0444	0.3702	3.6207	14.8025	38.7883
$q^{L1} = 64$	0.0113	0.1081	0.9672	3.7944	14.0045
$q^{L1} = 256$	0.0079	0.0673	0.5736	2.2688	6.4815

Table 1: Running times for the one-level-tiled matrix multiplication (coarse grain) with gcc

We apply the same test cases for the correlation matrix algorithm which has a more complicated code structure. As can be seen in Fig. 3 and Table 2, gcc is not able to vectorize its original version at all. However, again tiling the loops corresponding to the M-matrix-dimension (together with the corresponding code transformation) enables gcc to auto-vectorize the code already when setting q^{L1} to 4. With $q^{L1} = 256$, the running times are almost always faster than with PluTo's default one-level tiling.

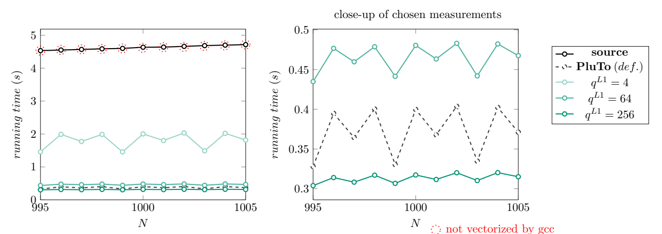


Figure 3: Running times for the one-level-tiled correlation matrix algorithm (fine grain) with gcc

3.1.2 Manual two-level tiling

Now, we consider a two-level tiling (of the vectorized loop and additionally the outermost loop of its nest) again using manually set tile sizes $q^{L1} = 4$, 64 and 256 for the vectorized loop and $q^{L2} = 2$, 4 and 8 for the outermost one. We

N	256	512	1024	1536	2048
source	0.0114	0.2140	4.8512	22.4826	60.9249
PluTo (def.)	0.0078	0.0599	0.4844	1.5089	3.8053
$q^{L1}=4$	0.0221	0.1844	2.0942	8.5166	21.8286
$q^{L1}=64$	0.0065	0.0573	0.5204	1.8351	5.7119
$q^{L1}=256$	0.0054	0.0404	0.3472	1.2619	3.7619

N	257	513	1025	1537	2049
source	0.0115	0.2152	4.9040	22.4878	61.0156
PluTo (def.)	0.0070	0.0533	0.4367	1.3892	3.3829
$q^{L1}=4$	0.0209	0.1745	1.8981	7.5941	20.9514
$q^{L1}=64$	0.0065	0.0557	0.5031	1.7586	5.5304
$q^{L1}=256$	0.0054	0.0403	0.3430	1.2401	3.7118

Table 2: Running times for the one-level-tiled correlation matrix algorithm (coarse grain) with gcc

configured PluTo to also generate a two-level tiling using its default sizes (see Fig. 23). The running times are depicted in Figures 4 and 5 and, for ease of comparison, the one-level tiling running times are shown dotted in the right graphs. As might have been expected, the additional blocking leads to shorter running times in all cases.

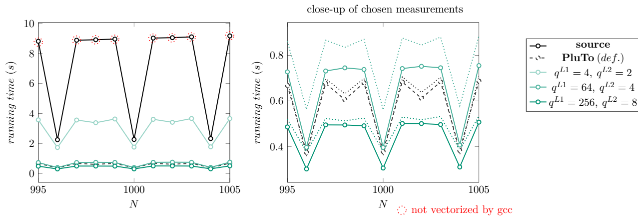


Figure 4: Running times for the two-level tiled matrix multiplication (fine grain) with gcc

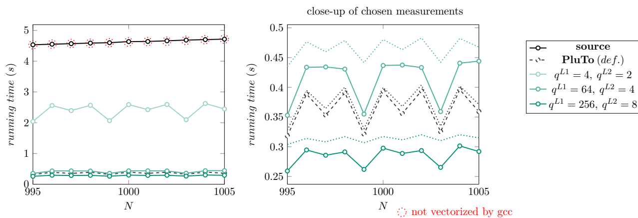


Figure 5: Running times for the two-level tiled correlation matrix algorithm (fine grain) with gcc

3.2 Ideas towards an automatic derivation of tile sizes

The results presented so far appear to support the hypothesis that a tiling of specific loops can be as effective as a tiling of all loops. Furthermore, they tell us that loop ranges should be *SIMD-specific*, i.e., fit to SIMD register widths, in order to leverage the full potential of gcc's automatic vectorizer. Until now, the running times seem to improve with increasing tile sizes. Clearly, the amount of increase that pays off must be limited. Our intuition was that a natural limitation should stem from the cache sizes. Ideally, the block size for the first level tiling should be fitted to the ratio of the size of the L1 cache and the amount of data read in one iteration of the loop to be vectorized. Similarly, the block size for the second level tiling should be fitted to the ratio between the L2 cache size and the L1 cache size. In the best case, this strategy could lead to the situation that all required data for one tile of the vectorized loop fits into the L1 cache and that such blocks of data can be successively pipelined from the L2 cache. With the larger innermost tile size, we should be able to move compulsory cache misses to the prefetcher and, with the *cache-specific* tiling, we should optimize for less capacity cache misses at the same time. Hence, we call the just described combination of both concepts a SIMD- and cache-specific (*SICA*) tiling. For the (optional) second-level tiling, we select the outermost loop of the corresponding nest. This strategy keeps changes to the inner loops as rare as possible which, as a heuristic, should be good for the prefetcher.

We set up a straightforward model to compute all the information needed for experiments to analyze whether this is indeed a promising approach. First of all, we need to know how many new operands must be loaded by each of a loops' iterations. This requires an analysis of the loops' statements, since, e.g., operands (or addresses) that are accessed multiple times should be loaded only once per iteration. Furthermore, constant values as well as operands that do not depend on the vectorized loop should be loaded even only once at all for the entire loop. As already stated in Sect. 2, the innermost loop may, in general, contain several statements which are considered to compose a statement-block. We calculate the total amount of data elements $\mathcal{E}$ to load per iteration for each of these blocks.

$$\text{Elements per Iteration} : \text{ElPeIt} = \mathcal{E} \tag{1}$$

Further, we need the following hardware parameters:

- $\mathcal{C}_{L1}$: The size of the L1 cache (in KBytes)
- $\mathcal{C}_{L2}$: The size of the L2 cache (in KBytes)
- R: The SIMD register width (in Bits)

Using this information, the number of elements of the given type (e.g. `float` or `double`) with size $\mathcal{D}$ (in Bytes) that fit into the L1 cache can be calculated as follows:

$$\text{cache size in elements} : \text{CaSiEl} = \frac{\mathcal{C}_{L1} * 1024}{\mathcal{D}} \tag{2}$$

The number of operands of the given data type with size $\mathcal{D}$ (in Bytes) that can be packed into the SIMD registers is denoted by:

$$\text{elements per register} : \text{ElPeRe} = \frac{\mathcal{R}}{8 * \mathcal{D}} \tag{3}$$

Since there might be other variables that should be cached, one might like to adjust the ratio of the L1 cache size to use to, e.g., only 90% or 80%. We therefore introduce an according parameter ρ with the meaning that $\rho = 1.0$ relates to 100%.

$$\text{ratio of cache to use} : \rho \tag{4}$$

For each block of statements we may now compute how many iterations shall be blocked so that all operands required by this block ideally fit into the L1 cache at once:

$$\text{iterations to block} : \text{ItToBl} = \rho * \frac{\text{CaSiEl}}{\text{ElPeIt}} \tag{5}$$

Finally, we want to make the first-level tile size q^{L1} a multiple of the SIMD-register width. Hence, we compute the greatest multiple of ElPeRe that fits into the L1 cache as follows:

$$q^{L1} = \left\lfloor \frac{\text{ItToBl}}{\text{ElPeRe}} \right\rfloor * \text{ElPeRe} \tag{6}$$

Summing up all calculations into a single formula yields:

$$q^{L1} = \left\lfloor \frac{\rho * \frac{\mathcal{C}_{L1}*1024}{\mathcal{D}} * \frac{1}{\mathcal{E}}}{\frac{\mathcal{R}}{8*\mathcal{D}}} \right\rfloor * \frac{\mathcal{R}}{8*\mathcal{D}} \tag{7}$$

$$= \left\lfloor \rho * \frac{\mathcal{C}_{L1} * 8192}{\mathcal{R} * \mathcal{E}} \right\rfloor * \frac{\mathcal{R}}{8*\mathcal{D}} \tag{8}$$

Now for the second level tiling, we simply calculate the ratio of the two cache sizes.

$$q^{L2} = \frac{\mathcal{C}_{L2}}{\mathcal{C}_{L1}} \tag{9}$$

3.2.1 *Example*

Consider the standard matrix multiplication for single precision floating point data with only one statement $C[i][\mathbf{j}] = C[i][\mathbf{j}] + \alpha * A[i][k] * B[k][\mathbf{j}]$ and vectorized **j**-loop. Two new data elements need to be loaded per **j**-iteration, namely $C[i][\mathbf{j}]$ and $B[k][\mathbf{j}]$. This leads to the following SICA L1 tile size for our test system with 32 KByte of L1 cache and 128 Bit SSE registers:

$$q^{L1} = \left\lfloor \rho * \frac{32 * 8192}{128 * 2} \right\rfloor * \frac{128}{8*4} \overset{\rho=1.0}{=} 4096 \tag{10}$$

Since our system has 256 KByte of L2 cache, the second block size evaluates to $q^{L2} = \frac{256}{32} = 8$.

3.3 SICA extensions to PluTo

We implemented our SICA tiling as an extension of PluTo together with several new parameters and functionalities that cause only neglectable overhead. It comprises adaptive components, like, e.g., procedures to determine hardware parameters by using the `CPUID` [15] instructions (they can be equally manually set via a configuration file) and new routines to calculate the amount of data loaded in one loop iteration. If an innermost loop contains multiple statements (possibly as a consequence of the polyhedral transformations), it is viable to group them into blocks for which the tiling is then performed independently. Unlike in PluTo's original tiling algorithm, every block of statements can be associated with individual tile sizes. This is necessary since different statement blocks (within the same loop nest) may require a different number of operands to be loaded per iteration. Nevertheless, one may still request a globally uniform tiling by adopting the minimal or maximal determined tile size for all statement blocks. As an example, a rectangular SICA tiling of a perfectly nested loop with only one statement S is depicted in Fig. 24.

4. ANALYSIS AND BENCHMARKS

The following experiments can be divided into two parts. First, we deliver a verification of the proposed correlation between the amount of data read within the vectorized loops' iterations, tile sizes, cache sizes and performance. After that, we evaluate the performance of the corresponding adaptive approach concerning running times, cache miss rates, TLB misses and the rate of issued SIMD instructions.

4.1 Verification of the approach

In order to evaluate the impact of the tile sizes only, we fix some (asymmetric) matrix sizes. We keep the matrix dimensions corresponding to non-vectorized loop dimensions small in order to be able to benchmark a large interval of sizes for the vectorized one and to obtain reasonable running times at the same time.

4.1.1 *SICA L1 tiling*

We again start our experiments with a one-level tiling. For the matrix multiplication, we set $M = 189, N = 139233$ and $K = 189$, since the loop corresponding to the N-dimension is the vectorized one. Then, we vary the tile sizes by successively changing the cache-ratio parameter ρ from 0 to 10 in steps of 0.01 units. This results in 1000 different versions of the code with tile sizes up to 36864. Fig. 6 shows their corresponding running times which are all within the shadowed area while the line is a cubic interpolation of them with some smoothing applied.

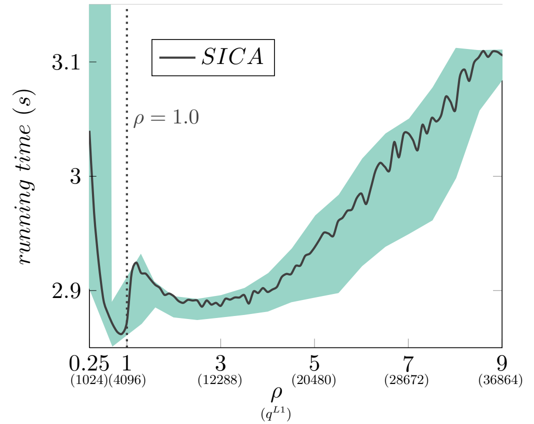


Figure 6: Impact of the tile size on the running time of the matrix multiplication code with gcc

There is a global optimum that corresponds to about 90% of the L1 cache size. This appears to confirm a correlation between performance, tile and cache sizes. Furthermore, we claimed a relationship between the optimum tile sizes and the amount of data that needs to be loaded within the vectorized loops' iterations. To verify this, we additionally measured the running times of codes performing the addition of two or even three matrix multiplications (the corresponding statements in the nested loop are depicted in Fig. 20). For each additional matrix multiplication, there is one additional operand to be loaded per iteration. Regarding our test system, this leads to tile sizes of $q^{L1} = 4096$ for a single matrix multiplication [*matmul1*], $q^{L1} = 2728$ for the sum of two of them [*matmul2*] and $q^{L1} = 2048$ for the sum of three of them [*matmul3*]. Again, Fig. 7, in which we scaled the running times of the different versions to a common ordinate (each individual range in seconds is denoted in brackets), shows that the best tile size is at about 80 to 90% of the theoretical optimum.

As before, the same experiments are repeated for the correlation matrix algorithm. Here, we set $M = 11923$ and $N = 89$ since loops corresponding to the M-dimension are vectorized. Since there are several statements and multiple loop nests (see Fig. 21 for the original code and Fig. 22

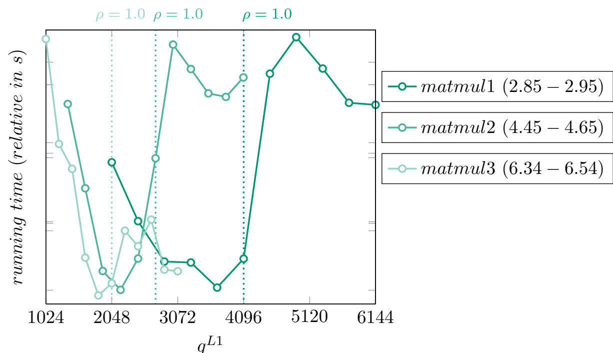


Figure 7: The effect of more data to be loaded in each iteration of the vectorized loop with gcc

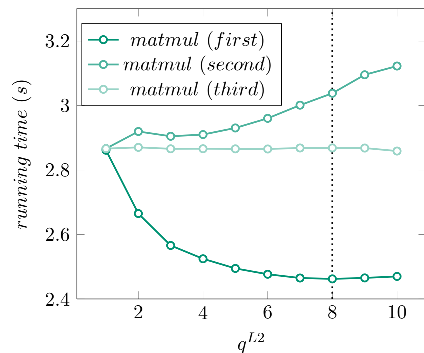


Figure 9: Second level tiling: The impact of the choice of a distinct loop and its tile size (with gcc)

for the SICA version), there is no unique tile size but an individual one for each statement block. This is why it is reasonable to measure the running times only by varying ρ and thereby scaling the tile sizes proportionally.

In Fig. 8, the best tile size turns out to be quite exactly the theoretical optimum. Another interesting observation are the local optima for $\rho = 2.0$ and $\rho = 3.0$, where the necessary data could be loaded from the cache in exactly two or three portions.

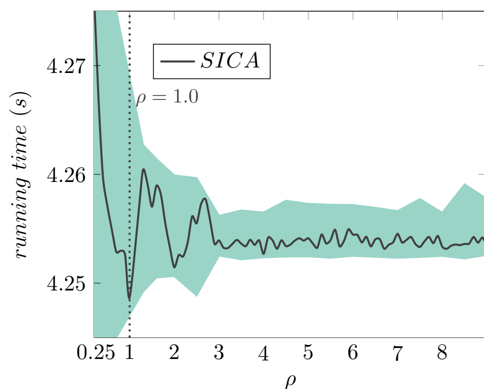


Figure 8: Impact of the tile size on the running time of the correlation matrix algorithm with gcc

4.1.2 SICA L2 tiling

For a two-level tiling, we already theoretically justified to choose the outermost loop from the nest of the vectorizable one. Now, we deliver an experimental justification of this decision using the matrix multiplication example. The resulting running times for each selection of one of the three nested loops for the second-level tiling (while tiling the first level with the calculated q^{L1}) are depicted in Fig. 9. Only when the outermost loop (first) is selected, the running time is improved. Furthermore, the calculated theoretical optimum of $q^{L2} = 8$ (the L2 cache on the test-system is 8 times larger than the L1 cache) leads to the best running times.

4.2 Performance counters

To further investigate the impact of the SICA tiling and to explain the improved running times, we measured PAPI [19] performance counters for the original source code and the two-level tilings by PluTo (default) and our extension with the asymmetric matrix sizes from Sect. 4.1. In particular, we focused on the effects concerning the L2 cache miss rate (as each L2 cache miss is preceded by an L1 cache miss and the L2 cache, being the last on-core level, is crucial for our intended pipeline), the rate of introduced SIMD instructions and the number of L1/L2-TLB misses. Concerning the cache behavior, it is important to consider miss rates instead of absolute numbers of cache misses. This is true since a successful prefetching of the accessed data may lead to an increase of the total number of cache misses and accesses at the same time. However, misses obtained like this do not significantly harm performance but impose cache hits when the operands are really needed. We additionally measured the cache hit rate but did not explicitly picture it. As could be expected, it sums up to 100% with the cache miss rate (besides minor deviation of the counters).

When using gcc, both the SICA and PluTo's default tiling largely improve the performance of the matrix multiplication code (cf. Fig. 10). While the original source code cannot be vectorized by gcc at all, the tiled versions enable the introduction of SIMD instructions. In case of the SICA tiling, in fact nearly all instructions are SIMD instructions. Tiling only two instead of all loop dimensions results in fewer cases where the control flows enters the innermost loop with 'remainders' of iterations that cannot be vectorized by gcc. This effect is in fact intensified by the choice of asymmetric matrix sizes for these experiments. The larger tile size for the vectorized loop (in comparison to PluTo) corresponds to many predictable accesses to the innermost (linearly stored) array dimension. Both tiled versions lead to a reduction of the L2 cache miss rate. This is especially true with the fitted tile sizes calculated by the SICA extension which also leads to a stronger reduction of TLB misses.

Fig. 11 shows that the internal optimizations done within icc applied to the original source code perform better than when applied to PluTo's default tiled version. This is especially true for the L2 cache miss rate. It is even marginally better for the original source than with the fitted SICA tile sizes. However, with the SICA tiling, icc is able to produce the fastest code since it can again turn nearly every instruction into a SIMD instruction. The situation concerning TLB misses is as before with the exception that the code produced by icc on the original source code leads to far less TLB misses than with gcc.

In case of the correlation matrix algorithm and gcc, the

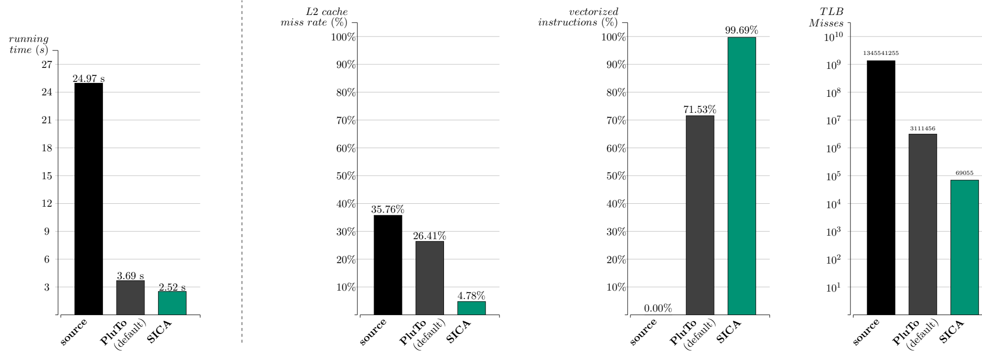


Figure 10: PAPI performance counters for the matrix multiplication code with gcc

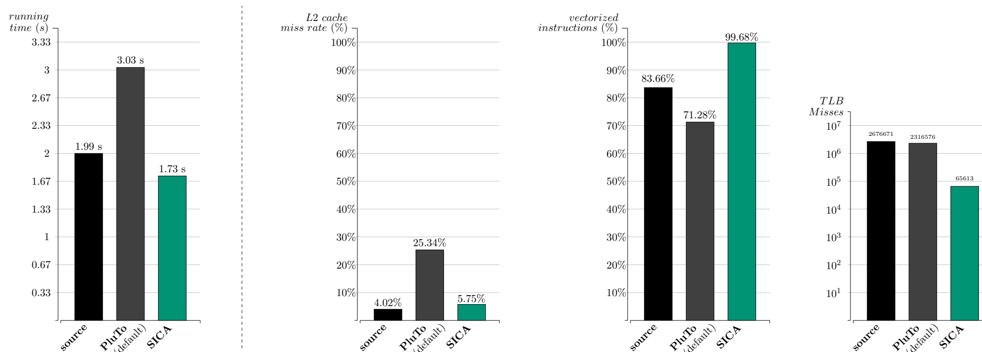


Figure 11: PAPI performance counters for the matrix multiplication code with icc

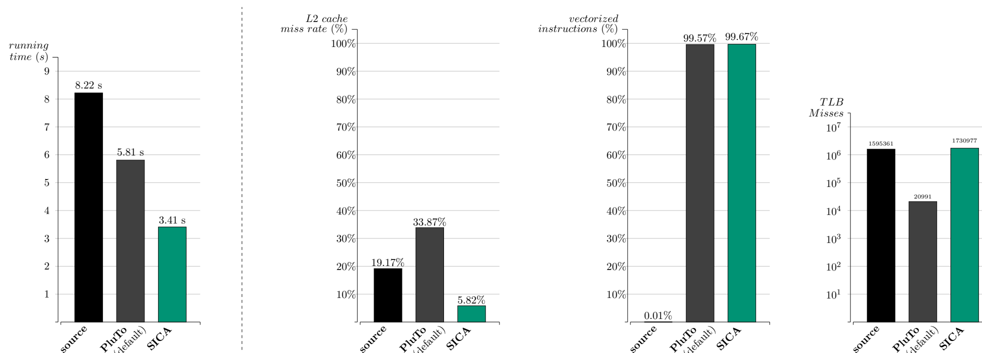


Figure 12: PAPI performance counters for the correlation matrix code with gcc

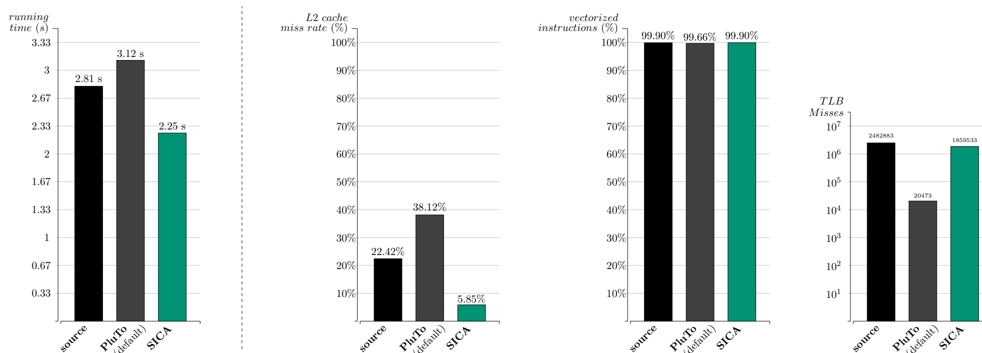


Figure 13: PAPI performance counters for the correlation matrix code with icc

source codes produced by PluTo and with the SICA tiling both improve the running time as is shown in Fig. 12. Even more, both tilings lead to a nearly perfect vectorization of the code. However, whereas the PluTo code leads to an increase of L2 cache misses compared to the original source code, the SICA tiling leads to a decrease which explains the better running time. As opposed to that and with less impact on the running time, PluTo performs by far best concerning TLB misses while the SICA version cannot even improve on the number of TLB misses that are produced with the original source code.

As with the matrix multiplication, PluTo's default tiling does not lead to a better running time compared to the original source code when compiled with icc. For the L2 cache miss rate and the TLB misses, the results are similar to the case of using gcc. However, icc is able to fully vectorize the original code and there is nearly no difference in the rate of vectorization using any of the three codes as input.

4.2.1 *Interim summary*

For the considered test cases and compared to the original source code and PluTo's default tiling, the SICA approach always produced the best running times, no matter if used as input for gcc or icc. A big advantage of the SICA tiling is that it typically enables the compilers to vectorize a larger number of instructions. Further, across all benchmarks, it leads to a small L2 cache miss rate which is the best one obtained except for the matrix multiplication with icc. Concerning TLB misses, possible improvements depend on the access pattern of the statements within the vectorized loop.

4.3 Performance benchmarks

Finally, we would like to verify the performance of the SICA two-level tiling for the two application codes across a larger range of symmetric matrix input sizes. We selected the sizes $i \cdot 1024$ for $i \in [2, 8]$. Since these are all multiples of four, they diminish the disadvantages of PluTo's default tiling in the benchmarks before, i.e., there will be no non-vectorizable 'remainders' of loop iterations anymore. Similarly, gcc will always be able to apply its auto-vectorization already to the original source code.

Fig. 14 and 16 show the corresponding running times for gcc and Fig. 15 and 17 those for icc. The output produced by the SICA tiling appears to support both compilers to produce faster code compared to the original source and PluTo's default tiling. For completeness, Table 3 depicts the average speedups obtained for the tested input sizes.

gcc	matrix multiplication	correlation matrix
PluTo (def.)	11.14	4.47
SICA	20.05	8.89
icc	matrix multiplication	correlation matrix
PluTo (def.)	1.01	3.73
SICA	1.31	7.54

Table 3: Average speedups (coarse grain)

5. CONCLUSION, ONGOING WORK AND OUTLOOK

We presented an adaptive hardware-aware tiling approach that has been implemented and evaluated as an extension to PluTo. In contrast to PluTo, it does not perform a tiling of all loops in a nest, but specific tilings of distinct loops that are beneficial for an effective vectorization. It is adaptive in that it performs an automatic analysis of the amount of data accessed by a loop's statements and is able to derive information about the underlying hardware such as cache sizes and SIMD register widths. These parameters are used to derive dynamic tile sizes that ideally lead to a software pipeline of blocks to be processed by the SIMD units and to be well prefetched through the cache hierarchy.

We verified and evaluated our approach experimentally on two source codes from PluTo's example suite, namely a standard matrix multiplication and a correlation matrix algorithm. As our results show, the proposed tiling strategy

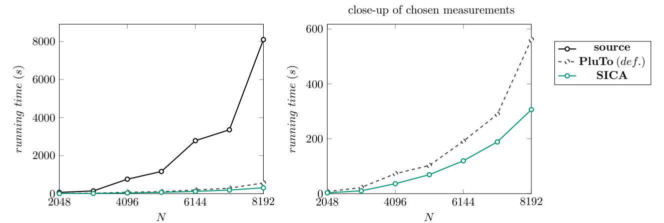


Figure 14: Running times for the two-level tiled matrix multiplication (coarse grain) with gcc

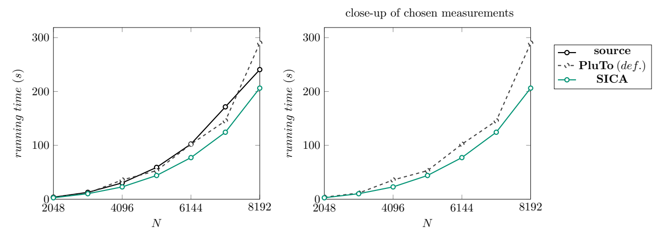


Figure 15: Running times for the two-level tiled matrix multiplication (coarse grain) with icc

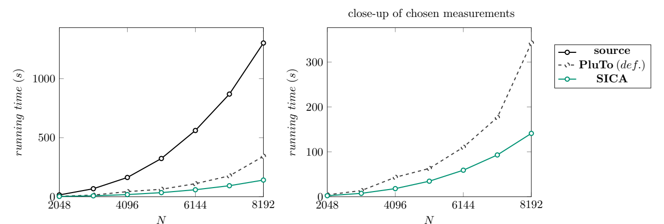


Figure 16: Running times for the two-level tiled correlation matrix (coarse grain) with gcc

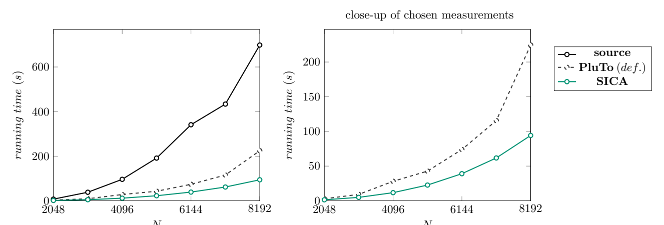


Figure 17: Running times for the two-level tiled correlation matrix (coarse grain) with icc

leads to better running times in comparison to PluTo's default tiling and the original source code when its output is compiled with gcc 4.6 or icc 13.0. An analysis with performance counters brought to light that these results can be mainly explained by an increase of issued SIMD instructions and a strong reduction of the L2 cache miss rate.

In further studies (see [10]), we examined the behavior of our extension on further source codes. They include applications that contain very deep loop nests and where a static all-dimension tiling therefore leads to blocks that are by far too large for today's usual cache sizes. This may considerably harm performance, whereas our extension can handle these cases by its dynamic analysis and the restriction to tile at most two loops.

However, we do not consider a tiling of at most two loops as a globally superior strategy. To the contrary, if the dependence structure of a statement block refers to multiple loop dimensions, a corresponding specific multi-dimensional tiling could be superior in terms of spatial locality and TLB performance. We plan to consider this in future work. Similarly, we would like to manipulate the loop transformations towards an automatic optimization for one-strided memory accesses. This could be potentially achieved by (a) separating the statements of a block according to their access patterns, (b) transforming them one by one targeting one-strided accesses and (c) vectorizing the resulting loop nests. We also plan to experiment with a L3 cache tiling as well as with the combination of our developments with automatic shared-memory parallelization in order to exploit the full concurrency potential of modern multicore processors.

Currently, our developments are ported to the PoCC [16] framework (that includes PluTo) in order to integrate our extensions into Polly and thereby into the LLVM infrastructure [14].

6. ACKNOWLEDGMENTS

Thanks to Uday Bondhugula for the development of the PluTo framework.

7. REFERENCES

[1] J. Abella, A. Gonzalez, J. Llosa, and X. Vera. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *Proc. Int. Parallel Processing Workshop*, pages 568 – 577, 2002.

[2] U. K. Banerjee. *Loop Parallelization.* Loop transformations for restructuring compilers. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

[3] D. Bernstein, D. Cohen, and A. Freund. Compiler techniques for data prefetching on the PowerPC. In *Proc. IFIP WG10.3 working Conf. on Parallel Architectures and Compilation Techniques*, PACT '95, pages 19–26, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[4] U. Bondhugula. PluTo: An automatic parallelizer and locality optimizer for multicores.

[5] U. Bondhugula. *Effective Automatic parallelization and locality optimization using the polyhedral model.* PhD thesis, Columbus, OH, USA, 2008.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[7] A. Cohen, M. Sigler, D. Parello, S. Girbal, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05)*, pages 151–160, 2005.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program. volume 21*, pages 313–348, 1992.

[10] D. Feld. Effiziente Vektorisierung durch semi-automatisierte Code-Optimierung im Polyedermodell, *available at http://publica.fraunhofer .de/documents/N-194546.html*, 2011.

[11] S. Ghosh, M. Martonosi, and S. Malik. Automated cache optimizations using CME driven diagnosis. In *Proc. 14th Int. Conf. on Supercomputing*, ICS '00, pages 316–326, New York, NY, USA, 2000. ACM.

[12] 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. *Int. J. Parallel Program. volume 34*, pages 261–317, 2006.

[13] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004.

[14] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - polyhedral optimization in LLVM. In *First Int. Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[15] Intel. Intel processor identification and the CPUID instruction. Technical report, Intel Corporation, 2011.

[16] L.-N. Pouchet. PoCC: the Polyhedral Compiler Collection.

[17] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software*, ISPASS '00, pages 146–153, Washington, DC, USA, 2000. IEEE Computer Society.

[18] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Proc. 21st Int. Conf. on Compiler Construction*, CC'12, pages 101–121, Berlin, Heidelberg, 2012. Springer.

[19] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. 2009.

[20] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proc. 18th Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[21] J. Xue. *Loop tiling for parallelism.* Kluwer Int. Series in Engineering and Computer Science. Kluwer Academic, 2000.

APPENDIX

A. STATIC CONTROL PARTS (SCoPs) OF THE CONSIDERED EXAMPLE CODES

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  for(k=0; k<K; k++)
   C[i][j] = C[i][j] + alpha*A[i][k]*B[k][j];
```

Figure 18: Original standard matrix multiplication

```
for(t1=0;t1<=floor(M-1,8);t1++) {
 for(t2=0;t2<=floor(N-1,3684);t2++) {
  for(t3=0;t3<=K-1;t3++) {
   for(t4=8*t1;t4<=min(M-1,8*t1+7);t4++) {
    {
     lbv=3684*t2; ubv=min(N-1,3684*t2+3683);
     #pragma ivdep
     #pragma vector always
     for(t9=lbv; t9<=ubv; t9++) {
      C[t4][t9]=C[t4][t9]+alpha*A[t4][t3]*B[t3][t9];;
     }
    }
}}}}
```

Figure 19: Matrix multiplication (SICA applied with $\rho = 0.9$)

```
C[i][j]=C[i][j]+A[i][k]*B[k][j];
C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j];
C[i][j]=C[i][j]+A[i][k]*B[k][j]+D[i][k]*E[k][j]
        +F[i][k]*G[k][j];
```

Figure 20: The different statements in matmul1, matmul2 and matmul3

```
/* Center and reduce the column vectors. */
for(i = 1; i <= N; i++)
 for(j = 1; j <= M; j++) {
  data2[i][j] -= mean[j];
  data2[i][j] /= sqrt(N) * stddev[j];
 }

/* Calculate the M * M correlation matrix. */
for(j1 = 1; j1 <= M-1; j1++) {
 symmat[j1][j1] = 1.0;
 for (j2 = j1+1; j2 <= M; j2++) {
  symmat[j1][j2] = 0.0;
  for (i = 1; i <= N; i++)
   symmat[j1][j2]+=(data2[i][j1]*data2[i][j2]);
  symmat[j2][j1] = symmat[j1][j2];
 }
}
```

Figure 21: Original correlation matrix SCoP

```
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(t2-920,921);t3<=floor(M,7368);t3++) {
  for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                            7368*t3+7366);t5++) {
   {
    lbv=max(7368*t3,t5+1); ubv=min(M,7368*t3+7367);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     symmat[t5][t10]=0.0;;
    }
   }
}}}
for(t2=1;t2<=M-1;t2++) {
 symmat[t2][t2]=1.0;;
}
for(t2=0;t2<=floor(N,8);t2++) {
 for(t3=0;t3<=floor(M,2456);t3++) {
  for(t5=max(1,8*t2);t5<=min(N,8*t2+7);t5++) {
   {
    lbv=max(1,2456*t3); ubv=min(M,2456*t3+2455);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     data[t5][t10]-=mean[t10];;
     data[t5][t10]/=sqrt(N)*stddev[t10];;
    }
   }
}}}
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
  for(t4=1;t4<=N;t4++) {
   for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                             3684*t3+3682);t5++) {
    {
     lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
     #pragma ivdep
     #pragma vector always
     for(t10=lbv; t10<=ubv; t10++) {
      symmat[t5][t10]+=(data[t4][t5]*data[t4][t10]);;
     }
    }
}}}}
for(t2=0;t2<=floor(M-1,8);t2++) {
 for(t3=ceil(2*t2-920,921);t3<=floor(M,3684);t3++) {
  for(t5=max(1,8*t2);t5<=min(min(M-1,8*t2+7),
                            3684*t3+3682);t5++) {
   {
    lbv=max(3684*t3,t5+1); ubv=min(M,3684*t3+3683);
    #pragma ivdep
    #pragma vector always
    for(t10=lbv; t10<=ubv; t10++) {
     symmat[t10][t5]=symmat[t5][t10];;
    }
   }
}}}
```

Figure 22: Correlation matrix SCoP (SICA applied with $\rho = 0.9$)

```
for(i=0; i<M; i++)
 for(j=0; j<N; j++)
  S(i,j);
```

```
for(ii=0; ii≤floor(M-1,32); ii++)
 for(jj=0; jj≤floor(N-1,32); jj++)
  for(i=32*ii; i≤min(M-1,32*ii+31); i++)
   for(j=32*jj; j≤min(N-1,32*jj+31); j++)
    S(i,j);
```

```
for(iii=0;iii≤floor(M-1,256);iii++)
 for(jjj=0;jjj≤floor(N-1,256);jjj++)
  for(ii=8*iii;ii≤min(floor(M-1,32),8*iii+7);ii++)
   for(jj=8*jjj;jj≤min(floor(N-1,32),8*jjj+7);jj++)
    for(i=32*ii;i≤min(M-1,32*ii+31);i++)
     for(j=32*jj;j≤min(K-1,32*jj+31);j++)
      S(i,j);
```

Figure 23: Perfectly nested loop (left) with one level (middle) and two level (right) traditional tiling by PluTo

```
for(i=0; i<M; i++)
 for(jj=0; jj≤floor(N-1,q^L1); jj++)
  for(j=q^L1*jj; j≤min(N-1,q^L1*jj+(q^L1 - 1)); j++)
   S(i,j);
```

```
for(ii=0; ii≤floor(M-1,q^L2); ii++)
 for(jj=0; jj≤floor(N-1,q^L1); jj++)
  for(i=q^L2*ii; i≤min(M-1,q^L2*ii+(q^L2 - 1)); i++)
   for(j=q^L1*jj; j≤min(N-1,q^L1*jj+(q^L1 - 1)); j++)
    S(i,j);
```

Figure 24: Loop from figure 23 with one level (left) and two level (right) SICA tiling for vectorizable j loop

SPolly: Speculative Optimizations in the Polyhedral Model

Johannes Doerfert
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
doerfert@st.cs.uni-saarland.de

Clemens Hammacher
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
hammacher@cs.uni-saarland.de

Kevin Streit
Saarbrücken Graduate School of Computer Science
Saarland University
Saarbrücken, Germany
streit@cs.uni-saarland.de

Sebastian Hack
Computer Science Department
Saarland University
Saarbrücken, Germany
hack@cs.uni-saarland.de

ABSTRACT

The polyhedral model is only applicable to code regions that form static control parts (SCoPs) or slight extensions thereof. To apply polyhedral techniques to a piece of code, the compiler usually checks, by static analysis, whether all SCoP conditions are fulfilled. However, in many codes, the compiler fails to verify that this is the case. In this paper we investigate the rejection causes as reported by *Polly*, the polyhedral optimizer of a state-of-the-art compiler. We show that many rejections follow from the conservative overapproximation of the employed static analyses. In *SPolly*, a speculative extension of Polly, we employ the knowledge of runtime features to supersede this overapproximation. All speculatively generated variants form valid SCoPs and are optimizable by the facilities of Polly. Our evaluation shows that SPolly is able to effectively widen the applicability of polyhedral optimization. On the SPEC 2000 suite, the number of optimizable code regions is increased by 131 percent. In 10 out of the 31 benchmarks of the PolyBench suite, SPolly achieves speedups of up to 11-fold as compared to plain Polly.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization, Retargetable compilers, Run-time environments*

General Terms

Performance

Keywords

Adaptive optimization, polyhedral loop optimization, just-in-time compilation, speculative execution

1. INTRODUCTION

The polyhedral model uses polyhedra to abstract essential information of a static control program (SCoP, a restricted form of nested DO loops). By abstracting loop nests as polyhedra, many popular loop optimizations can be elegantly formulated in terms of linear transformations. However, a piece of code has to meet the SCoP criteria [5] (or slight extensions thereof) to be tractable by polyhedral techniques. Those conditions are usually met by many numerical kernels from linear algebra, image processing, signal processing, etc. However, in more general benchmarks such as the SPEC 2000 benchmark suite, the SCoP conditions are often violated. One simple reason is, for example, that the compiler could not prove that all arrays used in the loop nest do not overlap or alias. Further reasons we encountered are: The implementation of functions called in the loop nest was not available because they are extern to the translation unit in question. However, at runtime, the functions turned out to be pure. Also, index expressions and/or loop bounds could not be classified as affine. Often, some parameter, which does not change in the loop nest, makes an expression non-affine. Hence, specializing the loop nest to the concrete value at runtime makes polyhedral techniques applicable.

In this paper, we want to advance the applicability of polyhedral optimizations by incorporating runtime information. For example, parameters that lead to non-affine expressions can be replaced by their concrete value. This way, using a just-in-time compiler, the program can prepare specially optimized versions for different parameter sets. In summary, this paper makes the following contributions:

- We investigate the applicability of Polly [8], a polyhedral framework for LLVM [10], on the SPEC 2000 benchmark suite and classify the causes that prevent the application of Polly to a candidate code region[1]. We conclude that up to 2.4 times more code regions in SPEC 2000 could be considered by Polly if runtime information is available.

- Based on those insights, we present *SPolly,* a prototypical extension of Polly to integrate runtime information. At the moment, SPolly handles two cases: First, it adds code to the program that checks whether all

[1]As code regions we consider all non-trivial regions according to Grosser et al. [8], i.e. single entry, single exit regions containing at least one loop.

referenced arrays of a loop nest do not overlap. Second, it uses profiling to record the values of parameters that lead to non-affine index expressions and/or loop bounds and generates specialized variants that can be selected during runtime. We demonstrate the benefit of these techniques on the PolyBench benchmark suite.

The rest of this paper is organized as follows: In Section 2 we present related work. Section 3 provides an evaluation of the most frequent rejection causes of polyhedral transformations; Section 4 explains how SPolly alleviates some of those causes. Finally Section 5 evaluates SPolly and Section 6 concludes.

2. RELATED WORK

The polyhedral literature is certainly too extensive to be discussed in full detail here. Furthermore, this paper is concerned with increasing the applicability of the polyhedral model by speculation and runtime analysis and not with inventing new polyhedral techniques. Therefore, we concentrate on the recent work which is directly related to ours.

In his thesis [7], Grosser describes a speedup of up to 8x for a matrix multiplication benchmark, achieved by his polyhedral optimizer Polly [8]. He also produced similar results for other benchmarks of the PolyBench [11] benchmark suite. Other publications [4, 1, 12] show similar results on PolyBench. PolyBench is certainly well suited to compare polyhedral techniques. However, it does not allow for assessing their width of applicability as we do in this paper.

Baghdadi et al. [1] reveal a huge potential for speculative loop optimizations as an extension to the formerly described techniques. They state that aggressive loop nest optimizations (including parallel execution) are profitable and possible, even though overestimated data and flow dependences would statically prevent them. Their manually crafted tests also show the impact of different kinds of conflict management. Overhead, and therefore speedup, differs from loop to loop, as the applicability of such conflict management systems does, but a trend was observable. The best conflict management system has to be determined per loop and per input, but all can provide speedup, even if they are not that well suited for the situation.

Bastoul et al. [2, 3] and Girbal et al. [6] also perform an evaluation of the applicability of the polyhedral model in several benchmarks. In contrast to this work however, he is more concerned with the structure and size of the SCoPs and not with the reasons of why other code regions could not be classified as SCoPs.

Jimborean et al. [9] employ the polyhedral model in the context of speculative parallelization. A static parallelization pattern is generated assuming linearity of loop bounds and memory accesses. Additionally, polyhedral transformations are proposed based on static dependence analysis. At runtime, the linearity of memory accesses and the applicability of the proposed candidates is assessed by collecting profiling information. In case of success, the statically generated pattern is instantiated with the gathered information and corresponding code is generated. As the performed transformations are speculative and not validated before execution of the loop, a speculation mechanism guards the execution. In case of detected conflicts, rollbacks are performed and execution switches to the safe, sequential version. The approach has some drawbacks: First of all, it relies on programmer annotations to identify parallelization candidates. Second, polyhedral optimization is limited to transformations that do not change the loop structure or reorder statements. The use of synthetic benchmarks in their evaluation does not allow judging in how far the approach can profitably extend the applicability of polyhedral optimization on widely used benchmarks such as the PolyBench suite.

3. MOTIVATION

In order to analyze which of the SCoP restrictions limit the applicability of the polyhedral model most, we conducted an evaluation on nine programs from the SPEC 2000 benchmark suite[2]. To this end we instrumented Polly to output all rejection causes that prevent a candidate code region from being considered a polyhedron[3]. In case of multiple rejection causes for one region, we record all of them.

Figure 1 shows the results of this evaluation. For each rejection cause in Polly, we report three numbers:

- The number of regions where the violation of condition i is a cause for not considering (Column A).
- The number of regions where the violation of condition i is the *only* rejection cause (Column B).
- The number of regions gained when ignoring all conditions 0 to i (Column C).

Over the nine tested programs, 1862 candidate regions are tested, out of which 1587 are rejected by Polly. The remaining 275 regions (14.8%) are considered legal SCoPs.

i	Rejection cause	A	B	C
0	Non-affine expressions	1230	84	84
1	Aliasing	1093	207	510
2	Non-affine loop bounds	840	6	660
3	Function call	532	72	928
4	Non-canonical indvars	384	0	1174
5	Complex CFG	253	31	1387
6	Unsigned comparison	199	0	1586
7	Others	1	0	1587

Figure 1: SCoP rejection causes on the SPEC 2000 benchmark suite

The rejection causes are ordered by the number of regions that they affect (column A). In the following, we will examine the conditions constituting the rejection causes in further detail and discuss if and how SPolly can alleviate their impact.

Non-affine expressions As explained in more detail in Section 4.2, all expressions used for memory accesses or predicates of conditional branches have to be affine expressions with respect to parameters and induction variables. If this is not the case, the code cannot be represented as polyhedron, thus preventing corresponding optimizations. This happens for example

[2]As SPolly's runtime environment is based on the Sambamba framework [13] we selected the programs that were contained in the Sambamba test suite: ammp, art, bzip2, crafty, equake, gzip, mcf, mesa, and twolf

[3]All programs were compiled with
`-sink -indvars -mem2reg -polly-prepare`

when a programmer chose to represent a 2-dimensional array by a 1-dimensional, flattened, one, translating the index (i, j) to $i * N + j$. In case i and j are iteration variables, and N is a parameter, this expression is not affine but quadratic. If however N can be detected as quasi-constant via profiling, it can speculatively be replaced by that constant to circumvent the non-affinity.

Aliasing Possible aliasing causes a region to be rejected whenever the base address of two memory accesses cannot be proven to be disjoint. In particular, this is the case when pointers originate in parameter values instead of global variables or stack-allocated memory, since the default alias analysis used by Polly only works intra-procedural. In most cases however, two arrays passed as parameters are disjoint; speculatively assuming non-aliasing may thus be profitable.

Non-affine loop bounds This constraint requires all loop bounds to be affine expressions with respect to the parameters and surrounding iteration variables. In our experiments we frequently observed that although bounds have not been affine at compile-time, they often turn out to remain constant, and thus affine, during all executions. This is for example the case if a loop iterates N^2 times, where N is a parameter. It is obvious that in these cases a specialized variant can be generated where the loop bounds are considered constant. This not only makes the loop representable as a polyhedron, but also enables better optimizations in the *isl*.

Function call Another major reason for rejecting a code region is contained function calls. In general, computing memory dependences through calls is a hard task; for external functions or indirect calls it is often impossible. Additionally, external functions may not terminate or have observable effects like text output or aborting the application. Polly rejects each region containing at least one call to a function that cannot be proven to be pure. In practice though, parallelization prohibiting side-effects—for example exceptions, or error reporting—might manifest only infrequently. In such case, speculatively ignoring the calls and specializing a region accordingly can make it amenable to polyhedral optimizations. To preserve the original semantics, the specialized code needs protection by a runtime speculation mechanism, e.g. in the form of transactional memory.

Non-canonical induction variables This constraint requires the induction variables to be in a canonical form, starting at zero and being incremented by one in each iteration. If LLVM's and Polly's preparation passes are unable to canonicalize all such variables, the code region is discarded.

Complex CFG This error is reported if complex terminators like switch instructions or are found, or the control flow has a "complex form" not representable via *while* and *if* constructs only.

Unsigned comparison During polyhedral optimizations, Polly may have to modify comparison operators or their operands, for example to alter the iteration space of a loop. Special care has to be taken to handle possible overflows correctly. To this end, this is only implemented for signed operations. Consequently, Polly rejects all code regions containing unsigned comparison.

Others This is a collection of minor technical limitations, for example rarely used LLVM instructions, which are not handled properly yet. The only occurrence in our tests is a cast from an integer to a pointer in the mcf program.

SPolly concentrates on the first three rejection causes. In our benchmarks, they make up for 42% of all SCoP rejections. Thus, assuming we could speculatively eliminate them in all cases, we could expect 660 new SCoPs to be detected, which is exactly 2.4 times the original number of SCoPs. The fourth cause, function calls, could be solved using runtime techniques as described, but in our experiments the introduced overhead of transactional execution did not pay off. Therefore we skipped that for now, and consider it future work to improve the performance of the transactional memory system. All other reported causes are either conceptual obstacles (induction variables, complex CFGs) that SPolly cannot remove or technicalities that will disappear when LLVM and Polly will mature further.

4. SPOLLY

SPolly extends the applicability of the loop nest optimizer and parallelizer Polly by deploying runtime information and speculatively specializing functions. It targets restrictions on loop nests arising as a consequence of overestimations in static analyses. By inferring common values for parameters, and providing additional conditions for memory accesses, it makes polyhedral optimizations applicable to more code locations, and allows for better optimization and code generation. Those specialized versions will coexist with the original sequential version and will be executed whenever the actual runtime values permit this. This section describes in detail how SPolly achieves this.

4.1 Possible Aliasing

Possible aliasing is the main rejection cause of Polly on the considered benchmarks of the SPEC 2000 suite and particularly well suited for speculation. An example for a loop with possibly aliasing accesses is given in Figure 2a. It shows two arrays accessed via pointers `A` and `B`. In case they point to addresses less than `N * sizeof(int)` bytes apart, parallel execution and other loop transformations possibly alter the semantics. Most alias analyses are conservative and assume aliasing if `A` and `B` could potentially point to the same allocated memory block. Using the latter definition, Polly would not optimize the presented loop without further information on `A` and `B`.

Polly offers the possibility to override conservative assumptions concerning possible aliasing. This can be done by either ignoring all aliasing, or by annotating individual code regions. Both these approaches require manual intervention and profound knowledge of the application to optimize. In contrast, SPolly is able to deal with possible aliases without any programmer-provided information. It does so by introducing alias checks preceding the subject loop nests to

```
void a(int *A, int *B) {
  int i;
  for (i=1; i<N; i++)
    A[i] = 3 * B[i];
}
```

(a) Loop with possible aliasing pointers

```
void b(int *A, int N) {
  int i, j;
  for (i = 1; i<N; i++)
    for (j = 0; j<N; j++)
      A[j*N+i] += A[j*N+i-1];
}
```

(b) Loop nest using non-affine expressions

Figure 2: Example loops rejected by Polly for different speculatively resolvable reasons

ensure the absence of conflicts between accesses to loop invariant base pointers. For the shown loop, those introduced checks are conclusive as the accessed range, relative to the base address, is known before entering the loop, thus non-aliasing of all accesses to the arrays can be checked a priori. This approach allows to optimize loops even if the used base pointers might point into the same allocated block, for example different parts of the same array. If the checks fail, the original, unmodified version is executed, otherwise the optimized version is chosen. An example that would not benefit from this approach because of possibly aliasing loop variant pointers is dynamic multidimensional arrays (arrays of pointers to arrays). Iterating over those will change the base pointer of the inner dimension for each execution of the outer loop, and checking that none of them alias is too expensive to be performed prior to execution of the loop.

4.2 Non-affine Parameters

Consider Figure 2b. C does not provide support for n-dimensional arrays of which the dimensions are not known statically. Hence, a common pattern is to implement n-dimensional arrays using a 1D array and performing the index arithmetic "manually". This pattern creates non-affine array subscripts on the 1D array. Using static analysis, one could infer that the 1D access is actually a 2D access; at least in this example. However, such an analysis is currently not implemented in Polly. Additionally, it is imaginable that the code is more complicated making the static analysis unsuccessful.

Thus, SPolly utilizes runtime information gained with a profiling version of the loop nest to identify reoccurring parameter values. For those values specialized loop versions are created with constant values plugged in for the problematic parameters. Dispatching code is inserted before the corresponding loop to decide at runtime whether a specialized version exists for the actual parameter values. If none is found, the original, less optimized version is used. In our example, by specialization the multiplications would become affine and therefore representable in the polyhedral model, and thus amenable for all polyhedral optimizations implemented in Polly.

Finally, we observed (see the next section) that replacing parameters with constants often leads to superior code because it enables more aggressive optimizations. Thus, even if a static analysis would analyze pseudo-1D array accesses and restate them into multidimensional accesses, executing specialized versions often leads to better performance.

5. EVALUATION

In order to evaluate the success of incorporating runtime information to speculatively optimize code regions, we will

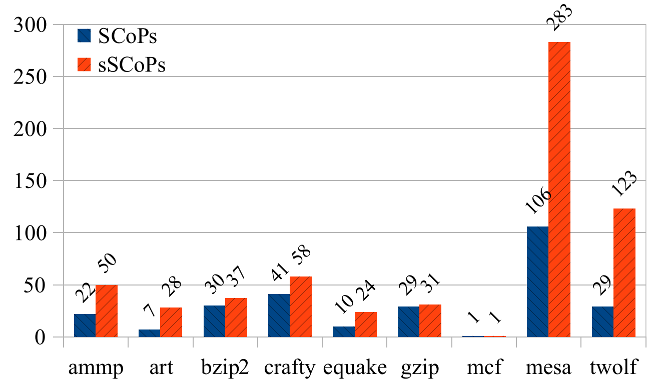


Figure 3: Quantitative analysis of the applicability of SPolly compared to Polly. Overall, SPolly provides 635 sSCoPs while Polly find 275 SCoPs.

investigate two hypotheses:

1. SPolly is able to handle substantially more regions than Polly. In other words, the number of speculative SCoPs (*sSCoPs*) is substantially larger than the number of SCoPs.

2. The extended applicability of the polyhedral model results in improved runtime performance.

Hypothesis 1 is motivated from Section 3, where we showed that the constraints that SPolly tackles are the main reasons for rejecting a SCoP. Hypothesis 2 follows from the primary goal of polyhedral optimizations, which is reducing the program runtime.

5.1 Extended Applicability

In order to evaluate the successful elimination of SCoP rejection causes, we executed SPolly on the SPEC 2000 tests that we already used in Section 3. We then compared the number of sSCoPs detected by SPolly against the number of SCoPs detected by Polly. A graphical representation of this comparison is shown in Figure 3.

You can see that for all test cases except of "mcf", our speculative extensions provided an increased number of code regions amenable to polyhedral optimizations. The overall number of sSCoPs is 635, as compared to the 275 detected SCoPs by Polly. This is an **increase of 131 percent** (360 additional SCoPs).

Compared to the expectations in Section 3, where we identified an upper bound of 660 additional SCoPs, we reached a **success rate of 55 percent**. The remaining sSCoP candidates contain code where our heuristics decided that speculation is not profitable or impossible. This is the case if,

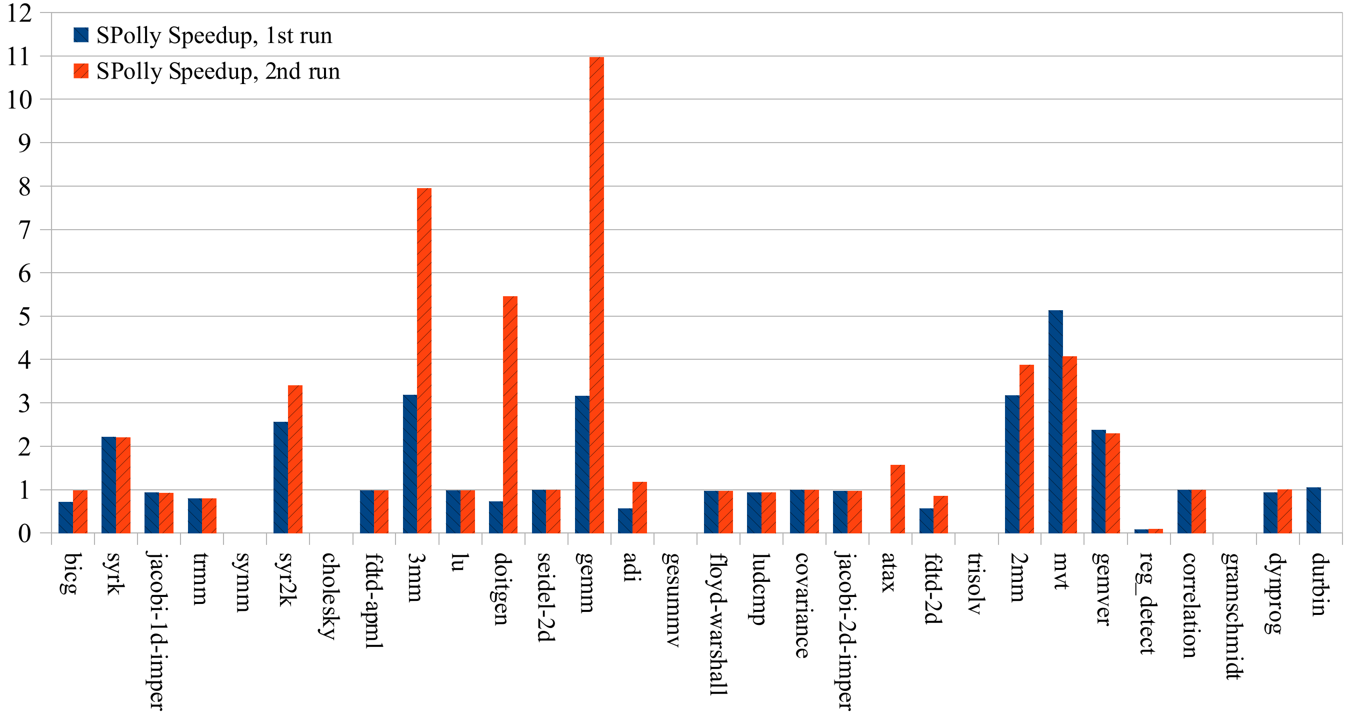


Figure 4: Speedups in execution time of SPolly-enabled programs on the PolyBench suite, normalized to the standard Clang-compiled version with all optimizations enabled

for example, an external function call is executed unconditionally, or indirect pointer loads are detected where aliasing cannot be checked a priori.

Overall, we conclude that SPolly in fact widens the applicability of polyhedral optimizations substantially.

5.2 Performance

After confirming that the number of SCoPs detected in SPEC 2000 is indeed increased, we investigated to which extent this actually improves the runtime of the programs. We observed that even though the additional SCoPs were valid and some of them were executed, the most interesting code regions were not optimized due to additional obstacles we could not eliminate. These are mainly external function calls and indirect memory accesses in the hot loops. So on these tests, we are unable to achieve significant speedups.

Thus we resort to the PolyBench suite [11], which has also been used by Grosser et al. [8, 7] before to evaluate the performance of Polly. We are using the current version 3.2, and the provided "large" data sets.

In order to compare the speedup in program execution achievable by applying polyhedral optimizations on statically validated SCoPs alone against that of speculatively created and optimized polyhedra, we run all tests

- on the standard toolchain of Clang and LLVM with all optimizations enabled
- using Polly after applying the standard preparation passes (see Section 3)
- using SPolly without any prior knowledge about the program
- using SPolly a second time, such that profiling information from the previous run is already available

All tests are conducted on an eight-core Intel Xeon machine running at 2.93 GHz with 24 GB of main memory.

Surprisingly, we are unable to reproduce earlier results of Polly on the PolyBench suite. Investigating the reasons why Polly rejects the SCoPs for the compute-intensive kernels of all of the benchmarks revealed, that in all cases this is due to aliasing problems. Since PolyBench 3.0, released in October 2011, the tests are using heap-allocated arrays, passed to the kernel via pointer arguments plus array sizes. Polly is unable to prove non-aliasing of those pointers, thus rejecting all possibly affected SCoP candidates. This is why we did not find any program for which Polly was able to improve the performance over a standard Clang-compiled program.

Figure 4 shows the speedups of the SPolly-enabled program runs, normalized to the runtime of the corresponding Clang-compiled program. Missing values in this bar chart indicate failures during optimization, which were mainly originating from the CLooG code generator, but also from creating the polytope in Polly. Nevertheless, for most of the programs we are able to report runtime results. It is not surprising that there are many programs where SPolly is not able to bring the kernel to a form amenable for polyhedral optimizations. Thus for these programs no speedup is achieved.

However, there are also different programs where SPolly indeed provides enormous speedups. For the first run, SPolly can make no use of any profiling data, so only those speculative transformations can be applied for which conclusive runtime checks can be synthesized. For these runs, the highest achieved speedup is 5.1-fold on the mvt program. In

the second run however, specialized versions can be created based on the profiles gathered in previous runs. This allows to replace loop bounds and parametric values by constant expressions, which not only makes those code regions representable as polyhedra, but also helps other transformations and the CLooG backend to create better code, e.g. by choosing the best tile size for loop tiling. Thus, the speedups achieved in the second runs are significantly higher in almost all cases, ranging to a maximum of 11-fold for the "gemm" program. In two cases (mvt and gemver), the speedup in the second run is slightly lower than in the first run. This is not due to overhead we introduce for checking whether a specialized code version can be chosen (this overhead was negligible for all our tests), but because the specialized variant actually executed slower than the original one. The problem of anticipating whether a code transformation, especially parallelization, will pay of at runtime is a well-known problem for which no general solution exists. Apart from that, it's not in the scope of this paper.

6. CONCLUSION

In this paper, we analyzed the most prominent causes that prohibit polyhedral optimizations for individual code regions. We observed that often promising regions are rejected by the polyhedral framework Polly because of over-approximation in the static analyses, and because of missing parameter values. Driven by this observation, we came up with runtime checks to dynamically switch to specialized variants of a code region, where the obstructive feature is removed. This approach is implemented in SPolly, a speculative extension to Polly. It enables the application of polyhedral optimizations to many more code locations, providing better runtime performance in those cases where the specialized variant could be used.

7. REFERENCES

[1] R. Baghdadi, A. Cohen, C. Bastoul, L.-N. Pouchet, and L. Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'10)*, June 2010.

[2] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, 2004.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 23–30. Springer-Verlag, Oct. 2003.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, 2008.

[5] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] 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. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[7] T. Grosser. *Enabling Polyhedral Optimizations in LLVM*. Diploma thesis, University of Passau, Apr. 2011.

[8] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - Polyhedral Optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Apr. 2011.

[9] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. VMAD: an Advanced Dynamic Program Analysis & Instrumentation Framework. In *CC - 21st International Conference on Compiler Construction*, pages 220–237, Mar. 2012.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[11] L.-N. Pouchet. Polybench, the Polyhedral Benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`, 2010.

[12] B. Pradelle, A. Ketterlin, and P. Clauss. Polyhedral parallelization of binary code. *ACM Transactions on Architexture and Code Optimization*, 8(4):39:1–39:21, Jan. 2012.

[13] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: A runtime system for online adaptive parallelization. In *Proc. 21st International Conference on Compiler Construction (CC)*, pages 240–243, Mar. 2012.

APPENDIX

	clang	clang -O2	clang -O3	SPolly 1st run	SPolly 2nd run	Speedup $1st$ run	Speedup $2nd$ run
bicg	0.66	0.22	0.22	0.32	0.23	0.72	0.99
syrk	49.03	12.16	12.16	5.48	5.50	2.21	2.21
jacobi-1d-imper	0.95	0.23	0.23	0.25	0.25	0.93	0.93
trmm	24.87	6.44	6.44	8.08	8.09	0.80	0.80
symm	129.49	119.75	119.75				
syr2k	86.57	25.76	25.76	10.03	7.56	2.57	3.40
cholesky	6.71	1.68	1.68				
fdtd-apml	12.42	10.69	10.69	10.84	10.86	0.98	0.98
3mm	294.77	233.01	233.01	72.96	29.30	3.19	7.95
lu	19.18	4.53	4.53	4.58	4.59	0.99	0.99
doitgen	45.07	10.95	10.95	14.95	2.00	0.73	5.46
seidel-2d	1.30	1.12	1.12	1.12	1.12	1.00	1.00
gemm	107.93	77.48	77.48	24.49	7.05	3.16	10.98
adi	12.56	9.31	9.31	16.10	7.88	0.57	1.18
gesummv	0.64	0.24	0.24				
floyd-warshall	72.31	13.70	13.70	14.05	14.05	0.98	0.98
ludcmp	31.62	20.94	20.94	22.31	22.26	0.94	0.94
covariance	71.80	64.70	64.70	64.78	64.88	1.00	1.00
jacobi-2d-imper	1.23	0.48	0.48	0.49	0.49	0.97	0.97
atax	0.81	0.19	0.19		0.12		1.57
fdtd-2d	5.26	2.07	2.07	3.63	2.43	0.57	0.85
trisolv	0.20	0.05	0.05				
2mm	206.63	155.09	155.09	48.75	39.92	3.18	3.88
mvt	1.69	1.18	1.18	0.23	0.29	5.13	4.06
gemver	2.33	1.30	1.30	0.55	0.57	2.36	2.28
reg_detect	0.43	0.06	0.06	0.80	0.72	0.08	0.09
correlation	71.81	64.73	64.73	64.80	64.75	1.00	1.00
gramschmidt	159.34	164.49	164.49				
dynprog	167.63	65.87	65.87	69.83	65.07	0.94	1.01
durbin	2.23	2.07	2.07	1.96		1.06	

Figure 5: Runtime results on the PolyBench suite, comparing clang with different optimization levels and SPolly. Empty cells represent runs which could not be completed due to technical issues with the used frameworks.