An Interactive Visual Tool for Code Optimization and Parallelization Based on the Polyhedral Model

Eric Papenhausen, Klaus Mueller
Computer Science
Stony Brook University
{epapenhausen, mueller}@cs.stonybrook.edu

M. Harper Langston, Benoit Meister and Richard Lethin
Reservoir Labs
{langston, meister, lethin}@reservoir.com

***Abstract*— Writing high performance software requires the programmer to take advantage of multi-core processing. This can be done through tools like OpenMP, which allow the programmer to mark parallel loops. Identifying parallelizable loops, however, is a non-trivial task. Furthermore, transformations can be applied to a loop nest to expose parallelism. Polyhedral compilation has become an increasingly popular technique for exposing parallelism in computationally intensive loop nests. These techniques can simultaneously optimize for a number of performance parameters (i.e. parallelism, locality, etc). This is typically done using a cost model designed to give good performance in the general case. For some problems, the compiler may miss optimization opportunities or even produce a transformation that leads to worse performance. In these cases, the user has little recourse, since there are few options for the user to affect the transformation decisions. In this paper we present PUMA-V, a visualization interface that helps the user understand and affect the transformations made by R-Stream, an industrial strength optimizing compiler based on the polyhedral model. This tool visualizes performance heuristics and runtime performance statistics to help the user identify missed optimization opportunities. Changes to the transformed code can be made by directly manipulating the visualizations. We show an example where performance is greatly improved over the polyhedral model alone by using our tool.**

Keywords—polyhedral model, source code optimization, visualization, performance analysis

I. INTRODUCTION

Good performance can be obtained either through careful hand-tuning or through automatic techniques. Hand-tuning requires an expert programmer to optimize for different features of the target architecture. This can be a very laborious and time-consuming process. Conversely, fully automatic techniques require little user intervention. Automatic techniques based on the polyhedral model [4][5] are particularly promising and have been shown to outperform hand optimized code in some cases [4].

Optimizing compilers based on the polyhedral model [4][18] take as input a traditional C style loop nest and output transformed code that is optimized for a number of performance characteristics (e.g. parallelism, locality, vectorization, etc.). Transformation decisions are made based on a cost model. The cost model is designed to make good decisions in the general case, but can miss optimization opportunities. In rare cases, the transformed code can perform worse than the input code. The user has little recourse when given a sub-optimal transformation, as there are few options available to affect the decisions made by the polyhedral compiler. To make any further changes, the user must resort to modifying the generated code by hand, a very time consuming and error prone task.

In this paper, we present PUMA-V, polyhedral user mapping assistant and visualizer. It is a tool that uses novel visualizations to show the internal workings of R-Stream [18], an advanced source-to-source compiler based on the polyhedral model. PUMA-V is a web-based tool that uses the popular D3.js [6] visualization library. Unlike existing polyhedral visualization tools, which require the user to perform all transformations, this tool was designed to help the user understand and augment the transformations made automatically by R-Stream. It also includes visualizations of heuristics and runtime performance data to help the user identify missed optimization opportunities and potential bottlenecks. Changes to the code can be made by directly interacting with the visualizations. As the polyhedral model becomes more widely adopted by popular compilers (e.g. GCC [22] and LLVM [11]), our PUMA-V tool is likely to become even more useful.

In the following section we present related work. In section 3 we present the PUMA-V tool and the different views to affect code transformation. Section 4 presents a user scenario showing the advantages of using our tool. Section 5 details future work and section 6 concludes the paper.

II. RELATED WORK

There have been many advancements in the realm of automatic parallelization through the polyhedral model [4][10][15]. These techniques involve representing a loop nest as a system of linear constraints. The lower and upper bounds of a simple *for* loop become linear constraints that define the boundary of a polyhedron. In programs with nested loops, each *for* loop represents an axis in a multi-dimensional coordinate system. This polyhedron is known as the iteration domain. Changing the shape of this polyhedron translates to a transformation of the code. Along with the iteration domain, dependences between statement instances are computed. A dependence occurs when there is a read and a write to the same memory location at different times during the program execution. The iteration domain and dependences together form a linear

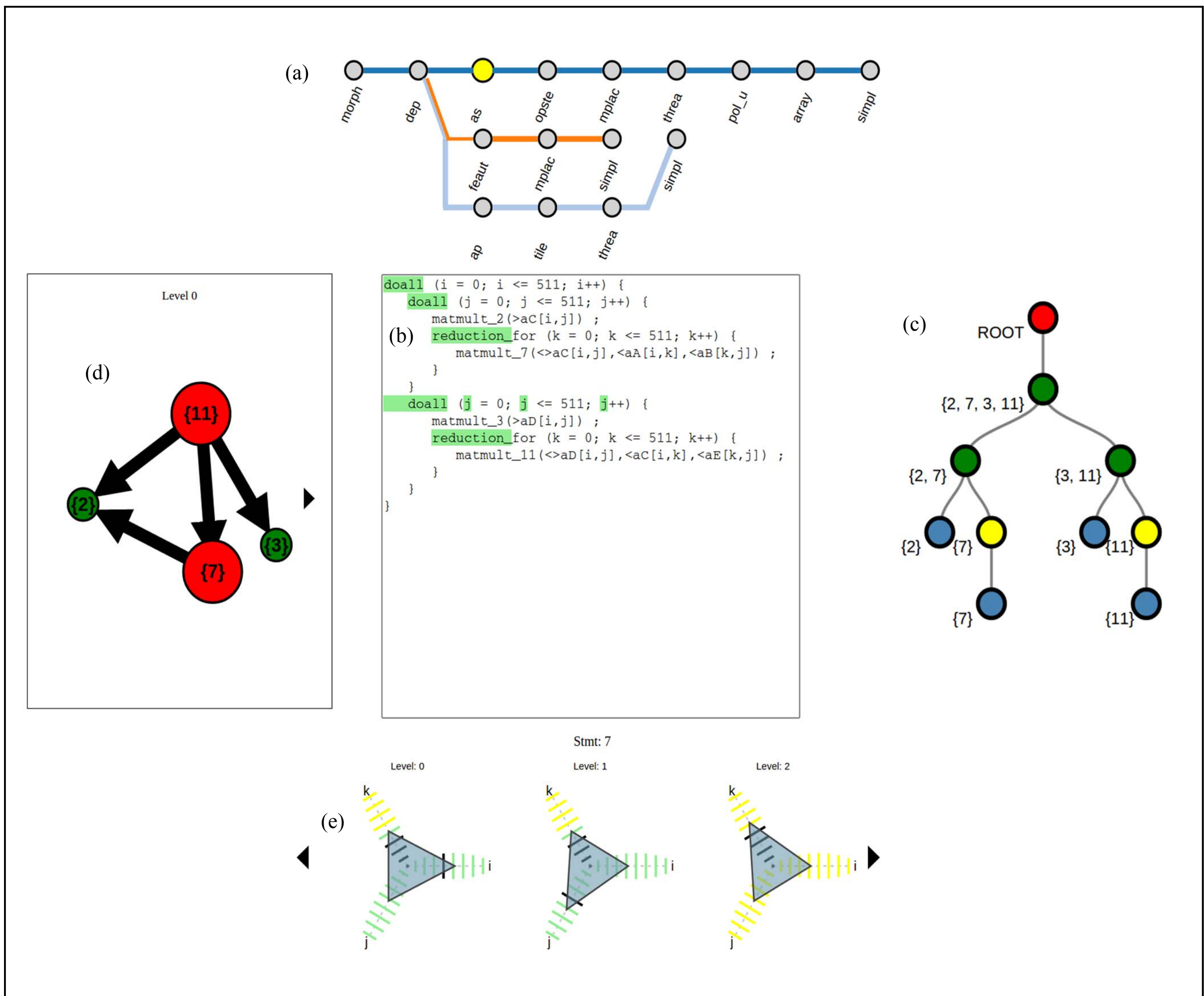


Figure 1: The PUMA-V interface. Each subway line in the tactic view (a) shows a different set of optimization passes. The yellow node indicates the current tactic. The code view (b) highlights the changes from the previous tactic in green. The beta tree view (c) shows a high level overview of the position of loops and statements. The amount of parallelism is signified by the color of the nodes. Nodes are labeled based on the id's of the statements they contain.The dependence graph view (d) shows the dependences. Edge width and length as well as node color and size are determined by various heuristics relating to cache locality. The transformation matrix is visualized via the star plot view (e). Tick marks are colored based on the level of parallelism that can be achieved by moving the values along the axis'.

programming problem, where the objective is to minimize the runtime by transforming the iteration domain in a way that exposes parallel loops. Algorithms that perform code transformation in this way are called polyhedral scheduling algorithms. Modern polyhedral compilers build off the work of [4], which exposes loop parallelism in a way that also allows the loops to be tiled (i.e. an important optimization for cache locality).

Techniques for visualizing the polyhedral model typically involve visualizing the iteration domain [12][17]. The 3D iteration space visualizer [28] visualizes the iteration domain and dependences of a loop nest. Dependences are visualized as vectors, and provide a convenient way for users to identify and mark parallel loops. Tulipse [27] is an Eclipse plugin that also includes a 3D visualization of the iteration domain and dependence vectors. This tool includes an editable code view as well as run time performance visualizations to help the user identify the bottlenecks in the code. ParaGraph [3] is another Eclipse plugin for visualizing and tuning parallel programs. This tool uses the CETUS [8] compiler to automatically identify parallel loops. CETUS performs some analysis to identify if a loop can be parallelized. However, it does not transform code to expose parallelism. The main visualization consists of a control flow graph augmented with dependence information.

The Clint tool [29], is an interface with multiple views for manipulating polyhedral transformations through visualizations. Users can use mouse interactions to change the shape and position of the iteration domain. An editable code view is then updated to reflect how those changes translate to source code transformations. In our previous work [21], we developed an Eclipse plugin to visualize the internal components of the R-Stream compiler. Users could manipulate the internal cost model that governs the types of transformations made by the compiler. Simple run-time performance visualizations were also included to help the user guide optimization decisions.

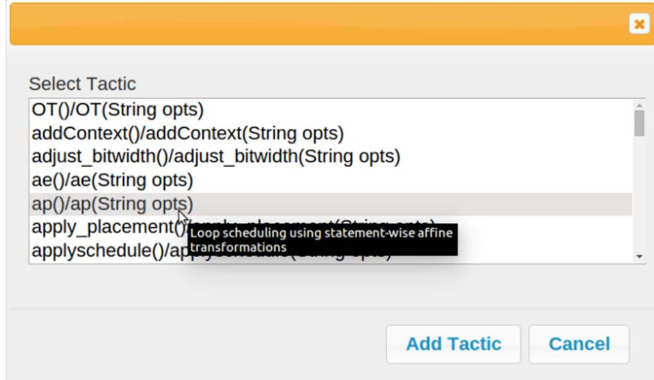


Figure 2: Pop-up that is triggered by clicking "Add Tactic". It displays a list of the available R-Stream optimizations. Hovering over a tactic displays a short description of what it does.

III. PUMA-V INTERFACE

Code generated by the polyhedral model quickly becomes too complex for a programmer to read. Optimization decisions made by R-Stream manifest through deep loop nests with complicated bounds and intricate array access functions. One of the goals that motivated the PUMA-V tool was to help the user better understand the properties of the transformed code. Reading polyhedral generated code can be very complex and intimidating. Augmenting the code with visualizations can make this process less daunting [2]. Our second goal was to help the user identify optimization opportunities missed by R-Stream, and make additional changes through the visualizations to take advantage of them. Current visualization tools based on the polyhedral model allow users to affect code transformations, but do not incorporate fully automatic optimization techniques. In this way, our tool is unique.

A. Optimization Passes

The R-Stream compiler applies a number of optimization passes that include both polyhedral and classical techniques. These optimizations are called tactics within R-Stream. By default, it only applies a small subset of tactics designed to give good performance in the general case. R-Stream has a large repository of tactics that are typically not used by the average user. These include alternative polyhedral scheduling algorithms [9][10][16], different tiling tactics, stencil specific optimizations [25], etc. Many of these tactics outperform the default tactics for certain programs, and so it is useful to provide a way of experimenting with different transformations.

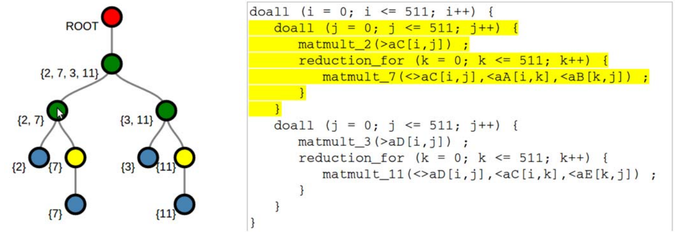


Figure 3: Clicking on a node in the beta tree highlights the corresponding text in the code view. Here the user has clicked on the innermost node containing both statement 2 and statement 7.

Figure 1(a) shows the tactic view. A subway visualization is used to show the tactics, where each tactic is represented by a "station". Tactics are applied from left to right. Each station takes the state of the code given to it from its left neighbor and applies a new tactic. Clicking on a station will update the other views to reflect the state of the code after the clicked tactic is applied. In the figure above, the larger yellow node signifies that the user has clicked on the "as" station.

The code view in figure 1(b) shows a pseudo-code representation of the transformed code. It is updated to reflect the state of the code after the current tactic is applied. Changes from the previous tactic are highlighted in green. The *doall* and *reduction* loops highlighted in figure 1(b) indicate that the application of the "as" tactic has resulted in parallel loops. This indicates that the "as" tactic was responsible for exposing parallelism. This feature increases the transparency of the optimizations. Instead of just viewing the transformed code, the user can see exactly how each tactic changes the code.

The user can also add new tactics at any station in the view. By right clicking a station and selecting "Add Tactic", a popup will show the list of available tactics, as seen in figure 2. Adding a new tactic causes the visualization to branch off a new subway line. This line is assigned an unused color and represents an alternative sequence of optimizations (see figure 1(a)). Different subway lines share the sequence of tactics up to the point where they diverge. In figure 1(a), all three subway lines apply the "morph" and "dep" tactics before diverging into separate optimization sequences. Adding a tactic at the end of a subway line extends the line. There is no limit to the number of tactics that can be assigned to a subway line. This allows the user to experiment with different optimizations while keeping older sequences in a color coded format.

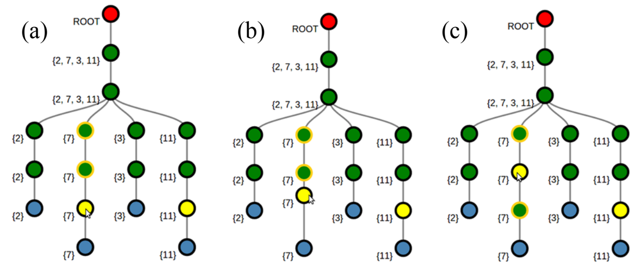


Figure 4: Simple loop interchange can be performed directly in the beta tree. Hovering over a node highlights legal interchangeable loops (a). Interchange can be performed via drag and drop interface (b)-(c).

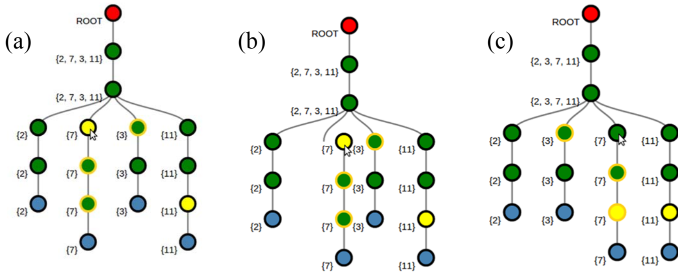


Figure 5: Entire loop nests can be moved via a drag and drop interface in the beta tree view. Hovering over a node highlights loop nests that it can be exchanged with (a). Dragging the node to a new position will exchange loop nests (b)-(c).

B. *Code Abstraction*

The tree visualization of figure 1(c) shows the lexicographic ordering of loops and statements. This is a high level view of the structure of the transformed code. Details like the bounds of the loops and array access functions are abstracted away. This view only shows how statements and loops are ordered in relation to each other. This can have important implications for performance. For example, two nodes that share a common ancestor correspond to statement instances that will be executed consecutively. If these statements share data, then there will likely be good cache locality. This view is generated directly from R-Stream's internal representation and is known as the beta tree view.

Within the beta tree, inner nodes correspond to loops and leaf nodes correspond to statements. Each node is labeled by the IDs of the statements that it contains. In figure 1(c), the node labeled {2,7}, for example, contains statement 2 and statement 7. Nodes with the same label indicate perfectly nested loops. Except for the root, each level in the tree corresponds to a loop level. Level 1 of the tree corresponds to the outer most loops. Level 2 corresponds to the next outer most loops, etc. Nodes below level 1 of the tree correspond to nested loops. Since R-Stream stores this loop position information in a tree data structure, using a tree visualization was a natural choice. The tree visualization also conveys a concise overview of the structure of the code. By looking at figure 3, for example, we can see that statement 3 is nested under two loops and that all statements share an outer loop.

A node is colored based on the amount of parallelism available for the loop it represents. A green node indicates that the corresponding loop is a *doall* loop. Yellow nodes represent *reduction* loops and red nodes represent *sequential* loops [26]. Blue nodes indicate leaf nodes corresponding to statements. *Doall* loops contain the maximum amount of parallelism. Loops that have *doall* parallelism carry no dependences, and so each iteration of the loop can be executed simultaneously. Loops that are marked as *reduction* carry a dependence that results from an operation that is associative (e.g. addition, subtraction, etc.) and can be executed via a parallel reduction. Loops marked *sequential* have no parallelism and become simple *for* loops in the transformed code. As an added convenience, clicking on a node in the beta tree highlights the related section of code in the code view. This includes any nested loops and statements, as seen in figure 3.

Additional code transformations can be applied by interacting with the beta tree. When the mouse hovers over a node in the beta tree, nodes that it can be interchanged with are highlighted (see figure 4). By dragging the node and dropping it to a new position, the user can perform a transformation called loop interchange (i.e. permuting the order of loops in the loop nest). Performing an interchange transformation with any of the highlighted nodes corresponds to a legal transformation (i.e. the transformed code is semantically equivalent to the input code). Loop interchange can be used to change the execution order of the loop nest to improve locality of reference. Entire loop nests can also be moved by dragging a node horizontally. This can improve locality by bringing statements that share data closer together. Figure 5 shows an example of moving the loop nests.

Loop fusion or loop fission can also be performed directly through the beta tree (see figure 6). Right clicking on a node gives the option to perform fusion or fission. Selecting fusion will highlight the nodes that are legal for loop fusion. Fusion has the benefit of improved cache locality for statements that share data. Fusing some loops, however, can lower the amount of parallelism, turning a *doall* loop into a *reduction* or *sequential* loop. Fission is the opposite of fusion and will split a single loop nest into multiple loop nests. The benefits of loop fission can include improved parallelism, turning a *sequential* loop into multiple *doall* loops. In our tool, fission is only allowed on loop nests that contain multiple statements and will cause each statement to separate into an isolated loop nest.

The transformations available through the beta tree view are especially useful when used in conjunction with the dependence graph view (figure 1(d)), which uses heuristics to visualize whether a statement has good cache locality. The dependence graph highlights likely performance bottlenecks, and the user can perform a simple transformation through the beta tree to remove it. The immediate feedback from updates in the dependence graph view encourages an exploratory strategy, where users can try many different optimizations and keep the best ones.

C. *Cost Model Visualization*

The R-Stream compiler performs a dependence analysis to identify the legal transformations and the amount of parallelism available in the program. A part of this process is

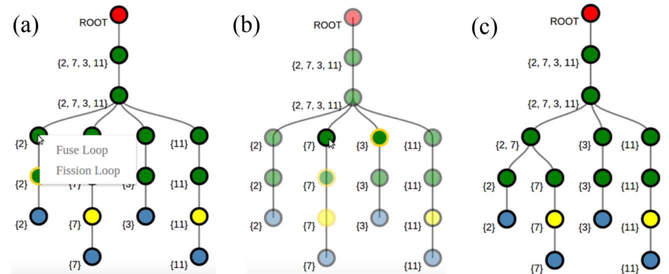


Figure 6: Loop fusion and fission can be performed via the beta tree view. Right clicking on a node gives the option to perform fusion or fission (a). Selecting "Fuse Loop" only keeps legal fusion candidates opaque (b). Clicking on a legal node fuses the loops (c).

the construction of a dependence graph. In this graph, nodes represent statements, and edges represent dependences. The polyhedral scheduling algorithm used by R-Stream adds additional information to this graph to indicate the desirability of certain transformations. Values are computed for each edge and each node to determine how to balance locality and parallelism enhancing transformations. This effectively turns the dependence graph into R-Stream's cost model. A cost model is constructed for each loop level of the program based on dependences and heuristics. Transformations are determined on a level by level basis. Optimization decisions are made for the outer most loop level first, then the next loop level, etc. Figure 1(d) shows the cost model as a dependence graph visualization for the outer most loop level (i.e. level 0). In general, the dependence graph view shows the cost model for a particular loop level and allows the user to navigate to other loop levels via black triangles on either side of the view.

The dependence graph view is implemented as a polymetric visualization [14]. This view contains visual clues conveying a variety of performance heuristics relating to cache locality. The color of the node indicates the amount of spatial or temporal locality available within a single statement. Good locality is achieved by minimizing the number of cache misses. Red indicates that the statement has poor spatial locality, meaning the requested data is likely to be off cache. Green indicates the statement has good spatial locality.

Edges between nodes indicate a dependence. This represents a producer / consumer relationship between the two statements in dependence. In figure 7, the arrow points from node {7} to node {2}, indicating that statement 2 writes to a memory location that statement 7 accesses (i.e. statement 7 depends on statement 2). The length of the edge is determined by the dependence distance (i.e. the number of loop iterations between the source and destination of the dependence). The width of the edge indicates the volume of data that is communicated between the two statements. Optimizations can be selected to change the length of the edge. Edge width, however, cannot be affected. Visualizing this metric can help users identify statements that communicate a lot of data and pick transformations that shorten the edge in the dependence graph view.

Dependences can also occur within a statement. This arises when a statement writes to a memory location at one loop iteration, and then reads from the same memory location at a later loop iteration. The size of the nodes in the dependence graph view indicates the intra-statement dependence distance. Larger nodes indicate a greater number of iterations between the write and read, suggesting that the data communicated has likely been evicted from the cache. We chose to use node size to represent intra-statement dependences because the view became cluttered and difficult to read when visualizing large self-loops.

We visualize performance heuristics relating to locality because memory bandwidth can often be a major performance bottleneck. It is also relatively easy to compute simple, effective heuristics to show the likelihood of a cache miss. Additionally, we already visualize the amount of parallelism through the beta tree view. Another performance heuristic we considered is the likelihood that a loop will be vectorized by a low level compiler (e.g. GCC). This, however, may not be as beneficial, since auto-vectorization greatly depends on the algorithms used by the low level compiler.

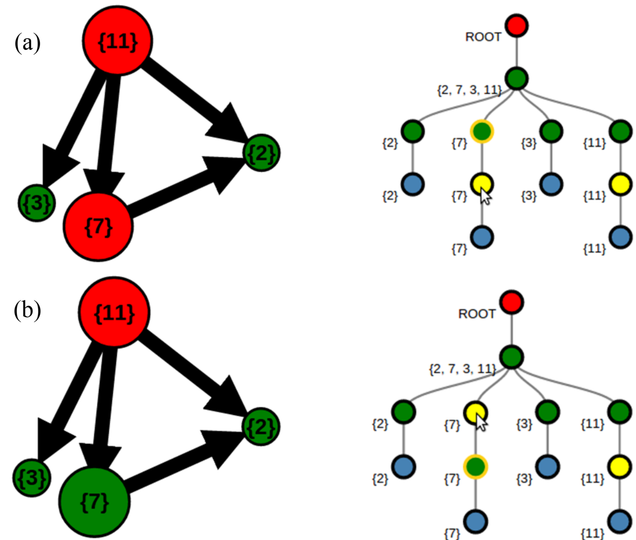
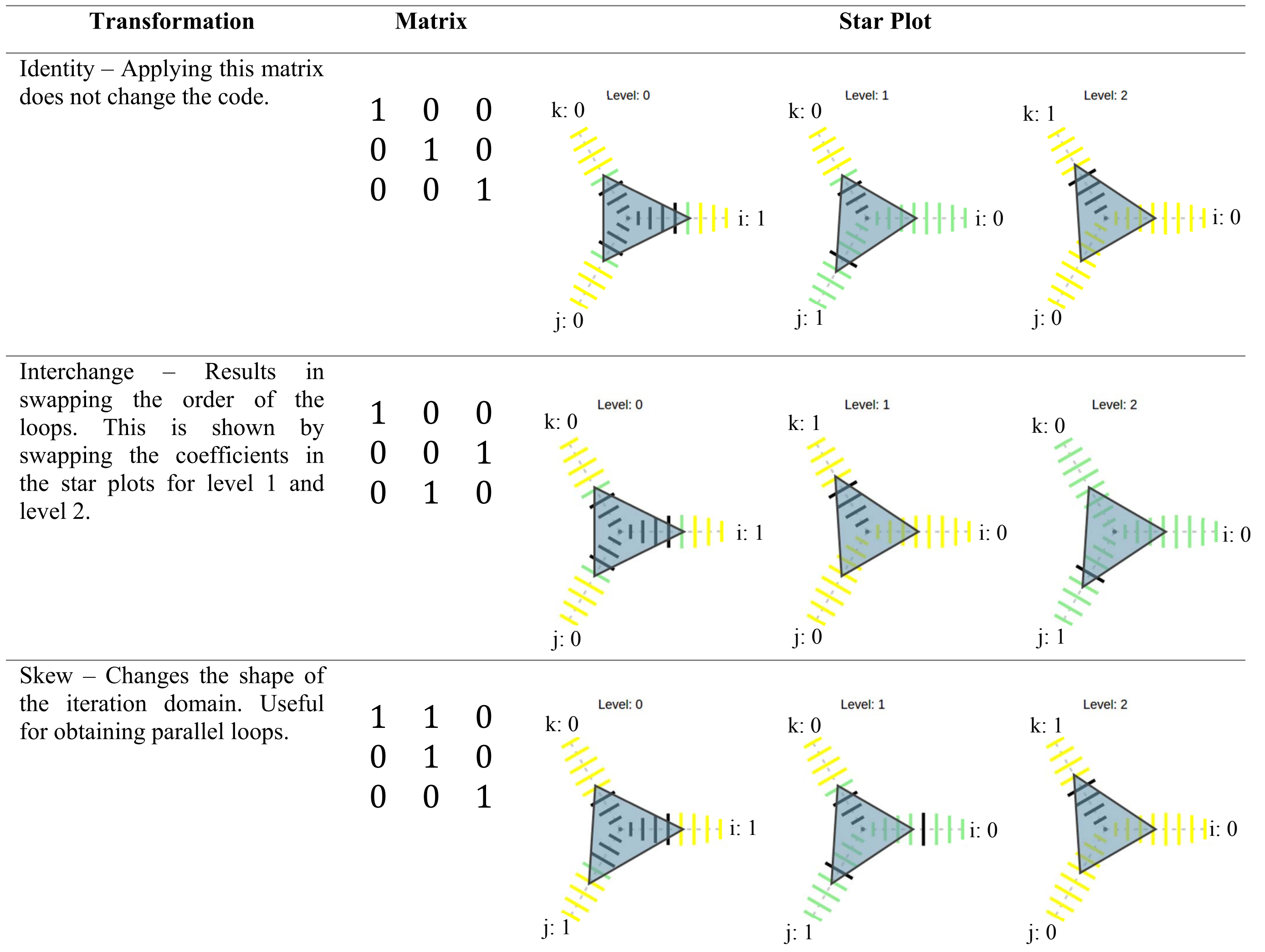


Figure 7: Beta tree view and the corresponding dependence graph view. Statement 7 has poor locality (a) that is improved by interchanging the loops (b). The dependence graph view is immediately updated to reflect the improved locality.

The size and color of the nodes as well as the length of the edges can all be affected by optimizations that change how data is accessed. In addition to the tactic view and the beta tree view, optimization decisions can also be affected by changing the dependence graph view. The values associated with the edges and nodes govern the desirability of certain transformations. A fusion score is associated with the dependence edges. This indicates the desirability of fusion among loop nests that contain the statements in dependence. The user can modify the fusion score by clicking on the appropriate edge in the view. This triggers a popup that allows the user to input a new fusion score. Setting a high fusion score on an edge will increase the likelihood that the statements will be fused, thus shortening the edge. Conversely, setting a high negative fusion score will encourage fission between the two statements, causing the edge to lengthen and possibly increasing parallelism. Similarly, the nodes contain a value for the execution cost and SIMD weight. The execution cost represents the amount of computation associated with the statement. A high execution cost will cause R-Stream to view parallelizing the loops surrounding the statement as a high priority. This can improve parallelism at the cost of locality. A high SIMD weight encourages optimizations that allow the low level compiler (e.g. GCC) to vectorize the code. This is a type of inner loop parallelism that R-Stream has less control over. PUMA-V exposes the fusion score, execution cost, and SIMD weight to the user for modification.

The dependence graph view acts as a proxy for expected performance. In figure 7, for example, the view shows that statements 2 and 3 have good spatial locality but statements 7 and 11 do not. The user can improve locality by interchanging the innermost loops in the beta tree. The interchange shown in



Transformation	Matrix	Star Plot
Identity – Applying this matrix does not change the code.	1 0 0 0 1 0 0 0 1	Level: 0 (k: 0, i: 1, j: 0); Level: 1 (k: 0, i: 0, j: 1); Level: 2 (k: 1, i: 0, j: 0)
Interchange – Results in swapping the order of the loops. This is shown by swapping the coefficients in the star plots for level 1 and level 2.	1 0 0 0 0 1 0 1 0	Level: 0 (k: 0, i: 1, j: 0); Level: 1 (k: 1, i: 0, j: 0); Level: 2 (k: 0, i: 0, j: 1)
Skew – Changes the shape of the iteration domain. Useful for obtaining parallel loops.	1 1 0 0 1 0 0 0 1	Level: 0 (k: 0, i: 1, j: 1); Level: 1 (k: 0, i: 0, j: 1); Level: 2 (k: 1, i: 0, j: 0)

Figure 8: Common transformations, their transformation matrix, and how it is represented in the star plot view. Each star plot corresponds to a row in the matrix. Each axis corresponds to a column in the matrix. The first column corresponds to the *i* axis. The second corresponds to the *j* axis and the third column corresponds to the *k* axis. Dragging a value along each axis corresponds to an update to the appropriate value in the transformation matrix.

figure 7 leads to a 1.4x speed-up for this example. Performing a similar interchange for statement 11 lead to a 5x speed-up. Recall this is the speed-up over R-Streams transformation, not the input code. In general, the user's goal is to find transformations that make the nodes small and green and the fat edges as short as possible. Changing the transformations can be done by selecting different R-Stream tactics in the tactic view, modifying the fusion or execution costs associated with the dependence graph, or by explicitly modifying the loop ordering in the beta tree view. Changes made in any of these views will cause the dependence graph to update to reflect the performance characteristics of the transformed code.

D. Transformation Matrix Visualization

Transformations within the polyhedral model are encapsulated through a matrix. A transformation matrix is assigned to each statement and encapsulates how the iteration order will change for that statement. This is a square matrix where the number of rows and columns is equal to the number of loops surrounding a statement. The transformation is applied by multiplying the matrix by a vector of loop iterators representing the surrounding loops. Table 1 shows a simple example of how transformations are applied in the polyhedral model. Each row in the matrix describes the transformation for a loop level as a linear combination of each of the loops in the input program. The first row corresponds the outer most loop of the transformed code. The second row corresponds to the next outer most loop, etc.

TABLE I. SIMPLE INTERCHANGE TRANSFORMATION

Input	Transformation	Output

`for i = 0 to N`
`for j = 0 to M`
`A[i][j] = . . .`
	$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{bmatrix}i\\j\end{bmatrix}$	`for j = 0 to M`
		`for i = 0 to N`
		`A[i][j]= . . .`

Figure 1(e) shows how the PUMA-V tool visualizes the transformation matrix. This view contains a series of star plot visualizations. A star plot [24] is a method of embedding a multidimensional vector into a two-dimensional visualization. Axes are organized in a circular fashion with the origin at the

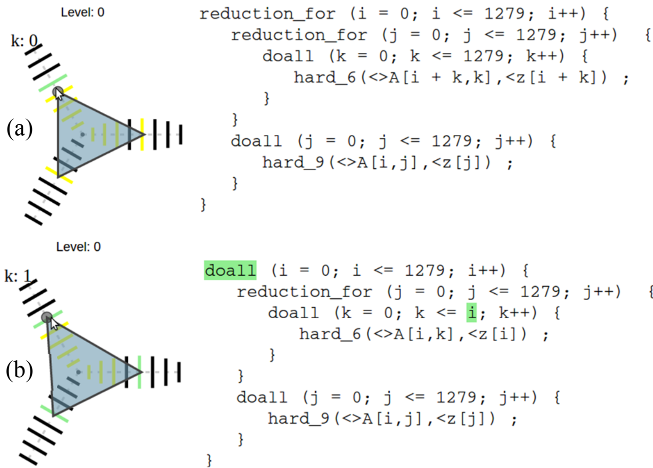


Figure 9: Code transformations can be set directly through the star plot view. In this example, a user is changing the parallelism for the outermost loop from a *reduction* (a) to a *doall* loop (b), by dragging the value along the *k* axis of the star plot for level 0.

center. The points of the plot represent values of the vector, which together form a recognizable shape (e.g. the triangle shown in figure 8).

The tick marks on each axis are colored based on the level of parallelism that can be achieved if the user changes the value along the axis. Green tick marks indicate *doall* parallelism. Yellow signifies a *parallel reduction* and red represents a *sequential* loop. Setting a value along the axis to a black tick mark results in an illegal transformation. This is a transformation that is not semantically equivalent to the input code. PUMA-V still allows the user to perform an illegal transformation, since the user may perform some other transformation to make the code legal again.

For each statement, there are a series of start plots. Each plot represents a row in the transformation matrix and each axis represents a column. The *i* axis corresponds to the first column. The *j* axis corresponds to the second column, etc. Figure 8 shows a number of common transformations along with their transformation matrices and how they are visualized using the star plot view. The values along each axis can be changed via a simple drag and drop interface. This gives the user direct control over the transformation matrix. Figure 9 shows an example of a user modifying the transformation matrix via the star plot view to get *doall* parallelism along the outermost loop.

We use the star plot visualization in lieu of an iteration domain visualization that is commonly used by other polyhedral visualization tools [28][29]. These visualization tools display a view of the iteration domain to represent code transformations. Iteration domain visualizations, however, do not scale very well to higher dimensions. To display multidimensional iteration domains corresponding to deep loop nests, previous tools would often resort to a scatterplot view of 2D projections. As the number of dimensions in the iteration domain increases, the number of views in the scatterplot grows exponentially. By using the star plot visualization, the number of views grows linearly with the dimensionality of the iteration domain (i.e. an additional star plot is needed to accommodate the added row in the transformation matrix). We considered using other high dimensional visualizations (e.g. parallel coordinates [13]), but the star plot views were more compact and required less screen real estate. The PUMA-V tool is able to represent high dimensional iteration domains concisely and without requiring a large amount of screen real estate.

E. *Embedded Performance Visualization*

Visualizing statistical data gathered from runtime execution has been shown to be quite effective at giving developers insight into the performance characteristics of their code. Using tools such as Vampir [20] and Tau [23] have allowed developers to identify performance bottlenecks. Like these tools, the PUMA-V tool provides a means for gathering runtime performance data of the transformed code. Unlike these tools, PUMA-V allows users to make changes to the code directly through the visualizations.

When gathering runtime performance data, C code is generated from the current transformations. The outermost loop in each loop nest is parallelized using an OpenMP [7] pragma. The code is then compiled and executed. Performance data is gathered with the help of HPCToolkit [1] and PAPI [19] performance counters. The execution time is also reported and the beta tree visualization is updated to reflect the performance.

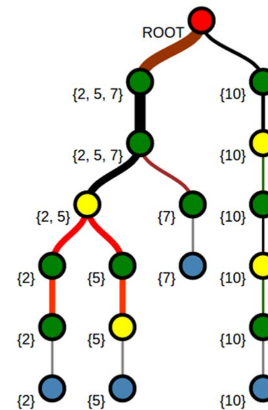


Figure 10: Runtime performance statistics are embedded in the beta tree view. Edge width denotes the distribution of execution time. Green edges indicate a low L2 cache miss rate (low CPI) and red edges indicate a high L2 cache miss rate (high CPI). Black edges indicate that we are missing cache miss rate data.

Figure 10 shows the beta tree after runtime performance data is gathered. The width of the edges is updated to reflect the distribution of execution time. Thicker edges indicate that more time is spent processing the statements in that branch of the tree. Edges are also colored to show additional performance metrics. The user can toggle between setting edge color to represent the L2 cache miss rate or cycles per instruction (CPI). This gives a representation of the memory bandwidth and instruction bandwidth respectively. Green edges indicate a low L2 cache miss rate (low CPI) and red indicates a high L2 cache miss rate (high CPI). HPCToolkit does not always gather performance metrics for every branch of the tree. Black edges indicate that HPCToolkit has gathered the timing data but has not gathered data relating to the L2 cache miss rate or the CPI.

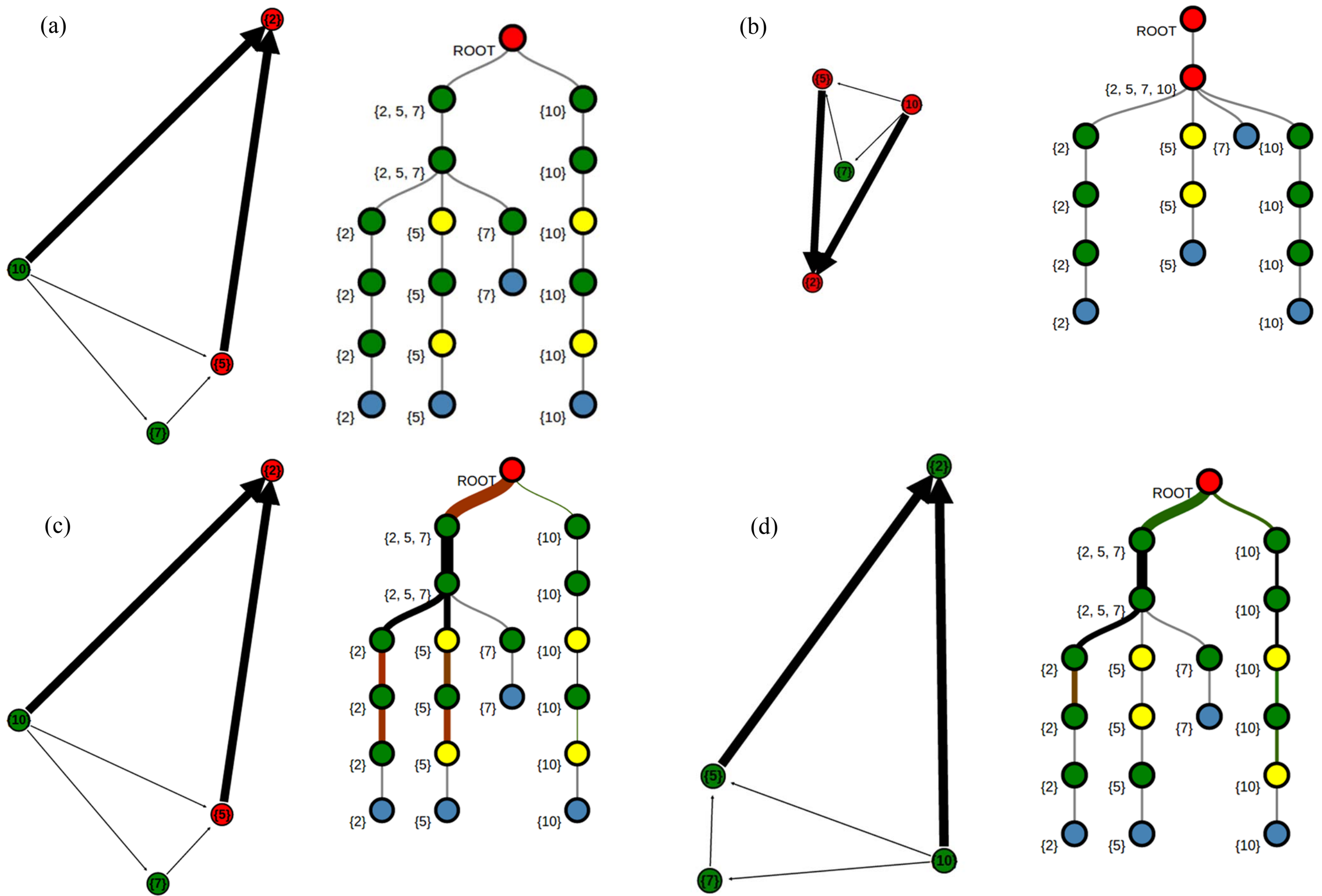


Figure 11: Beta tree and dependence graph views of the various optimizations tried during the user scenario. (a) shows the views prior to any additional changes. Increased locality can come at the cost of parallelism (b). The runtime performance data showing poor locality among the loop nests containing statements 2 and 5 (c). Performing a loop interchange via the beta tree has yielded a 1.9x speed-up (d). This interchange is evidenced by the ordering of the innermost 2 loop nodes containing statement 5. The nodes have gone from "green" "yellow" in (c), to "yellow" "green" in (d). Note that in (d), all dependence graph nodes are green, indicating that each statement likely has good locality of reference.

Thin gray edges indicate that HPCToolkit has not gathered any data for that branch in the beta tree. The runtime evaluation allows users to take an iterative approach to optimizing code. Embedding the runtime performance visualization into the beta tree helps users see the problem areas of the code and perform appropriate transformations to alleviate the bottleneck.

Visualizing the runtime performance through the beta tree view has a few advantages. First, it helps conserve screen real estate. It is important for the user to be able to see all the views on one screen. We would not be able to add a new view for performance visualization without pushing some other view off the screen. Second, embedding the runtime information in the beta tree allows the user to see which loop nests contain bottlenecks. This is particularly useful for the L2 cache miss rate, which is often improved through a simple loop interchange. A user is able to identify a bottleneck, and resolve it through a single view.

IV. User Scenario

In this section we present a scenario showing how a user, Tom, would use the PUMA-V tool to optimize a sequence of matrix-vector multiplications. After loading the input file into the tool, the views update to show the result after the default optimization passes are applied by R-Stream. The result is shown in figure 11(a). The first thing Tom does is he traverses the subway visualization to get a sense of what transformations have been applied and how they affect performance. Next, Tom is ready to start making changes to the transformation. Tom does not have much experience with the polyhedral model, and so he takes a more exploratory strategy.

Tom starts affecting transformations by adding new optimization passes to the subway visualization. He adds new subway lines that use alternative polyhedral scheduling algorithms. Tom applies transformations that sound like they might be beneficial according to their descriptions (see figure 2). The beta tree view shows that these alternative algorithms have different fusion / fission structures but they consistently expose less parallelism. He is unable to find optimizations that expose more parallelism than the default sequence of optimizations. He also experiments with different tiling tactics. The performance characteristics for each tiling tactic are displayed through the dependence graph view and Tom sees that the performance is similar for each of the tiling tactics.

Looking at the dependence graph, he sees that there are long thick edges between statements 5 and 2 and statements 10 and 2. To improve locality between these statements, he sets a high fusion weight on these edges (e.g. a weight of 1000). Figure 11(b) shows the result of this transformation. Locality has been improved between these statements, as seen by the fusion at the outermost loop level in the beta tree. The cost, however, is reduced parallelism. The outer most loop is now sequential. Tom realizes that this is too much of a cost to pay for improved locality and he resets the fusion weight to its prior values.

Next, Tom performs a runtime evaluation of the code. The tool tells him that the execution time for this version of the code is 0.35 seconds. Figure 11(c) shows the beta tree view updated with performance counter data from the runtime evaluation. Tom sees that most of the execution time is spent processing statements 2, 5, and 7 (i.e. the left branch of the beta tree). The red edges in the beta tree along the loop nests containing statements 2 and 5 indicate that there is a high L2 cache miss rate for these statements. Nodes 2 and 5 in the dependence graph view are also red, indicating poor locality. Tom realizes that poor locality on these statements is likely the bottleneck of this application. He performs a simple loop interchange of the innermost nodes of statements 2 and 5 through the drag and drop interface available via the beta tree. The effect of this is seen in the dependence graph shown in figure 11(d). The nodes are now all green indicating good locality for each statement. Tom performs another runtime evaluation on the transformed code. Figure 11(d) shows the results. The execution time for this program is 0.18 seconds, yielding a 1.9x speed-up over the original transformation selected by R-Stream.

V. FUTURE WORK

We have shown the PUMA-V tool to potential users and have gained valuable feedback that directs our future work. One direction for future work involves setting tile sizes in a way that provides visual feedback about the expected locality improvements. Setting good tile sizes can have a large impact on performance. We propose using a slider to set the tile sizes. Changing the values on the slider would then update the size of the nodes representing loop nests based on whether the tile fits in L1, L2, or L3 cache.

The PUMA-V tool would also benefit from letting users manually edit the code via the code view. R-Stream, however, does not currently provide the ability to mix manual edits with the automatic transformations. Other polyhedral visualization tools provide this feature (e.g. [29]), and so it is technically feasible. The challenge is to enforce that the source code edits do not change the semantics of the input program, while allowing maximum flexibility. This remains an item for future work.

Our current runtime performance visualization only supports two metrics (i.e. L2 cache miss rate and cycles per instruction). We will expand this to include any metric that can be constructed from the available performance counters. This feature will be similar to HPCToolkit's derived metrics. The user can specify which metrics he or she is interested in and use radio buttons to select the metric to be mapped to edge color. We also want to explore the possibility of integrating the PUMA-V tool with more traditional performance visualization tools (e.g. Vampir [20] and Tau [23]). This could streamline the optimization process, where different transformations could be performed rapidly through PUMA-V and then immediately evaluated using popular profiling tools.

Finally, the current PUMA-V tool only supports optimizing for multi-core systems, and parallelizes using OpenMP [7]. R-Stream, however, can perform automatic optimization for GPUs and multi-node systems as well. We intend to extend the capabilities of our tool to support optimizing for these other architectures.

VI. CONCLUSION

In this paper, we presented PUMA-V, a tool combining fully automatic and manual techniques for optimizing source code. The scenario presented in this paper suggests that combining automatic methods with user intuition can lead to significantly better performance compared to automatic methods alone. Using effective visualizations through multiple views helps users seize optimization opportunities missed by the auto-parallelizing compiler. Embedding heuristics and runtime performance into the visualization assists users in identifying bottlenecks.

PUMA-V can also be valuable to compiler writers. Over time, the tool has exposed a number of deficiencies in the R-Stream compiler, which can now be addressed by the compiler writers. Hence, PUMA-V is not only useful in making a given code faster, it can also help identify effective optimization strategies that users make, and attempt to automate them.

In addition to optimizing performance, PUMA-V can also serve as an important educational tool for teaching the polyhedral model. We performed user studies, to be presented elsewhere, in which users note that the PUMA-V tool greatly improved their understanding of the transformations made by R-Stream. Students can get a sense of how to perform some basic optimizations and why they are important. By manipulating the star plot view and seeing the impact on the code, they can become more accustomed to the transformation matrix. Runtime performance visualization helps students see how certain transformations can affect performance characteristics.

REFERENCES

[1] L. Adhianto, et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs." *Concurrency and Computation: Practice and Experience* 22.6 (2010): 685-701.

[2] Ball, T. and Eick, S.G., 1996. "Software visualization in the large". *Computer*,*29*(4), pp.33-43.

[3] I. Bluemke, J. Fugas, "A tool supporting C code parallelization." Innovations Comput. Sci. Soft. Eng., 259–264 (2010)

[4] U. Bondhugula, A. Acharya, A. Cohen, "The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests." ACM Transactions on Programming Languages and Systems (TOPLAS), accepted in Dec 2015.

[5] U. Bondhugula, O. Gunluk, S. Dash and L. Renganarayana, "A Model for Fusion and Code Motion in an Automatic Parallelization Compiler." IBM Research Report RC24967 (W1001-053), Jan 2010.

[6] M. Bostock, "D3. js." *Data Driven Documents* (2012).

[7] L. Dagum, and R. Enon. "OpenMP: an industry standard API for shared-memory programming." *Computational Science & Engineering, IEEE* 5.1 (1998): 46-55.

[8] C. Dave, H. Bae, S.J. Min, S. Lee, R. Eigenmann, S. Midkiff: Cetus: A source-to-source compiler infrastructure for multicores. Computer 42, 36–42 (2009)

[9] A. Darte, L. Khachiyan, and Y. Robert. "Linear scheduling is nearly optimal". *Parallel Processing Letters*, *1*(02), 73-81. (1991)

[10] P. Feautrier. "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time", In International Journal of Parallel Programming, vol. 21(5), pp. 313—318, 1992.

[11] T. Grosser, Hongbin Zheng, Raghesh A, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet, "Polly – Polyhedral Optimization in LLVM," in proceedings of the 2011 International Workshop on Polyhedral Compilation Techniques (IMPACT '11), Apr 2011.

[12] T. Grosser, "islplot." https://github.com/tobig/islplot

[13] A. Inselberg, and B. Dimsdale. "Parallel coordinates." In *Human-Machine Interactive Systems*, pp. 199-233. Springer US, 1991.

[14] M. Lanza and S. Ducasse. "Polymetric views - a lightweight visual approach to reverse engineering". IEEE Transactions on Software Engineering, Vol. 29(9):782-795, September 2003.

[15] R. Lethin and A. Leung and B. Meister and N. Vasilache, "Methods And Apparatus For Joint Parallelism And Locality Optimization In Source Code Compilation." U.S. Patent Application No. 12561152, Sep 2009.

[16] A. W. Lim and M. S. Lam. "Maximizing parallelism and minimizing synchronization with affine partitions". Parallel Computing, 24(3-4):445–475, 1998.

[17] V. Loechner, "VisualPolylib." *http://icps.u-strasbg.fr/PolyLib/*

[18] B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, A. Leung and Richard Lethin, "R-Stream Compiler." In Encyclopedia of Parallel Computing, pp. 1756—1765. David Padua Ed. 2011.

[19] P.J. Mucci, S. Browne, C. Deane, and G. Ho. "PAPI: A portable interface to hardware performance counters." In *Proceedings of the department of defense HPCMP users group conference* (pp. 7-10). 1999, June

[20] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," Supercomputer, vol. 12, no. 1, pp. 69–80, January 1996.

[21] E. Papenhausen, B. Wang, M.H. Langston M. Baskaran, T. Henretty, T. Izubuchi, A. Johnson, C. Jung, M. Lin, B. Meister and K. Mueller, 2015, September. "Polyhedral user mapping and assistant visualizer tool for the r-stream auto-parallelizing compiler." In *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on* (pp. 180-184).

[22] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber and N. Vasilache, GRAPHITE: Loop optimizations based on the polyhedral model for GCC." Proceedings of the 4th GCC Developper's Summit, pp. 179—198, Ottawa, Canada, Jun 2006.

[23] S. S. Shende and A. D. Malony, "The tau parallel performance system," Int. J. High Perform. Comput. Appl., vol. 20, pp. 287–311, May 2006.

[24] R. Spence, "Information visualization". New York: Addison-Wesley; 2001.

[25] K. Stock, M. Kong, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. "A framework for enhancing data reuse via associative reordering". In PLDI, 2014

[26] M. Stone, "A field guide to digital color". CRC Press; 2013 Jul 31.

[27] Y. W. Wong, T. Dubrownik, W. T. Tang, W. J. Tan, R. Duan, R. S. M. Goh, S.-h. Kuo, S. J. Turner, and W.-F. Wong. Tulipse: a visualization framework for user-guided parallelization. In Euro-Par 2012 Parallel Processing, pages 4–15. Springer, 2012.

[28] Y. Yu and E. D'Hollander. Loop parallelization using the 3d iteration space visualizer. Journal of Visual Languages & Computing, 12(2):163–181, 2001.

[29] O. Zinenko, C. Bastoul, and S. Huot, "Manipulating visualization, not codes," in Int. Workshop Poly. Comp. Tech. 2015 (IMPACT), 2015, p. 8.