FEATURE ARTICLE

PUMA-V: Optimizing Parallel Code Performance Through Interactive Visualization

Eric Papenhausen
Computer Science Department, Stony Brook University, Stony Brook, NY

M. Harper Langston
Benoit Meister
Richard Lethin
Reservoir Labs, Inc. New York, NY

Klaus Mueller
Computer Science Department, Stony Brook University, Stony Brook, NY

Performance optimization for parallel, loop-oriented programs compromises between parallelism and locality. We present a visualization interface which allows programmers to assist the compiler in generating optimal code. It greatly improves the user's understanding of the transformations that took place and aids in making additional transformations in a visually intuitive way.

Visual analytics enables users to participate in machine-based optimization processes, contributing elements of human creativity, ingenuity, and expertise as well as commonsense knowledge. The benefits of visual analytics have been demonstrated in many domains, such as science, business, and medicine. In this work, we take advantage of visual analytics to allow users to reason about parallel code generation. The visual tool we developed, PUMA-V, is especially useful for compiler developers who are looking to expose and patch weaknesses in the compiler's optimization pipeline, as well as general users who want to optimize their code but do not know how to best focus their efforts.

Because of its deep level of analysis and powerful abstractions, we focus particularly on compilers based on the polyhedral model[1-4] in our work. The cost models of these compilers can be augmented through compiler options. However, expert knowledge of the compiler tactics is typically required to achieve deeper levels of optimization. The average user has little recourse to improve the resulting performance, since the options available to affect the transformation decisions are generally non-intuitive and require extensive background in the polyhedral model to understand their affects. Furthermore, the complexity of the generated code makes it nearly impossible for a user to understand the optimization decisions or the reasoning behind them.

PUMA-V stands for *P*olyhedral *U*ser *M*apping Assistant and *V*isualizer. It allows users to affect, at a fine level of detail, many of the optimization decisions made by R-Stream, a polyhedral based source-to-source compiler. [1] The main contribution of PUMA-V, compared to previous work on polyhedral visualizations, is its tight integration with a fully-automatic compiler via linking to the compiler's library, calling functions such as scheduling, dependence analysis, etc. The automatic compiler uses linear programming to produce a program schedule, optimizing a cost function that favors locality, parallelism, and other factors. PUMA-V empowers the user, possibly after an automatic optimization, to improve the schedule further. The views of PUMA-V are augmented with static and runtime performance analyses to help guide the user in making manual transformation decisions to the most problematic parts instead of just applying boiler-plate transformations. A simple drag-and-drop interface makes it easy and intuitive to use.

User studies we conducted show that the semi-automatic approach available through this tool enables better performance as well as a greatly improved understanding of the transformations made by the compiler. Although we apply this tool to a compiler based on the polyhedral model, many of the visualizations and interactions we use can be applied to compiler optimizations in general. With the widespread adoption of polyhedral techniques into popular compilers like GCC and LLVM, our tool will become even more useful in the future.

THE POLYHEDRAL MODEL

Much work has been done on the polyhedral model for loop parallelization.[1] These techniques involve representing a loop nest as a system of linear constraints. This is an abstraction where the lower and upper bounds of a simple for loop become linear constraints that define the boundary of a polyhedron, or iteration domain for a statement. In programs with nested loops, each loop represents an axis in the multi-dimensional iteration domain. Each point in the iteration domain represents an instance where the statement will be executed and is assigned a logical execution date based on the surrounding loop iterators. Changing the shape of this polyhedron translates to a transformation of the code because it changes the execution dates of the statement instances. New execution dates can be computed for statement instances via a scheduling relation. This is a multi-dimensional vector that can represent a number of common program transformations (e.g. loop interchange, fusion / fission, skewing, etc.).

Along with the iteration domain, dependences between statement instances are computed. A dependence occurs when two statement instances access the same memory location and at least one of these accesses is a write. Dependences help define the semantics of the program (i.e., preserving dependences will preserve program semantics). The iteration domain and dependences together form a linear programming problem; where the objective is to minimize the runtime by transforming the iteration domain in a way that exposes parallel loops. C code is then generated from this abstraction and compiled by a low level compiler (e.g. GCC). Algorithms that perform code transformation in this way are called polyhedral scheduling algorithms. Modern polyhedral compilers expose loop parallelism in a way that also allows the loops to be tiled (i.e. an important optimization for cache locality).

Polyhedral compilers expose loop-level parallelism, optimize for cache locality, expose SIMD parallelism, etc. In practice they make reasonably good optimization decisions that result in better performance. Because the polyhedral model is based on heuristics and cost models, some optimization decisions are sub-optimal. Hence, while performance improvement is generally obtained, there is no guarantee on the optimality of the performance of the optimized programs.

THE PUMA-V INTERFACE

The interface we present here builds on our previous system published in [20] (see sidebar for a description). The main view (Figure 1), chord view (Figure 7), and global view (Figure 9(b)) together give a detailed account of the optimizations made by R-Stream and various performance characteristics of the transformed code. These views are all linked and transformations made in one view will trigger an update in the others to show the state of the transformed code. Current tools allow users to make polyhedral transformations; but do not incorporate automatic polyhe-

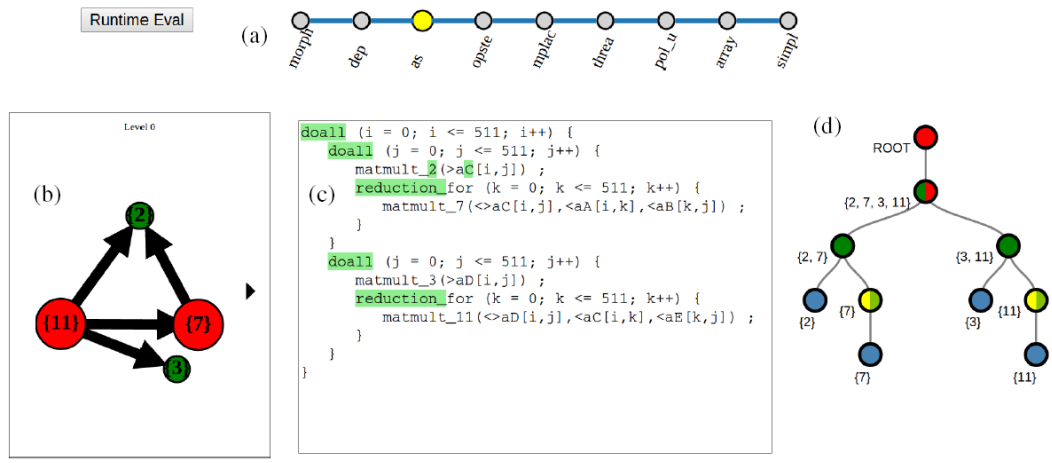

Figure 1 The PUMA-V tool showing the state of the code – a consecutive matrix multiplication task – after the "affine scheduling" (as) optimization (or tactic) has been applied. (a) The tactic view is a subway visualization, where each tactic is represented by a station, applied sequentially left to right. (b) The dependence graph view shows a node-link diagram where nodes represent statements and edges represent dependences. (c) The code view lists the nested-loop program code that is being optimized. (d) The beta tree view shows the lexicographic ordering of loops and statements. Each branch is a nested loop where the numbers refer to the numbers appended to the code statements in the code view.

dral scheduling techniques. To the best of our knowledge, PUMA-V is the first tool that visualizes exposes the internal mechanics of an advanced polyhedral based compiler.

Tactic View

The R-Stream compiler applies a sequence of polyhedral and classical optimizations called *tactics.* 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 (close to 100). Some of the more popular ones are *dep()* which performs dependence analyses, *morph()* which builds cost functions based on the processor's characteristics (cache size, number of cores, etc.) and *as()* which performs affine scheduling. Others are typically not used by the average user such as alternative polyhedral scheduling algorithms, different tiling tactics, stencil specific optimizations, etc.[2-4] Many of these tactics outperform the default set of tactics for certain programs; and so we strived for a visual interface that can provide users with an easier way of experimenting with different transformations.

Tactics are applied in a sequential manner. A familiar paradigm to visualize sequential processes is the subway visualization.[5] Figure 1(a) shows the tactic view using our subway visualization view, 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. A user can visit a station by clicking on the node. This will update the other views to reflect the state of the code after the clicked tactic is applied. In Figure 2, the larger yellow station signifies that the user has clicked on the "affine scheduling" (as) node.

The code view in Figure 1(c) is updated to reflect the state of the code after the current tactic is applied. This view contains a pseudo-code representation of the transformed code. The suffix of a nested loop body statement displays its unique identifier. For example, the "7" in "matmult 7" indicates that this is statement 7. This view also distinguishes between a read and write to arrays within a statement. The ">" sign indicates a write while a "<" indicates a read. For example, statement 7 in Figure 1(c) reads and writes to a C[i,j] and only reads from a A[i,k] and a B[k,j].

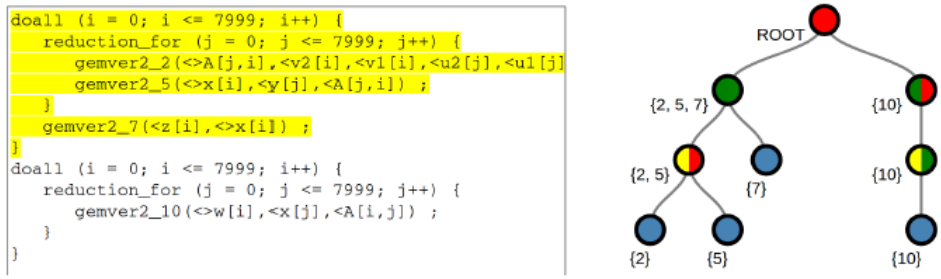

Figure 3: Result of clicking on the node labeled (2,5,7). The corresponding text in the code view is highlighted. The code here is from the Polybench benchmark suite [12].

Changes from the previous tactic are highlighted in green. The *doall* and *reduction* loops highlighted in Figure 1(c) indicate that the application of the "affine scheduling" (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. 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 2). Different subway lines share the sequence of tactics up to the point where they diverge. In Figure 2, both the orange and light blue 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.

Beta Tree View

The beta tree view of Figure 1(d) shows the lexicographic ordering of loops and statements. It is a visualization of the β component of the scheduling relation of Figure 1(c). Since R-Stream stores this loop position information in a tree data structure, using a tree visualization was a natural choice. Inner nodes correspond to loops and leaf nodes correspond to statements. Each node is labeled by the IDs of the statements that it contains. The node labeled {2,7}, for example, contains nested loop body statement 2 and statement 7 (matmult_2 and matmult_7, see previous section).. Except for the root, each level in the tree corresponds to a loop level. Level 0 of the tree corresponds to the outer most loops. Level 1 corresponds to the next outer most loops, etc. Nodes below level 0 of the tree correspond to nested loops. 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 10 is nested under two loops and that statement 2, 5, and 7 share an outer loop. Additionally, clicking on a node in the beta tree highlights the related section of code in the code

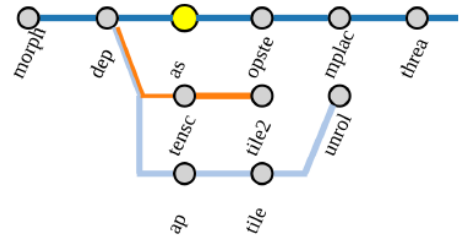

Figure 2: The subway visualization used in the tactic view. Different colored lines represent different optimization paths that can be taken by the compiler. The orange and light blue lines diverge from the dark blue line after the "dep" station.

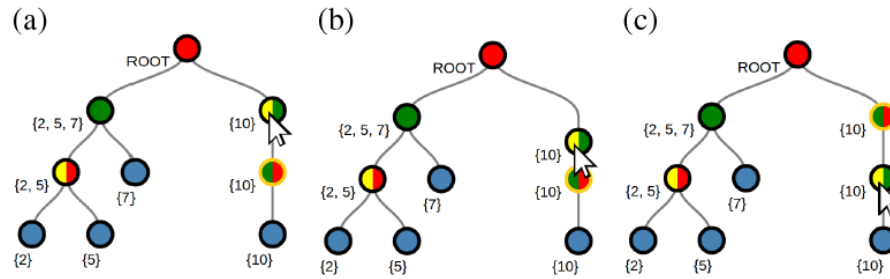

Figure 4: Loop interchange can be performed directly through the beta tree. Hovering over the node in (a) highlights permutable loops. A drag and drop action interchanges the loops (b)-(c).

view. This includes any nested loops and statements; as seen in Figure 3.

The two most important performance metrics in parallel computing are parallelism and locality which often present tradeoffs. In order to communicate both of these metrics to the user simultaneously we split the inner nodes (which represent loops) into two halves and colored each half with respect to one of the two metrics. The leaf nodes on the other hand are colored blue to indicate that they represent statements. The left half of an inner node is colored based on parallelism, while the right half is colored based on the array access stride of the loop. For parallelism, green corresponds to *doall* loops. Yellow represents *reduction* loops and red represents a *sequential* loop. 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.

The right half of the nodes are colored from red to yellow to green based on a linear scaling of the stride. Red indicates a large array access stride, while green represents a low stride. The stride of a loop has important implications for locality. A low stride loop at the innermost position indicates good spatial or temporal locality (i.e. data is likely to be in cache when it is needed). An optimal ordering for loops with respect to stride is to have the red nodes at the top of the tree, yellow nodes in the center, and green nodes at the bottom. An example of this split color can be seen in node {2,5} of Figure 3. This node is colored yellow and red to indicate it represents a *reduction* loop with a high memory access stride. Although node colors are split based on parallelism and locality by default, radio buttons at the bottom of the tree allow the user to toggle between showing parallelism only or stride only views.

We also wanted to define a set of visual interactions by which users could perform code transformations directly in 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). 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 data reuse by bringing statements that share data closer together. Figure 5(a-c) shows an example of moving the loop nests.

Users can also perform loop fusion or loop fission through the beta tree (see Figure 5(d-f)). 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. Fusing statements that share data increases the likelihood that data is already in cache when it is needed. 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. In our tool, fission is

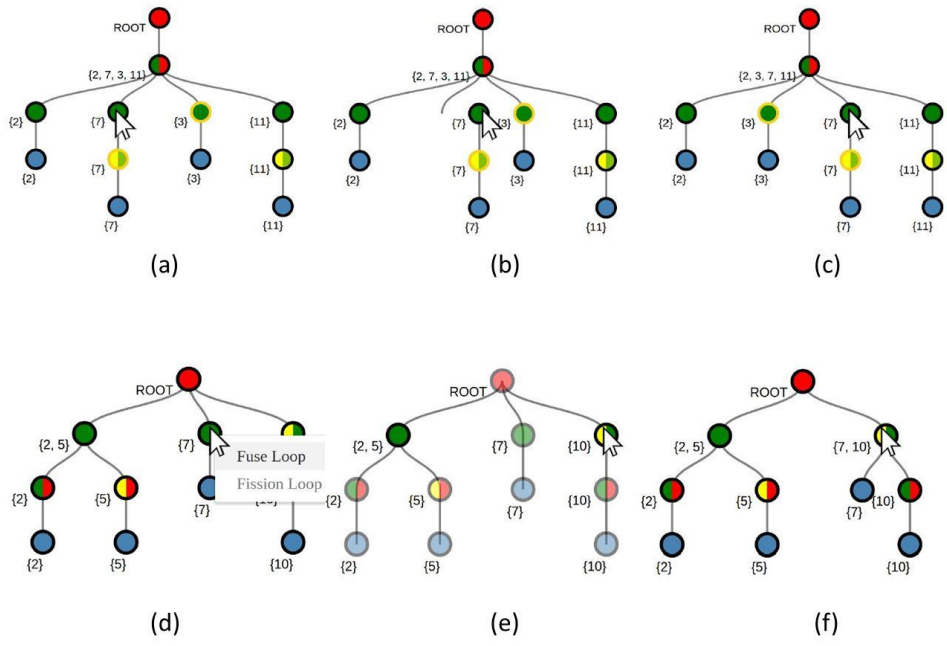

Figure 5: Some beta-tree operations. Reordering loop nests directly through the beta tree. Hovering over the node in (a) highlights swappable loop nests. A drag and drop action changes the loop nests (b)-(c). Right clicking on node 7 brings the option to perform fusion or fission (d). Selecting “Fuse Loop” highlights only loops where fusion is legal (e). Clicking node 10 causes the loops to fuse (f). Note that we lose the parallelism of the doall loop on 7.

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 (Figure 1(b)); 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.

Dependence Graph View

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 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. 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 heuristics and on the dependence representation. 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(b) 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.[6] This view contains visual clues conveying a variety of performance heuristics relating to cache locality. Figure 1(b-c) show sample code and its dependence graph. 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 locali-

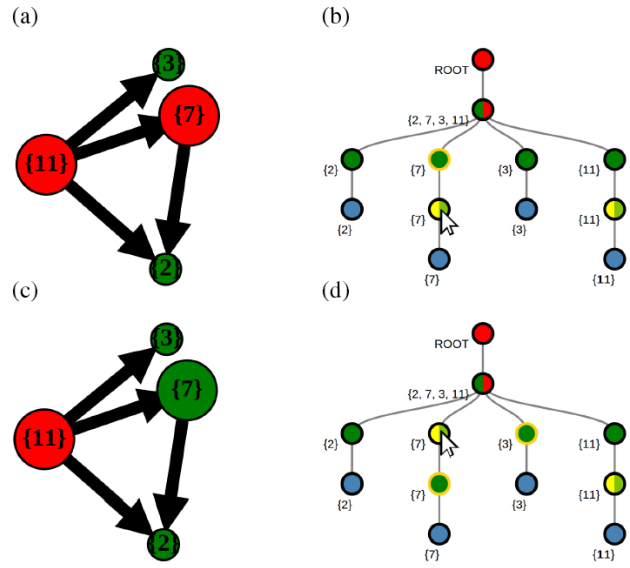

Figure 6: Performing transformations in the beta view causes the dependence graph view to update. Statements 7 and 11 in (a) exhibit poor locality. By performing loop interchange on statement 7 (b)-(d) the locality is improved (c).

ty; meaning the requested data is likely to be off cache. This is commonly caused by a high access stride of the innermost loop. 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 1(b), 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. Shortening an edge through fusion, for example, reduces the amount of time between the execution of two dependent statements; thus, improving the likelihood that data is in cache. 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 consecutive accesses; suggesting that the data communicated has likely been evicted from the cache. An example of this can be seen with statement 7 in Figure 1. 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 effective; since auto-vectorization greatly depends on the algorithms used by the low level compiler.

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, optimiza-

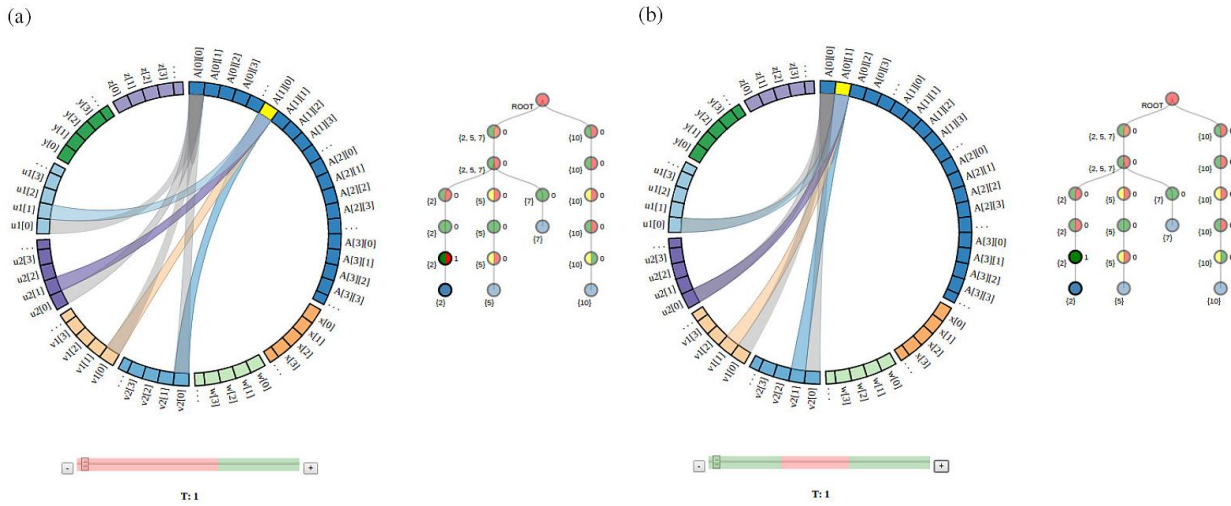

Figure 7: The chord view shows poor locality along accesses to the A array (a) and how permuting the inner loops of statement 2 leads to a better access pattern (b). The colored arcs show the elements accessed at timestep 1 while the gray, shadow arcs show the accesses of the previous timestep. Note that the leaf node labeled f2g and the node directly above it in the beta tree are opaque to indicate that the current timestep refers to the first iteration of the innermost loop of statement 2.

tion decisions can also be affected by changing the dependence graph view. 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. 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. 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 enables through the use of pragma directives. 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 good performance. 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. Figure 6 shows an example of how the dependence graph view highlights performance bottlenecks that can be fixed via the beta tree view.

Chord View

The chord view is a new type of visualization which shows the access pattern of the program via a chord diagram. A chord diagram is a visualization used to show relationships between entities. Arcs are drawn between nodes arranged on a circle to show that the nodes have something in common. For our purposes, the nodes represent array elements and the arcs represent memory accesses. The goal of this view is to help non-expert users better understand concepts like spatial and temporal locality and their impact on performance.

This view maps arrays to a circle as seen is Figure 7. The nodes are colored based on the array in which it belongs. Arcs are colored based on the array being read from at the current timestep. The node highlighted in yellow represents the array element that is written to in the current timestep (i.e. the write element). For each timestep, arcs are drawn from the write element to the array elements that are read. This shows, in a single view, the memory footprint of a single iteration of one statement.

A slider at the bottom allows the user to select the timestep shown in the chord diagram. Shadow arcs are drawn for the previous timesteps to help clarify the access pattern. Large gaps between the color and shadow arcs indicate poor locality of reference. In Figure 7(a), for example, the program suffers from a bad access pattern as seen by the large gap between the color and shadow arcs along array *A*. Regions of the slider bar are colored from green (i.e. good locality) to red (i.e. bad locality) based the access pattern of the timesteps they represent. Specifically, the color is selected from a linear scale based on the stride of the innermost loop. This helps users quickly identify the regions that need inspection.

The beta tree view is replicated and shown next to the chord view. In addition to allowing the user to perform transformations like fusion / fission and interchange, this beta tree is augmented to provide additional context to the chord view. Nodes that do not relate to the current timestep are made transparent while opaque nodes indicate which iteration of a statement is being executed at the current timestep. The integer to the right of the inner nodes of the beta tree show the current iteration of the loop it represents. Together, this implies a mapping from timestep to statement and loop iteration.

Because loops can be parametric, we cannot perform this type of static analysis for each iteration. Even for loops with a constant trip count, displaying a chord diagram for each iteration is impractical, since there may be thousands of iterations. The size of arrays can also be too large to display each element as a node in the chord diagram. To address this scalability issue, chord diagrams are only produced for the first four iterations of each loop. The idea is that only a few iterations of each loop is needed to give the user a good idea of the access pattern. This limits the number of timesteps for which we need to produce visualizations. Similarly, only the first four elements of each dimension are displayed for each array. This is followed by a special ellipsis array element to account for accesses that do not fall within the first four elements of a dimension. The ellipsis element acts as a representative for the rest of the array and accesses not within the first four elements have arcs drawn to this node.

Although locality of reference is represented elsewhere in PUMA-V, the chord view displays a finer level of detail than the other views can provide. Whereas the beta tree and dependence graph can highlight which statements have bad locality, the chord view shows which array accesses specifically are the problem. The benefits of fusion are also more clearly shown in this view, as users can see exactly which array elements are shared between statements. Conversely, it is also obvious when R-Stream fuses two statements that should not be fused. This is observed as a type of ping-ponging effect where different arrays are accessed at each timestep. With the chord view a user can rapidly identify a bad access pattern at a range of timesteps, determine which statement the timesteps refer to, and perform a transformation in the beta tree to achieve better locality.

Tile Size View

Tiling is an important transformation that groups iterations of statements together so that they can benefit from data reuse. The goal is to maximize the likelihood that the data has already been read into cache when it is needed again. This is especially useful for statements that have an undesirable access pattern that cannot be addressed through loop interchange. Tiling is a combination of a strip-mine and an interchange. A loop is first strip-mined by a number of iterations (i.e. the tile size), and then the strip-mined loop is sunk to the innermost position.

The tile sizes determine how many iterations are grouped together and have important impacts on performance. If the tiles are too large then performance will suffer from cache capacity misses. If the tiles are too small, then the cache may not be fully utilized. Tiling is performed by a tiling tactic in R-Stream where the tile sizes are selected based on a heuristic. In PUMA-V, tile

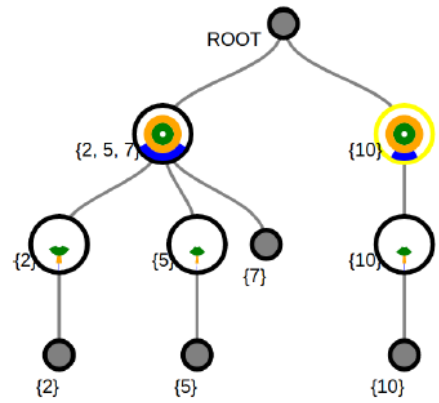

Figure 8: The tile beta tree view shows which loops will be tiled. The concentric arcs show the saturation of the three levels of cache based on the current tile size. The green, orange, and blue arcs refer to the L1, L2, and L3 caches respectively. The node outlined in yellow indicates that changes to the tile slider will affect the degree to which the corresponding loop will be strip-mined.

size selection is exposed to the user through a simple slider interface with a novel tree-based visualization to help guide him.

Figure 8 shows the tile size visualization. It is built from the beta tree prior to tiling. The large nodes represent loops that will be tiled. Clicking on any of these nodes switches the focus to the clicked node. A slider can then be used to set the tile size of the highlighted node. The nodes corresponding to tiled loops contain a visualization using three concentric arcs. These represent the saturation of the three levels of cache. The inner green arc represents L1 cache. The orange arc represents L2 cache and the outer blue arc represents L3 cache. Moving the slider changes each of the arcs. The appropriate level of cache is fully saturated when the arc completes to form a circle. This allows the user to make informed decisions about the size and shape of a tile.

The arc sizes within a node are computed based on the data footprint of the arrays accessed by the statements nested within the loop it represents. Specifically, for each statement, we take the image of the iteration domain under the access function of each array. Parametric upper bounds are added for each loop iterator. This produces a parametric footprint domain for each array. When the tile sizes are set, the parametric upper bounds are replaced with constants based on the tile sizes. The saturation is then computed by summing the area of each footprint domain. Since R-Stream performs rectangular tiling, the area is computed as the product of the length of each of the tile dimensions.

The arcs of a node at height *d* in the tree are drawn with respect to the first *d* dimensions of the tile. For example, the innermost node has a height of one in the tree and its arcs are drawn with respect to the first tile dimension. A node directly above it has a height of two and its arcs are drawn with respect to the first two tile dimensions. By changing the tile size of one node, all of its ancestors' arcs are updated as well. This lets the user see how each tile dimension adds data to the cache and set saturation limits based on the tile dimension.

Scalability

One potential weakness with the visualizations in this tool is scalability. As the number of statements increase, some of the views can become cluttered. The beta tree and dependence graph views are particularly vulnerable to visual clutter. A large number of statements can cause the beta tree to become too wide, thus preventing the entire tree from being displayed at once. The dependence graph view can become a tangled mess of nodes and edges where some dependences are occluded.

To address the issue of scalability, we include an additional "global view", shown in Figure 9(a). This view contains a single visualization of the beta tree. This view gives users a way to focus on

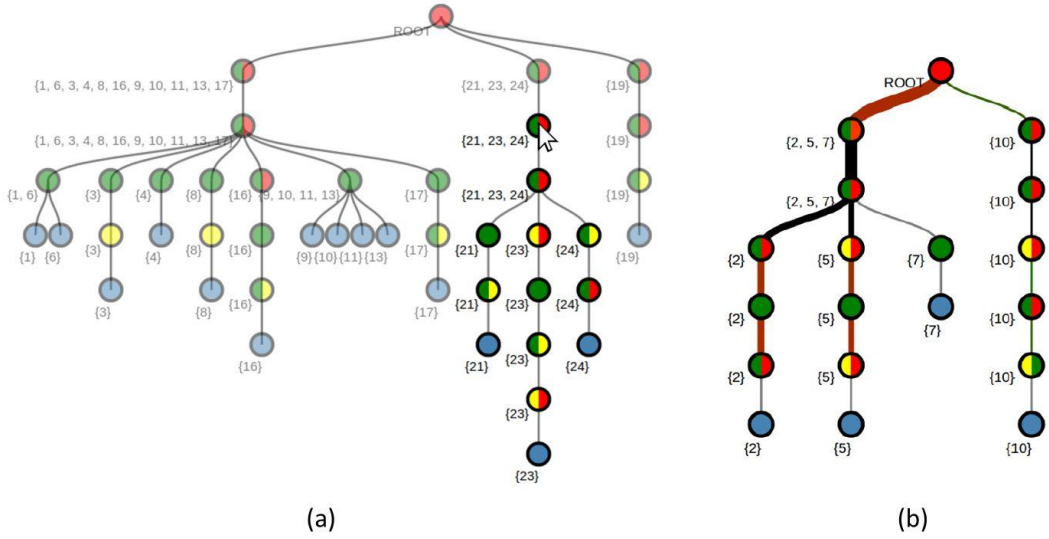

Figure 9: Beta-tree operations for scalability. (a) Clicking on the node in the global view will make all nodes transparent that do not contain statements 21, 23, and 24. The other views will be updated to show only these statements. (b) Updated beta tree view augmented with runtime performance data. The width of the first left edge is thicker than the right; indicating more time is being spent processing the left branch. Red branches indicate a high L2 cache miss rate.

a particular part of the program containing the performance bottleneck. By clicking on a node in the tree, the other views in PUMA-V will be filtered to only show the statements contained in the subtree rooted at the clicked node. This simple mechanism allows users to focus only on problem areas of the code by removing the visual clutter induced by insignificant statements.

Runtime Evaluation

The PUMA-V tool provides a means for gathering runtime performance data of the transformed code. In the upper left corner of Figure 1 is the “Runtime Evaluation” button. When clicked, C code is generated and the outermost loop in each loop nest is parallelized using OpenMP. The code is then compiled and executed. Performance data is gathered with the help of HPCToolkit and PAPI performance counters. The execution time is then reported and the beta tree visualization is updated to reflect the performance.

Figure 9(b) 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 auxiliary 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. 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 al-

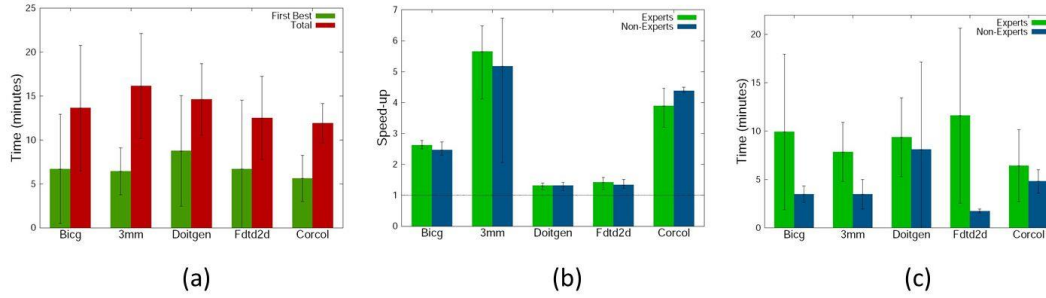

Figure 10: User study results. (a) Average time to the first transformation better than baseline R-Stream and the total completion time. Users spent an average of 7 minutes interacting with the tool before identifying a better transformation. Error bars show one standard deviation. (b) Average speed-up obtained for experts and non-experts. Experts achieved an average speed-up of 2.98 whereas non-experts obtained a speedup of 2.93. Error bars show the min and max speed-ups. (c) Average time to the first transformation better than baseline R-Stream for experts and non-experts. Non-expert users were able to identify better transformations twice as quickly, on average, compared to expert users. Error bars show standard deviation.

lows 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.

USER STUDY

We conducted a user study to evaluate the effectiveness of the PUMA-V tool. In this study, users were given full access to all features of the tool; including a number of polyhedral scheduling algorithms as well as classical compiler optimizations. The goal was to use the visualizations presented in this paper to apply manual transformations on several codes to optimize performance. The runtime of the user manipulated code was compared to that of the "baseline R-Stream" transformation for R-Stream version 3.15. Baseline R-Stream applies a set of default tactics without any options or additional optimizations. It is the transformation applied if R-Stream is run without any arguments.

In this experiment, we recruited six participants to perform semi-automatic optimization using the PUMA-V tool. All users had experience programming with the C programming language. Three of the users had a deep understanding of the polyhedral model and were considered experts. The rest of the users had no polyhedral background and were considered non-experts. The task was to optimize five separate programs of varying difficulty taken from the PolyBench benchmark suite v1.0 [7] which has code for matrix multiplications, stencil operations (derivatives, finite differences, etc.), BLAS (basic linear algebra subprograms), and others.

Performance was evaluated on a 4 core machine with Intel Core i5-2520M CPU @ 2.50GHz and 32KB of L1 cache, 256KB L2 cache, and 3MB L3 cache. Our hypothesis was that users would be able to find a better transformation compared to the baseline R-Stream by using the PUMA-V tool. Participants would start with a visualization of the baseline transformations made by the R-Stream compiler. They were allowed to use all features of the PUMA-V tool. There was a time limit of 20 minutes on each problem but users were allowed to withdraw at any time. A trial would end when the user withdrew, exceeded the time limit, or felt that no other useful optimizations could be made. The goal was to outperform the optimizations made by the baseline R-Stream. Each session lasted roughly two hours.

For each experiment we measured the execution time of the transformed code and the completion time. We also measured the amount of time that had passed until a transformation was found that outperformed the baseline R-Stream transformed code. This acted as a measure of the amount of effort required to outperform baseline R-Stream. At the end of the study, each user was asked to fill out a survey about their experience using the tool.

We observed an average speed-up of 2.9, with respect to execution time, across all experiments. In general, baseline R-Stream did a good job at exposing parallelism; but would often make the wrong transformation with respect to locality. In many examples R-Stream would fuse statements where fission and loop interchange yielded better performance.

Figure 10(a) shows the average completion time and the time to the first transformation that outperformed the baseline R-Stream. Users spent an average of 14 minutes on each problem. On average, the first transformation that performed better than baseline R-Stream was found after 7 minutes and users continued interacting with the tool for another 7 minutes. This was especially true in examples with multiple statements that shared a significant amount of data. Users would find initial success by performing loop permutation and spent the remaining time exploring the tradeoffs between fusion / fission and loop permutation.

Figure 10(b) compares the speed-ups obtained by expert versus non-expert users. Experts outperformed non-experts in nearly every example; but only marginally. Expert users had an average speed-up of 2.98 versus 2.93 for non-experts. Figure 10(c) shows the amount of time before users arrived at a transformation that outperformed baseline R-Stream. On average, experts took twice as long before achieving a speed-up.

We observed that all of the expert users spent a significant amount of time reading the code. These participants were more accustomed to reading the code generated by a polyhedral compiler. They knew what performance characteristics to look for in the generated code and wanted to gain a better understanding of the computation being performed. They were also more deliberate when performing transformations. Experts would look at the code view to understand how transformations like fusion or interchange affected the generated code and to verify that the changes were taking place as they intended.

Conversely, users who were not familiar with the polyhedral model relied mostly on the visualizations. These users would avoid reading the code and took a more exploratory strategy. They quickly became more comfortable with the visualizations and were able to identify the transformations needed to remove the bottlenecks conveyed by the visualizations. This generally led to performance improvements in less time compared to expert users who relied heavily on the code view. Although inspecting the code view was time consuming, it did provide some useful insights.

If several loops have a similar access stride, the difference in color might be too subtle to differentiate in the beta tree view. This would be the case if, for example, one loop yielded spatial locality (i.e. a stride of one) whereas another loop yielded temporal locality (i.e. a stride of zero). Inspecting the code view would be required to determine the optimal loop ordering. This could be shown in the beta tree by ranking the loops based on stride, and coloring the nodes based on this ranking. This remains an item for future work.

Five of the six users agreed that the PUMA-V tool improved their understanding of the transformations made by R-Stream. The most popular views were the beta tree and chord view. A majority of the transformations were performed through beta tree interactions (i.e. fusion / fission and permutation). This was largely due to the simplicity of the interactions and how they relate to changes in the code (e.g. interchanging nodes in the tree corresponds to interchanging loops in the code). Users also noted that the chord diagram was particularly useful in determining whether to fuse statements. Expert users utilized the tile size visualization to varying degrees. One expert, in particular, noted that the arc visualizations were especially useful in guiding tile size selection. All users agreed that the polyhedral view was the least useful because it lacked a mechanism for guiding transformations.

CONCLUSION

The study presented in this paper suggests that combining automatic methods with user intuition can lead to significantly better performance compared to automatic methods alone. We found that users were able to further exploit opportunities for optimization that R-Stream exposed with its initial transformations. In addition, our study also revealed some limitations in R-Stream's baseline transformations. Although R-Stream found optimal skewing / shifting and exposed

maximal parallelism, the baseline transformations occasionally focused too heavily on fusion at the expense of access stride. This suggested that the cost model used by baseline R-Stream was not evaluating some of these tradeoffs properly. This may be resolved by using different optimizations or auto-tuning, but PUMA-V allows the user to fine tune the performance in a visually intuitive way. With these observations in mind, our expert users agreed that PUMA-V has high potential to be a useful tool for compiler developers. It can be used to identify frequently missed optimizations that developers should address and enrich the compiler with new optimization strategies.

The chord view is currently limited to only showing the first four iterations of each loop. This is to avoid visual clutter associated with large or parametric loops. The dependence and access patterns, however, may change throughout the execution of a program. To remedy this, we intend to allow the user to choose which iterations to visualize. This can also be augmented with a mechanism to highlight iterations of a loop that have an "interesting" dependence or access pattern (i.e. the pattern is different from most of the iterations). This remains an item for future work.

Finally, in addition to optimizing performance, PUMA-V can also serve as an important educational tool for teaching the polyhedral model. Some experts in our study had suggested this. Students can get a sense of how to perform some basic optimizations, why they are important, and how they impact the generated code.

ACKNOWLEDGMENTS

This research was partially supported by NSF grant IIS 1527200 and DOE STTR Phase I/II grants DE-FOA-0000760/DEFOA-000101. Further support came from the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ICT Consilience Creative Program supervised by the IITP (Institute for Information and Communication).

SIDEBAR

Software Visualization

Visualization has long been used to make sense of program structure. Ball and Eick convey information about large programs through the use of pixel or line representations, where each line in the view represents a line of code.[8] Color is used to show code change history, compare programs, convey software complexity, and show execution hotspots where the programmer should focus his optimization efforts.

De Pauw et al. present a visual tool for exploring a JAVA program's runtime behavior.[9] A histogram shows CPU and memory resource consumption, which can also be used for hotspot detection. A graph view is available to convey references between objects which was found to be particularly useful for identifying memory leaks. An execution view that consists of color-coded stripes to show the execution time of different functions can help users identify bottlenecks. JaVis is another JAVA based software visualization tool.[10] It uses program traces to visualize and debug concurrent programs, with an emphasis on detecting deadlocks. UML diagrams are employed to visualize the traces.

Performance Visualization

Performance visualization is important for identifying bottlenecks and the best way to resolve them. VAMPIR is a performance visualization tool for analyzing MPI program traces. [11] It is capable of gathering and visualizing performance statistics and includes a zoomable timeline visualization for viewing problems at any level of detail. Jumpshot is another tool for visualizing MPI performance. [12] It is capable of detecting anomalous durations; thus bringing users' atten-

tion to problem areas of the program. A timeline view is used to show the state of the different MPI nodes over time. Histograms were also used to convey performance metrics.

The memory trace visualizer is presented by Choudhury et al.[13] This tool enables analysis of memory access patterns throughout program execution. Visual representations are presented for different cache levels. Color is used to indicate read/write operations as well as cache hits and misses. The tool also includes animation to convey cache accesses over time. Cache behavior is represented via colored glyphs placed on three concentric rings representing main memory, L2, and L1 cache.[14] Animation is used to show how data is read from main memory to L2 and L1 cache levels and eventually evicted from cache.

Polyhedral Model Visualization

Techniques for visualizing the polyhedral model typically involve visualizing the iteration domain The 3D iteration space visualizer visualizes the iteration domain and dependences of a loop nest.[15] Dependences are visualized as vectors, and provide a convenient way for users to identify and mark parallel loops. Tulipse is an Eclipse plugin that also includes a 3D visualization of the iteration domain and dependence vectors.[16] 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 is another Eclipse plugin for visualizing and tuning parallel programs.[17] This tool uses the CETUS compiler to automatically identify parallel loops.[18] 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 is an interface with multiple views for manipulating polyhedral transformations through visualizations.[19] 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. Our own work is a precursor of the system presented here.[20] This early prototype, however, lacks several important features and it also did not provide a user evaluation to gauge efficacy. For the current work, the beta tree view was augmented to convey array access stride and parallelism type simultaneously. Second, we created the chord and tile size visualizations to convey array access pattern and tile size/shape, respectively. Finally, a "global view" was added to allow the user to focus in on specific parts of the program being optimized.

REFERENCES

1. B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, A. Leung, and R. Lethin, "R-stream compiler," Encyclopedia of Parallel Computing, pp. 1756–1765, 2011.
2. A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine partitions," Parallel computing, vol. 24, no. 3, pp. 445–475, 1998.
3. C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," ACM Sigplan Notices, vol. 26, no. 7, pp. 39–50, 1991.
4. K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rostello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," ACM SIGPLAN Notices, vol. 49, no. 6, pp. 65–76, 2014.
5. J. Scott, "Automatic layout of metro maps using multicriteria optimisation," Ph.D. dissertation, Kent University, 2008.
6. M. Lanza and S. Ducasse, "Polymetric views-a lightweight visual approach to reverse engineering," IEEE Transactions on Software Engineering, vol. 29, no. 9, pp. 782–795, 2003.
7. L.-N. Pouchet. (2012) Polybench: The polyhedral benchmark suite. [Online]. Available: http://web.cse.ohio-state.edu/ pouchet/software/polybench
8. T. Ball and S. G. Eick, "Software visualization in the large," Computer, vol. 29, no. 4, pp. 33–43, 1996.
9. W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of java programs," Software Visualization, pp. 647–650, 2002.

10. K. Mehner, “Javis: A uml-based visualization and debugging environment for concurrent java programs,” Software Visualization, pp. 163–175, 2002.
11. W. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach, “Vampir: Visualization and analysis of MPI resources,” 1996.
12. O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward scalable performance visualization with jumpshot,” The International Journal of High Performance Computing Applications, vol. 13, no. 3, pp. 277–288, 1999.
13. I. Choudhury, K. C. Potter, and S. G. Parker, “Interactive visualization for memory reference traces.” Computer Graphics Forum, vol. 27, no. 3, pp. 815–822, 2008.
14. I. Choudhury and P. Rosen, “Abstract visualization of runtime memory behavior,” Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on, 2011.
15. Y. Yu and E. H. D’Hollander, “Loop parallelization using the 3D iteration space visualizer,” Journal of Visual Languages and Computing, vol. 12, no. 2, pp. 163–181, 2001.
16. Y. W. Wong, T. Dubrownik, W. T. Tang, W. J. Tan, R. Duan, R. S. M. Goh, S. hao Kuo, S. J. Turner, and W.-F. Wong, “Tulipse: a visualization framework for user-guided parallelization,” in European Conference on Parallel Processing, pp. 4–15.
17. I. Bluemke and J. Fugas, “A tool for supporting c code parallelization,” Innovations in Computing Sciences and Software, pp. 259–264, 2010.
18. C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, “Cetus: A source-to-source compiler infrastructure for multicores,” Computer, vol. 42, no. 12, 2009.
19. O. Zinenko, C. Bastoul, and S. Huot, “Manipulating visualization, not codes,” in International Workshop on Polyhedral Compilation Techniques (IMPACT), 2015.
20. E. Papenhausen, K. Mueller, M. H. Langston, B. Meister, and R. Lethin, “An interactive visual tool for code optimization and parallelization based on the polyhedral model,” in Parallel Processing Workshops (ICPPW), 2016 45th International Conference on, 2016, pp. 309–318.