Polyhedral User Mapping and Assistant Visualizer Tool for the R-Stream Auto-Parallelizing Compiler

Eric Papenhausen*, Bing Wang*, M. Harper Langston†, Muthu Baskaran†, Tom Henretty†, Taku Izubuchi‡, Ann Johnson†, Chulwoo Jung‡, Meifeng Lin‡, Benoit Meister†, Klaus Mueller*, Richard Lethin†
*Stony Brook University - State University of New York : {epapenhausen, wang12, mueller}@cs.stonybrook.edu
†Reservoir Labs, Inc. : {langston, baskaran, henretty, johnson, meister, lethin}@reservoir.com
‡Brookhaven National Laboratory : {izubuchi,mlin,chulwoo}@bnl.gov

***Abstract*—Existing high-level, source-to-source compilers can accept input programs in a high-level language (e.g., C) and perform complex automatic parallelization and other mappings using various optimizations. These optimizations often require trade-offs and can benefit from the user's involvement in the process. However, because of the inherent complexity, the barrier to entry for new users of these high-level optimizing compilers can often be high. We propose visualization as an effective gateway for non-expert users to gain insight into the effects of parameter choices and so aid them in the selection of levels best suited to their specific optimization goals.**

A popular optimization paradigm is polyhedral mapping which achieves optimization by loop transformations. We have augmented a commercial polyhedral-model source-to-source compiler (R-Stream) with an interactive visual tool we call the Polyhedral User Mapping and Assistant Visualizer (PUMA-V). PUMA-V is tightly integrated with the R-Stream source-to-source compiler and allows users to explore the effects of difficult mappings and express their goals to optimize trade-offs. It implements advanced multivariate visualization paradigms such as parallel coordinates and correlation graphs and applies them in the novel setting of compiler optimizations.

We believe that our tool allows programmers to better understand complex program transformations and deviations of mapping properties on well understood programs. This in turn will achieve experience and performance portability across programs architectures as well as expose new communities in the computational sciences to the rich features of auto-parallelizing high-level source-to-source compilers.

I. Introduction

As highlighted in [1], there is a constant need to optimize and parallelize codes for newer architectures. However, aside from utilizing the limited number of expert engineers, achieving optimal performance is often difficult, but sophisticated compilers can aid in optimizations and auto-parallelization.

High-level source-to-source compilers are powerful tools for generating optimized versions of complex input codes for specific architectures, or they can be utilized for translating codes from one hardware specification to another. These tools additionally often require users to be involved in the mapping process due to various parameter choices to guarantee optimal outputs. In order to address these issues for users, we have developed PUMA-V , which uses visualizations to guide users through the mapping process. In particular, we have built PUMA-V to be used with the R-Stream compiler, a source-to-source compiler based on the polyhedral model [2], a mathematical abstraction to represent and reason about programs in a compact representation [3]. We chose the R-Stream compiler for incorporation into PUMA-V due to its various strengths and its flexibility. In particular, R-Stream accepts input programs in a high-level language (e.g., `C`) and performs automatic parallelization and other mappings using optimizations framed in the polyhedral model and outputs a program to be processed by a low-level compiler. For example, R-Stream can produce output in `C+OpenMP`, which can be accepted by scalar compilers like `icc`, or in `CUDA`, which can be accepted by the Nvidia compiler. R-Stream performs optimizations such as parallelization, tiling, direct memory access (DMA) communication generation, distributed scratchpad memory management, and can target a range of symmetric, heterogeneous, and hierarchical targets under the control of a machine model description.

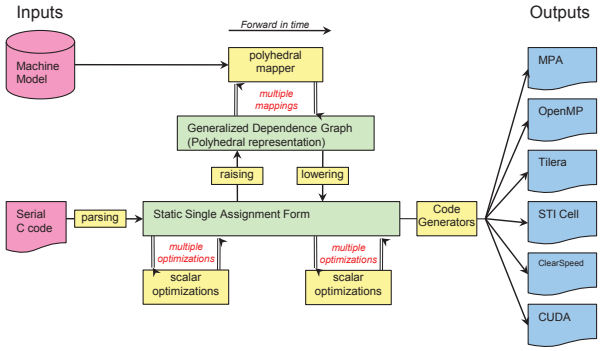

Fig. 1. R-Stream's architecture

The polyhedral representation of programs makes it possible to compactly construct a search space of all the legal loop transformations for a given program. The polyhedral model and R-Stream topics are too rich to fully cover here, so we give a brief overview in section II and refer to [3] for more detail and [2], [4] for the origins of the polyhedral model.

Source-to-source compilers such as R-Stream are powerful

This material is based upon work supported by the U.S. Department of Energy under Award Number DE-SC0009678.

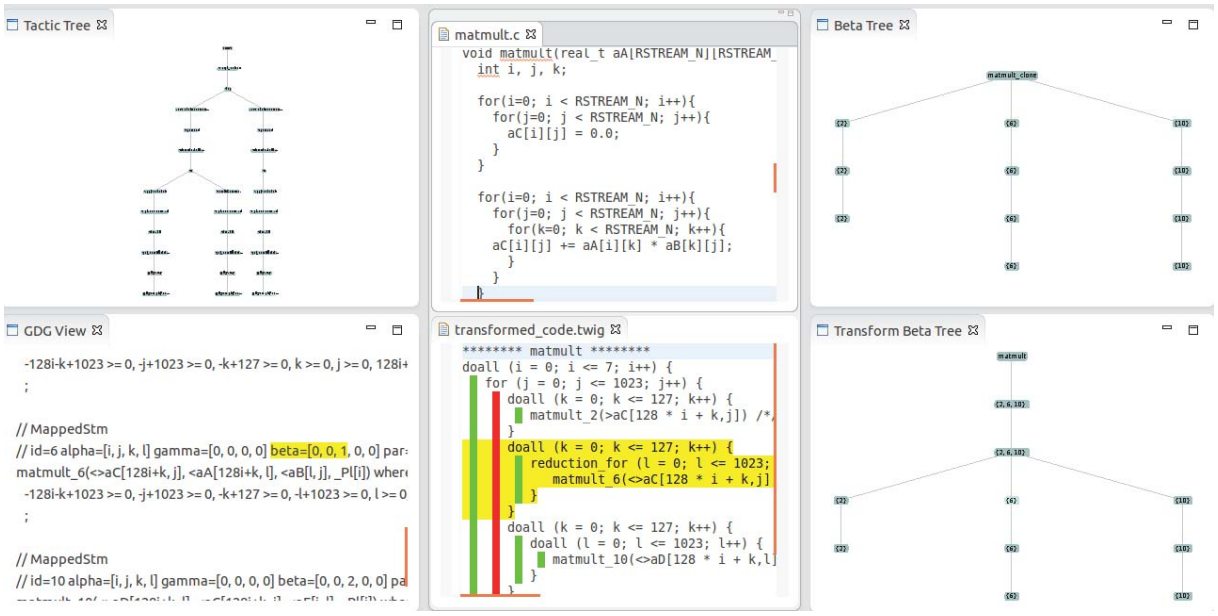

Fig. 2. ***Top Left***: The *Tactic Tree View* (*TTV*) lists transformations selected and applied by R-Stream. Multiple branches in the tree represent different potential sequences of transformations; ***Top Center***: the ***Source Code View*** (*SCV*) shows the original source code on which the transformations will be applied; ***Top Right***: the ***Beta Tree View*** (*BTV*) shows the "BetaTree" representation of the source code, the relative lexicographic nesting of each statement with respect to other statements. The leaves of the tree represent statements, and the inner nodes of the tree represent loops at various nesting levels. ; ***Bottom Left***: the ***GDG View*** (*GDGV*) shows a textual representation GDG for the transformed source code; ***Bottom Center***: the ***Transformed Code View*** (*TCV*) shows the resulting transformed code, a pseudo-code representation resulting from the current sequence of transformations. The number of vertical lines, covering the textual length of each loop, determine the nesting depth, and the lines are color coded to show the amount of parallelism at the corresponding loop (i.e, green for maximum parallelism, yellow for some, and red for none); ***Bottom Right***: the ***Transformed Beta Tree View*** (*TBTV*) shows the beta tree representation of the transformed code, often the most convenient form to view the effects of a transformation.

tools for accelerating runtimes; however, as mentioned, the complex nature of these tools are often a barrier-to-entry for new users. For example, simple matrix multiplication has 912 possible transformations. Visualization techniques can help new and experienced users to see the trade-offs for various optimization decisions made through an underlying cost model. In [5] a visualization framework utilizing the polyhedral model was developed to allow users to manipulate the iteration space of a program. Similarly, PUMA-V uses many state-of-the-art visualization techniques to make interaction with the polyhedral method more intuitive and interactive. With PUMA-V the user influences the optimization choices by modifying the cost model as opposed to in [5], where the user explicitly defines the transformation by altering the iteration space. Further, PUMA-V scales to deeper loop nests since the number of views increases by $\binom{N}{2}$ for a loop nest of depth N.

One technique employed by PUMA-V involves parallel coordinate visualizations [6], an approach that is used to represent high dimensional and multivariate data and is well-suited to analyzing the output from high-level compilers. Parallel coordinate visualizations represent data points as polylines crossing a series of parallel axes. Each axis represents one data dimension and the position where a polyline crosses the axis shows the value of that data point on this dimension. The high-dimensional space is unrolled into a serialization of axis pairs, giving good visual access onto the space. Parallel coordinates render complex datasets in a single 2D image, and with proper dimension reordering or range brushing, it can further reveal relationships between neighboring dimensions [7]. Parallel coordinates have been successfully applied in the area of visual clustering [8], high dimensional data analysis [7], finance [9] and even data generation [10]. In [11], parallel coordinates are used in the EPOsee software package for visualizing software development archives.

Section II briefly introduces components of the R-Stream compiler, followed by a deep discussion of the PUMA-V tool in section III. Section IV presents a user scenario for a typical input, including a powerful technique for visualizing performance effects of user decisions and how this guides the user to better decisions and ultimately accelerated codes.

II. R-Stream Polyhedral Model Compiler

While R-Stream handles high-level transformations, the resulting source code still needs to be compiled via a traditional *low-level compiler*. To utilize R-Stream, regions to be mapped are tagged with a simple pragma, a compiler directive in the source code, and R-Stream automatically determines a mapping based on the target machine and emits transformed code. R-Stream creates a polyhedral abstraction from the input source, and this abstraction is encapsulated by the **generalized dependence graph** (GDG). Most of the visualizations created within PUMA-V are generated from the GDG. Ultimately, changes the user makes through PUMA-V are translated to changes in the GDG, affecting the transformations that take

place within R-Stream. Fig. 1 shows where the GDG is located within R-Stream's pipeline.

In the polyhedral model, the problem of selecting which transformations to apply is translated to a linear programming problem. Certain criteria (e.g. execution time) are optimized subject to specified constraints (e.g. preservation of program semantics). These constraints and objective functions are encapsulated within the GDG and represented via a graph, where nodes represent strongly connected components (SCC) containing statements and edges indicate statement dependencies.

Since many factors affect the execution time of a program (e.g. utilization of cache and instruction throughput), a cost model is integrated into this graph to govern the desirability of certain transformations. Each node in the graph is augmented with an "execution cost" variable. This determines how desirable it is to parallelize the statement. The edges are augmented with a "fusion cost" variable which determines whether we should fuse the loops (where multiple loops are replaced or *fused* into a single loop) of those two statements. These costs are typically determined by R-Stream, but PUMA-V exposes these costs to the user; thus allowing him to indirectly affect the transformations chosen by R-Stream.

III. Visualizer for R-Stream Optimizations

As a popular open source IDE that facilitates the development of software in various languages and due to its strong track record for supporting the development of plugins, the Eclipse environment was chosen to augment R-Stream with visual capabilities for PUMA-V . The Eclipse plugin we have developed for PUMA-V extends the functionality of R-Stream by including visualizations to help the user better understand transformations and optimizations and allowing the user to guide them in a visually intuitive way.

PUMA-V presents the user with six views as shown in Fig. 2, providing an overview of the transformations while preventing the screen from becoming too cluttered. Many of the visualizations shown are used throughout the polyhdedral literature, and this "six view" representation has the additional benefit of linking and brushing between views. For example, in Fig. 2 the user has clicked on the first "6" node in the *Transform Beta Tree View* (*TBTV*), highlighting the corresponding loop in the transformed code and the corresponding beta coordinates in the *GDG View* (*GDGV*).

A. Interfacing with the R-Stream Transformation System

A list of transformations applied to a GDG can be visualized in the *Tactic Tree View* (*TTV*) as seen in Fig. 2 (*Top-Left*). Left-clicking on a transformation node triggers all the transformations to be run up to the selected node. This is particularly useful in understanding the transformation process by means of inspection of the BetaTree, GDG, and transformed code after transformations are applied. This process can be applied iteratively and results in a step-by-step analysis of the transformations performed by R-Stream.

Some transformation nodes have extra functionality. When selected, the *AffineScheduling* node opens a view listing the strongly connected components (SCC), on which the *AffineScheduling tactic* (*AST*) has operated. Fig. 3 shows the *SCC List View* (*SCCLV*), containing one graph per schedule dimension and a visualization of a selected SCC graph. Users can change the costs associated with each SCC node (i.e., parallelism score) and each edge (i.e., fusion score). Re-running the *AST*, adds a new branch to the *SCCLV* and updates other views to reflect changes.

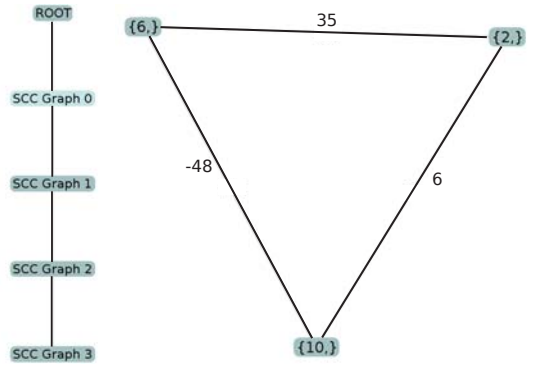

Fig. 3. ***Left***: *SCC List View* (*SCCLV*) of graphs; ***Right***: the corresponding SCC graph for "SCC Graph 0".

The ability to explore the solution space, through modification of the cost model, allows the user to develop intuitions about the legal transformation space for an application. This intuition is further assisted by the *Parallel Coordinates View* (*PCV*) and the *Correlation Graph View* (*CGV*).

B. Parallel Coordinates View and Correlation Graph View

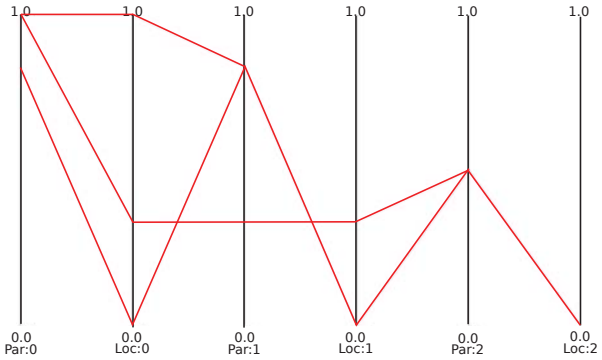

Fig. 4. The *PCV* shows the performance of different cost models: each line represents a mapping by R-Streamwith a different, user-defined, cost model.

PUMA-V uses parallel coordinates to visualize the effects on performance for a particular transformation. In the *PCV* each dimension represents either parallelism or locality at a particular level in the transformed program. This allows the user to view the various tradeoffs between parallelism and locality. An example of this within the visual plugin is illustrated in Fig. 4. In conjunction with the *SCCLV*, the *PCV* visually quantifies the effects of scheduling transformations, thereby providing a mechanism for systematically searching the space

of available transformations. A user scenario highlighting the effects of various transformations is presented in section IV.

The *CGV* acts as an auxiliary view to the *PCV*. In the correlation graph, vertices represent parallelism and locality at the various depths (i.e. dimensions in the *PCV*), whereas edges show how the parallelism and locality at the different depths interact with each other. An example is shown in Fig. 5, indicating that parallelism at depth 1 is negatively correlated with locality at depth 1 (edge colored in red).

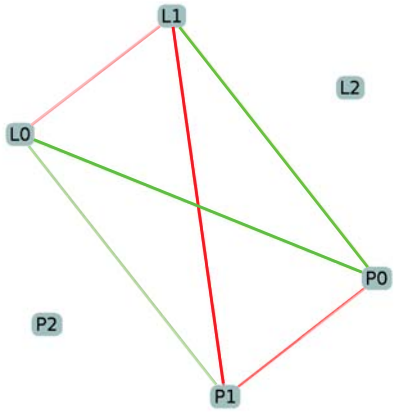

Fig. 5. The *CGV* shows tradeoffs between parallelism and locality at various levels. Red and green edges indicate negative and positive correlations, respectively, and color intensity is determined by correlation strength.

The benefits of these visualizations are best seen with PUMA-V's *randomized run* functionality, which invokes R-Stream multiple times. During each invocation, random values are assigned to the cost model, acting as a "first pass" to discover transformations R-Stream can produce for a given application. The user can then fine-tune the transformations, using the visualization as a guide.

IV. User Scenario

As a practical scenario, let us imagine a physicist, Tom, modeling particle interactions in a high dimensional space. During each timestep of the simulation, each particle's position is updated through multiplication by a matrix; formed by successive multiplications of rotation matrices. After some simple profiling, Tom has identified the function in Fig. 6 as a major bottleneck. Next, Tom opens the PUMA-V tool, loads the appropriate C file, and clicks the ***Apply*** button to run the default transformations. After the transformations are applied, each of the views from Fig. 2 is updated:

- ***TTV*** (*Tactic Tree View*) is updated, displaying the list of transformations where each node corresponds to a specific, applied transformation.
- ***BTV*** (*Beta Tree View*) is updated, displaying the relative nesting of statements in the input code.
- ***GDGV*** (*GDG View*) is updated, giving a more detailed textual description of the transformations.
- ***TCV*** (*Transformed Code View*) is updated, showing a pseudo-code description of the current transformation.

```
void matmult(real_t aA[n][N], real_t aB[N][N]
real_t aC[N][N], real_t aD[N][N]
real_t aE[N][N]) {
   int i, j, k;
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         aC[i][j] = 0.0;
   }}
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         for (k=0; k<N; k++) {
            aC[i][j] += aA[i][k]*aB[k][j];
   }}}
   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         for (k=0; k<N; k++) {
            aD[i][j] += aC[i][k]*aE[k][j];
   }}}
```

Fig. 6. Successive matrix multiply input source code to R-Stream.

- ***TBTV*** (*Transformed Beta Tree View*) is updated, displaying the nesting of statements in the transformed code.

Tom starts by clicking on nodes in the *TBTV*. By doing this, loops are highlighted in the *TCV*. This type of linking and brushing affect gives Tom a better understanding of the transformations that have taken place.

Next, Tom clicks on the **PerfVis** menu item, opening a parallel coordinates view with four dimensions (vertical lines). This view visualizes performance counter data, gathered from profiled executions of the transformed code. To generate this data, we used the HPCToolkit [12], a collection of tools to measure program performance, which uses PAPI [13] to obtain the hardware performance. The four dimensions of the parallel coordinates are Level 3 cache miss rate, Level 2 cache miss rate, Instructions per cycle, and execution time.

The first thing Tom notices when looking at this view is that it is devoid of polylines because no physical executions of the code have occurred. To gather performance data, Tom clicks the **Execute** menu item, and two polylines are drawn: one for the input code, and one for the default transformation. The lines drawn are shown in Fig. 8 as Blue for the original input code and Orange for default transformation.

Tom is pleased as the default transformation leads to a significant improvement in performance. He suspects, however, that he can do better. The L2 miss rate is quite high: the cost model may favor parallelism over locality, so Tom decides modify the cost model to favor locality.

First he clicks on the affine scheduling node (*as*) in the *Tactic Tree View*. This causes the *SCC List View* to open (Fig. 3 ***Left***); displaying a list of strongly connected components representing R-Stream 's cost model. By clicking on the first SCC node, a view displaying the SCC graph opens (Fig. 3 ***Right***), allowing him to affect transformations at the outermost loop level. To improve locality, Tom sets the weights on the edges of the graph to be very high. Tom does this by clicking on each edge in the graph, and setting the numerical value to 1000. The cost model will now heavily favor locality and fuse

loops wherever possible.

Tom then runs the transformation with the new cost model and clicks **Execute** to generate a new polyline in the parallel coordinate view (Green in Fig. 8). He is disappointed. The L2 miss rate was only reduced a small amount, while the total execution time has increased compared to the default transformation. To better understand what is happening he looks more closely at the *Transformed Code View*. A snippet of this view is shown in Fig. 7. Tom notices the locality of the array accesses of statement `matmult_6` shows B's access pattern is cache-inefficient since each iteration of the k loop requires a value that is N elements away.

```
doall (j = 0; j <= 1023; j++) {
   reduction_for (k = 0; k <= 1023; k++) {
      matmult_6(<>aC[i,j],<aA[i,k],<aB[k,j])
   }
}
```

Fig. 7. Snippet from the *TCV*.

Tom notices that the loops are parallel. He suspects that R-Stream has again favored parallelism over locality; this time at the innermost loop level. He goes back to the *SCC List View*, and this time he clicks on the last SCC node; allowing Tom to affect transformations at the innermost loop level. He clicks on each of the nodes in the corresponding SCC graphs allowing him to modify the execution cost of each statement. He does so by setting the cost to zero; thus greatly reducing parallelism at the innermost loop level. After running the new transformation, a new polyline is generated (Red in Fig. 8). Tom is thrilled to see that the L2 miss rate has been greatly reduced, resulting in reduced execution time as well.

V. Conclusion

We have presented the novel PUMA-V tool for visualizing complex transformations and optimizations performed by the high-level source-to-source R-Stream compiler, allowing a user-in-the loop to guide R-Stream to desirable results using intuitive visualization techniques. Through a user scenario we have further shown how PUMA-V allows a user to delve deeply into the finer details of compiler optimizations for a variety of machine models. To the best of our knowledge, this is the first interactive visualization tool allowing for direct manipulation of the polyhedral cost model.

The incorporation of HPCToolkit performance counters into PUMA-V has opened up a new avenue for visualization within our tool. Currently, we visualize performance counters at the function level. HPCToolkit, however, is able to gather performance counters at the loop, and even statement level. We are currently working to incorporate this finer level of detail, allowing the user to select a node in the beta tree, view its performance data, and thereby identify where the performance bottlenecks are in the application.

Future plans include exposing the user to more direct control over the output code. For example, allowing the user

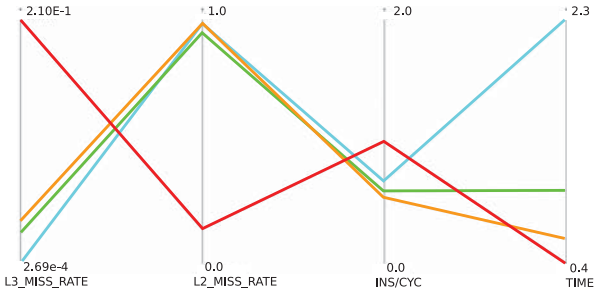

Fig. 8. *PCV* for performance evaluation of user-guided R-Stream transformations. ***Blue***: No transformations; ***Orange***: Standard R-Stream optimizations result in high $L2$ cache miss rate; ***Green***: Heavily favoring fusion in the cost model shows slight improvement in cache performance though the miss rate is too high; ***Red***: By discouraging inner loop parallelism through the *GCCV*, the user guides R-Stream to the desired result with PUMA-V .

to interchange loops by dragging nodes in the beta tree would permit imply interchanging loops without having to change the cost model. Additionally, we plan to test the tool more extensively with users (compiler experts and novices) , evaluate the performance gains from tuning optimizations identified through visualizations, and ascertain the level of background and training necessary to allow users to optimally tune their source codes.

References

[1] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" *Commun. ACM*, vol. 58, no. 5, pp. 77–86, Apr. 2015.

[2] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–348, Oct. 1992.

[3] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-stream compiler," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1756–1765.

[4] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Trans. Comp.*, vol. 54, no. 10, pp. 1242–1257, Dec. 2005.

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

[6] A. Inselberg, "The plane with parallel coordinates," *The Visual Computer*, vol. 1, no. 2, pp. 69–91, 1985.

[7] Z. Zhang, K. T. McDonnell, and K. Mueller, "A network-based interface for the exploration of high-dimensional data spaces," in *IEEE Pacific Visualization Symposium, PacificVis*, 2012, pp. 17–24.

[8] H. Zhou, X. Yuan, H. Qu, W. Cui, and B. Chen, "Visual clustering in parallel coordinates," *Computer Graphics Forum*, vol. 27, no. 3, pp. 1047–1054, 2008.

[9] J. Alsakran, Y. Zhao, and X. Zhao, "Tile-based parallel coordinates and its application in financial visualization," *Proc. SPIE*, vol. 7530, 2010.

[10] B. Wang, P. Ruchikachorn, and K. Mueller, "Sketchpadn-d: Wydiwyg sculpting and editing in high-dimensional space," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 19, no. 12, pp. 2060–2069, 2013.

[11] M. Burch, S. Diehl, and P. Weissgerber, "Visual data mining in software archives," in *Proc. of the 2005 ACM Symp. on Sof. Vis.*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 37–46.

[12] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[13] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.