Michelle Strout
November 22, 2005 and updated August 30, 2010

Review of *A Data Locality Optimizing Algorithm [WL91]*

What problem did the paper address? Who is the intended audience?
The big picture problem is how can we improve program performance given the large latency between the processor and memory. The audience is compiler researchers and writers because they are focusing on an existing compilation technique called tiling, which was developed to avoid memory access latency . The paper addresses the problem that not all loops are initially tileable. Specifically they answer the question, what combination of loop permutation, skewing, and reversal (unimodular transformations) should be used to make a loop tileable in a way that will improve data locality? The problem is hard. Just finding a legal set of unimodular transformations is exponential in the number of loops [SD90].

Is it important/interesting? What was the context for the paper?
The authors show that performance improvements due to tiling can be significant by presenting the speedup they observe with tiled matrix multiply. The matrix multiply computation is both legal and advantageous to tile in its original form. The authors report a speedup of 2.75 on a single processor of the CGI 4D/380 machine. Tiling matrix multiply also improves the scaling of the parallel version of matrix multiply. The un-tiled version experiences a speedup of 4.5 on 8 processors, and the tiled version experiences a speedup factor over 7.

This paper has been very influential (according to citeseer it has been cited 465 times). I think this paper has been so influential because it formulates tiling with rectangular tiles, generalizes unimodular transformations, and provides guidance for selecting unimodular transformations that enable tiling. Irigoin and Triolet [IT88] were the first to describe tiling, which they referr to as supernode partitioning (the term tiling was introduced by Wolfe [Wol89]). In the supernode paper, tiles were parallelpipeds. Since Wolf and Lam based their formulation on hyper-rectangular tiles, it makes code generation easier while not actually restricting tile shape (because it is possible to skew the iteration space). Unimodular transformation frameworks for doubly-nested loops were introduced by Banerjee [Ban90]. The Wolf and Lam paper generalizes unimodular transformations and perhaps more importantly provides guidance for selecting such transformations based on reuse analysis.

What is the approach used to solve the problem?
They create a unimodular transformation that results in loop experiencing reuse becoming fully permutable and therefore tileable. We summarize their approach using the following loop as an example:

```
for i = 0 to N
  for j = 0 to N
    A[i,j] = A[i-1,j] + A[i,j-1]
```

The reuse analysis first divides array accesses within the loop into uniformly generated reference sets, which are those accesses that differ only be a constant vector. In our example, all three array references are in the same uniformly generated reference set, where `A[`$H \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} c_i \\ c_j \end{bmatrix}$`]` and $H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

The next step in the reuse analysis is to calculate the self-temporal R_{ST}, self-spatial R_{SS}, group temporal R_{GT}, and group spatial R_{GS} reuse vector spaces. The self-temporal reuse vector space indicates which individual array references access the same location at different iterations in the loop. For our example, each array access reads or writes a different location at each iteration, therefore $R_{ST} = \emptyset$. Self-spatial reuse indicates that an array access is reusing data within a cache-line. For our example, $R_{SS} = ker H_s = ker \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = span\{(0,1)\}$. Group temporal reuse occurs when two different array accesses read or write

the same storage location in different iterations. For our example, $R_{GT} = span\{(1,0),(0,1)\}$. Finally, if two memory accesses refer to the same cache line in the same iteration then there is group-spatial reuse. In our example, there is group-spatial reuse between `A[i,j]` and `A[i,j-1]`, therefore $R_{GS} = span\{(0,1)\}$.

After reuse analysis, the authors apply the SRP algorithm to determine a unimodular transformation that will make all loops experiencing reuse tileable. In our example, both loops experience reuse and the loop nest is already tileable.

How does the paper support or otherwise justify the conclusions it reaches?
They implement their algorithm in the SUIF compiler. Using the SUIF compiler they generate serial and parallel code for the SGI 4D/380 for the following benchmarks: LU kernel on 1, 4, and 8 processors using a matrix of size 500x500 and tile sizes of 32x32 and 64x64 and SOR kernel on 500x500 matrix for 30 time steps. For SOR the wavefront version exploits fine-grained parallelism, but results in bad performance due to bad data locality. The 2D tiling for SOR results in better performance than wavefront, but doesn't exploit temporal reuse along the outermost time stepping loop. The 3D tile version results in the best performance.

What problems are explicitly or implicitly left as future research questions?
At the end of section 5, the authors suggest that the performance of their SRP algorithm could be improved with a branch and bound formulation.

Other research questions that arise from this paper are as follows:

- How should the tile sizes be selected? Here they select the tile size so that it is smaller than the cache size, and they apply copying to handle collisions in the cache. Specific guidance for the tile size and when copying should occur is possible future research.
- After performing tiling, what set of transformations should be performed to further improve performance? They perform some optimizations by hand such as inner loop unrolling, but they do not provide a model for guiding the application of such optimizations.
- What is the relationship between storage reuse, data locality, and parallelism? This paper assumes that the mapping of computation to storage is set. What if we were selecting the storage mapping as well as the schedule in order to maximize storage reuse and data locality and effectively exploit parallelism?

How does the paper address the questions we are asking about programming models this semester?
Many researchers in the parallel programming model communities have been recommending that programming models enable some way to express locality properties in algorithms [CPP02, NPP+00, TaMS+08]. Languages like X10 [CGS+05] and Chapel [CCZ04, CCZ07] have concepts like locales to do such a thing.

The paper being reviewed [WL91] is suggesting a technique that tiles loops to improve the data locality. Their approach was originally developed to be incorporated into a compiler and therefore not exposed to the programmer. Current research [HNP09, YSY+07, HCS+09] is investigating ways to expose performance transformations like tiling to the performance programmer. The approaches currently being proposed are along the lines of separate scripting languages and/or pragmas.

I think we should consider enabling transformations like tiling in the main programming model, but provide some form of encapsulation. Chapel iterators [JCD06] enable the orthogonal specification of a tiled loop schedule, but it places to much of the effort on the programmer. How could we provide orthogonal encapsulation of tiling while having the compiler do most of the difficult code generation work?

References

[Ban90] Uptal Banerjee. Unimodular transformations of double loops. In *Languages and Compilers for Parallel Computing*, August 1990.

[CCZ04] D. Callahan, B.L. Chamberlain, and H.P. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[CPP02] Barbara M. Chapman, Amit Patil, and Achal Prabhakar. Achieving performance under openmp on ccnuma and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14:713–739, 2002.

[HCS+09] Mary Hall, Jacqueline Chame, Jaewook Shin, Chun Chen, Gabe Rudy, and Malik Murtaza Khan. Loop transformation recipes for code generation and auto-tuning. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2009.

[HNP09] Albert Hartono, Boyana Norris, and Sadayappan Ponnuswamy. Annotation-based empirical performance tuning using Orio. In *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS) Rome, Italy*, May 2009.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages*, pages 319–329, 1988.

[JCD06] Mackale Joyner, Bradford L. Chamberlain, and Steven J. Deitz. Iterators in chapel. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, April 2006.

[NPP+00] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade;eacute;. Is data distribution necessary in openmp? In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC)*, page 47, Washington, DC, USA, 2000. IEEE Computer Society.

[SD90] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, 1990.

[TaMS+08] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors (MAW)*, pages 377–384, New York, NY, USA, 2008. ACM.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.

[Wol89] Michael J. Wolfe. More iteration space tiling. In ACM, editor, *Proceedings, Supercomputing '89, Reno, Nevada*, pages 655–664, Reno, Nevada, November 1989. ACM Press.

[YSY+07] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Proceedings of the Parallel and Distributed Processing Symposium*, 2007.