Compiler and Run-time Support for Improving Locality in Scientific Codes (extended abstract)

Hwansoo Han, Gabriel Rivera, Chau-Wen Tseng

Department of Computer Science, University of Maryland, College Park, MD 20742

1 Introduction

Modern microprocessors provide high performance by exploiting data locality with carefully designed multi-level caches. However, advanced scientific computations have features such as adaptive irregular memory accesses and large data sets that make utilizing caches effectively difficult. Traditional program transformations are frequently inapplicable or insufficient. Exploiting locality for these applications requires compile-time analyses and run-time systems to perform data layout and computation transformations. Run-time systems are needed because many programs are not analyzable statically, but compiler support is still crucial both for inserting interfaces to the run-time system and for directly applying program transformations where possible. Cooperation between the compiler and run-time will be critical for advanced scientific codes. We investigate software support for improving locality for advanced scientific applications on both sequential and parallel machines. We examine issues for both irregular adaptive and dense-matrix codes.

2 Irregular Adaptive Codes

Irregular adaptive codes are scientific applications with irregular and changing memory access patterns, where poor spatial and temporal locality can cause high cache miss rates. Unlike applications with regular memory access patterns, irregular applications are hard to optimize at compile time, since their access patterns are unknown until run time. They consist of lists of nodes and edges, computing values for each edge based on its nodes. Locality is improved by rearranging nodes and edges so that related nodes are placed nearby and accessed closely in time, increasing the probability they remain in cache. An inspector/executor paradigm can be used to optimize these irregular applications. The compiler inserts calls to an *inspector*, which analyzes memory access patterns and transforms data layout and computation order at run time. The transformed code, or *executor*, then performs the computation more efficiently. The overhead of the inspector is amortized in scientific applications performing computations for multiple time steps, since the inspector does not need to be re-executed each time.

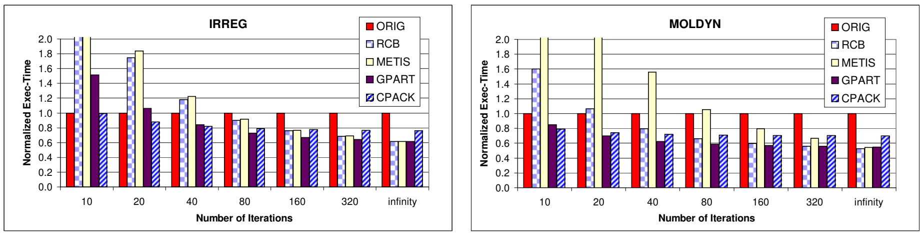

Fig. 1. Normalized Execution Time for Optimizations (ORIG = 1.0)

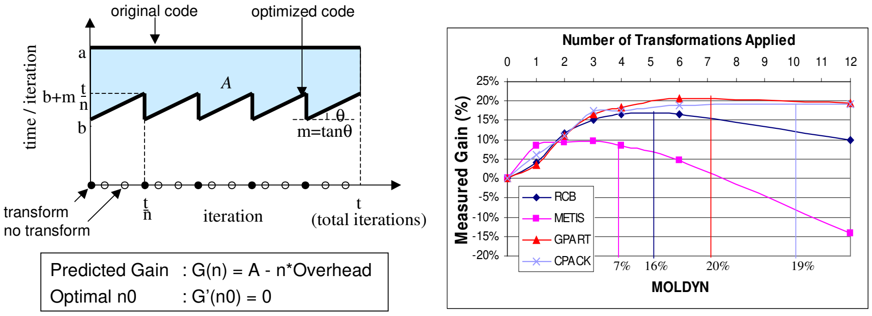

Fig. 2. Optimization for Adaptive Codes

We are investigating the effectiveness of a number of algorithms for improving locality. Simple sorting algorithms (CPACK) based on original memory access patterns reduce cache misses with low preprocessing overhead [2]. More sophisticated partitioning schemes based on geometric coordinate information (RCB) or multi-level graph refinement (METIS) yield greater locality, but with much higher overhead [1, 4]. We develop a graph partitioning algorithm (GPART) based on hierarchical graph clustering, which randomly combines nodes until partitions approach the L1 cache size, then lays out nodes based on partitions [3]. It achieves performance close to RCB and METIS with lower overhead. Figure 1 shows normalized execution times for 2 irregular kernels Irreg and Moldyn as the total number of iterations varies. Performance is improved by 30–50% for long-running applications. Complications arise for *adaptive* computations, where the access pattern changes. Benefits of locality transformations are reduced in proportion to the disruption, but reapplying transformations after every change is too expensive. By considering different overheads and benefits of each locality transformation, we developed a cost model to select the type and frequency of transformations [3]. Our model needs to run a few time steps to measure the benefit of transformations and the percentage of access pattern changes. After that, it chooses an algorithm and frequency of application for the best performance. The first graph in Figure 2 illustrates how to choose optimal number of transformations applied (n_0). The second graph in Figure 2 shows preliminary verification of the model, comparing experimentally measured gain (curves) with predicted gain (vertical bars) by our cost model.

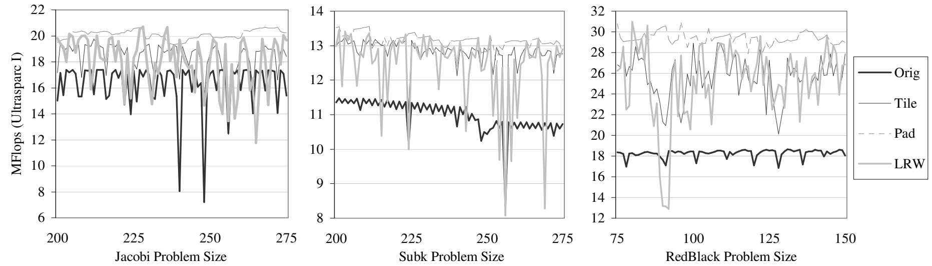

Fig. 3. Uniprocessor Performance of Tiled 3d Codes

3 Dense-Matrix Codes

Dense-matrix codes have simple, consistent memory access patterns. Locality is generally excellent, but reuse is lost for certain pathological (but common) problem sizes due to ping-pong or severe conflict misses. For instance, when unit stride accesses A(i,j) and B(i,j) map to the same cache set, spatial reuse across inner loop iterations is lost. Also, stencil codes exhibit group-temporal reuse across outer loop iterations between references typically of the form A(i,j), A(i,j+1). Often variables are laid out so that conflict misses prevent this reuse.

In earlier work, simple compiler padding techniques are shown to be effective in eliminating both forms of conflict misses [6, 7]. Generally only a few cache lines of inter-variable padding are needed to avoid ping-pong conflict misses. Larger pads (though still smaller than the L1 cache) are used to preserve group-temporal reuse. Existing inter-variable padding algorithms which solve this problem (and can improve performance of some SPEC95 benchmarks by up to 15%) target only the L1 cache.

We extend these techniques to avoid conflicts in additional levels of cache. Ping-pong conflict misses are avoided simply by using the largest cache line size when testing conflicts. When padding cannot preserve all group-temporal reuse on the L1 cache, we obtain reuse on the L2 cache by spacing out variables on this cache using inter-variable pads which are multiples of the L1 cache size. Such pad sizes will simultaneously preserve the best L1 layout while improving L2 reuse. Experiments show these enhanced techniques can avoid conflict misses and reduce miss rates, but actual performance improvements are not significant for current processor architectures. Overall, we find most locality optimizations do not need to explicitly target multi-level caches.

Tiling is a well-known locality optimization which reorganizes the iteration space into small tiles or blocks. In linear algebra codes, such as matrix multiplication, we execute the tiles repeatedly, improving data locality. When applied in conjunction with intra-variable padding or copying, tiling for the L1 cache has been shown to improve the performance of linear algebra codes by over 200% on some architectures. A possible modification to tiling is to select tile sizes in order to improve reuse at additional levels of the memory hierarchy. However, results indicate L1-sized tiles offer three or more times the performance improvement of L2-sized tiles, since they can improve both L1 and L2 cache performance.

Researchers have usually evaluated tiling for linear algebra codes, in which the simplicity of data dependences permits large performance improvements by targeting the L1 cache. However, for scientific computations such as 3D partial differential equation (PDE) solvers, tiling may also be applied to preserve group-temporal reuse where inter-variable padding fails due to capacity constraints. Figure 3 compares the performance of three tiling approaches with ORIG, the untiled version, as we vary the problem sizes of three 3D scientific codes. As in the case of linear algebra codes [8], we find combining intra-variable padding with tile size selection (PAD) to be more effective than tile size selection alone (LRW) [5] or selecting tile sizes irrespective of array dimensions (TILE). These results show an improvement of about 23% for Jacobi and Subk and 61% for RedBlack SOR as a result of tiling with padding.

4 Conclusion

Preliminary experimental results on representative program kernels show significant performance improvements are possible. However, much work remains to automate compile-time transformations and make run-time systems efficient for more realistic scientific applications. The goal is worth pursuing because it holds the promise of making high-performance computing more accessible for scientists and engineers.

References

1. M. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, 37(12):570–580, 1987.
2. C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proceedings of the SIGPLAN PLDI*, Atlanta, May 1999.
3. H. Han and C.-W. Tseng. Improving locality for adaptive irregular scientific codes. Technical Report CS-TR-4039, Dept. of Computer Science, University of Maryland, College Park, September 1999.
4. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proceedings of the 24th ICPP*, Oconomowoc, August 1995.
5. M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the ASPLOS-IV*, SantaClara, April 1991.
6. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN PLDI*, Montreal, June 1998.
7. G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *Proceedings of the ICS*, Melbourne, July 1998.
8. G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th Conference on Compiler Construction*, Amsterdam, March 1999.