Improving random walk performance

Ilya Safro[1], Paul Hovland[1], Jaewook Shin[1], and Michelle Strout[2]

[1] Argonne National Laboratory, USA, (safro,hovland,jaewook)@mcs.anl.gov*
[2] Colorado State University, USA, mstrout@cs.colostate.edu

Abstract. Random walk simulation is employed in many experimental algorithmic applications. Efficient execution on modern computer architectures demands that the random walk be implemented to exploit data locality for improving the cache performance. In this research, we demonstrate how different one-dimensional data reordering functionals can be used as a preprocessing step for speeding the random walk runtime.

1 Introduction

Random walk simulation is an important ingredient of many scientific applications. It is employed, for example, in data clustering [14], image segmentation [13], circuit clustering [3], and computation of different kinds of data similarities and scorings [9,18]. Usually, the relationships between the data elements in such applications can be modeled by a graph, either directed or undirected, with a relevant portion of information assigned to each node and edge. Typically, the simulation of a random walk consists of sequentially visiting a set of adjacent nodes and their neighborhoods within some small distance. The computational complexity of this process can be compared to that of pointwise relaxation methods such as Jacobi or Gauss-Seidel (improvement of which was studied in [21]).

For modern architectures on which random walk applications are executed, accessing main memory is an order of magnitude slower than accessing cache, which is smaller but faster memory closer to the processor. Thus, one should exploit cache as much as possible for efficient execution of such applications. For applications with regular memory accesses, a huge body of research work has been devoted to loop transformations targeting efficient use of cache [2,7,10,11,12,16,22,23], and dramatic performance improvement has been achieved. For applications with irregular memory accesses such as random walk, however, cache is not very helpful in improving memory access time. In [21], Strout and Hovland described runtime reordering transformations for data in memory and for iterations of loops for better use of cache in applications with irregular memory accesses. In [21] different metrics and reordering algorithms were proposed and tested. As part of this work, we have tried to improve their results. However, although the improved ordering reduces these metrics by up to 10-15%, we did not observe an

* This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

improvement in runtime when we applied state-of-the-art heuristics to transformations reordering for the finite-element instances from [21].

In this paper we demonstrate how the improvement of data locality can influence the performance of a random walk simulation, which typically represents a highly irregular part of scientific codes. In contrast to the approach proposed in [21], here the data reordering is done as a preprocessing of a particular application that uses random walk. Thus there is no need to know the runtime data access pattern but only the connectivity of the graph on which the random walk is performed.

2 Definitions and methods

Consider a random walk [17] on an edge-weighted directed graph $G = (V, E)$ with a weighting function on the set of edges $w : E \to \mathbb{R}^+ \cup \{0\}$. Denote by w_{ij} the nonnegative weight of the directed edge ij between nodes i and j. If $ij \notin E$, then $w_{ij} = 0$.

Random walk may be viewed as a process of sequential vertex visiting. Starting at node i_0, at the tth step of a walk, we move to i_t, one of the i_{t-1}'s neighbors, with probability

$$P_t = \frac{w_{i_{t-1}i_t}}{\sum_{i_t \in N(i_{t-1})} w_{i_{t-1}i_t}} \ , \tag{1}$$

where $N(i) = \{j \in V : \ ij \in E\}$. Because of model limitations, we cannot make changes in the reordering transformations as was done in [21]. Thus, only data locality improvements can be employed.

We have tested three data reordering metrics based on the p-sum functionals [15] with $p = 1, 2$ and ∞ and the workbound functional [4]. Let π be a bijection

$$\pi : \ V \ \longrightarrow \ (1, 2, ..., n) \ .$$

The following functional is minimized for the minimum p-sum problem over all possible permutations π:

$$\sigma_p(G, \pi) = \sum_{ij \in E} \left(w_{ij} |\pi(i) - \pi(j)|^p \right)^{1/p} \ . \tag{2}$$

In particular, we concentrate on the minimum bandwidth problem (when $p = \infty$), which seeks a linear layout that minimizes the maximal stretched edge, namely,

$$bw(G) = \min_{\pi} \max_{ij \in E} w_{ij} |\pi(i) - \pi(j)| \ . \tag{3}$$

The minimization functional of the workbound reduction problem is defined as

$$wb(G, \pi) = \sum_{i} \max_{\substack{j \\ \pi(j) < \pi(i)}} w_{ij} (\pi(i) - \pi(j))^2 \quad . \tag{4}$$

While the 1-sum functional is known metric for data locality improvement algorithms [21], other functionals were not used for this purpose. In particular,

we are interested in the workbound functional which represents a combination of two p-sum functionals and may be viewed as a measure for minimizing the maximum edge length per vertex.

Because of the NP-hardness of minimization of the above functionals, we used four multilevel solvers [20] for approximate minimization of the functionals. These solvers provide results that are at least comparable to the best-known heuristics, while keeping linear running time. In addition, we tested the Cuthill-McKee bandwidth reduction algorithm, which produces results of significantly poorer quality (in comparison to many other state-of-the-art heuristics [20]) but is extremely fast and easily implementable. We used the symmetric version of the reverse Cuthill-McKee solver implemented in Matlab as function 'symrcm'.

3 A random walk simulation algorithm

Algorithm 1 shows simulation of a random walk that we tested. In general, the tight upper bound of the random walk cover time is very big [8] to perform a full simulation, and we used a restart strategy to cover fully each graph faster. The algorithm consists of 1000 restart iterations of a random walk in order to ensure better coverage of graphs in case some of them contain hidden 'highly connected' components, that is, the components that ensure with high probability that the hitting time inside them is much smaller. Each sweep of a random walk consists of $100|V|$ steps, and in general we did not observe a situation when the graph was not covered by 1000 sweeps. The calculation performed at each step of the random walk is computationally comparable to one step of the Gauss-Seidel process.

Input: graph $G = (V, E)$
for $i = 1$ **to** 1000 **do**
  $\forall\, i \in V$ define $w_v = rand()$;
  $\forall\, ij \in E$ $w_{ij} = rand()$;
  $t = 0$;
  $i \leftarrow$ randomly chosen vertex;
  while $t < 100|V|$ **do**
    $w_i = w_i + \sum_{ij} w_{ij} w_j / \sum_{ij} w_j$;
    $i \leftarrow$ randomly chosen vertex among the neighbors of i with probability P_t;
    $t = t + 1$;
  end
end

Algorithm 1: Random walk with 1000 restarts

In many implementations a graph model-based data structure contains an appropriate amount of data at each node or edge. Two data allocation models were tested in order to compare the improved data locality reordering schemes

and metrics. In the first *homogeneous* allocation model ($\mathcal{HM}$) the amount of data (in bytes) stored at each node is slightly bigger than the maximum node degree on G, and there is no difference between the memory performances on high- and low-degree nodes. Low-degree nodes were artificially filled by useless information and this information was accessed while visiting these nodes. The second model, called *nonhomogeneous* ($\mathcal{NHM}$), allows one to keep only a few numbers of double precision, i.e., the most of the memory space allocated at each node is occupied by the pointers and information on its neighbors. In both models, the access time of a next neighbor node was $O(1)$, and the graphs were stored in continuous segments of memory without accessing a hard disk.

4 Experimental results

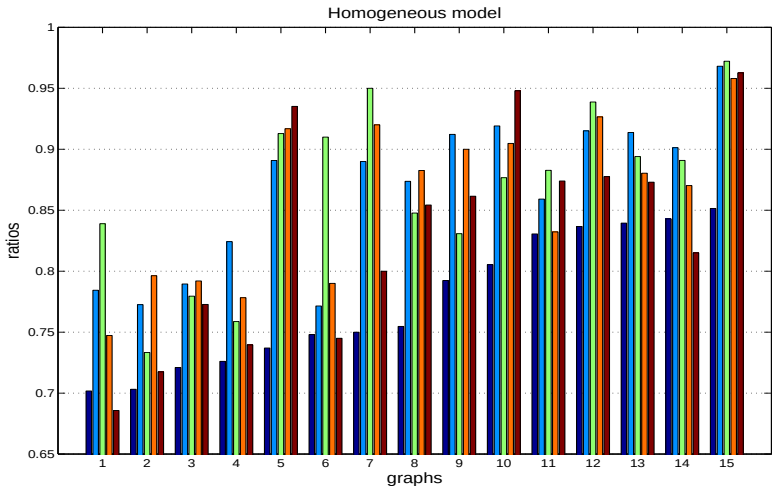


Fig. 1. Homogeneous model

To estimate how the runtime of a random walk behaves with the improved data locality, we compared the runtime of a random walk with and without reordering. The experimental results are summarized in Figures 1 and 2 that represent the comparisons of the $\mathcal{HM}$ and $\mathcal{NHM}$ models, respectively.

We chose 15 graphs (see Table 1) of different size and structure from [5] and measured the runtime of the same random walk on the randomly disordered graphs, originally ordered, and on the ordered by five previously mentioned heuristics. All experiments were performed under operating systems Linux and Windows XP. In general, no difference was observed between these two cases.

Each 5-tuple of bars (presented in Figures 1 and 2) corresponds to the five ratios between the runtime measurements of ordered and disordered graph, re-

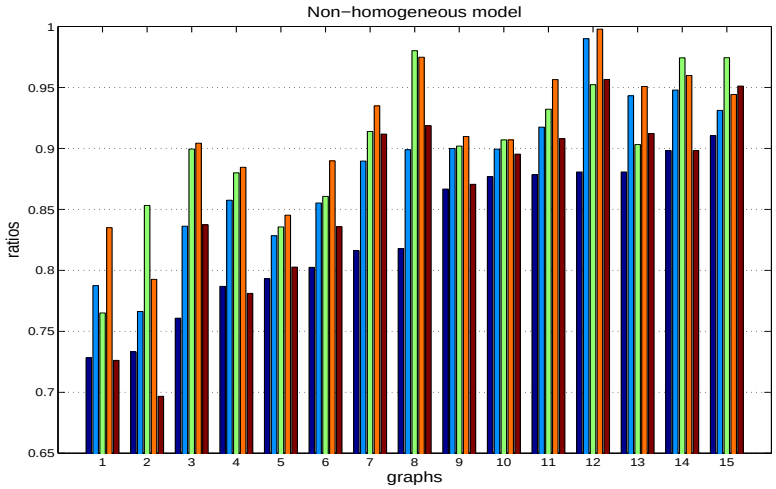


Fig. 2. Non-homogeneous model

Graph	$\|V\|$	$\|E\|$	min deg	max deg	avg deg
diw0079	11821	22516	1	4	3.81
horse	48485	145449	3	16	6.00
f117	48518	293581	3	39	12.10
nasa4704	4704	50026	5	41	21.27
torso	168930	1254712	6	46	14.85
finan512	74752	261120	2	54	6.99
onetone2	36057	201173	1	66	11.16
plgr10000_2	8467	13661	1	77	3.23
net100	28084	966960	2	180	68.86
bcsstk30	28924	1007284	3	218	69.65
fxm4_6	18892	239476	3	308	25.35
memplus	17753	41534	1	352	4.68
3dtube	45330	1584144	9	2363	69.89
p2p_7	11174	23409	1	2389	4.19
gupta3	16783	4653322	32	14671	554.53

Table 1. Experimental graphs

spectively. The first (2nd, 3rd, 4th, and 5th) bar correspond to the ratio between the runtime of the ordered graph by applying 1-sum (workbound, 2-sum, bandwidth, and Cuthill-Mckee) algorithm and the runtime of the disordered graph.

The most successful metric for the data locality improvement for a random walk was the 1-sum functional with the corresponding minimization algorithm [19]. The next most competitive heuristic was the Cuthill-Mckee algorithm which also produces the 1-sum orderings at compatible costs for graphs 1, 2, 4, 6, and 14. However, there are several graphs with big ratio gaps in favor of the 1-sum functional. Although the Cuthill-Mckee algorithm is fast and can be easily implemented, it has a serious disadvantage: its sensitivity to the initial vertex ordering it starts from can lead to many unexpected problems. On the other hand, the runtime of 1-sum heuristics [19] is also linear. Thus, we conclude that having a good-enough algorithm for the minimization of 1-sum functional can lead to the best results among the tested functionals.

Experimenting with power-law graphs. Graphs with vertex degree sequence proportional to the power-law distributions model many real-life processes, such as Internet connectivity, biological networks, and various social networks. All previously mentioned techniques of data reordering were applied to a set of ten power-law graphs of different sources. The differences in the observed improvement among five reordering methods were not as big as they were for other general graphs. Moreover, it is almost impossible to distinguish between three $1(2,\infty)$-sum minimizers and Cuthill-Mckee algorithm, which exhibited an improvement between 0.80 and 0.83 in terms of ratios in Figures 1 and 2. The improvement obtained by the minimum workbound solver was less significant: 0.85 on average.

Experimenting with original orderings. To complete the comparison of different orderings, we examined the original graph orderings. In practice, the original ordering of a graph depends on the database. However, typically, applications use their own graphs that are created by local connection ordering (BFS- or DFS-based orderings), that is, according to the order of vertex creation. This ordering can be much worse (in terms of minimizing all previously mentioned metrics) if the graph (the finite element or another structure) is created by a parallel algorithm when different portions of data from different regions are saved in parallel. Indeed sometimes this ordering can have even worse minimization results than randomized expected costs. Such orderings can be suitable when the graphs have very strong local, but not global, connectivity. The situation with local connection orderings turns out to be much worse when the graph has a structure that is not very similar to the finite element, for example when there are global connections that can destroy the beauty of local connection orderings. In general, in this paper a randomized ordering is used as a more meaningful upper bound than different local ordering methods (see [6]) that have a big variability. However, we compared our results with those by original ordering schemes and observed that the difference between runtime after a reverse Cuthill-McKee algorithm and randomized ordering and between runtime obtained by original ordering and randomized ordering is about 30%. Thus,

according to our observations, the reverse Cuthill-McKee algorithm represents the best version among different original local orderings.

4.1 Discussion

To understand the experimental result shown in Figure 2, we performed the same experiment again but on another machine and measured hardware performance counters using PAPI [1]. The machine has 2.5 GHz AMD Phenom processors, and the memory hierarchy incorporates three levels of cache (64 KB L1 data cache, 512 KB L2 cache, 2 MB L3 cache) and 4 GB of memory. Level 1 cache is fastest and closest to the processor but smallest whereas level 3 cache is largest in size but slowest next to the main memory. Another important part of memory hierarchy is *translation lookaside buffer* (TLB), which keeps the mapping between virtual addresses issued by the processor and the corresponding physical addresses. If the mapping is not found in TLB, the page table in main memory has to be referenced and the mapping is brought into TLB similar to the way data cache works. The machine we used in this experiment has 48 entries at L1 data TLB and 512 entries at L2 data TLB.

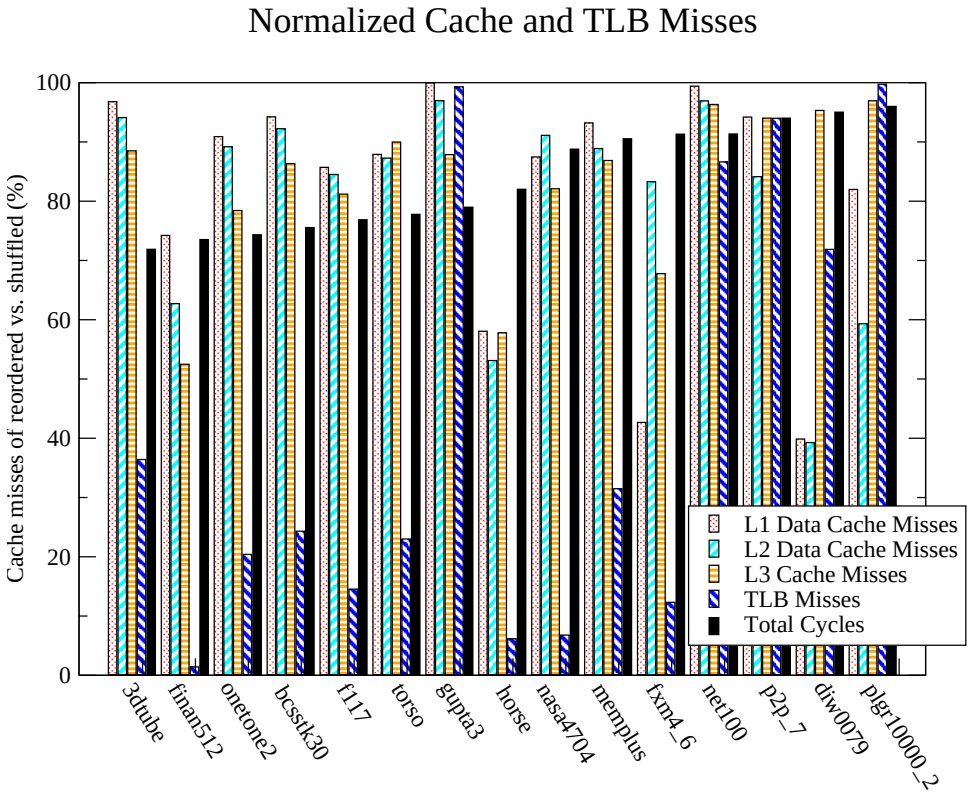
Figure 3(a) shows cache and TLB misses for the reordered graphs normalized by the corresponding misses for the randomly shuffled graphs. The graphs on the x-axis are sorted in decreasing order of performance improvements (or increasing order of normalized runtimes) of the reordered graphs over the shuffled ones so that general trends are easier to see. In all cases, cache and TLB misses have decreased for the reordered graphs explaining the performance improvements. Also, for the larger performance improvements in the left side of the graph, the reductions in TLB misses are larger, with the exception of `gupta3`.

Since our graph reordering improves the data locality, we checked the relationship between the performance and working data size. Figure 3(b) shows graph sizes in memory normalized by the biggest size for `gupta3` and `L1Sum`s (1-sum functionals) normalized by the `L1Sum`s of the unoptimized graphs. In general, for the larger speedups on the left side of the graph, graph sizes are larger and `L1Sum`s are smaller. We expect that the benefit of the technique proposed in this paper increases as the graph size gets bigger, machine's cache size gets smaller, and the normalized L1Sum is smaller.

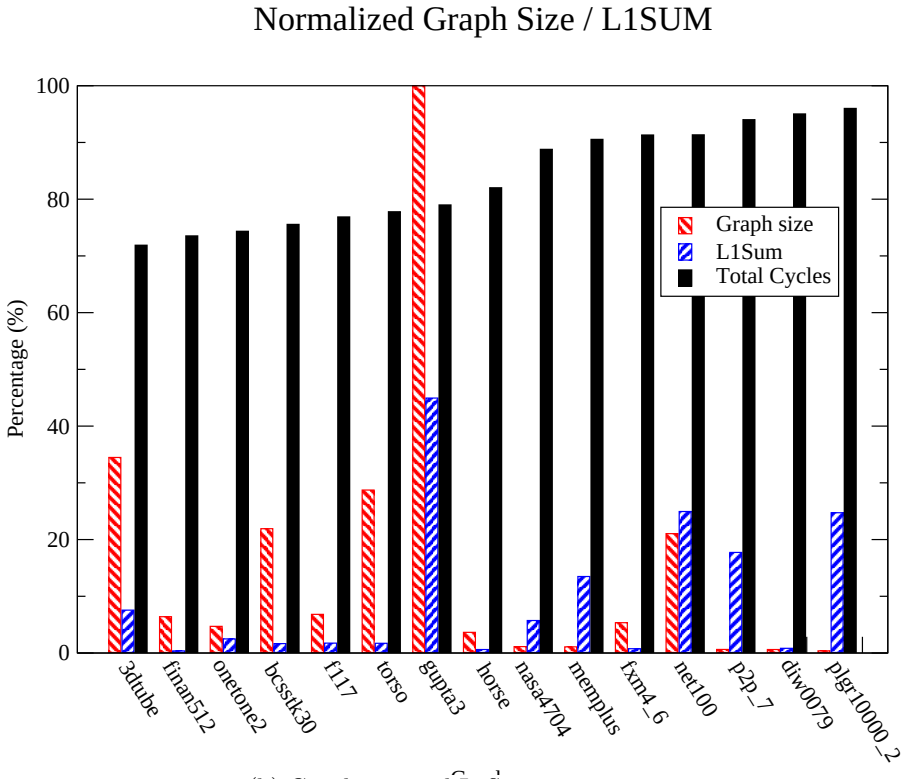
5 Conclusions

Our research demonstrates how to improve the data locality for a random walk simulation, which can be a highly irregular part of scientific code. In particular, we conclude that minimizing a 1-sum functional at the preprocessing stage can lead to the best performance results. On the other hand, a reverse Cuthill-McKee reordering scheme can produce slightly longer runtime simulation but its implementation is extremely easy.

The main difference between qualities of the reverse Cuthill-McKee and multilevel solvers is that the multilevel solver allows to improve a global ordering



(a) Cache and TLB misses



(b) Graph size and L1Sum

Fig. 3. Analysis of the speedups for L1Sum.

rather than the local only. Thus, if the experimental instance has high local connectivity only then Cuthill-McKee algorithm is preferable for use, otherwise it is very likely to obtain much better runtime with a multilevel 1-sum solver.

References

1. S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
2. Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
3. J. Cong, L. W. Hagen, and A.B. Kahng. Random walks for circuit clustering. In *Proc IEEE Conf. on ASIC*, pages 14.2.1–14.2.4, 1991.
4. Gianna M. Del Corso and Francesco Romani. Heuristic spectral techniques for the reduction of bandwidth and work-bound of sparse matrices. *Numerical Algorithms*, 28(1–4):117–136, December 2001.
5. T. Davis. University of florida sparse matrix collection. *NA Digest*, 97(23), 1997.
6. J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.
7. Karim Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.
8. U. Feige. A tight upper bound on the cover time for random walks on graphs. *Random Struct. Algorithms*, 6(1):51–54, 1995.
9. Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):355–369, 2007.
10. Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995.
11. Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
12. Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, October 1998.
13. Leo Grady. Random walks for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1768–1783, 2006.
14. David Harel and Yehuda Koren. Clustering spatial data using random walks. In *Knowledge Discovery and Data Mining (KDD'01)*, pages 281–286, 2001.
15. Martin Juvan and Bojan Mohar. Optimal linear labelings and eigenvalues of graphs. *Discrete Appl. Math.*, 36(2):153–168, 1992.
16. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.
17. L. Lovasz. Random walks on graphs: A survey. *Bolyai Soc. Math. Studies*, 2:1–46, 1993.

18. Augusto Pucci, Marco Gori, and Marco Maggini. A random-walk based scoring algorithm applied to recommender engines. In *WEBKDD*, pages 127–146, 2006.
19. I. Safro, D. Ron, and A. Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *Journal of Algorithms*, 60(1):24–41, 2006.
20. I. Safro, D. Ron, and A. Brandt. Multilevel algorithms for linear ordering problems. *J. Exp. Algorithmics*, 13:1.4–1.20, 2008.
21. Michelle Mills Strout and Paul D. Hovland. Metrics and models for reordering transformations. In *MSP '04: Proceedings of the 2004 workshop on Memory system performance*, pages 23–34, New York, NY, USA, 2004. ACM.
22. Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *ACM International Conference on Supercomputing*, Portland, OR, November 1993.
23. Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.