Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences

Ziang Hu Juan del Cuvillo Weirong Zhu Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716, U.S.A
{hu,jcuvillo,weirong,ggao}@capsl.udel.edu

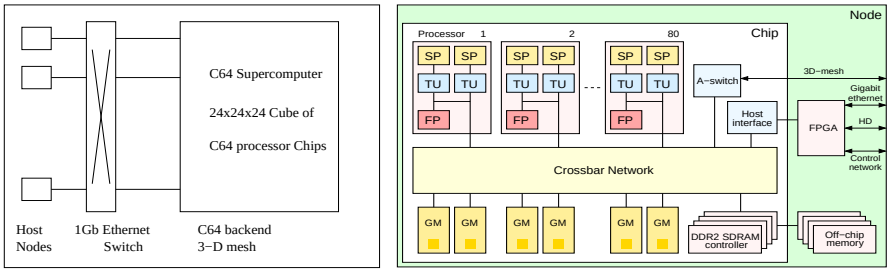
Abstract. This paper presents a study of performance optimization of dense matrix multiplication on IBM Cyclops-64(C64) chip architecture. Although much has been published on how to optimize dense matrix applications on shared memory architecture with multi-level caches, little has been reported on the applicability of the existing methods to the new generation of multi-core architectures like C64. For such architectures a more economical use of on-chip storage resources appears to discourage the use of caches, while providing tremendous on-chip memory bandwidth per storage area.
This paper presents an in-depth case study of a collection of well known optimization methods and tries to re-engineer them to address the new challenges and opportunities provided by this emerging class of multi-core chip architectures. Our study demonstrates that efficiently exploiting the memory hierarchy is the key to achieving good performance. The main contributions of this paper include: (a) identifying a set of key optimizations for C64-like architectures, and (b) exploring a practical order of the optimizations, which yields good performance for applications like matrix multiplication.

1 Introduction

Cyclops-64 (C64) [1, 2] is a petaflop supercomputer project under development at IBM. As shown in Figure 1(a), a C64 system is built from thousands of C64 chips that employ a unique multiprocessor-on-a-chip design. Each chip consists of 160 thread units and the same number of SRAM memory banks connected by an on-chip crossbar network (see Figure 1(b)). C64 chip architecture features massive intra-chip parallelism and on-chip memory bandwidth (320GB/s). Given such a novel architecture, the challenge is how to use these two features to obtain high sustained performance for scientific and engineering applications.

During the past two decades, there has been a considerable amount of work on how to optimize dense matrix applications on shared memory architectures with multi-level caches. However, it is not clear whether the existing methods are applicable to the new generation of multi-core architectures, such as C64.

This paper presents an in-depth case study of how a collection of well known optimization methods can be applied to address the new challenges and opportunities that the emerging class of multi-core chip architectures may present. The *phase ordering*



(a) Cyclops-64 Supercomputer (b) Cyclops-64 Chip Architecture

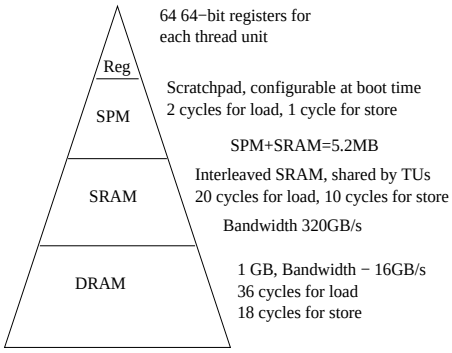
Fig. 1. Cyclops-64 Architecture

of different optimizations has long been challenging but interesting research problem that still remains open [3]. Furthermore, previous work [4], which has established the optimization order for cache-based architectures, may or may not be applicable to a cacheless architecture like C64. In this work, we apply specific optimizations following the order dictated by our experience and knowledge of the problem at hand. However, we do not in any way claim that this order is optimal. Our goal is to demonstrate that overall, for a given dense matrix operation, it is possible to derive a good order of optimization. We hope that the experience reported in this paper will prove to be useful for developers, in designing compilers and runtime systems for C64-like multi-core architectures.

2 Cyclops64 chip architecture

The work described in this paper focuses on a single C64 chip [1, 2], the main component of a C64 node (see Figure 1(b)). Within a C64 chip there are 80 processors, each consisting of two thread units, a floating-point unit, and two SRAM memory banks of 32KB each. Hence, the total on-chip memory is approximately 5MB. A 32KB instruction cache, not shown in the figure, is shared among five processors.

At boot time, SRAM banks are partitioned into two segments. One segment contributes to the globally shared interleaved on-chip memory. Processors and interleaved memory are logically arranged in a dancehall configuration with processors and memory banks on opposite sides connected by a one-level crossbar switch. The other segment, called scratchpad memory (SPM), is regarded as local memory since the corresponding thread unit has fast access to its own SPM. The C64 architecture also provides four DRAM controllers. Each one is attached to a 256MB bank, hence a C64 node features 1GB off-chip DRAM. As a summary, Figure 2(a) reflects the current size, latency (when there is no contention) and bandwidth of each level of the memory hierarchy. The C64 instruction set architecture incorporates efficient support for thread level execution, hardware barriers, and atomic in-memory operations.



(a)

Memory	Cycles	Load Delay	FLOPS
SPM	13378	0	306 M
SRAM	128154	73728	32 M
DRAM	193690	139264	21 M

(b)

Fig. 2. Cyclops-64 Memory Segments and Baseline Numbers

3 The Problem and Experimental Method

This paper is a case study of square matrix multiplication (MM), which is a widely used computation kernel for scientifc and engineering applications. For our baseline, we choose a straightforward implementation of the sequential algorithm. To parallelize matrix multiplication, we partitioned the three matrices into t^2 blocks and we assign each thread unit the computation of a number of such blocks. The computation of a $\mathbf{C}_{m,n}$ block requires t block multiplications and additions according to the following expression:

$$\mathbf{C}_{m,n} += \sum_{k=0}^{t-1} \mathbf{A}_{m,k} \times \mathbf{B}_{k,n} \tag{1}$$

To exploit spatial locality, it is best to assign the calculation of $\mathbf{C}_{m,n}$ to a single thread, as the resultant matrix block does not need to move around.

To study matrix multiplication on the C64 architecture we used the FAST simulator [5]. FAST is an functionally-accurate simulator that, among other features, models the memory hierarchy of C64 architecture, including the latencies and bandwidth of each memory segment.

4 Evolutionary Performance Tuning

In this paper, 128×128 and 256×256 matrix multiplications are simulated on up to 68 thread units. The former ensures the matrix fits into on-chip memory, the latter forces some blocks of the matrix to be stored in DRAM. However, the results hereby presented can be extrapolated to larger matrices.

The study begins with the sequential version, where the code always resides in off-chip DRAM, and data is placed in each of the three memory segments one at a

time. We compare the performance and memory latencies of the three cases. Then a straightforward parallel version of the MM is introduced, with data stored in SRAM and DRAM, respectively. In the following sections, we improve the performance of the parallel implementation and measure the effectiveness of various optimizations.

4.1 Sequential Matrix Multiplication

We start by comparing the performance achieved by the sequential implementation, with matrices placed in SPM, interleaved SRAM, and DRAM, respectively. SPM size is quite limited. In addition it holds both the runtime stack and thread-private data. Hence, the maximum size allowed for each matrix is 16×16 only. The results from this experiment are shown in Figure 2(b).

It is apparent that the performance difference comes from the latency incurred by load operations accessing different memory segments. We may conclude that data should always be loaded into SPM first before starting the computation. However, data needs to be loaded from SRAM/DRAM into registers first and stored into SPM afterwards on this architecture. If data reuse rate is low, it is not worth performing this "prefetching". Therefore, data reuse is a key issue for achieving high performance on C64. Matrix multiplication has the potential for high data reuse as the memory size is O(n^2) and computation is O(n^3).

4.2 Matrix Multiplication Parallelization in On-chip SRAM

We implemented a straightforward parallel version, which places three 128×128 matrices into interleaved SRAM. This version will be used as the baseline version for performance comparison. The matrices are partitioned into 8^2 blocks, each with 16×16 size. At most 64 thread units are used in this experiment, as there are 64 blocks in total. Thus, it is natural to assign one resultant matrix block to each thread, as well as all the computation for that block. We encapsulate the computation for one resultant block into one task. Also notice that the resultant block can be reused 8 times while the other blocks are used only once for each task. A task array is employed to store the tasks. Each task consists of a pointer to the resultant block, and two arrays of pointers that point to 8 pairs of source blocks.

Each thread tries to obtain the next available task from the task pool. When successful, it performs the computations, writes the resultant block back, and attempts to get a new task until the task pool is empty. The result is shown in Table 1. Although we get near linear speed up, the overall performance is still low - up to 1.3GFLOPS for 64 threads - 4% of the peak performance (32GFLOPS with 64 thread units).

Next, we will study a sequence of optimizations to improve the parallel performance.

Using SPM The next step is to use the SPM as a high speed buffer to accelerate the corresponding thread unit in the computation. We still perform the 16×16 matrix multiplication in SPM. The matrices are copied into SPM block by block. The computation is conducted and the result is stored back into SRAM. It is worth copying the resultant

Table 1. Baseline Parallel Version

Num of Threads	Cycles	FLOPS	Speedup
1	93,435,509	22.5M	1.00
2	46,750,840	44.9M	2.00
4	23,413,382	89.6M	3.99
8	11,783,500	178.0M	7.93
16	5,942,832	352.9M	15.72
32	3,207,410	653.9M	29.13
64	1,627,767	1.3G	57.40

block into SPM as it will be used 8 times. Since the two source matrices are only used once, they are not copied into the SPM. Implementing this yielded 1.79GFLOPS. This represents a 38% performance improvement over the base version (See "Using SPM" in Table 2).

Table 2. 128x128 MMM Incremental Optimizations in SRAM

Optimizations	GFLOPS	Speedup Over Baseline Parallel Version	Speedup Over Sequential Version	Incremental
Baseline	1.29	1.00	40.31	0%
Using SPM	1.79	1.38	55.94	38%
Tiling+Unrolling	2.77	2.15	88.56	55%
Reg. Tiling	5.05	3.91	157.81	82%
Inst. Sched.	10.02	7.77	313.12	99%
Reg. Alloc.	11.03	8.55	344.69	10%
Sync. Opt.	13.70	10.61	428.12	24%

Loop Tiling and Unrolling Loop tiling is a very effective optimization for architectures with caches. The tile size is chosen to allow all the data accessed by the inner most tile to fit into the cache. For matrix multiplication, the 16×16 matrix is split into two levels of 4×4 tiles.

A simple tiling does not bring performance gain as the number of branch instructions and code size are increased. By unrolling the next level of inner loops, 2.77GFLOPS, which is a 55% improvement over "Using SPM", is achieved.

Register Tiling (Manually) For the inner most 3 loop nests, there are total $4 \times 4 \times 3 = 48$ data elements that can fit into 64 registers of C64. The data reuse rate is 4 for each element of A and B, and 32 for C.

Because of the current limitation in the compiler, we manually did the register tiling by allocating registers properly to the data elements of the 3 matrices, as well as other index variables. Those elements are used in the 2 inner most loop nests, with A and B inside and C one level outside. After manually performing register tiling and allocation, the optimized code achieved 5.05GFLOPS, which is an 82% improvement over the simple tiling plus unrolling.

Instruction Scheduling (Manually) After register tiling, by properly scheduling the instructions in the innermost loop, we can hide the latencies of most memory and floating point operations and achieve 10.02GFLOPS - another 99% improvement over the register tiling. By moving accesses to C outside of the inner most loop, the performance reaches 11.03GFLOPS.

A good instruction scheduler is very important to the MM application as well as other programs. The key issue is that the scheduler should be aware of the different latencies when accessing different memory segments (SPM, SRAM and DRAM). Most existing compilers assume cache latency when they do instruction scheduling. For this architecture, there is no data cache and each load/store may have different latency depending on the target memory segment. Explicit multi-level memory hierarchy aware instruction scheduling is a key optimization for the C64 architecture. In fact, loop tiling, register tiling and instruction scheduling have to be tightly coupled, and the aggregation of the 3 optimizations is the key to generate optimal code for even a simple matrix multiplication.

Remove Unnecessary Synchronization In all the above experiments, mutex is used to control the access to the task pool. When one thread is getting a task from the task pool and updating the status of the allocated task, all other threads have to wait for the release of the mutex lock.

Since MM is a regular application, an alternative approach is to statically assign workload, i.e., each thread is assigned to a fixed number of tasks. As a result, the mutex lock is not needed. After removing the mutex, we get 13.70GFLOPS, which is 42.8% of the potential peak performance (32GFLOPS for 64 threads).

All of the above results are based on the assumption that 3 matrices are stored into on-chip SRAM. The memory bandwidth (320GB/s) is enough to sustain the computation. However, when the matrices become larger and larger such that they cannot be stored into on-chip SRAM, bandwidth of DRAM becomes a major issue. In the next section, we are going to investigate bandwidth optimizations to bring high performance to the algorithm assuming that data resides in off-chip DRAM.

4.3 Parallelizing Matrix Multiplication in DRAM

Off-chip DRAM is the largest memory resource of the C64 architecture. Most data and code will be stored there for real applications. On-chip SRAM and SPM are smaller and more expensive resources, and should be used more carefully.

To demonstrate the optimizations, we use 256×256 matrices that need to be stored in DRAM with 128×128 sub-banks buffered in SRAM. Therefore, the application has

to move data between DRAM and SRAM. In this section we study the impact of DRAM bandwidth limitation on the application's performance and how to tackle this problem by hiding the communication latency between DRAM and SRAM with computation. A nice feature of C64 is that thread units are not expensive - there are very many of them. On-chip memory resources are more expensive. We can use a set of thread units to do the computation and another group of thread units to move data between DRAM and SRAM. In this case study, we use two sets of SRAM banks (double buffering). One set for computation and another set for preloading, and switch between them during the computation.

DRAM Bandwidth For the first version of C64 chip design, the DRAM can transfer at most 32 bytes every cycle. Hence, the total DRAM bandwidth is 16GB/s.

To make the best utilization of the DRAM bandwidth, load multiple and store multiple (of 8 doublewords or 64 bytes) instructions should be used and the starting address should be 64 byte aligned.

Bandwidth limitation is the major challenge here. For 128×128 matrix multiplication, the total number of memory accesses is $128 \times 128 \times 128 \times 8 \times (3 + 1)$ bytes (3 loads, 1 store), or $67,108,864$ bytes. Then, the ideal access to memory time is $67,108,864/32$, or $2,097,152$ cycles. Even excluding load/store conflicts and ignoring other instructions, the peak performance can only be 1GFLOPS.

We may assume the C array is loaded and stored in the second innermost loop. The total bytes to be accessed becomes $128 \times 128 \times 128 \times 8 \times 2 + 128 \times 128 \times 8 \times 2$, or 33,816,576 bytes. In this case, the ideal performance increases to 1.98 GFLOPS. But we are still far from the peak performance (32GFLOPS for 64 threads).

This means that we have to use on-chip SRAM and/or SPM to buffer matrix blocks, perform the computation in SRAM/SPM, and store the results back to off-chip DRAM. In other words, we have to reduce the DRAM bandwidth requirements via the on-chip data reuse.

Using LDM and STM One optimization is to use LDM and STM instructions to aggregate multiple memory accesses. Four LDD (load doubleword) are combined into one LDM and four STD are combined into one STM. Hence, DRAM requests are effectively reduced to 1/4 of its original number, and DRAM bandwidth has been better utilized here. The best case is to combine 8 LDD into one LDM and 8 STD into one STM. But for register tiling, 4x4 is a better choice. If we do 8x8, although we can load sub-blocks into registers, we cannot consume them and have to store them into on-chip memory. This is not good for matrices A and B.

Using On-chip Memory To reduce the bandwidth requirement to DRAM, we try to move sub-blocks of matrices into SRAM, and move intermediate results back to DRAM whenever it is necessary. We also pipeline the process by using two SRAM blocks for each matrix: one for computation and the other for load/store.

In this study, we assume the original size of the three matrices is 256×256 and they reside in DRAM. The on-chip block size is 128×128. Each matrix has two blocks in

SRAM and half of each is loaded into SRAM. We assume $c1$ and $c2$ for matrix C, $a1$ and $a2$ for A, and $b1$ and $b2$ for B. While one set of SRAM blocks is used for computation, the other set can be used to load or store . The pipeline is designed as follows:

Computation Threads	Memory Access Threads
	load c00 (to c1), a00(a1),b00 (b1)
compute c00/a00/b00 in c1/a1/b1	load a01(to a2) b10 (to b2)
compute c00/a01/b10 in c1/a2/b2	load c01(to c2) b01 (to b1)
	store c00
compute c01/a00/b01 in c2/a1/b1	load b11(to b2)
compute c01/a01/b11 in c2/a2/b2	load c11(to c1) a10 (to a1)
	store c01
compute c11/a10/b01 in c1/a1/b1	load a11(to a2)
compute c11/a11/b11 in c1/a2/b2	load c10(to c2) b00 (to b1)
	store c11
compute c10/a10/b00 in c2/a1/b1	load b10(to b2)
compute c10/a11/b10 in c2/a2/b2	
	store c10

Fig. 3. Execution Steps When the Matrices are in DRAM

The total DRAM accesses: $128\times128\times8\times$(4 loads of C+4 stores of C+4 loads of A+ 6 loads of B) $= 2,359,296$ bytes. The ideal DRAM access time in this case is $73,728$ cycles, which is equivalent to 56.9GFLOPS without considering other computations.

Synchronization Overhead To implement the above pipelined scheme, a barrier is inserted at the end of each step. There are 12 barrier invocations in the implementation. This guarantees that computation happens after loading all the required data, and storing follows the corresponding computation stage. C64 has hardware barrier support with low cost. A barrier can be completed in as little as dozens of cycles.

Optimized memcpy() The standard C library features an optimized version of memcpy(), which is up to 20 times faster than the initial straightforward implementation. It takes into account possible unalignment at the source and destination, as well as different copy lengths. It is also capable of pipelining the three basic stages: loading from the source array, address computation and storing into the destination array.

Using More Threads for Load/Store In previous sections, only one thread handles the work of loading and storing. To further improve the performance, we assign three more threads, four in total. Three threads are responsible for preloading each of the three matrices, and the main thread handles the task pool creation and stores the resulting sub matrices back to DRAM.

The final result we achieve is $1,206,048$ cycles and 13.9 GFLOPS for a 256×256 problem size, which is 43.4% of the peak performance (we use 68 threads in this case: 64 threads for computation, 4 threads for load/store).

Table 3. Optimizations for Matrices in DRAM

Optimizations	Size	Cycles	Mem/Delay	FLOPS	Speedup
No Opt	128	6,499,276	5,401,783	322.7M	
No Opt	256	42,078,325	35,060,687	398.7M	1.00
LDM/STM	128	1,745,340	1,439,301	1.2G	
LDM/STM	256	13,996,754	11,652,068	1.2G	3.00
All Opt	256	1,206,048	810,997	13.9G	34.86

5 Conclusions

Our results demonstrate that efficiently exploiting the multi-level memory hierarchy is the key to achieve good performance on C64. When data fits into SRAM, tiling, loop unrolling, register allocation, and instruction scheduling are the most important optimizations. SPM can also be used to buffer frequently accessed data. When data does not fit in SRAM, DRAM bandwidth becomes the bottleneck. To overcome this issue, first we use SRAM to buffer blocks of DRAM data, which additionally reduces the bandwidth requirements to DRAM. Second, we overlap DRAM accesses with computation in SRAM to dramatically improve the performance.

For compiler designers, inner most register tiling is very important. The instruction scheduler should be aware of the latency for each memory segment. High level loop optimization should be able to automatically choose SPM buffers for SRAM data and/or SRAM buffers for DRAM data.

6 Related and Future Work

Locality optimizations have been studied by numerous researchers which resulted in many publications on cache-based architectures. Loop transformations have been investigated to exploit computation parallelism and data locality for scientific applications [6–11]. Loop tiling is a well known loop transformation to increase cache locality (see [12, 7, 9, 13, 14] and their references). We use loop tiling (and register tiling) to map a matrix block into the register file, SPM, and SRAM. Bandwidth optimization has also been extensively explored in [15–21] and their references. Indeed, we have shown that an efficient utilization of the memory bandwidth is critical for C64 when data is stored in DRAM. Phase order problem has been studied in [4, 3] and their references. We identify a set of useful optimizations for C64-like architectures. Moreover, we explore a practical sequence order of optimizations for the matrix multiplication that yields 14GFLOPS.

As future work we intend to the study other representative benchmarks. The identified optimizations will be implemented in the C64 compiler. Traditional loop optimizations may be extended to support automatic storage and thread unit management by allocating SPM and SRAM to the hot data at certain computation phases, and automatically overlap memory transfer with computation.

Acknowledgments

We acknowledge support from IBM, in particular, Monty Denneau, Henry Warren, José Castaños, and Christos Georgiou, ETI, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. Thanks to many CAPSL members for helpful discussions, in particular, Fei Chen, Brice Dobry, Geoff Gerfin, Ge Gan, Wesley Toland, and John Tully.

References

1. Denneau, M., Warren, Jr., H.S.: 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, Yorktown Heights, NY (2005)
2. Denneau, M., Warren, Jr., H.S.: 64-bit Cyclops principles of operation part II: Memory organization, the A-switch, and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, NY (2005)
3. Almagor, L., Cooper, K.D., Al., E.: Finding effective compilation sequences. In: LCTES'04, Wahsington, DC, USA (2004)
4. Wolf, M.E., Maydan, D.E., Chen, D.K.: Combining loop transformations considering caches and scheduling. In: Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, IEEE-CS TC-MICRO and ACM SIGMICRO (1996) 274–286
5. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In: Workshop on Modeling, Benchmarking and Simulation (MoBS'05) of ISCA'05, Madison, Wisconsin (2005)
6. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kauffmann Publishers (2001)
7. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario (1991) 30–44 *SIGPLAN Notices*, 26(6), June 1991.
8. Carr, S., McKinley, K.S., Tseng, C.W.: Compiler optimizations for improving data locality. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society (1994) 252–262 *Computer Architecture News*, 22, October 1994; *Operating Systems Review*, 28(5), December 1994; *SIGPLAN Notices*, 29(11), November 1994.
9. Anderson, J.M., Lam, M.S.: Global optimizations for parallelism and locality on scalable parallel machines. In: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, New Mexico (1993) 112–125 *SIGPLAN Notices*, 28(6), June 1993.
10. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing, Boston, MA (1995)
11. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris (1997) 201–214
12. Wolfe, M.: Iteration space tiling for memory hierarchies. (SIAM) Parallel Processing for Scientific Computing (1987) 36–361
13. Andonov, R., Bourzoufi, H., Rajopadhye, S.: Two-dimensional orthogonal tiling: from theory to pratice. In: HiPC 1996, Trivandrum, India (1996)

14. Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers (2000)
15. Calder, B., Krintz, C., John, S., Austin, T.: Cache-conscious data placement. In: Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society (1998) 139–149 *Computer Architecture News*, 26, October 1998; *Operating Systems Review*, 32(5), December 1998; *SIGPLAN Notices*, 33(11), November 1998.
16. Ding, C., Kennedy, K.: Improving cache performance in dynamic applications through data and computation reorganization at run time. In: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia (1999) 229–241 *SIGPLAN Notices*, 34(5), May 1999.
17. Kennedy, K., Kremer, U.: Automatic data layout for distributed memory machines. ACM Transactions on Programming Languages and Systems **20**(4) (1998)
18. Chilimbi, T.M., Davidson, B., Larus, J.R.: Cache-conscious structure definition. In: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia (1999) 13–24 *SIGPLAN Notices*, 34(5), May 1999.
19. N.Gloy, Smith, M.D.: Procedure placement using temporal-ordering information. ACM Transactions on Programming Languages and Systems **21**(5) (1999)
20. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. Parallel and Distributed Computing **64**(1) (2004)
21. Ding, C., Orlovich, M.: The potential of computation regrouping for improving locality. In: SuperComputing 2004, Pittsburgh, PA. (2004)