Cache Optimization for Coarse Grain Task Parallel Processing using Inter-Array Padding

Kazuhisa Ishizaka, Motoki Obata, and Hironori Kasahara

Department of Computer Science, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{ishizaka,obata,kasahara}@oscar.elec.waseda.ac.jp

Abstract. The wide use of multiprocessor system has been making automatic parallelizing compilers more important. To improve the performance of multiprocessor system more by compiler, multigrain parallelization is important. In multigrain parallelization, coarse grain task parallelism among loops and subroutines and near fine grain parallelism among statements are used in addition to the traditional loop parallelism. In addition, locality optimization to use cache effectively is also important for the performance improvement. This paper describes inter-array padding to minimize cache conflict misses among macro-tasks with data localization scheme which decomposes loops sharing the same arrays to fit cache size and executes the decomposed loops consecutively on the same processor. In the performance evaluation on Sun Ultra 80(4pe), OSCAR compiler on which the proposed scheme is implemented gave us 2.5 times speedup against the maximum performance of Sun Forte compiler automatic loop parallelization at the average of SPEC CFP95 tomcatv, swim hydro2d and turb3d programs. Also, OSCAR compiler showed 2.1 times speedup on IBM RS/6000 44p-270(4pe) against XLF compiler.

1 Introduction

Multiprocessor architectures are currently used in wide range of computers including high performance computers, entry level servers and games embedding chip multiprocessors. To improve usability and effective performance of multiprocessor systems, automatic parallelizing compilers are required. To this end, automatic parallelizing compilers have been researched. For example, Polaris[1] compiler exploits loop level parallelism by using symbolic analysis, runtime data dependence analysis, range test and so on. Loop parallelization considering the data locality optimization using unimodular transformation, affine partitioning and so on has been researched in SUIF compiler [2].

Since various kinds of loops can be parallelized by those advanced compilers, to further improve the effective performance of multiprocessor systems, the use of different grains of parallelism such as the use of coarse grain task parallelism among loops and subroutines and fine grain parallelism among statements and instructions in addition to loop level parallelism should be considered. NANOS compiler[3] uses multi level parallelism by using the extended OpenMP API. PROMIS compiler[4] integrates loop level parallelism and instruction level parallelism using a common intermediate language. Multigrain parallel processing which has been realized in OSCAR compiler [5], APC compiler(Advance Parallelizing Compiler developed by Japanese millennium project IT21)[6] uses coarse grain task parallelism among loops and subroutines and near fine grain parallelism among statements.

Also, optimization for memory hierarchy to minimize the memory access overhead that is getting larger with the speedup of a processor is important to improve the performance. Loop restructurings such as loop permutation, loop fusion and tiling to change data access pattern in a loop are researched as the cache optimization by the compiler. Data layout transformations including strip mining and array permutation to make data access pattern contiguous are also researched. Intra-array padding and inter-array padding to reduce conflict misses in a single loop or a fused loop are proposed[7]. Also, the loop fusion scheme using peeling and shifting of loop iteration to allow fusion and maintain loop parallelism has been used to enhance data locality[8]. Furthermore, after loop fusion, conflict misses can be reduced by cache partitioning[9].

The performance of physically-indexed cache depends on the page placement policy of operating system such as page coloring and bin hopping[10]. Runtime recoloring scheme using the extended hardware such as Cache Miss Lookaside buffer to traces cache conflict misses has been proposed [11]. Low overhead recoloring using extended TLB to record the cache color is also researched[12]. In addition to these approaches requiring the extended hardware, OS and compiler cooperative page coloring scheme without hardware extension using information on access pattern of program provided by compiler is proposed[13].

This paper proposes the padding scheme to reduce conflict misses to improve the performance of the coarse grain task parallel processing. In the cache optimization for coarse grain task parallel processing [14], at first, complier divides loops into smaller loops to fit data size accessed by loops to cache size. Next, the compiler analyzes parallelism among tasks including the divided loops using Earliest Executable Condition analysis and schedules tasks which shared the same data to the same processor so that the tasks can be executed consecutively accessing the shared data on the cache. After that, cache line conflict misses among tasks which are executed consecutively are reduced by padding proposed in this paper. Although ordinary cache optimizations by the compiler target a single loop or a fused loop, the proposed scheme optimizes cache performance over loops.

The rest of this paper is organized as follows. In section 2, the coarse grain task parallel processing is described. Section 3 describes the cache optimization scheme using data localization for the coarse grain task parallel processing. Section 4 proposes the padding scheme to reduce conflict misses over loops. The effectiveness of the proposed schemes is evaluated on the commercial multiprocessors using several benchmarks in SPEC CFP95 in section 5. Finally, concluding remarks are described in section 6.

2 Coarse Grain Task Parallel Processing

This section describes coarse grain task parallel processing to which the proposed cache optimization scheme is applied. In the coarse grain task parallel processing, a source program is decomposed into three kinds of coarse grain tasks, or macro-tasks, namely block of pseudo assignment statements(BPA) repetition block(RB), subroutine block(SB). Also, macro-tasks are generated hierarchically inside of a sequential repetition block and a subroutine block.

2.1 Generation of Macro-task Graph

After the generation of macro-tasks, compiler analyzes data flow and control flow among macro-tasks in each layer or each nested level. Next, to extract parallelism among macro-tasks, the compiler analyzes Earliest Executable Condition(EEC)[5] of each macro-task. EEC represents the conditions on which macro-task may begin its execution earliest.

EEC of macro-task is represented in macro-task Graph (MTG) as shown in Fig.1. In macro-task graph, nodes represent macro-tasks. A small circle inside nodes represents conditional branches. Solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means ordinary control dependency and the condition on which a data dependent predecessor macro-task is not executed. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relation ship.

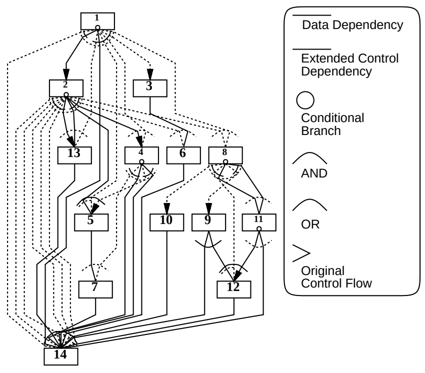

Fig. 1. An Example of Macro-Task Graph

2.2 Macro-Task Scheduling

In the coarse grain task parallel processing, static scheduling and dynamic scheduling are used for assignment of macro-tasks to processors.

If a macro-task graph has only data dependencies and is deterministic, static scheduling is selected. In the static scheduling, assignment of macro-tasks to processors is determined at compile time by the scheduler in the compiler. Static scheduling is useful since it allows us to minimize data transfer and synchronization overhead without runtime scheduling overhead.

If a macro-task graph has control dependencies, dynamic scheduling is selected to cope with runtime uncertainties like conditional branches. Scheduling routine for dynamic scheduling are generated by compiler and embedded into a parallelized program with macro-task code.

2.3 Code Generation

OSCAR compiler has several backends and generates the parallelized code for multiple target architectures. In this paper, OpenMP backend is used to generate OpenMP FORTRAN from sequential FORTRAN. OSCAR compiler generates the portable code for various shared memory multiprocessors by using "one-time single code generation" technique[5, 15]. Furthermore, by using native compiler as the backend of OSCAR compiler, general optimizations and machine specific optimizations provided by it are applied to the generated code. Therefore, the performance of OSCAR compiler can be used as a performance booster of the native compiler on the state of the art multiprocessor.

3 Cache Optimization for Coarse Grain Task Parallel Processing

If macro-tasks that access the same data are executed consecutively on the same processor, shared data can be transffered among these macro-tasks using fast memory such as cache. This section describes cache optimization using data localization[16] to enhance the performance of coarse grain task parallel processing.

3.1 Loop Aligned Decomposition

To avoid cache misses among the macro-tasks, Loop Aligned Decomposition(LAD)[16] is applied to loops that access large size data. LAD divides a loop into partial loops with the smaller number of iterations so that data size accessed by the divided loops is smaller than cache size.

Partial loops are treated as coarse grain tasks and the Earliest Executable Condition(EEC) analysis is applied. Partial loops connected by data dependence edge on the macro task graph are grouped into "Data Localization Group(DLG)". Partial loops, or macro-tasks, inside a DLG are assigned to the same processor as consecutively as possible by static or dynamic scheduler.

In macro-task graph of Fig.2(a), it is assumed that macro-tasks 2, 3 and 7 are parallel loops and they access the same shared variables and their size exceeds cache size. In this example, loops are divided into four partial loops by the LAD. For example, macro-task 2 in Fig.2(a) is divided into macro-task 2_A through 2_D in Fig.2(b). Also, DLGs are defined, for example, 2_A, 3_A, 7_A are grouped into DLG_A.

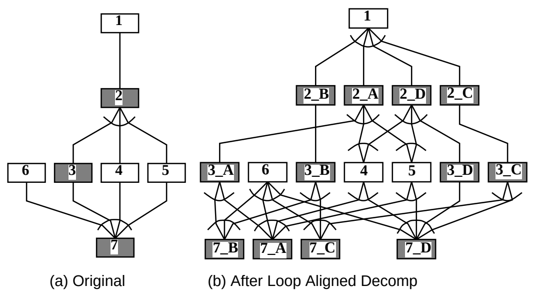

Fig. 2. Example of Loop Align Decomposition

3.2 Scheduling for Consecutive Execution of Macro-tasks

Macro-tasks are executed in the increasing order of the node number on the macro-task graph in the original program. For example, the execution order of macro-tasks 2_A to 3_D is 2_A, 2_B, 2_C, 2_D, 3_A 3_B, 3_C, 3_D. In this order, macro-tasks in the same DLG are not executed consecutively.

However, the earliest executable condition shown in Fig. 2(b) means that macro-task 3_B, for example, can be executed immediately after macro-task 2_B because macro-task 3_B depends on only macro-task 2_B.

In the proposed cache optimization scheme, a task scheduler for the coarse grain tasks assigns macro-tasks inside a DLG to the same processor as consecutively as possible[14] in addition to "critical path" priority. Fig.3 shows a schedule when the proposed cache optimization is applied to macro-task graph in Fig.2(b) for a single processor. As shown in Fig.3, macro-task 2_B, 3_B, 8_B in DLG_B and macro-task 2_C, 3_C, 7_C in DLG_C are executed consecutively to use cache effectively.

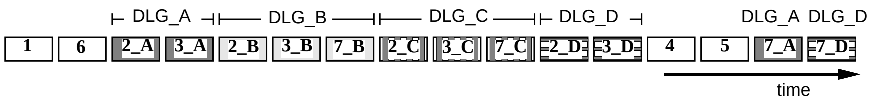

Fig. 3. Example of Scheduling Result on Single Processor

4 Reduction of Cache Conflict Misses

This section describes the data layout transformation using padding to reduce conflict misses among macro-tasks in a DLG.

4.1 Conflict Misses in a DLG

In the data localization, loops accessing the same shared variable larger than cache size are divided to smaller loops or macro-tasks. Furthermore, macro-tasks in the same DLG are executed consecutively on the same processor. This enables the shared data to be reused before cache out. However, if data accessed by macro-tasks in a DLG share the same line on the cache, data may be removed from the cache because of line conflict miss even though data size accessed in a DLG is not larger than the cache size.

Conflict misses in a DLG are reduced by data layout transformation by inter-array padding. In this section, SPEC CFP95 swim is used as an example for the proposed padding scheme. Swim has 13 single precision 513x513 arrays and each size is about 1MB. Fig.4 shows the data layout image on cache where 13 arrays are allocated to 4MB direct map cache. In this figure, boxes framed by thick lines show arrays. Horizontal direction represents 4MB cache space. This figure means that arrays on the same vertical position are allocated to the same cache lines and they cause line conflict misses. For example, arrays U, VNEW, POLD and H are allocated to the same part of cache.

Dotted lines in the figure show the partial arrays accessed by the divided loops by the LAD when loops are divided to 4 smaller loops. Gray part of each array shows a partial array accessed by the divided loops in a DLG. As shown in the figure, conflict misses may be caused among the partial arrays accessed in a DLG, or on a vertically same position.

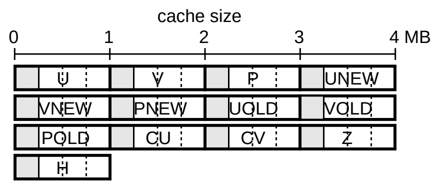

Fig. 4. Data Layout Image on Cache of Swim

This conflict misses interfere the data reuse among the consecutively executed macro-tasks. Data layout transformation by the padding to reduce conflict misses in a DLG is required for the cache optimization among loops or macro-tasks. This section describes the padding scheme to reduce conflict misses in a DLG.

4.2 Inter-Array Padding

This section describes an inter-array padding procedure using array declaration size change.

Step1 Select Target Arrays Since OSCAR compiler on which the proposed scheme is implemented generates the parallelized OpenMP FORTRAN, the actual data layout is determined by the machine native compiler which is used as the back end of OSCAR compiler. Therefore, in the current implementation, OSCAR compiler chooses arrays of the same size as the target of the proposed padding and changes declaration size of the target arrays to realize inter-array padding.

Arrays in FORTRAN “common block” are also chosen as the target of inter-array padding if a common block has the same shape over all program modules because changing declaration size of such arrays dose not break the program semantics. Padding for arrays in common block that has different shapes are described in section 4.3.

Step2 Generate Data Layout Image on Cache Next, a compiler calculates addresses of selected arrays and generates data layout image on cache as shown in Fig.4. In this step, because all target arrays have the same size, a compiler can determine the data layout image regardless of the actual data layout determined by the native compiler.

Step3 Calculate Minimum Division Number A compiler calculates the minimum division number (*div_num*) to make data size accessed in a DLG smaller than the cache size by dividing total array size of target arrays by cache size. In the example in Fig.4, total array size is 13MB and cache size is 4MB. Then, *div_num* is $ceil(13/4) = 4$.

Step4 Calculate Maximum DLG Access Size The maximum data size accessed in a DLG (*part_size*, gray range in Fig.4) is calculated by dividing array size by *div_num*. If there are overlaps among partial arrays of *part_size* in data layout image on cache, it means that conflicts may be caused among arrays accessed in a DLG. If there is no overlap, padding is not applied.

Step5 Calculate Padding Size To remove conflict, the distance on the cache between the base address of first array (array U in Fig.4) and the base address of first array after cache size (array VNEW) should be *part_size*. Padding size to remove conflict between U and VNEW is $cache\_size + part\_size - base\_address$ where *base_address* is the base address of VNEW. Similarly, same size pads are inserted to remove all conflicts as shown in Fig.5(a).

Step6 Change Array Size In the proposed scheme, pads inserted among certain arrays as shown in Fig.5(a) are distributed to all arrays so that the data layout dose not depends on the specific order of arrays. In practice, the rightmost dimension of each array is changed to increase array size by $padding\_size/narrays$, where *narrays* is the number of arrays in the range from the beginning to $cache\_size + part\_size$(4 in this example). Fig.5(b) shows the data layout image on cache after the proposed padding by changing array size.

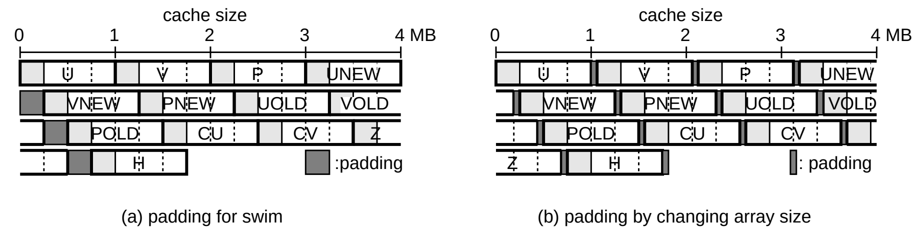

Fig. 5. Inter-Array Padding for Swim

4.3 Padding for Common Block

Some program modules may have the different array declarations size for a common block. Because padding among arrays in such common block may change the program semantics, it is difficult to apply inter-array padding to such arrays. Therefore, a compiler merges such common blocks to single large common block and inserts pads among common blocks to maintain program semantics and reduce conflict misses among arrays in common blocks.

4.4 Set Associative Cache

In the current implementation, the proposed padding targets LRU replacement policy for a set associative cache. A set associative cache is treated as a direct map cache of same size. If padding removes conflicts on a direct map cache, the number of overlaps on n-way cache is smaller than n because the data layout image on cache of n-way set associative is same as that of a direct map cache of 1/n size. Therefore, there is no conflict on an n-way cache because a cache set of n-way cache can hold n lines.

4.5 Page Placement Policy of Operating System

Data layout transformation by a compiler is made on virtual address. Therefore, page placement policy of operating system to map a virtual address to a physical address affects it on a physically-indexed cache.

A simple page coloring maps sequential virtual pages to sequential physical pages. Therefore, a page conflicts with the page apart from it by cache size. Data transformation by a compiler is effective in this policy because continuity of the address on virtual address is kept on physical address.

In bin hopping, sequential physical pages are assigned to virtual pages in the order of page fault, irrespective of their virtual address. Continuity on the virtual address is not remained on physical address in this policy. Therefore, it is difficult that a compiler applies data layout transformation effectively beyond the page size on virtual address.

5 Performance Evaluation

This section describes the performance evaluation of the proposed scheme on Sun Ultra 80 and IBM RS/6000 44p-270. Ultra80 has four 450MHz Ultra SPARC-IIs with 4MB direct map L2 cache(LRU) for each processor and RS/6000 has four 375MHz Power3s with 4MB 4-way set associative L2 cache. Both caches are physically-indexed caches. Solaris 8 on Ultra 80 and AIX 4.3 on RS/6000 support page coloring and bin hopping.

In the evaluation, sequential FORTRAN programs are translated into parallelized OpenMP FORTRAN programs using OSCAR compiler on which the proposed scheme has been implemented. Three kinds of compilation, namely OSCAR with the proposed padding, OSCAR without the padding and automatic parallelization by the machine native compiler are compared.

SPEC CFP95 tomcatv, swim, hydro2d and turb3d are used in this evaluation. Original sources code of SPEC are used by both OSCAR and native compiler for tomcatv, swim and hydro2d. However, turb3d is preprocessed by APC compiler[6] in order to parallelize some loops containing subroutine calls because both OSCAR and native compilers currently cannot parallelize such loops.

Since data size of programs used in this evaluation are about ten MB, the target of the proposed padding with data localization in this evaluation is L2 cache that has larger miss penalty and larger impact on performance than L1 cache of 32KB or 64KB. In this evaluation, the number of loops generated from a loop by loop division is same as the number of processors. Therefore, performance improvement is obtained mainly by the proposed padding.

The proposed inter-array padding extends 513x513 2-dimensional array to 513x573 for tomcatv, 513x513 to 514x544 for swim, 66x64x64 to 66x64x71 for turb3d. The padding for common blocks is applied to hydro2d. Four common blocks, VAR1, VAR2, VARH and SCRA, are merged to a common block and a dummy array of 318696 bytes is inserted between VAR2 and VARH.

5.1 Performance on Sun Ultra 80

Solaris 8 supports Hashed VA, V.addr=P.addr and bin hopping as the page placement policy. V.addr=P.addr method keeps continuity on virtual address on physical address. Hashed VA is similar to V.addr=P.addr but it inserts a small gap every L2 cache size (4MB) to avoid conflict miss among two addresses, distance among which is just L2 cache size. Default policy of Solaris 8 is Hashed VA.

Speedups for 4PEs against sequential execution by Sun Forte 6 update 2 compiler on Sun Ultra 80 are shown in Fig.6. Numbers above the bar in the figure show execution times. In addition, the number of cache misses measured by CPU Performance count of Ultra SPARC-II is shown in Fig.7.

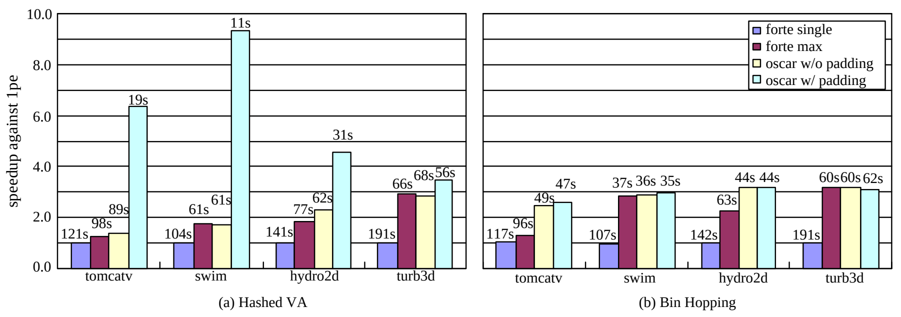

Fig. 6. Speedups on Sun Ultra 80

Speedups on Hashed VA by the automatic parallelization of Forte for tomcatv, swim and hydro2d are only 1.2, 1.7 and 1.8 times against sequential execution respectively as shown in Fig.6(a). Also, speedups by OSCAR without padding are 1.4, 1.7 and 2.3 times, since conflict misses prevent the scalability.

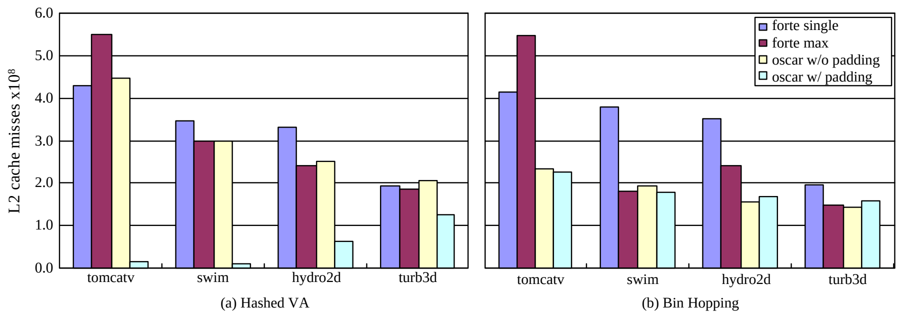

Fig. 7. L2 Cache Misses on Sun Ultra 80

For example, the number of cache misses of swim by Forte automatic parallelization is 300 million and that of OSCAR without padding is also 300 million as shown in Fig.7(a). These are not much reduced compared with that of the sequential execution (350 million) in spite of the quadruple cache size on 4PEs. On the other hand, turb3d has two kinds of loops. The first access is sequential and it causes conflict misses as show in section 4. However, because second access pattern is interleaved, cache performance is better than other three programs.

Since Ultra 80 used in this evaluation has a single memory bank, memory accesses are serialized and the bottleneck of scalability. Therefore, reduction of conflict misses to improve the L2 cache performance is important. Speedups by OSCAR with padding on Hashed VA are 6.3 times for tomcatv, 9.4 for swim, 4.6 for hydro2d and 3.4 for turb3d on 4PEs against the sequential execution. Also, padding increases the performance of OSCAR without padding 4.7, 5.5, 2.0 and 1.2 times respectively. The number of cache misses are decreased by padding to 3.5% of OSCAR without padding for tomcatv, 4.2% for swim, 25% for hydro2d, 61% for turb3d as shown in Fig.7.

Speedups by OSCAR without padding against sequential execution on bin hopping are 2.4 times for tomcatv, 3.0 swim, 3.2 for hydro2d and 3.2 for turb3d as shown in figure 6(b). These are 1.8, 1.7, 1.4 and 1.1 times better than OSCAR without padding on Hashed VA. The reason is that conflict misses assumed on virtual address dose not appear on physical address. Speedups by OSCAR with padding on bin hopping are 2.5, 3.0, 3.2, 3.0 times for each program and only few percentage speedups compared with OSCAR without padding.

In this evaluation, the best performance on Ultra 80 is given by OSCAR with padding on Hashed VA. Execution times by it are 19 seconds for tomcatv, 11 seconds for swim, 31 seconds for hydro2d and 56 seconds for turb3d and minimum execution times on bin hopping are 47 seconds, 35 seconds, 44 seconds and 60 seconds respectively.

5.2 IBM RS/6000 44p-270

Fig.8 shows speedups for 4PEs against sequential execution on IBM RS/6000 44p-270 with 4-way set associative L2 cache(LRU). Default page placement policy of AIX 4.3 is bin hopping and page coloring is supported.

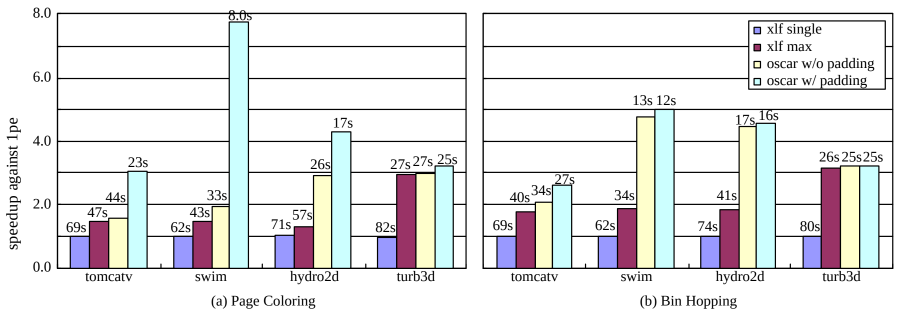

Fig. 8. Speedups on RS/6000 44p-270

As shown in Fig.8(a), speedups by OSCAR with padding against sequential execution on bin hopping are 2.6 times for tomcatv, 5.0 for swim, 4.6 for hydro2d and 3.2 for turb3d. They are 27%, 4.6%, 2.3%, 0.2% better than OSCAR without padding.

Speedups by OSCAR without padding on page coloring are 1.6, 1.9, 2.9, 3.0 times against sequential execution and less than on bin hopping. Bin hopping show 1.2 times better performance for XLF automatic parallelization, 1.5 times better for OSCAR without padding compared with page coloring. However, OSCAR with padding gave us 3.0 times speedup for tomcatv, 7.8 for swim, 4.3 for hydro2d and 3.2 for turb3d against sequential execution on bin hopping. Padding increases the performance by OSCAR without padding 2.0, 4.1, 1.5 and 1.1 times for each program.

Execution times by OSCAR with padding on page coloring are 23 seconds for tomcatv, 8 seconds for swim, 17 seconds for hydro2d and 25 seconds for turb3d and minimum execution times on bin hopping are 27 seconds, 12 seconds, 16 seconds and 25 seconds for respectively. OSCAR with padding on page coloring gave us the best performance on RS/6000 44p-270.

6 Conclusions

This paper has described the cache optimization with data localization for coarse grain tasks parallel processing on SMP machine. In the proposed scheme, loops are divided into smaller loops to fit the cache and loops accessing the shared data are executed on the same processor as consecutively as possible to improve temporal locality over different loops. Moreover, cache line conflicts among loops are reduced by inter-array padding.

The proposed scheme is implemented in OSCAR compiler as a core compiler of APC compiler developed in the Japan METI Advanced Parallelizing Compiler project in a part of Millennium Project IT21[6] and it was evaluated on the two commercial SMP workstations having different cache configurations with popular page placement policies of operating system. In the evaluation on the Sun Ultra 80(4pe) which has 4MB direct map L2 cache, the proposed padding scheme gave us 5.9 times speedup against sequential execution at the average of 4 programs of SPEC CFP95, tomcatv, swim, hydro2d and turb3d, on the default page placement policy called Hashed VA. OSCAR with padding on page coloring also gave us 4.6 times speedup against sequential execution on the RS/6000 44p-270(4pe) having 4MB 2-way set associative L2 cache. The evaluation on two multiprocessors shows that OSCAR with padding on page coloring gave us the best performance on both machines.

Acknowledgments

This research is supported by METI/NEDE millennium project IT21 "advanced Parallelizing Compiler" and STARC (Semiconductor Technology Academic Research Center).

References

1. R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.
2. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.
3. X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitatio in numa multiprocessors. *Proc. of the 1999 International Conference on Supercomputing*, June 1999.
4. C. J. Brownhill, A. Nicolau, S Novack, and C. D. Polychronopoulos. Achieving multi-level parallelization. *Proc. of the International Symposium on High Performance Computing*, 1997.
5. H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. *Proc. of 13 th International Workshop on Languages and Compilers for Parallel Computing 2000*, Aug. 2000.
6. APC. Advanced parallelizng compiler project. http://www.apc.waseda.ac.jp.
7. G. Rivera and C-W. Tseng. Eliminating conflict misses for high performance architectures. *Proc. of the 1998 ACM International Conference on Supercomputing*, July 1998.
8. Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *Proc. of the 24th International Conference on Parallel Processing*, Aug. 1995.
9. Naraig Manjikian and Tarek S. Abdelrahman. Array data layout for the reduction of cache conflicts. *Proc. of 8th International Conference on Parallel and Distributed Computing Systems*, Sep. 1995.

10. R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM transaction of Computer Systems*, Nov. 1992.
11. Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Brandley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. *Proc. of the Sixth Internatinal Symposium of Architectural Support for Programing Languages and Operating Systems*, Oct. 1994.
12. Timothy Sherwood, Brad Calder, and Joel Ember. Reducing cache misses using hardware and software page placement. In *Proc. of the International Conference of Supercomputing*, June 1999.
13. E. Bugnion, J. M. Anderson, T. C. Mowry, M. R. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. *Proc. of the Seventh Internatinal Symposium of Architectural Support for Programing Languages and Operating Systems*, Oct. 1996.
14. K. Ishizaka, M. Obata, and H. Kasahara. Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In *Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing*, Aug. 2001.
15. H. Kasahara et al. Performance of multigrain parallelization in japanese millennium project it21 advanced parallelizing compiler. In *Proc. of 10th International Workshop on Compilers for Parallel Computers (CPC)*, Jan. 2003.
16. H. Kasahara A. Yhoshida, K. Koshizuka. Data-localization using loop aligned decomposition for macro-dataflow processing. *Proc. of 9th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1996.