Near Fine Grain Parallel Processing Using Static Scheduling on Single Chip Multiprocessors

Keiji Kimura and Hironori Kasahara
Dept. of Electrical, Electronics and Computer Engineering, Waseda University.
Okubo, Shinjuku-ku, Tokyo, Japan, 169-8555, TEL: +81-3-5286-3371
E-mail: {kimura, kasahara}@oscar.elec.waseda.ac.jp
URL : http://www.kasahara.elec.waseda.ac.jp/

Abstract

With the increase of the number of transistors integrated on a chip, efficient use of transistors and scalable improvement of effective performance of a processor are getting important problems. However, it has been thought that popular superscalar and VLIW would have difficulty to obtain scalable improvement of effective performance in future because of the limitation of instruction level parallelism. To cope with this problem, a single chip multiprocessor (SCM) approach with multi grain parallel processing inside a chip, which hierarchically exploits loop parallelism and coarse grain parallelism among subroutines, loops and basic blocks in addition to instruction level parallelism, is thought one of the most promising approaches. This paper evaluates effectiveness of the single chip multiprocessor architectures with a shared cache, global registers, distributed shared memory and/or local memory for near fine grain parallel processing as the first step of research on SCM architecture to support multi grain parallel processing. The evaluation shows OSCAR (Optimally Scheduled Advanced Multiprocessor) architecture having distributed shared memory and local memory in addition to centralized shared memory and attachment of global register gives us significant speed up such as 13.8% to 143.8% for four processors compared with shared cache architecture for applications which have been difficult to extract parallelism effectively.

1. Introduction

Advances in semiconductor technology allow us to integrate a lot of execution units, memory or even processors on a single chip[12]. These resources have been used for extracting instruction level parallelism (ILP) in superscalar and VLIW architectures. However, it has been said that there is the limitation of ILP in programs and it is difficult to obtain scalable improvement of effective performance for such ILP processors. Therefore, to use these resources effectively, next generation microprocessor architectures[2, 11, 18, 19, 12] and their compilers[17, 13, 16] have been widely researched. Among these architectures, a single chip multiprocessor (SCM) architecture, which uses thread level parallel processing with speculative execution[18, 17, 13] or integrates many simple processing elements[16, 19, 12], is one of the most promising architectures. The authors think that the SCM architecture to support multigrain parallel processing, which can exploit multiple grains of parallelism hierarchically, allows us to develop a scalable and cost effective computer system.

This paper evaluates effectiveness of the SCM with a shared cache, global registers, distributed shared memory and/or local memory for near fine grain parallel processing as the first step of research on SCM architecture to support multi grain parallel processing.

The rest of this paper is organized as follows. Section 2 gives a brief overview of multigrain parallel processing and detailed explanation of near fine grain parallel processing. Section 3 describes the SCM architecture for supporting the near fine grain parallel processing, and architectures to be evaluated in this paper. Section 4 evaluates performance of the architectures in near fine grain parallel processing using real application programs and OSCAR multigrain parallelizing compiler[14, 6].

2. Multigrain Parallel Processing

This section gives a overview of the multi grain parallel processing which allows us to exploit much more parallelism than instruction level parallelism on a single chip multiprocessor.

The multigrain parallel processing[8] hierarchically applies the macro-dataflow processing[3], which uses coarse

grain parallelism among loops, subroutines, and basic blocks, the loop parallel processing that uses iteration level parallelism and near fine grain parallel processing[7] which uses statement level parallelism inside a basic block.

The multi grain parallelization is automatically performed by OSCAR Fortran multigrain parallelizing compiler[14, 6] used for the performance evaluation of SCM architectures in this paper.

2.1. Coarse-grain Task Parallel Processing

In the macro data flow processing, firstly, a Fortran program is decomposed into three kinds of coarse grain tasks or macrotasks (MTs), such as Block of Pseudo Assignment statements (BPA), Repetition Block (RB) and Subroutine Block (SB)[3].

A BPA is usually defined as an ordinary basic block (BB). However, it is sometimes defined by decomposing a basic block into independent blocks to extract larger parallelism or by fusing multiple basic blocks into a coarser macrotask, or BPA[14].

A RB is a Do loop or a loop generated by a backward branch, namely, an outermost natural loop.

As to a SB, the compiler defines subroutines, to which the in-line expansion technique cannot be efficiently applied, as SBs.

Furthermore, SBs and RBs can be hierarchically decomposed into sub-macrotasks. For the sub-macrotasks, the macro dataflow processing scheme can be hierarchically applied to exploit parallelism inside SB and RB[15].

After generation of macrotasks, the compiler analyzes control flow and data flow among macrotasks. The result of analysis is represented by a directed acyclic graph called Macro-Flow-Graph (MFG)[3, 4]. Figure1 shows an example of a macroflow graph. In this MFG, nodes represent macrotasks, such as BPAs, RBs, and SBs. Dotted edges represent control-flow. Solid edges represent data dependencies among macrotasks. Small circles inside nodes represent conditional branch statements inside macrotasks.

Next, in order to find the maximum parallelism among macrotasks considering control dependencies and data dependencies, the compiler analyzes an earliest-executable-condition for the macrotask[3, 4]. The earliest-executable-condition of macrotask i (MT_i), is a condition on which MT_i may begin its execution earliest.

These earliest-executable-conditions of macrotasks are represented by a directed acyclic graph called MacroTask-Graph (MTG)[3, 4], as show in Figure2. In MTG, nodes represent macrotasks. Dotted edges represent extended control dependencies. Solid edges represent data dependencies.

The extended control dependence edges are classified into two types of edges, namely, ordinary control dependence edges and co-control dependence edges. The co-

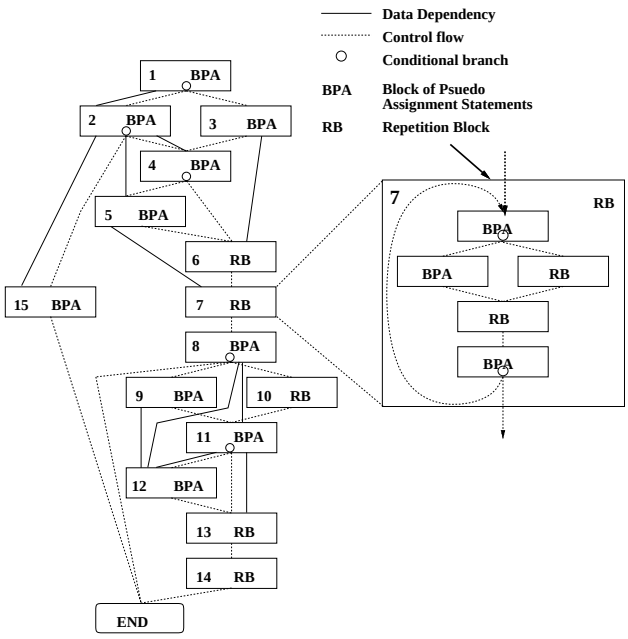

Figure 1. Macroflow graph(MFG)

control dependence edges represent conditions under which the data dependence predecessor of MT_i (namely, MT_k on which MT_i is data dependent) is not executed.

In addition, a data dependence edge, or a solid edge, originating from a small circle has two meanings, namely, an extended control dependence edge and a data dependence edge. Arcs connecting edges at their tails or heads have two different meanings. A solid arc means that edges connected by the arc are in an AND relationship. A dotted arc means that edges connected by the arc are in an OR relationship. Small circles inside nodes represent conditional branch statements. For example, earliest-executable-condition of MT_8 is that MT_1 branches to MT_3 or MT_2 branches to MT_4 and the condition of MT_6 is that MT_3 finishes execution or MT_2 branches to MT_6.

After generation of MTG, if there exists runtime uncertainties inside a target program, such as, conditional branches among macrotasks and a variation of macrotask execution time, macrotasks are assigned onto processor-clusters (PCs), processors or single chip multiprocessors at runtime by a dynamic scheduling routine generated exclusively for the target program and embedded into user program by the compiler. Otherwise, macrotasks are assigned by static scheduling in compile time.

2.2. Loop Level Parallel Processing

Macrotasks are assigned to processor-clusters (PCs) dynamically or statically as mentioned in the previous sub-

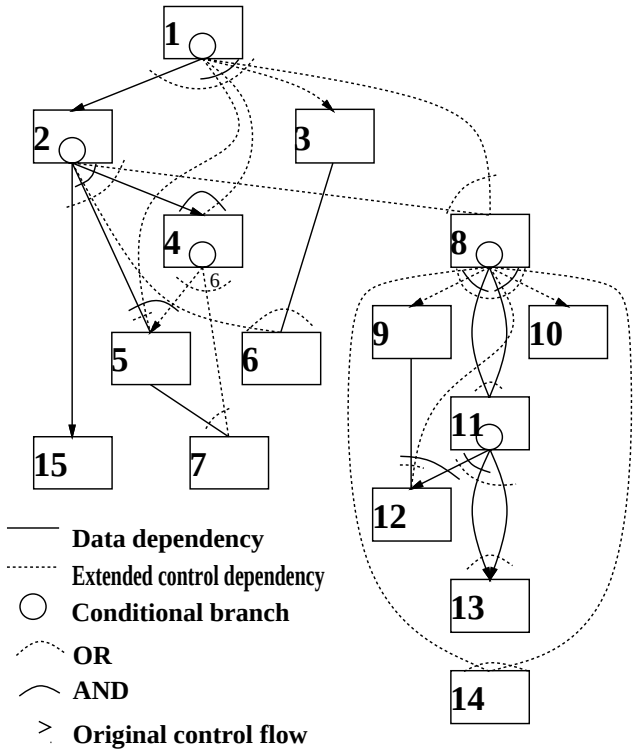

Figure 2. Macrotask graph(MTG)

section. If a macrotask assigned to a PC is a Doall loop, the macrotask is processed in the medium grain, or iteration level grain, by processing elements (PEs) inside a PC.

2.3. Near-fine grain parallel processing

If a macrotask assigned to a PC is BPA or sequential loop, it is decomposed into the near fine grain tasks, each of which consists of a statement, and processed in parallel by PEs inside a PC or a single chip multiprocessor.

Figure3 shows an example of BPA containing 17 statements that solves a random sparse matrix using Crout method. This kind of basic block is generated by the symbolic generation technique, which has been used in electronic circuit simulators like SPICE and by partial evaluation.

The compiler analyzes data dependencies among the statements and generates a task graph that represents data dependencies among near fine grain tasks. Figure4 shows an example of task graph for the BPA in Figure3. In the task graph, the dependencies, or precedence constraints, among the generated tasks can be represented by edges, and each task corresponds to a node. In Figure4, figure inside a node circle represent the task number, i, and those beside it represent a task-processing time on a PE, t_i. An edge directed from node N_i toward N_j represents the partially ordered constraint that task T_i precedes task T_j. When we consider a data transfer time between tasks, each edge generally has a variable weight. Its weight, t_{ij} will be a data transfer time between task T_i and T_j if T_i and T_j are assigned to different PEs. In Figure4, it is assumed that data transfer and synchronization takes nine clocks. It will be zero or a time to access register or local data memory if the tasks are assigned to the same PE.

These tasks are assigned onto processors statically since there exist only data dependencies among near fine grain tasks inside BPA. However, since this static scheduling problem is strong NP hard, the OSCAR compiler uses four heuristic scheduling algorithms CP/DT/MISF, CP/ETF/MISF, ETF/CP and DT/CP[4] and chooses the best schedule automatically.

After scheduling, the compiler generates the machine codes for each PE by putting together instructions for tasks assigned to the PE and by inserting instructions for data transfer and synchronization into the required places using a statically scheduled result. In this time, the compiler can optimize the codes by making full use of information obtained from the static schedule. For example, when a task should pass shared data to other tasks assigned to the same PE, the data can be passed through registers on the PE. In addition, the compiler minimizes the synchronization overhead by considering the information about tasks to be synchronized, the task assignment, and the execution order[7]. The first optimization is the elimination of redundant synchronization as shown in Figure5. In this figure, tasks A, B, and C are allocated to PE1, task D is allocated to PE2, and task E is allocated to PE3. The edges among tasks show data dependencies. Therefore, edges across PEs mean data transfer and synchronization among PEs. If the data transfers and synchronization among PEs are realized by using the centralized shared memory, task E does not need to check the synchronization flag that indicates the completion of task D, because completion of tasks has already been confirmed by tasks B and C, which are precedent tasks of E. For the architecture having global register file explained in section 3.5, the compiler also can assign near fine grain data transfer and synchronization to global registers efficiently with statically scheduled information.

3. Architecture for Near Fine Grain Parallel Processing

This section describes architectural supports in SCM for near fine grain parallel processing.

In multigrain parallel processing, as mentioned before, near fine grain parallel processing is normally applied to a basic block and the compiler can minimize data transfer and synchronization overhead by static scheduling. In

<< LU Decomposition >>

1) $u_{12} = a_{12} / l_{11}$
2) $u_{24} = a_{24} / l_{22}$
3) $u_{34} = a_{34} / l_{33}$
4) $l_{54} = -l_{52} * u_{24}$
5) $u_{45} = a_{45} / l_{44}$
6) $l_{55} = a_{55} - l_{54} * u_{45}$

<< Forward Substitution >>

7) $y_1 = b_1 / l_{11}$
8) $y_2 = b_2 / l_{22}$
9) $b_5 = b_5 - l_{52} * y_2$
10) $y_3 = b_3 / l_{33}$
11) $y_4 = b_4 / l_{44}$
12) $b_5 = b_5 - l_{54} * y_4$
13) $y_5 = b_5 / l_{55}$

<< Backward Substitution >>

14) $x_4 = y_4 - u_{45} * y_5$
15) $x_3 = y_3 - u_{34} * x_4$
16) $x_2 = y_2 - u_{24} * x_4$
17) $x_1 = y_1 - u_{12} * x_2$

Figure 3. Near fine grain tasks.

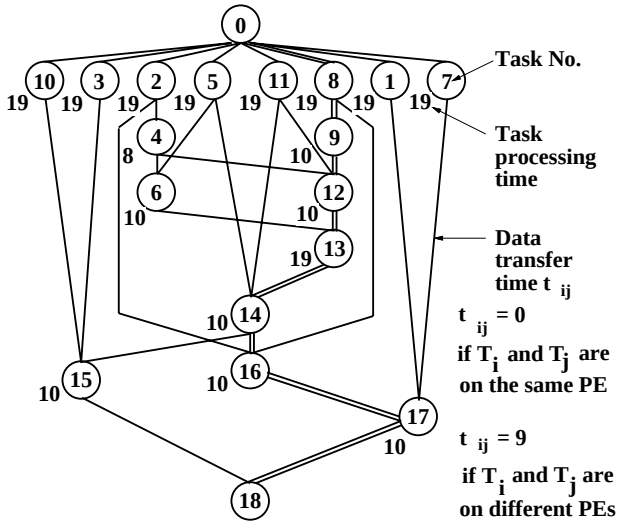

Figure 4. Task graph for near fine grain tasks.

order to generate most efficient parallel machine code precisely scheduled in clock level, every instruction should be executed in fixed clock cycles.

In addition, low-latency data transfer and low-overhead synchronization mechanisms are required to minimize the overhead among processors. To this purpose, distributed shared memory (DSM) having two ports, shared cache and/or global register seems to be useful. Especially, DSM can transfer shared data without preventing remote PE execution and minimize data synchronization overhead since busy wait for synchronization flag is performed inside a processor without consuming network and centralized shared memory band width.

Also, PE local data memory (LDM), which is used for storing PE local data and can have twice larger memory than DSM since local memory only have a single port, can be used effectively[1, 5]. Furthermore, global registers among processors allow us to reduce data and synchronization overhead with good register allocation technique in the compiler.

3.1. Traget Architectures

Considering architecture supports mentioned above, this section describes single-chip multiprocessor architectures evaluated in this paper.

The authors prepared shared cache type architecture and OSCAR type architecture, which has distributed shared memory and local data memory in addition to centralized shared memory. Effectiveness of global register files attached to the both architectures is also evaluated.

3.2. Common Specification

Each SCM architecture has one through four processing elements (PEs) on a chip and each of these has OSCAR RISC architecture CPU core[10, 9]. This CPU core is typical single instruction issue load/store architecture with 32bit fixed instruction length. This CPU also has 64 general-purpose-registers which can be used for integer and floating point operation, and can execute every instruction including floating multiply and floating addition in one clock cycle and the other floating operation in fixed clock cycles.

Each PE has local-program-memory (LPM) which stores program code exclusively generated to the PE by compiler. This LPM can supplies instruction in one clock cycle. This assumption is made because it is thought that performance difference between LPM and instruction cache is small with the workload which is not so large and fits in cache size at hot start time.

Furthermore, each SCM processor has centralized shared memory (CSM) outside the chip. Access latency to the CSM is assumed as 20 clock cycles.

3.3. Shared Data Cache Type Architecture

The shared data cache type architecture assumed in this paper has a large data cache shared among PEs as shown in Figure6. Each PE has a CPU core and local program memory (LPM).

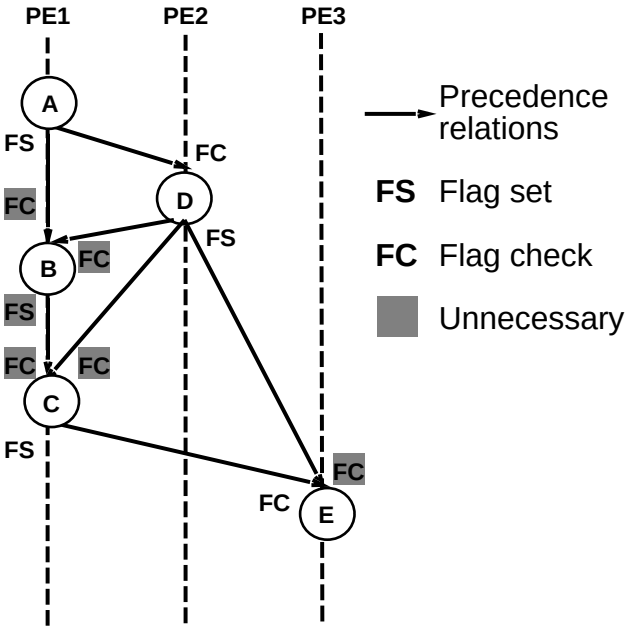

Figure 5. Elimination of redundant synchronization.

This data cache has multiple banks. Each bank is connected to each port by a switch and if multiple cache accesses occur to the same bank, only one access is taken at the clock. The associativity of this cache is 4-way set associative and write policy is write back. The capacity of this shared cache is 4M bytes and access latency is one clock cycle.

In this shared data cache architecture, storing data, loading data and setting synchronization flag take one clock cycle respectively. Checking synchronization flag takes three clock cycles.

3.4. OSCAR Type Architecture

The OSCAR type single chip multiprocessor architecture based on multi processor system OSCAR[10] as show in Figure7 is evaluated.

In the OSCAR type architecture, each PE has CPU, LPM, local data memory (LDM), distributed shared memory (DSM) having two ports and data transfer unit(DTU) which can be used for overlapping of data transfer and computation by compiler control though this function is not used in this paper. These PEs are connected by three buses.

The DSM on each PE can be accessed simultaneously by the PE itself and another PE, and is used for direct data transfer and synchronization among PEs.

The capacity of LDM is 1M bytes per PE respectively

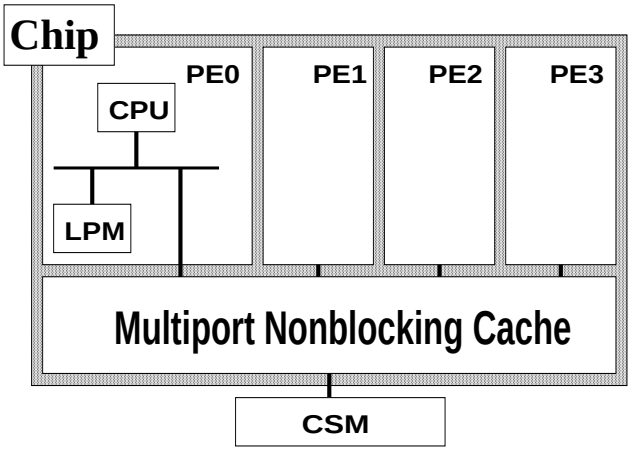

Figure 6. Shared data cache architecture.

and its access latency is one clock cycle. Similarly, the capacity of DSM is 16K bytes per PE respectively and local DSM access latency is one clock cycle and that of remote DSM is four clock cycles.

Using this DSM, near fine grain data transfer does not interfere remote PE execution and can minimize data synchronization overhead since busy-waiting for synchronization flag is performed inside a PE.

In this OSCAR type architecture, storing data and setting synchronization flag take four clocks respectively, loading data takes one clock cycle and checking synchronization flag takes three clock cycles.

3.5. Global Registers

In this paper, effectiveness of multi-port global register (GR) attached to the previous two architectures as sown in Figure8 are evaluated. Number of the global registers is limited in sixteen because of the usable register field in OSCAR instruction set.

Every PE can access GR simultaneously in one clock cycle. These registers are used for near fine grain data transfer and synchronization. Therefore, using global register file, storing data and setting synchronization flag take one clock cycle respectively.

4. Performance Evaluation

In this section, the shared data cache type architecture and the OSCAR type architecture for near fine grain parallel processing are evaluated.

For the evaluation, the following workloads are used.

Random sparse matrix solution

This program is Fortran loop-free code that consists of

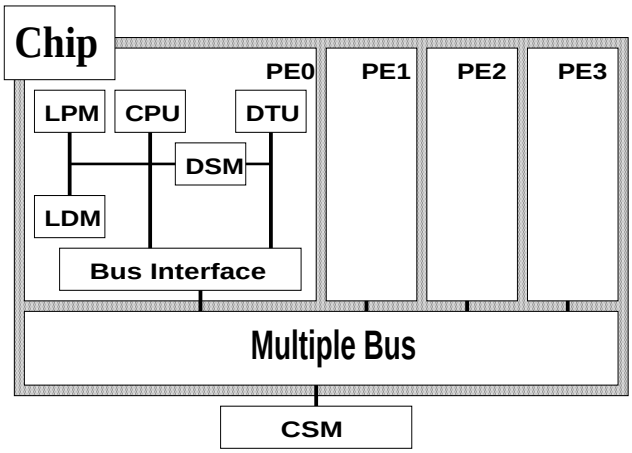

Figure 7. OSCAR type architecture.

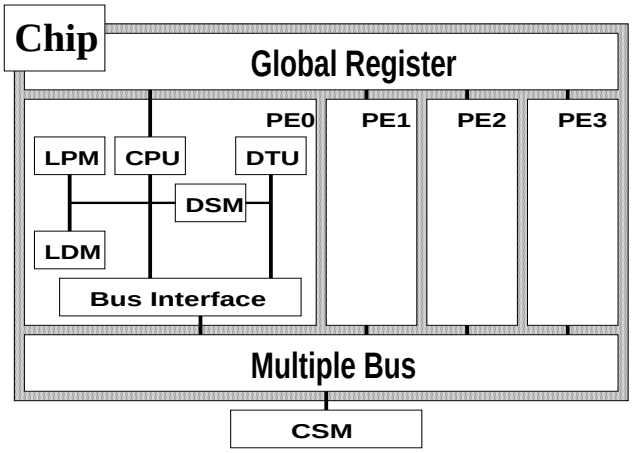

Figure 8. OSCAR type architecture with global register.

arithmetic assignment statements, or 94 near fine grain tasks.

NS3D

This program is a part of CFD program "NS3D" developed by National Aerospace Laboratory in Japan. It is a loop body inside the largest loop inside a subroutine SUB4. This loop body has 429 near fine grain tasks.

FPPPP

This program is a subroutine "FPPPP" of program "FPPPP" from SPECfp95 benchmark programs. This subroutine consumes about 35% of total execution time and has 333 near fine grain tasks.

Electrical circuit simulation

This program is similar to Spice circuit simulation program using the Crout method as a matrix solver. Innermost loop of this program has 221 statements.

These programs are processed in near fine grain on four types of single chip multi processor architectures, such as shared cache (shown as CACHE-WB in Figure9 to Figure12), OSCAR type (OSCAR), shared cache with global registers (CACHE-WB/GR) and the OSCAR with global registers (OSCAR/GR) having 1, 2 or 4 processors.

Figure9 to Figure12 show the speed-up ratio based on sequential execution time on the OSCAR architecture.

Figure9 shows that the OSCAR gives us 3.02 times speed-up for four PEs and the CACHE-WB gives us 2.76 times speed-up. However, Figure10 shows that CACHE-WB achieves 1.76 times speed-up for two PEs and decreases its performance to 1.25 times for four PEs, while the OSCAR achieves 1.50 times speed-up for two PEs and 2.14 times for four PEs.

Figure11 and Figure12 give us almost same result. In Figure11, the OSCAR gives us 2.27 times speed-up for four PEs and the CACHE-WB gives us 1.03 times speed-up. Figure12 shows that the OSCAR gives us 1.55 times speed-up for four PEs and the CACHE-WB gives us 0.91 times speed-up. These results show that the OSCAR type architecture can increase its performance with the increase of number of PEs.

The reason of performance degradation of the CACHE-WB is considered as bank conflict by large amount of data transfer and synchronizations among PEs because all of data transfer and busy-waiting for synchronization flag check use the shared cache. However, in the case of OSCAR type architecture, each PE can minimize the interferences to other PE because it stores PE private date in LDM and directly stores shared data and synchronization flag to the remote DSM.

To examine such performance differences between OSCAR type and CACHE type, the ratio of memory access clocks to total program execution clocks for four PEs is measured. In Figure13 and Figure14, DATA(R) shows the ratio of "read" operations, or "load" to the total clocks. DATA(W) shows the ratio of "write" operations, or "store". SYNC+DT(R) shows the ratio of "check" operations for synchronization flag to "read" operations for received data. SYNC+DT(W) shows the ratio of "send" operations for shared data to "set" operations for synchronization flag. In the figures, EXEC shows the ratio of integer and floating point instruction execution clocks to the total clocks.

In Figure13, the whole memory access ratio of the OSCAR type and that of the CACHE type are almost same. However, SYNC+DT for the CACHE type is lower than the OSCAR type. This is because the CACHE type can send data in one clock, while the OSCAR type takes four clocks in remote DSM access. On the other hand, in Figure14,

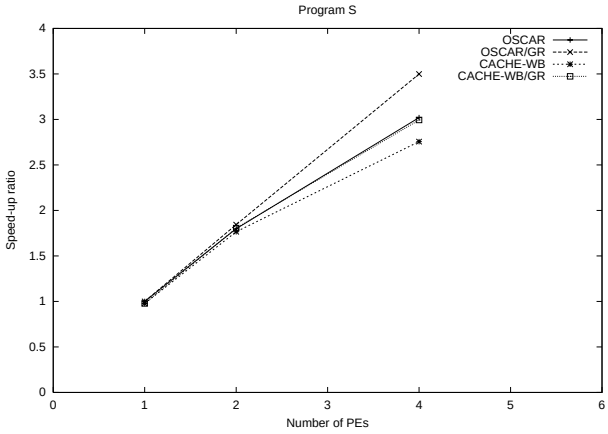

Figure 9. Speed up ratio of random sparse matrix solution.

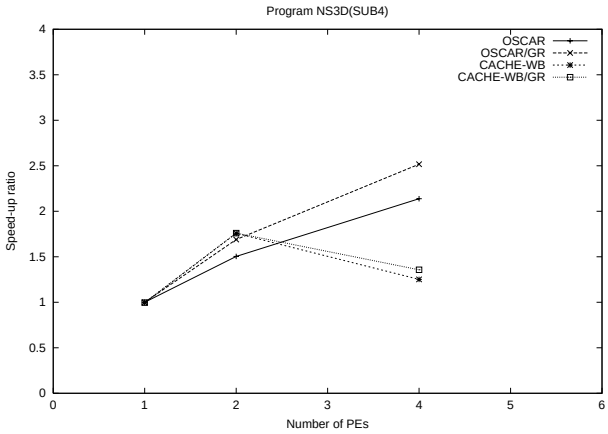

Figure 10. Speed up ratio of NS3D.

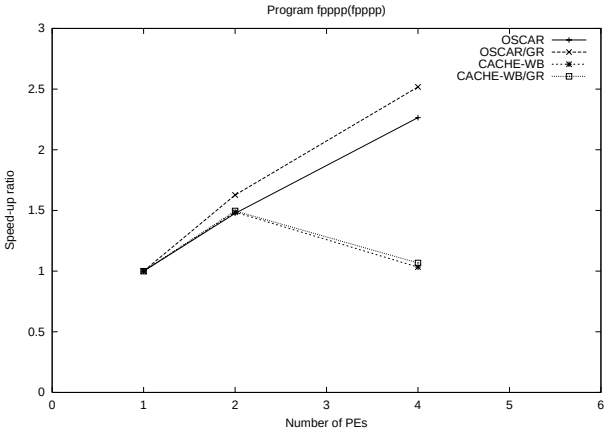

Figure 11. Speed up ratio of FPPPP.

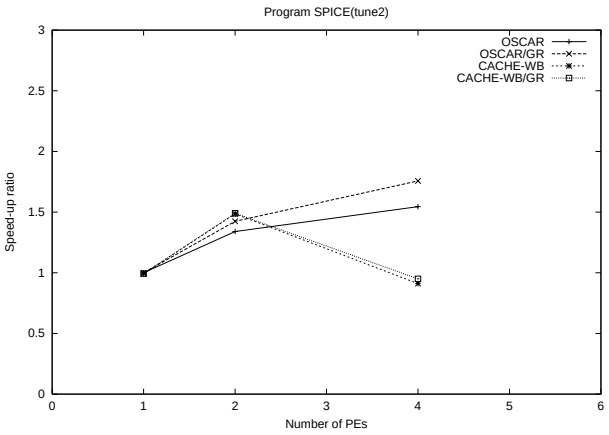

Figure 12. Speed up ratio of Electrical circuit simulation.

the whole memory access ratio is higher than Figure13, and the ratio of DATA(R/W) for the CACHE type is higher than the OSCAR type. This shows that FPPPP needs more data transfer and synchronization than the random sparse matrix solution. These data transfer causes bank conflicts among PEs and interferes loading and storing data. In such an application which gives pressure to shared cache memory, the OSCAR type gives us better performance than the CACHE type.

As to effectiveness of global register, especially OSCAR/GR in Figure10 improves its performance obviously. In this figure, OSCAR/GR improves its performance 17.5% for four PEs than the OSCAR. The attachment of global register is very effective for this kind of application that need many data transfers and synchronization among PEs.

5. Conclusions

This paper has evaluated effectiveness of the four kinds of single chip multiprocessor (SCM) architectures, such as shared data cache type, OSCAR type and those with global registers for near fine grain parallel processing as the first step of research on SCM architecture for supporting multi-grain parallel processing.

The evaluation shows OSCAR type architecture assuming the compiler optimization having local memory and distributed shared memory gives us better performance than shared data cache type architecture in near fine grain parallel processing. For instance, OSCAR type architecture gives us 9.6% better performance for four PEs than shared data cache type in the case of solution of random sparse matrix. OSCAR type also attained 2.19 times better per-

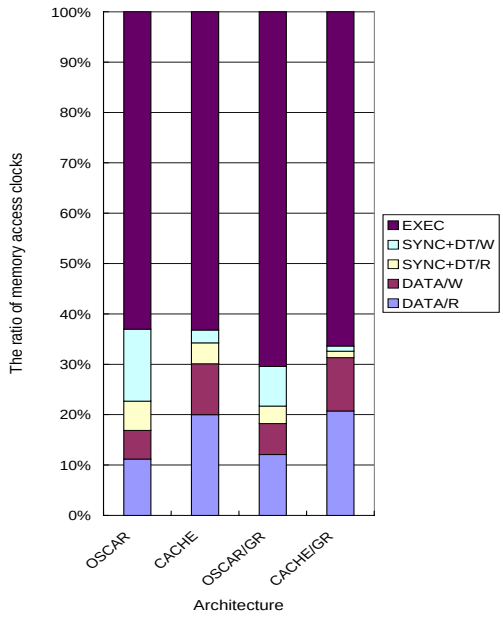

Figure 13. Memory access statistics for random sparse matrix solution.

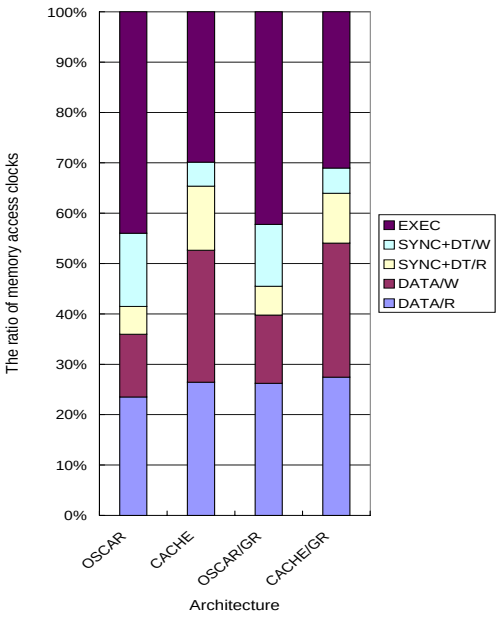

Figure 14. Memory access statistics for FPPPP.

formance for four PEs than shared data cache type in the "FPPPP". Furthermore, adding 16 global registers can improve performance by 10 – 17% for the OSCAR type architecture.

Evaluation of various single chip multi processor architectures including snoop cache and/or different ILP processor architecture like SPARC or PowerPC using more programs from Perfect and SPECfp benchmarks are next research topics.

References

[1] A.Yoshida, K.Koshizuka, and H.Kasahara. Data Localization Using Loop Aligned Decomposition for Macro-Dataflow Processing. In *Proc. of 9th Workshop on Languages and Compilers for Parallel Computers*, Aug 1997.

[2] C.Kozyrakis and D.Patterson. A New Direction for Computer Architecture Research. *Computer*, 31(11):24–32, 1998.

[3] H.Honda, M.Iwata, and H.Kasahara. Coarse Grain Parallelism Detection Scheme of a Fortran Program. *Trans. of IEICE(D-I)*, J73-D-I(12):951–960, 1990.

[4] H.Kasahara. *Parallel Processing Technology*. CORONA PUBLISHING CO.,LTD, 1991.

[5] H.Kasahara and A.Yoshida. A Data-Localization Compilation Scheme Using Partial Static Task Assignment for Fortran Coarse Grain Parallel Processing. *Journal of Parallel Computing*, Special Issue on Languages and Compilers for Parallel Computers, May 1998.

[6] H.Kasahara, H.Honda, A.Mogi, A.Ogura, K.Fujiwara, and S.Narita. A Multi-grain Parallelizing Compilation Scheme for OSCAR(Optimally Scheduled Advanced Multiprocessor). In *Proc.4th Workshop on Lang. And Compilers for Parallel Computing*, Aug 1991.

[7] H.Kasahara, H.Honda, and S.Narita. Parallel processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR (Optimally Scheduled Advanced Multiprocessor). In *Proc. of IEEE ACM Supercomputing '90*, Nov 1990.

[8] H.Kasahara, H.Honda, and S.Narita. A multigrain parallelizing compilation scheme for oscar. In *Proc.4th Workshop on Lang. And Compilers for Parallel Computing*, Aug 1991.

[9] H.Kasahara, M.Okamoto, A.Yoshida, W.Ogata, K.Kimura, G.Matsui, H.Matsuzaki, K.Aida, and H.Honda. OSCAR Multi-grain Architecture and Its Evaluation. In *Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Oct 1997.

[10] H.Kasahara, S.Narita, and S.Hashimoto. Architecture of OSCAR(Optimally SCheduled Advanced multiprocessoR). *Trans. of IEICE*, J71-D(8), 1988.

[11] J.Smith and S.Vajapeyam. Trace processors: Moving to fourth generation microarchitectures. *Computer*, 30(9):68–74, 1997.

[12] K.Diefendorff. Power4 focuses on memory bandwidth. 13(13), 1999.

[13] K.Okulotun, L.Hammond, and M.Willey. Improving the performance of speculatively parallel applications on the hydra cmp. In *Proc. of the 1999 ACM Int'l Conf on Supercomputing*, June 1999.

[14] L.Bic, A.Nicolau, and M.Sato, editors. *Parallel Language and Compiler Research in Japan*. KLUWER ACADEMIC PUBLISHERS, 1995.

[15] M.Okamoto, A.Yoshida, M.Miyazawa, H.Honda, and H.Kasahara. A Hierarchical Macrodataflow Computation Scheme of OSCAR Multi-grain Compiler. *Trans. of IPSJ*, 35(4):513–521, 1994.

[16] R.Barua, W.Lee, S.Amarasinghe, and A.Agarwal. Maps:a compiler-managed memory system for raw machines. In *Proc. of ISCA-26*, June 1999.

[17] T.N.Vijaykumar and G.S.Sohi. Task Selection for a Multiscalar Processor. In *31th Int'l Conf. on Microarchitecture (MICRO-31)*, Nov-Dec 1998.

[18] J.-Y. Tsai, Z. Jiang, E. Ness, and P.-C. Yew. Performance study of a concurrent multithreaded processor. In *Proc.4th Int'l Conf. on HPCA-4*, Fec 1998.

[19] Y.Kang, M.Huang, S.Yoo, Z.Ge, A.Keen, V.Lam, P.Pattnaik, and J.Torrellas. Flexram:an advanced intelligent memory system. In *Proc. Int'l Conf. on Computer Design*, Oct 1999.