Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor

Keiji Kimura, Yasutaka Wada, Hirufumi Nakano, Takeshi Kodaka,
Jun Shirako, Kazuhisa Ishizaka and Hironori Kasahara
Dept. of Computer Science Waseda University.
3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan, 169-8555
{kimura,yasutaka,hnakano,kodaka,shirako,ishizaka,kasahara}@oscar.elec.waseda.ac.jp

Abstract

This paper describes multigrain parallel processing on a compiler cooperative chip multiprocessor. The multigrain parallel processing hierarchically exploits multiple grains of parallelism such as coarse grain task parallelism, loop iteration level parallelism and statement level near-fine grain parallelism. The chip multiprocessor has been designed to attain high effective performance, cost effectiveness and high software productivity by supporting the optimizations of the multigrain parallelizing compiler, which is developed by Japanese Millennium Project IT21 "Advance Parallelizing Compiler". To achieve full potential of multigrain parallel processing, the chip multiprocessor integrates simple single-issue processors having distributed shared data memory for both optimal use of data locality and scalar data transfer, local data memory for processor private data, in addition to centralized shared memory for shared data among processors. This paper focuses on the scalability of the chip multiprocessor having up to eight processors on a chip by exploiting of the multigrain parallelism from SPECfp95 programs. When microSPARC like the simple processor core is used under assumption of 90 nm technology and 2.8 GHz, the evaluation results show the speedups for eight processors and four processors reach 7.1 and 3.9, respectively. Similarly, when 400 MHz is assumed for embedded usage, the speedups reach 7.8 and 4.0, respectively.

1. Introduction

With the improvement of semiconductor technology as described in the Semiconductor Industry Association's report [1], current processor designers can use huge amount of transistors. However, they have faced the problems with power consumption, memory wall, and performance scalability [2, 3]. Cooperative work between software and hardware is important for overcoming these problems.

When focusing on the performance scalability of a microprocessor, limitations in the current superscalar architecture are caused by the limitations of Instruction Level Parallelism (ILP) [4]. Design complexity and the differences between transistor switching speed and wiring delay also complicate performance improvement because of superscalar architecture [5, 6]. Chip Multiprocessors (CMP) (e.g. Stanford Hydra, MIT RAW, Wisconsin Multiscalar, UT Austin TRIPS, IBM POWER4) [7, 8, 9, 10, 11] will serve as the next generation architectures because it can use larger parallelism than ILP. In addition, the multiple processor cores on a CMP can reduce the risk of wiring delay because of the essentially clustered structure.

Software support is crucial to fully exploit the availability of parallelism on a CMP. Therefore, several studies about cooperative work between software and CMP architecture have been completed in RAW, Multiscalar and Hydra [12, 13, 14]. In contrast, multigrain parallel processing has been proposed aiming to boost the effective performance of multiprocessors and to provide high productivity of application development on multiprocessor platforms [15, 16]. In multigrain parallel processing, a compiler exploits three kinds of parallelism from a source program such as an ordinary loop iteration level parallelism, and coarse grain task parallelism among loops and subroutines, in addition to ILP. This parallel processing scheme is especially efficient for CMP because CMP can treat finer grains of parallelism than ordinary multiprocessors, and can naturally treat coarser grains of parallelism than current superscalar processors.

We have developed OSCAR (Optimally SCheduled Advanced multiprocessoR) Chip Multiprocessor (OSCAR CMP) which cooperatively works with a multigrain parallelizing compiler. The goal of OSCAR CMP is to build a scalable, highly effective performance and cost effective computer system for various targets, from embedded computing like mobile phones, PDAs and game machines, to high performance computing. The development of OSCAR CMP includes several projects such as exploit-

ing multigrain parallelism, using data locality, hiding data transfer overhead, and so on by developing the architecture and the multigrain parallelizing compiler simultaneously. This paper especially focuses on the performance of multigrain parallel processing on OSCAR CMP, considering the realistic parameters from current semiconductor technology. The contributions of this paper are that (1) proposing OSCAR CMP and showing how much scalability can be achieved for SPECfp benchmark programs using multigrain parallel processing; and, (2) describing the impact of architectural parameters under multigrain parallel processing for both, low clock frequency of embedded systems, and high clock frequency of high end systems.

The rest of this paper is organized as follows: Section 2 lists several previous works. Section 3 provides an overview of the multigrain parallel processing and generated code image. Section 4 describes an overview of the proposed OSCAR CMP and its architectural support for multigrain parallel processing. Section 5 presents performance evaluation using SPECfp benchmark programs.

2. Related Works

As mentioned in Section 1, much research has been conducted on chip multiprocessor architecture [7, 8, 9, 10, 11].

RAW architecture has somewhat similar architectural approach to OSCAR CMP [8]. RAW architecture integrates many processing elements (named tiles) having a simple processor core and local memory. These resources are managed by a parallelizing compiler.

TRIPS also tries to exploit multilevel parallelism, such as ILP, thread level parallelism (TLP) and data level parallelism (DLP) [10]. TRIPS has an array of GridProcessor and Memory tiles to build both larger dimensions and operate at high clock speeds. Although the TRIPS project also aims to exploit various grains of parallelism simultaneously from one source program like OSCAR CMP, the current evaluation shows each of ILP, TLP and DLP.

Kasahara and his colleagues originally proposed multigrain parallel processing and the OSCAR multiprocessor system [16, 17]. They also evaluated OSCAR multigrain parallelizing compiler using commercial SMP servers, without near-fine grain parallel processing [18]. In addition, they evaluated OSCAR CMP using OSCAR multigrain parallelizing compiler [19, 20]. This paper differs from these prior works in two ways: (1) multigrain parallel processing, including near-fine grain parallel processing, is evaluated on OSCAR CMP using SPEC benchmark programs, then it is described how to exploit and use multigrain parallelism from the applications; and, (2) the architectural parameter of OSCAR CMP is derived using current semiconductor technology.

3. Multigrain Parallel Processing

Multigrain parallelism [16] is defined as the hierarchical use of three types of parallelism, such as coarse grain parallelism among loops, subroutines and basic blocks [16]; loop parallelism among loop iterations; and, near-fine grain parallelism [17] among statements inside a basic block. This multigrain parallelism can be automatically exploited by the OSCAR FORTRAN multigrain parallelizing compiler as a core compiler of Japanese Millennium Project IT21 "Advanced Parallelizing Compiler Project" [21]. At first, the compiler exploits coarse grain task parallelism from a source program. Next, the compiler exploits loop parallelism and near-fine grain parallelism from each coarse grain task hierarchically, as shown in Figure 1. Thus, multigrain parallel processing fully exploits parallelism from a whole source program to obtain scalability.

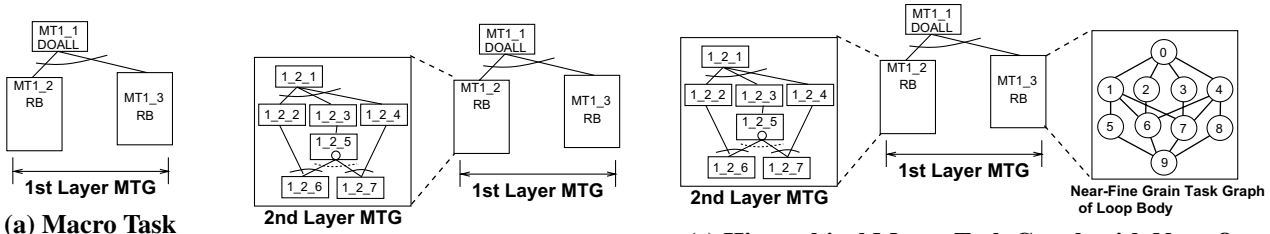
This section provides an overview of multigrain parallel processing. More detailed descriptions were given in the prior works by Kasahara and his colleagues [16, 17, 15]. This section also describes the generated code image of multigrain parallel processing.

3.1. Coarse-grain Task Parallel Processing

The compiler decomposes a source program into three kinds of coarse grain tasks, namely MacroTasks (MTs), such as the Block of Pseudo Assignment statements (BPA), the Repetition Block (RB) and the Subroutine Block (SB) in coarse grain task parallel processing [16]. A BPA is basically defined as an ordinary basic block. However, a basic block is decomposed into several BPAs to extract larger parallelism when that basic block includes independent blocks. If the compiler detects several small basic blocks, the compiler may fuse them into a coarser BPA to reduce scheduling overhead. An RB is a Do loop, or a loop generated by a backward branch. An SB is a subroutine call to which the in-line expansion cannot be efficiently applied.

After generating the MacroTasks, the compiler analyzes control flow and data dependency among MacroTasks. The compiler represents the results of the analysis as a Macro Flow Graph (MFG) as shown in Figure 2 (a). In this figure, the nodes represent MacroTasks. The dotted edges represent control flow. The solid edges represent data dependencies among MacroTasks. The small circles inside the nodes represent conditional branch statements inside MacroTasks.

Next, the compiler analyzes the Earliest Executable Condition of each MacroTask to find maximum parallelism from a MFG [16]. The Earliest Executable Condition of MacroTask i (MT_i) represents a condition under which MT_i



(a) Macro Task Graph

(b) Hierarchical Macro Task Graph

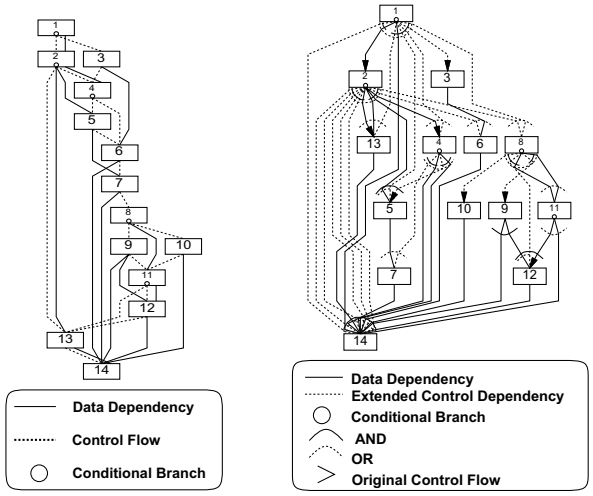
(c) Hierarchical Macro Task Graph with Near-fine grain Task Graph

Figure 1. An Example of Hierarchically Exploiting Multigrain Parallelism from a Source Program.

may begin its execution earliest. These Earliest Executable Conditions for each MacroTask are represented by a directed acyclic graph called Macro Task Graph (MTG), as shown in Figure 2 (b). In the MTG, the nodes also represent MacroTasks. The small circles inside the nodes represent conditional branch statements. The solid edges represent data dependencies. The dotted edges represent extended control dependencies. The extended control dependency means ordinary control dependency and the condition under which data dependent predecessors are not executed. A solid arc means that the edges connected by the arc are in an AND relationship. A dotted arc means that the edges connected by the arc are in an OR relationship.

If a MacroTask of RB or SB has enough coarse grain task parallelism inside its body, the body of this MacroTask can be hierarchically decomposed into sub-MacroTasks, as shown in Figure 1 (a) and (b). Figure 1 (a) shows a MTG extracted from a whole source program. The compiler defines this MTG as the first layer MTG. If the compiler detects coarse grain task parallelism inside MT1_2, the compiler applies the Earliest Executable Condition analysis to the body of MT1_2 and generates the MTG, as shown in Figure 1 (b), as the second layer MTG. In this way, the compiler extracts coarse grain task parallelism hierarchically by applying the Earliest Executable Condition analysis recursively.

Each MacroTask in a MTG is assigned onto a Processor Group (PG), each of which is a group of processors virtually defined by the compiler. If runtime uncertainties exist such as conditional branches among MacroTasks and fluctuations of MacroTask execution time in the target program, MacroTask assignment is decided at runtime by a dynamic scheduling routine. The compiler generates a dynamic scheduling routine exclusively for the target source code and embeds it into a parallelized object code. In contrast, MacroTasks are assigned onto PGs statically in compile time when runtime uncertainties do not exist.



(a) Macro Flow Graph (MFG)

(b) Macro Task Graph (MTG)

Figure 2. A Macro Flow Graph (MFG) and A Macro Task Graph.

3.2. Loop Iteration Level Parallel Processing

MacroTasks are assigned to PGs dynamically or statically, as mentioned in the previous sub-section. If a MacroTask assigned to a PG is a Doall loop, this MacroTask is processed in the medium grain, or iteration level grain, by processors inside the PG.

3.3. Near-fine Grain Parallel Processing

If a MacroTask is a BPA or a sequential RB, this MacroTask is decomposed into statement level near-fine grain tasks. These near-fine grain tasks are processed in parallel by processors inside a PG, as shown in Figure 1 (c) [17]. In this figure, RB of MT1_3 is a sequential loop and its loop body is decomposed into near-fine grain tasks. The compiler generates a near-fine grain Task Graph for MT1_3.

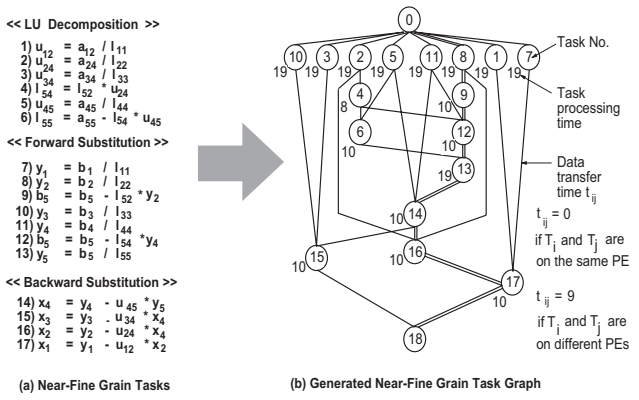


Figure 3. An Example of Near-Fine Grain Tasks and Near-Fine Grain Task Graph.

Figure 3 (a) shows an example of a 17-statement BPA, which solves a random sparse matrix using the Crout method. The compiler analyzes data dependencies among statements and generates a task graph that represents data dependencies among near-fine grain tasks. Figure 3 (b) shows an example of a task graph for the BPA in Figure 3 (a). In Figure 3 (b), a number inside a node circle represents a task number, i, and a number beside the node represents a task-processing time on a processor, t_i. An edge directed from node N_i toward N_j represents a partially ordered constraint caused by the data dependence that the task T_i precedes task T_j. Each edge has a variable weight to represent the data transfer time between tasks. If T_i and T_j are assigned onto different processors, the weight t_{ij} is considered the data transfer time between task T_i and T_j.

These near-fine grain tasks are assigned onto processors statically since there exist only data dependencies among tasks inside a BPA. The compiler uses four heuristic scheduling algorithms, such as CP/DT/MISF, CP/ETF/MISF, ETF/CP and DT/CP [17], and chooses the shortest scheduled result automatically.

After scheduling, the compiler generates a machine code of near-fine grain tasks for each processor. The compiler also embeds machine code for data transfer and synchronization into the required places using statically scheduled results.

3.4. Generated Code Image for Multigrain Parallel Processing

This section gives a generated code image for multigrain parallel processing using Figure 4. Figure 4 shows a code image of the MTG in Figure 1, using eight processors in a chip. As shown in this figure, each processor has its own program code to avoid complexity and overhead of hierarchical parallel task control, such as thread dispatching. At the beginning of program execution, each processor starts its own code. Then, processors communicate with each other for data transfer, synchronization, task control, and so on, using the compiler's appropriately embedded code.

Figure 1 (a) shows that the first layer MTG does not have conditional branches and this MTG has only "2" of parallelism. Therefore, eight processors are grouped into two PGs, each of which has four processors. MT1_1 and MT1_2 are assigned onto processor group0 (PG0) statically. Similarly, MT1_3 is assigned onto processor group1 (PG1). In addition, the compiler places code for sending a synchronization flag immediately after MT1_1 on Processor0 (MT_Sync_S(1)), and places code for receiving a synchronization flag immediately before MT1_3 on Processor4 to Processor7 to maintain the synchronization between MT1_1 and MT1_3.

The second layer MTG inside MT1_2 is processed in parallel on PG0. Distributed dynamic scheduling is applied to this MTG as shown in Figure 4. In distributed dynamic scheduling, each processor group has its own scheduler code to schedule its next MacroTask. Each processor inside PG0 has all the codes for the MacroTasks inside MT1_2 to prepare runtime behavior of that MTG. Scheduling information, such as the Earliest Executable Condition and ready task queue, is shared by all processor groups that process the MTG. When a MacroTask is finished, a scheduler code updates scheduling information exclusively to other processors immediately after that MacroTask, and assigns the next ready MacroTask to its own processor. Then the processor transfers control to assigned MacroTask and executes it. In addition to distributed dynamic scheduling, the compiler also uses centralized dynamic scheduling, which means that one processor is occupied by a scheduler code and no MacroTask is executed on it. Therefore, only a scheduler processor maintains scheduling information.

As to the number of processor groups for MT1_2, PG0 is further divided into two PGs, such as processor group0_0 (PG0_0) and processor group0_1 (PG0_1), each of which has two processors inside it. Each MT inside MT1_2 is processed in parallel by loop parallel processing or near-fine grain parallel processing on these two processors inside PG0_0 or PG0_1.

The loop body of MT1_3 is processed by near-fine grain parallel processing using four processors inside PG1. Each processor (Processor4 to Processor7) has its own code according to the scheduled results of the MT1_3's body. Code for near-fine grain data transfer and synchronization are also placed among near-fine grain task code as described in Section 3.3. In Figure 4, Ti means near-fine grain task code,

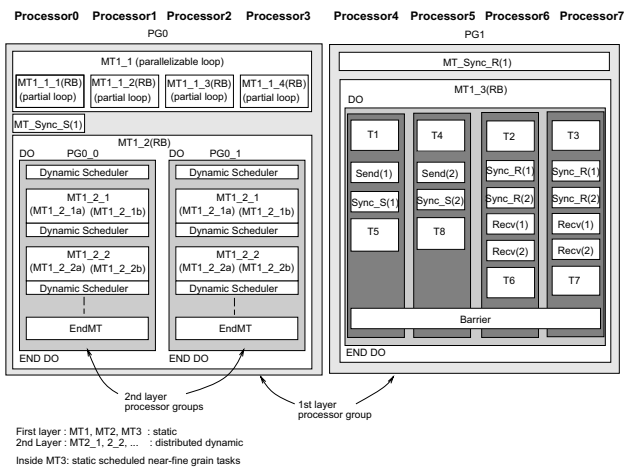


Figure 4. A Generated Code Image for Multigrain Parallel Processing.

Send(j) means near-fine grain data transfer code, Recv(j) means data receive code, Sync_S(k) means sending near-fine grain synchronization flag code, and Sync_R(k) means receiving synchronization flag code. In this example, Processor4 sends data generated by the near-fine grain task T1 at Send(1), and then sends the synchronization flag, which means T1 is finished, at Sync_S(1). In contrast, Processor6 waits for synchronization flags from Processor4 and Processor5 at Sync_R(1) and Sync_R(2), respectively, and then receive data transfered by these processors at Recv(1) and Recv(2), respectively. Finally, a BPA is finished with a barrier synchronization code.

4. Architectural Support for Multigrain Parallel Processing

Proposal of the OSCAR CMP and its architectural support for multigrain parallel processing, especially coarse grain task parallel processing and near-fine grain parallel processing, are described here. One of the most important goals of developing the OSCAR CMP is building a scalable computer system by cooperatively working with a multigrain parallelizing compiler. Therefore, OSCAR CMP heavily depends on the characteristics of multigrain parallel processing. In other words, OSCAR CMP must deal with various sizes of task granularity and communication granularity for coarse grain task parallelism, loop parallelism and near-fine grain parallelism. The principal design concept of OSCAR CMP is multigrain parallelizing compiler controllable simple architecture.

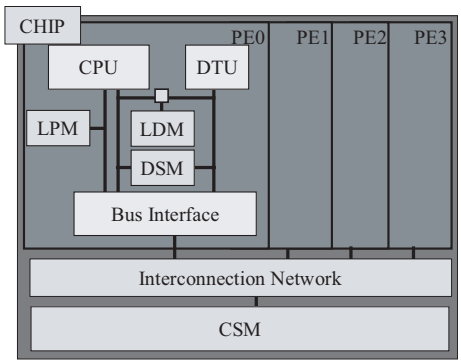


Figure 5. An Overview of OSCAR Chip Multiprocessor.

4.1. OSCAR Chip Multiprocessor Architecture

Figure 5 shows an overview of the OSCAR CMP. OSCAR CMP consists of multiple processor elements (PE) and an interconnection network using multiple buses or a crossbar network. On-chip centralized shared memory (CSM) is also assumed in this paper. Each PE has a simple single-issue CPU core, local program memory (LPM), local data memory (LDM), distributed shared memory (DSM) having two ports, and a data transfer unit (DTU) [19]. Similar architecture, which has local memory or distributed shared memory and simple core, is also proposed from commercial area [22].

LPM stores program code generated by the compiler for each PE. DSM is a dual port memory. DSM provides low-latency data transfer and low-overhead synchronization for coarse grain task parallel processing and near-fine grain parallel processing. LDM stores PE private data. LDM can have twice larger memory size than DSM since LDM only needs a single port.

Note that OSCAR CMP in this paper does not have any cache memory system. Data consistency of DSM is kept by the compiler explicitly.

4.2. Architectural Support for Coarse Grain Task Parallel Processing

As described in Section 3, OSCAR multigrain parallelizing compiler logically forms a different configuration of processor groups (PGs) for each program or even each portion of a program considering the best task granularity and parallelism of a source program. Multiple PEs in each processor group communicate with each other, especially, when a MacroTask executed on that PE is processed in near-fine grain parallel processing as shown in Figure 4. In such a situation, network boundary caused by an in-flat interconnection network is inefficient for processor grouping by the

compiler. OSCAR CMP adopts a flat interconnection network such as a crossbar or multiple buses to support such flexible processor grouping by software.

The compiler uses static and dynamic scheduling depending on a MTG. When the compiler chooses dynamic scheduling, shared data among MacroTasks are assigned to CSM, since tasks that access the shared data are assigned to processors at runtime. In addition, when distributed dynamic scheduling is applied, scheduling information such as a ready task queue and Earliest Executable Conditions are allocated on CSM.

For both static and dynamic scheduling, data communication by a few words among the PGs are required. In static scheduling, a one-word synchronization flag is sent from a predecessor MacroTask to successor MacroTasks when these dependent MacroTasks are assigned to different processors as shown in Figure 4. In dynamic scheduling, scheduling signals such as task starting signals, task finishing signals and conditional branch signals are exchanged among processors. These scheduling signals are also one word. These short-length communications are processed via DSM with a little latency.

Shared array data among RBs and loops should be passed through DSM or LDM for data locality optimization. The compiler decomposed loops and array data so that the decomposed data can be passed through DSM or LDM among the decomposed loops [23]. Furthermore, overlapped array data transfer behind MacroTask execution has been proposed [24]. DTU is used for loading array data from CSM, storing array data to CSM and data transfer among PGs with high throughput.

4.3. Architectural Support for Near-Fine Grain Parallel Processing

Since near-fine grain tasks consist of a few to several tens of machine instructions, it is important that each processor core precisely processes near fine-grain tasks according to the statically scheduled results in near-fine grain parallel processing. Wrong scheduling caused by the unpredictable runtime behavior of processor cores prevents OSCAR CMP from efficient parallel processing. This is one of the reasons that OSCAR CMP uses simple single-issue processor cores since the simple core allows a compiler to predict its runtime behavior.

As to data transfer and synchronization in near fine grain parallel processing, scalar data and synchronization flags come and go across PEs as shown in Figure 4. These data transfer and synchronizations are processed via DSM, similar to scheduling information in coarse grain task parallel processing. Busy waiting loops for synchronization flags at a local DSM area can be performed inside the PE because sender PEs transfer flags to remote PE's DSM directly. This busy wait loops does not degrade the bandwidth of the interconnection network.

Private data detected at compile time using static scheduling information can be located on LDM.

5. Evaluation

This section presents a performance evaluation of multigrain parallel processing on OSCAR CMP. For performance evaluation, two configurations are used under assumptions of low frequency in the embedded system and high frequency in the high-end system.

5.1. Evaluation Environment

To evaluate the scalability of cooperative work between OSCAR CMP and OSCAR multigrain parallelizing compiler, the numbers of processors, or PEs, inside a chip are 1, 2, 4 and 8. The size of LDM, DSM and on-chip CSM are 128KByte, 32KByte and 3MByte, respectively. Memory access latency for each memory is shown in Table 1, in which two parameter sets of memory parameters are evaluated considering both, the low frequency for the embedded system and the high frequency for the high-end system. The parameter set of 400 MHz is assumed to have an aggressive memory parameter because this low memory latency makes the availability of multigrain parallelism clear. These parameters are estimated using CACTI 3.0 [25] under the assumption of 90nm technology. As to the interconnection network and CSM structure, triple buses and four-banked CSM are assumed. A processor core inside a PE is a simple single-issue core like microSPARC [26] although instruction set architecture is SPARC V9 [27].

For this evaluation, seven FORTRAN77 programs from SPECfp95 are used. Data size of each program is reduced, as described in Appendix A, to allocate all data on on-chip CSM and to reduce long evaluation time by the precise architecture simulator. These programs are compiled by the OSCAR multigrain parallelizing compiler using SPARC back-end and processed on the clock accurate OSCAR CMP architecture simulator. This paper especially focuses on evaluating multigrain parallel processing on OSCAR CMP. Data locality optimization and data transfer optimization are out of the scope of this paper even though these are two of the most important topics. Therefore, almost all data are allocated on CSM, and DTU is not used. However, near-fine grain data transfer and synchronization are performed via DSM. Data transfers for dynamic scheduling are also performed via DSM.

Frequency [MHz]	400	2800
LDM (128KB) [clocks]	1	2
DSM (32KB, 2 Ports) [clocks]	1	2
CSM (3MB) [clocks]	2	8
Network [clocks]	2	10

Table 1. Memory Access Latencies for Performance Evaluation

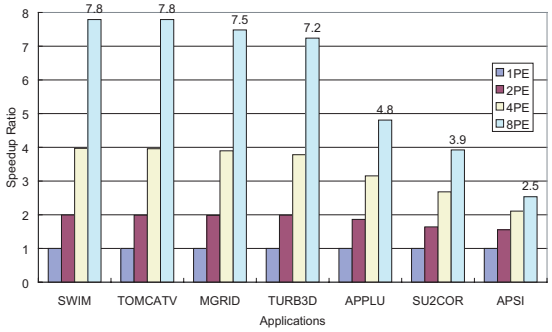


Figure 6. Speedup of SPEC fp programs under the Assumption of 400 MHz, Triple Buses and Four CSM Banks.

5.2. Evaluation Results of Rich Loop Parallelism Applications

Figure 6 and Figure 7 show the results of the performance evaluation under the assumptions of 400 MHz and 2.8 GHz clock frequency, respectively. Each bar in these figures shows speedup against sequential execution time for 2, 4 and 8 processors, or PEs, respectively. Figure 6 shows that OSCAR CMP achieves good scalability under the assumption of 400 MHz for SWIM, TOMCATV and MGRID, which have rich Doall parallelism. Figure 7 also shows good scalability under the assumption of 2.8 GHz.However, their speedups are lower than that of 400 MHz because of relatively higher CSM and remote DSM access latency for 2.8 GHz.

As a result, for these rich Doall parallelism applications, OSCAR CMP achieves scalable performance improvement efficiently for both 400 MHz and 2.8 GHz. Furthermore, there is an opportunity of performance improvement by data locality optimization and data transfer optimization with efficient use of LDM, DSM and DTU, especially in the case of 2.8 GHz.

The following sub-sections present the evaluation results of TURB3D, APPLU and APSI, respectively.

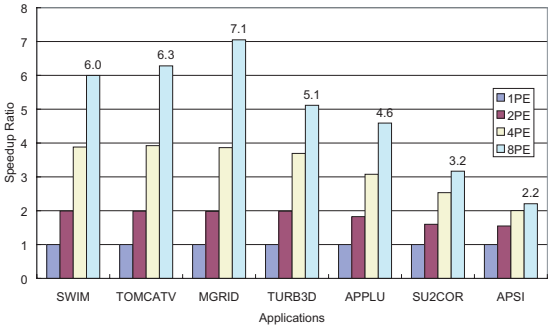


Figure 7. Speedup of SPEC fp programs under the Assumption of 2.8 GHz, Triple Buses and Four CSM Banks.

5.3. Evaluation Results of TURB3D

Subroutine "TURB3D" consumes most execution time of program TURB3D. This subroutine has several coarse grain loops each of which calls subroutine "XYFFT" and "ZFFT". Each of these loops can be processed by loop iteration level parallel processing. In addition, coarse grain task parallelism among these loops can be exploited after applying loop distribution.

Figure 8 shows the speedups of TURB3D, both of results using Doall parallelism only, namely "w/o Coarse Grain", and Multigrain parallelism, namely "w/ Coarse Grain". This figure shows both parallel processing give as almost same speedups until four PEs. However, multigrain parallel processing gives us 1.24 times better performance for 400MHz and 1.13 times for 2.8GHz when eight PEs are used. This result shows multigrain parallel processing can exploit parallelism that conventional loop parallel processing cannot use.

On the other hand, TURB3D also suffers from frequent memory access in the case of 2.8GHz. This result also shows that data locality optimization and data transfer optimization is required.

As a result, OSCAR CMP having eight PEs gives us 7.2 times speedup for 400MHz and 5.1 times speedup for 2.8GHz.

5.4. Evaluation Results of APPLU

APPLU is known for an application in which Doacross parallel processing is effective [28]. However, multigrain parallel processing is applied to this application, other than Doacross parallel processing. Each subroutine "BLTS" and "BUTS", which prevents from simple Doall parallel processing, has several Doall loops inside an outermost triply

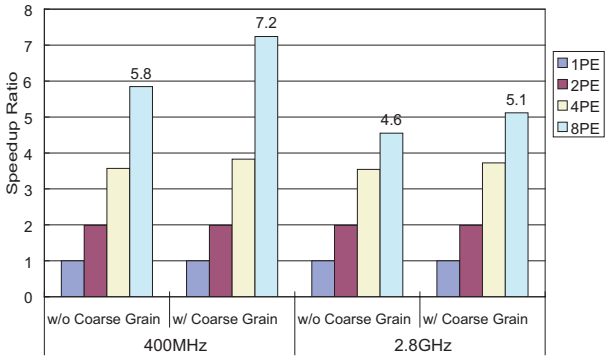


Figure 8. The Speedup Ratio of TURB3D with or without Coarse Grain Task Parallel Processing. Each Bar shows the Speedup against Sequential Execution Time of "w/ Coarse Grain".

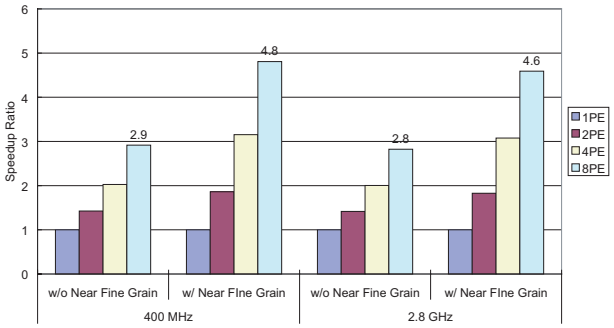


Figure 9. The Speedup Ratio of APPLU with or without Near Fine Grain Parallel Processing. Each Bar shows the Speedup against Sequential Execution Time of "w/ Near Fine Grain".

nested loops. These Doall loops are unrolled and restructured into one large BPA because they have only small fixed number of iterations. The compiler can exploit near fine grain parallelism from subroutine "BLTS" and "BUTS". Of course, Loop parallelism is also exploited from other Doall loops. This optimization is easier than Doacross parallel processing that requires both complex analysis and difficult restructuring.

Figure 9 shows the speedups of APPLU including results that use only Doall parallelism, namely "w/o Near Fine Grain". When eight PEs are used, multigrain parallel processing including near-fine grain parallelism and loop parallelism gives us 1.65 times performance improvement for 400 MHz and 1.63 times performance improvement for 2.8 GHz against Doall only parallel processing.

As a result, OSCAR CMP having eight PEs gives us 4.8 times speedup for 400 MHz and 4.6 times speedup for 2.8 GHz. This is a good example of flexibility and scalability of the multigrain parallel processing.

5.5. Evaluation Results of APSI

APSI is a difficult program for parallel processing. Although this program has many Doall loops, many of these loops have small number of iterations. For instance, 34.1% of sequential execution time is consumed for such small loops whose number of iterations is less than or equal to eight. In addition, there is little near-fine grain parallelism in this application. It seems enough that OSCAR CMP achieves 2.5 times speedup for 400 MHz and 2.2 times for 2.8 GHz, respectively.

6. Conclusions

This paper proposes compiler cooperative OSCAR CMP and describes multigrain parallel processing on OSCAR CMP. The goal of OSCAR CMP is building a scalable, high effective performance and cost effective computer system for various targets from embedded system to high performance system by cooperative work between software and hardware. This paper especially focuses on the scalability of OSCAR CMP using multigrain parallel processing. When 90 nm technology is assumed, the performance evaluation using a simple single-issue processor core shows that OSCAR CMP, having eight processors, achieves 2.5 – 7.8 times speedup against sequential execution under the assumption of 400 MHz for embedded usage, and 2.2 – 7.1 times speedup under the assumption of 2.8 GHz for high performance usage. OSCAR CMP achieves such a good scalability by efficiently exploiting not only each type of grain of multigrain parallelism, such as coarse grain task parallelism, loop level parallelism and near-fine grain statement level parallelism, but also hierarchically combining these types of parallelism. The evaluation results also show that there is an opportunity of more performance improvement by data locality optimization and data transfer optimization using local memory and a data transfer unit, especially in the case of 2.8 GHz.

In addition to data locality optimization and data transfer optimization, performance evaluation using multimedia applications and integer applications are the next important issues. Comparing OSCAR CMP with other memory architectures, such as typical shared cache CMP architectures, is also an important topic.

7. ACKNOWLEDGEMENTS

A part of this research has been supported by STARC "Automatic Parallelizing Compiler Cooperative Single Chip Multiprocessor", Grants-in-Aid for JSPS Fellows(# 1501202), and JSPS Grants-in-Aid for Young Scientists(B)(# 15700074). The authors thank to Mr. Miyamoto (STARC), Mr. Takahashi (Fujitsu), Mr. Yasukawa (Toshiba) and Mr. Edahiro (NEC).

References

[1] Semiconductor Industry Association. International technology roadmap for semiconductors, 2003 edition. In *http://public.itrs.net/*, 2003.

[2] M. R. Stan and K. Skadron. Power-aware computing. *Computer*, 38(12):35–38, Dec. 2003.

[3] M. V. Wilkes. High performance memory systems. *IEEE Transactions on Computers*, 50(11):1105, Nov. 2001.

[4] D. W. Wall. Limits of instruction-level parallelism. In *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Apr. 1991.

[5] J. Hennessy. The future of systems research. *Computer*, 32(8):27–33, 2000.

[6] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proc.of the 27th International Symposium on Computer Architecture (ISCA-27)*, June 2000.

[7] L. Hammond, B. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford HYDRA CMP. *IEEE MICRO Magazine*, 20(2):71–84, 2000.

[8] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler support for scalable and efficient memory systems. *IEEE Trans. on Computers*, 50(11):1234–1247, Nov. 2001.

[9] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA-22)*, 1995.

[10] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture (ISCA 2003)*, June 2003.

[11] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *Technical White Paper*, Oct. 2001.

[12] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA-26)*, June 1999.

[13] T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proc. of the 31st International Conference on Microarchitecture (MICRO-31)*, Nov.–Dec. 1998.

[14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of 8th ACM Conference on Architectural Support for Programming Language and Systems (ASPLOS-VIII)*, Oct. 1998.

[15] H. Kasahara, M. Obata, K. Ishizaka, K. Kimura, H. Kaminaga, H. Nakano, K. Nagasawa, A. Murai, H. Itagaki, and J. Shirako. Performance of Multigrain Parallelization in Japanese Millennium Project IT21 Advanced Parallelizing Compiler. In *Proc. of 10th International Workshop on Compilers for Parallel Computers (CPC) Amsterdam, Netherland.*, June 2003.

[16] 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.

[17] 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 Supercomputing '90*, Nov. 1990.

[18] K. Ishizaka, T. Miyamoto, J. Shirako, M. Obata, K. Kimura, and H. Kasahara. Performance of oscar multigrain parallelizing compiler on smp servers. In *Proc. of 17th International Workshop on Language and Compilers for Parallel computing (LCPC2004)*, Sep. 2004.

[19] K. Kimura and H. Kasahara. Near fine grain parallel processing using static scheduling on single chip multiprocessor. In *Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and systems (IWIA'99)*, pages 23–31, Nov 1999.

[20] K. Kimura, T. Kodaka, M. Obata, and H. Kasahara. Multigrain parallel processing on oscar cmp. In *Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and systems (IWIA'03)*, pages 56–65, Jan. 2003.

[21] Advanced Parallelizing Compiler Project. *Advanced Parallelizing Compiler Home Page*, 2003. http://www.apc.waseda.ac.jp/.

[22] M. Suzuoki and T. Yamazaki. Computer architecture and software cells for broadband networks. *United States Patent Application Publication*, (US 2002/0138637 A1), September 2002.

[23] 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.

[24] H. Kasahara, M. Kogou, T. Tobita, T. Masuda, and T. Tanaka. An automatic coarse grain parallel processing scheme using multiprocessor scheduling algorithm considering overlap of task execution and data transfer. In *Proc. of SCI99 and ISAS*, pages 82–89, Aug. 1999.

[25] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. *WRL Technical Report*, Feb. 2001.

[26] Texas Instruments. *TMS390S10 Microprocessor*, 2000. http: //www.ti.com/ corp/ docs/ company/ history/ microsparc.shtml.

[27] D. L. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice-Hall, Inc, 1994.

[28] B. Pottenger. Parallelism in loops containing recurrences. Technical report, Univ. of Illinois at Urbana-Champaign, June 1996.

A. Evaluated Programs

Evaluated programs and their modified points are described below.

SWIM This program is 102.swim. This is a shallow water modeling program. In this evaluation, array size "M" and "N" in the "test" data set is changed from 512 to 192.

TOMCATV This program is 101.tomcatv. This is a mesh-generation program. In this evaluation, array size "N" in the "test" data set is changed from 257 to 193. The number of loop iterations "ITACT" is also changed from 500 to 10. This program is measured only main loop to reduce the influence of file I/O.

MGRID This program is 107.mgrid. This is a multi-grid solver in 3D potential field program. In this evaluation, parameter "LMI" and "NIT" in the "test" data set are changed from 7 to 5, and 40 to 4, respectively.

TURB3D This program is 125.turb3d. This program simulates isotropic, homogeneous turbulence in a cube. In this evaluation, the parameter "NSTEPS" in the "train" data set is changed from 11 to 6, "NAVG" is changed from 10 to 5, "M1" is changed from 6 to 3, respectively. In addition, "IX", "IY" and "IZ" are changed from 64 to 16.

APPLU This program is 110.applu. This is a parabolic/elliptic partial differential equations program. In this evaluation, the number of loop iterations "ITMAX" in the "train" data set is changed from 50 to 5.

SU2COR This program is 103.su2cor. This is a Monte Carlo simulation program. In this evaluation, parameter "LSIZE(4)" in the "test" data set is changed from (8, 8, 8, 16) to (4, 4, 4, 8), respectively.

APSI This program is 141.apsi. This program solves problems regarding temperature, wind, velocity and distribution of pollutants. In this evaluation, the parameter "NTIME" in the "train" data set is changed from 720 to 3.