OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers

Keiji Kimura, Masayoshi Mase, Hiroki Mikami, Takamichi Miyamoto, Jun Shirako, and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan,
{kimura,mase,hiroki,miyamoto,shirako}@kasahara.cs.waseda.ac.jp,
kasahara@waseda.jp,
http://www.kasahara.cs.waseda.ac.jp/

Abstract. OSCAR (Optimally Scheduled Advanced Multiprocessor) API has been designed for real-time embedded low-power multicores to generate parallel programs for various multicores from different vendors by using the OSCAR parallelizing compiler. The OSCAR API has been developed by Waseda University in collaboration with Fujitsu Laboratory, Hitachi, NEC, Panasonic, Renesas Technology, and Toshiba in an METI/NEDO project entitled "Multicore Technology for Realtime Consumer Electronics." By using the OSCAR API as an interface between the OSCAR compiler and backend compilers, the OSCAR compiler enables hierarchical multigrain parallel processing with memory optimization under capacity restriction for cache memory, local memory, distributed shared memory, and on-chip/off-chip shared memory; data transfer using a DMA controller; and power reduction control using DVFS (Dynamic Voltage and Frequency Scaling), clock gating, and power gating for various embedded multicores. In addition, a parallelized program automatically generated by the OSCAR compiler with OSCAR API can be compiled by the ordinary OpenMP compilers since the OSCAR API is designed on a subset of the OpenMP. This paper describes the OSCAR API and its compatibility with the OSCAR compiler by showing code examples. Performance evaluations of the OSCAR compiler and the OSCAR API are carried out using an IBM Power5+ workstation, an IBM Power6 high-end SMP server, and a newly developed consumer electronics multicore chip RP2 by Renesas, Hitachi and Waseda. From the results of scalability evaluation, it is found that on an average, the OSCAR compiler with the OSCAR API can exploit 5.8 times speedup over the sequential execution on the Power5+ workstation with eight cores and 2.9 times speedup on RP2 with four cores, respectively. In addition, the OSCAR compiler can accelerate an IBM XL Fortran compiler up to 3.3 times on the Power6 SMP server. Due to low-power optimization on RP2, the OSCAR compiler with the OSCAR API achieves a maximum power reduction of 84% in the real-time execution mode.

Key words: Multicore API, Parallelizing Compiler, Power Reduction

1 Introduction

Multicore processors have been widely used in a variety of applications such as embedded (consumer electronics) systems, PCs, workstations, and high-performance computers. In consumer electronics applications, various types of multicore processors are used in a wide variety of applications such as image and audio processing, face recognition, and real-time controls. For example, from the memory architecture point of view, while there are conventional SMP multicores, many multicores in consumer electronics are equipped with a local memory (or scratch pad memory) such as CELL/BE[1], RP1[2], and RP2[3]. Furthermore, some multicores have a distributed shared memory such as FR1000[4], RP1, and RP2 and some have an on-chip shared memory such as MP211[5], RP1, and RP2 in addition to a local memory. On the other hand, many multicores are equipped with some kind of power control mechanisms inside a chip, such as DVFS; fine-grained clock gating; and per-core power gating for ensuring long battery life, preserving reliability, and enabling small fan-less package.

While current multicores for consumer electronics have several features that are responsible for achieving high performance with low power, the cost for application development is increasing since programmers must take care of usage for those architectural features such as memory management and power management.

Some compilers, programming languages, and APIs have been developed to mitigate the application development cost for consumer electronics multicores. Source-to-source parallelizing compilers[6, 7] have been developed for ordinary multiporcessor systems. These compilers preserve portability among different multiprocessor systems by generating parallelized C or Fortran programs. However, multi-platform parallelizing compilers for embedded computing systems have not yet been developed since there are few multi-platform parallel APIs. Though the Multicore Association has developed multi-platform multicore APIs and the MCAPI has been released as a communication API[8], other APIs are still under development. OpenCL has been developed as a multi-platform parallel API[9]. However, it is intended to be mainly used on accelerators like GPGPU.

On the other hand, the OSCAR multigrain parallelizing compiler has been developed to fully exploit the potential of both the ordinary multiprocessors and the multicores. The OSCAR compiler enables multigrain parallel processing[10–12]; memory control under capacity restriction for cache memory, local memory, distributed shared memory, and on-chip/off-chip shared memory[13, 14]; data transfer using DMA controller[15]; and power reduction control using DVFS, clock gating, and power gating[16]. In order to apply these optimization using the OSCAR compiler for a variety of consumer electronics multicores, the OSCAR API has also been developed. That is the OSCAR compiler generates parallelized C or Fortran programs with OSCAR API, and backend compilers for target multiprocessors and multicores generate executable object files from those parallelized programs.

The OSCAR API is designed on a subset of OpenMP[17], which is the defact standard of parallel programing for shared memory multiprocessors. The

OSCAR API consists of four directives from the OpenMP and 12 newly prepared directives. The main differences between OSCAR API and prior parallel languages and APIs are that the OSCAR API supports distributed shared memory and on-chip centralized shared memory in addition to local data memory in a simple manner, and it employs a user-level power control API. The basic design policy of the OSCAR API is to keep its specification as simple as possible.

In this paper, the directives in the OSCAR API and its compatibility with the OSCAR compiler are described. Then, performance evaluation of a combination of OSCAR compiler and OSCAR API using the IBM Power5+ workstation, the IBM Power6 SMP server, and the newly developed consumer electronics 8-core low-power multicore RP2 is carried out.

The rest of this paper is organized as follows. Section 2 provides an overview of the OSCAR compiler and its execution model. Section 3 describes the directives in the OSCAR API. The code example related to the OSCAR compiler is also shown in this section. Section 4 shows the performance evaluation of a combination of OSCAR compiler and OSCAR API. Finally, Section 5 summarizes the main conclusion of this paper.

2 OSCAR Parallelizing Compiler and Its Execution Model

This section provides an overview of the OSCAR multigrain parallelizing compiler and its execution model. In order to minimize the runtime overhead of parallel processing, the OSCAR compiler adopts one-time single level thread creation as its execution model. This execution model creates threads at the program start point. Synchronizations and scheduling require only two parallel processing primitives. This execution model affects the design decision of the OSCAR API.

2.1 Overview of OSCAR Parallelizing Compiler

Multigrain parallel processing exploits multiple grains of parallelism such as coarse grain task parallel processing, loop iteration level parallel processing, and statement level near fine grain parallel processing. In this study, loops, function calls, and basic blocks are defined as coarse grain tasks.

In order to apply multigrain parallel processing to an ordinary sequential program, the OSCAR compiler firstly decomposes a source C or Fortran program into coarse grain tasks, namely macro-tasks (MTs), such as basic block (BPA), loop (RB), and function call or subroutine call (SB). Then, the compiler analyzes both the control flow and the data dependencies among MTs and represents them as a macro-flow-graph (MFG). Next, the compiler applies the earliest executable condition analysis, which can exploit parallelism among MTs associated with both the control dependencies and the data dependencies. The analysis result is represented as a macro-task-graph (MTG). If an MT is a subroutine call or a loop that has coarse grain task parallelism, the compiler hierarchically generates inner

MTs inside that MT. Then, the compiler groups processor cores into processor groups (PG) logically and hierarchically,

These MTs are assigned to processor cores by the compiler. If the MTG has conditional branches or runtime fluctuations, dynamic scheduling is applied to it. Otherwise, static scheduling is applied.

After generating MTGs, the compiler applies loop iteration level parallel processing if an MT has loop iteration level parallelism. If an MT does not have loop iteration parallelism but has statement level parallelism, such an MT is processed by statement level near fine grain parallel processing[12].

Data locality optimization and data transfer optimization can be applied after generating MTGs. If multiple MTs share same data, whose size is greater than that of the cache memory or the local memory, the OSCAR compiler decomposes these MTs into smaller MTs in order to fit the shared data accessed by each MT into the cache or the local memory by loop aligned decomposition[13]. Then, these decomposed MTs are scheduled onto processor cores in order to assign MTs, which access same smaller data, successively as much as possible[14]. If the target architecture has a local memory, the compiler assigns processor private data to the local memory and generates data transfer codes between the main memory and the local memory. These data transfer codes are overlapped MT execution as much as possible by data transfer optimization[15].

If there are idle or busy-waiting periods between MTs in a statically scheduled MTG, the compiler tries to minimize total power dissipation by prolonging the execution time of MTs with DVFS or applying clock gating and power gating during the idle periods. This execution mode is named as the fastest execution mode. Similarly, if the deadline of an MTG is given and there are sufficient idle periods until the deadline, the compiler also applies DVFS, clock gating, and power gating[16]. This execution mode is named as the deadline execution mode. If a power-optimized MTG with deadline is processed iteratively as in the case of a movie player, this execution mode is named as real-time execution mode.

2.2 Execution Model of Multigrain Parallel Processing

The OSCAR compiler adopts one-time single level thread creation as its runtime execution model. This execution model generates threads for each processor core only once at the program start time. Each thread is assumed to be bound to only one processor core during the program execution time. MTs generated by the OSCAR compiler are mapped onto these thread codes. In the case of hierarchical parallel processing control, the scheduling and synchronization codes are processed by an ordinary program code with two runtime primitives. Therefore, the compiler minimizes the runtime overhead for parallel execution.

Here, the one-time single level thread creation model is shown in Fig.1. In this figure (Fig.1-(a)), the source program has four MTs such as MT1_1 – MT1_4 at the top-level of the program (the first layer MTG). MT1_3 and MT1_4 have internal MTGs each of inside them (second layer MTG). These MTs are to be assigned eight threads, as shown in Fig.1-(b).

The first layer MTG is scheduled statically at compile time since there is no conditional branch inside it. The program codes for MT1_1, MT1_2, and MT1_4 are mapped onto the Processor Group 0 involving Thread0 – Thread3, and MT1_3 are mapped onto the Processor Group 1 involving Thread4 – Thread7, respectively. Two second layer MTGs are scheduled dynamically at runtime using dynamic scheduling codes generated by the compiler, due to the presence of conditional branches. In the case of the one-time single level thread creation model, execution codes for all MTs are generated onto all threads when dynamic scheduling is applied to the target MTG. At the scheduled time, a dynamic scheduling code selects the next MT and assigns it onto the appropriate thread by sending a scheduling information to that thread.

Considering the synchronizations of this example, there are synchronization codes, i.e., "SYNC SEND" after MT1_1 and "SYNC RECV" before MT1_3, generated by the data dependence between these MTs. The OSCAR compiler generates an assignment statement to a synchronization flag variable as "SYNC SEND", and a busy-waiting loop against this flag variable as "SYNC RECV." Similarly, barrier synchronizations are carried out after MTs inside MT1_4 are processed by combinations of assignment statements and busy-waiting loops with barrier flag variables. Note that such synchronization codes require a memory consistency primitive such as "flush" directive in the OpenMP.

With regard to dynamic scheduling, the OSCAR compiler generates distributed scheduler codes and centralized scheduler codes according to the parallelism of the target MTG and the number of available processor cores. Distributed scheduler codes are embedded after each MT, as shown in MT1_4. Centralized scheduler codes occupy one thread, as shown in Thread7 of MT1_3. In the case of distributed scheduler codes, scheduling information such as ready task queue and scheduling table must be processed exclusively in critical sections. Therefore, a lock primitive such as "critical" directive in the OpenMP is required.

In summary, thread creation, memory consistency primitive and lock primitive are required for one-time single level thread creation.

3 OSCAR Application Program Interface

This section describes the OSCAR API. This paper especially focuses on how to associate the OSCAR API with the OSCAR compiler by showing code examples. Detailed specifications of the OSCAR API v1.0 is available at our website [18]. Note that in this paper, there are a few points that differ between OSCAR API v1.0 and the OSCAR API, since the OSCAR API is being still discussed for heterogeneous multicores and many cores. These points are presented in the remaining of this section.

3.1 Overview of OSCAR API

The OSCAR API is designed on a subset of OpenMP for preserving portability over a wide range of multicore architectures. An OpenMP-based design can

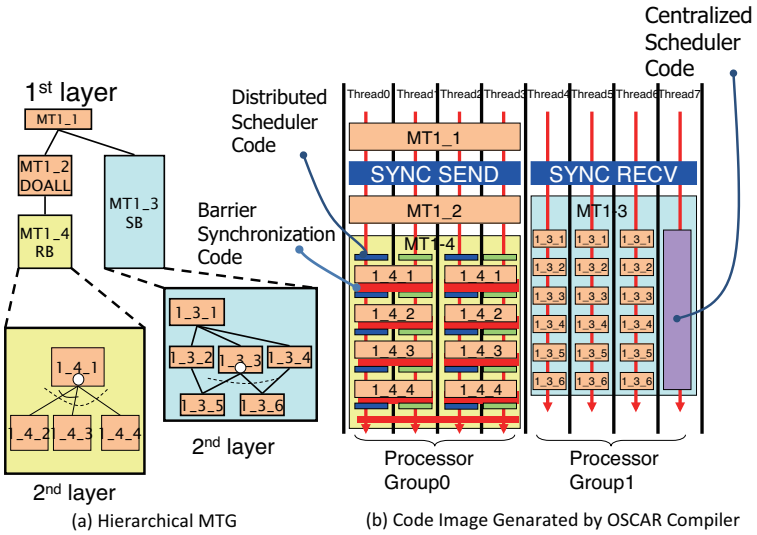


Fig. 1. Example of hierarchical MTG (a) and a code image of parallelized code generated by OSCAR compiler (b).

support both C and Fortran programs. However, in order to avoid the complexity of a backend compiler and runtime routines, only three directives are chosen from the OpenMP, such as "parallel sections," "flush," and "critical," which enable one-time single level thread creation, as described in Section2.2. Note that nested parallelism is not required for the OSCAR API.

In addition to these three directives, one OpenMP directive (threadprivate) is extended, and 12 directives are newly added to support the previously mentioned parallel optimizations carried out using the OSCAR compiler, whose specifications are simple as possible. Furthermore, the OSCAR memory architecture[12] is defined as the model multicore architecture of the OSCAR API, as shown in Fig.2.

The OSCAR memory architecture consists of multiple multicore chips and an off-chip CSM (Centralized Shared Memory) module. Each multicore chip has multiple processor cores and an on-chip CSM. Each processor core has a CPU, an LDM (Local Data Memory) for core private data, a DSM (Distributed Shared Memory) for synchronization flags and shared data, a DTC (Data Transfer Controller), a TIMER (Timer Unit), an FVR (Frequency and Voltage Control Register) and a GROUPBAR (Group Barrier Synchronization module). Each module in the OSCAR memory architecture may have the FVR. The directives in the OSCAR API are related to those modules in the OSCAR memory architecture. If the target architecture does not have some of those modules, directives related with those modules can be ignored.

Fig.3 shows a list of directives in the OSCAR API. These directives are classified into six categories such as Parallel Execution API, Memory Mapping API, Synchronization API, Data Transfer API, Power Control API, and Timer API. In this paper, Parallel Execution API, Memory Mapping API, Power Control API, and Timer API are explained in the remaining of this section.

The compile flow of the OSCAR compiler with the OSCAR API is described as following: First, a sequential C or Fortran program are parallelized by the OSCAR compiler. If a source program is a C program, this program is written in "Parallelizable C," which stands for C with some restriction around pointer usage for ease of parallelization by the compiler. The OSCAR compiler generates a parallelized C or Fortran code with the OSCAR API. This parallelized code is then compiled by an OpenMP compiler for a server platform, or it is translated into C or Fortran program with runtime library calls generated by the API analyzer in front of the backend compiler of the target multicore. Finally, the backend compiler generates an executable object for the target multicore.

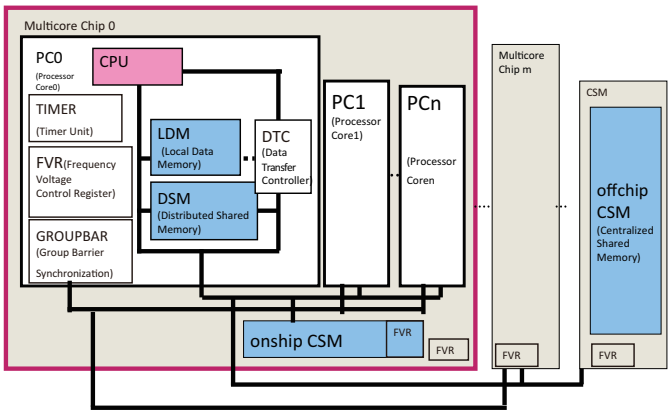


Fig. 2. OSCAR Memory Architecture as the Model Multicore Architecture

3.2 Parallel Execution API and Memory Mapping API

Parallel Execution API consists of four directives, i.e., parallel section, flush, critical, and execution. Memory Mapping API also consists of three directives, i.e., threadprivate, distributedshared, and onchipshared.

Fig.4 shows an example of one-time single level thread creation by Parallel Execution API. The program shown in Fig.4-(a) is written in C and that shown in Fig.4-(b) is written in Fortran. As shown in these figures, all the threads are created at the program start time only once. In the OSCAR API, the processor cores involved in parallel execution at runtime are called as Virtual Cores

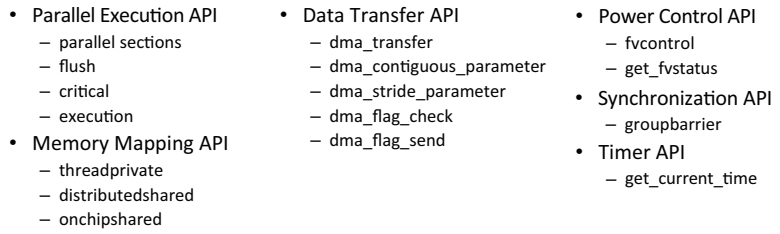


Fig. 3. List of Directives in OSCAR API

(VCs)[1]. Each thread is statically bound to one VC with an increasing order of VC number. For example, `main_VC0` is executed on VC0, `main_VC1` is executed on VC1, and so on.

These figures also show an example of one-to-one synchronization between VC0 and VC4. This synchronization is processed via variable `syncflag` and `myversion`. Note that `syncflag` is a shared variable among VCs. On the other hand, `myversion` is thread private data, which is stored in the local memory if the target architecture has a local memory. The synchronization code generated by the OSCAR compiler is realized by the version number method. Threadprivate variable `myversion` is incremented at each synchronization time. The receiver VC, i.e., VC4 in this example, waits for VC0 to send a flag by comparing the shared variable `syncflag` with the private variable `myversion`. Memory consistency is appropriately maintained by the compiler-inserted flush directives.

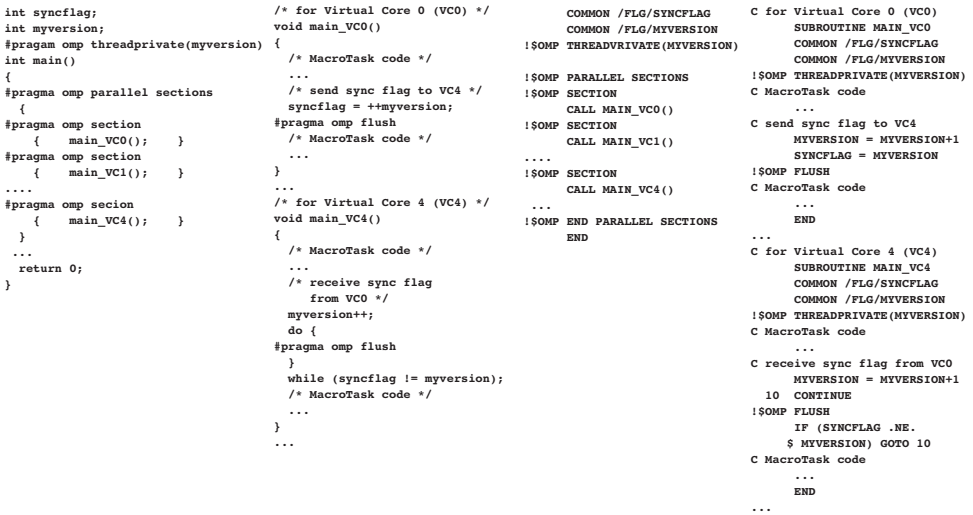
Here, if `syncflag` is specified in an onchipshared directive such as `myversion` in a threadprivate directive, `syncflag` is stored in the onchip centralized shared memory if the target architecture has an onchip centralized shared memory. Similarly, if `synclfag` is specified in a distributedshared directive, for example, just after the fourth parallel section directive, `syncflag` is stored in the distributed shared memory inside VC4.

3.3 Power Control API and Timer API

The Power Control API consists of the fvcontrol and the get_fvstatus directives. The fvcontrol directive sets the power status of a module to a specified value. The get_fvstatus acquires the current power status from a specified module. The power status notation used in the Power Control API is an integer value ranging from -1 to 100. The value from 0 to 100 represents the percentage of clock frequency of the specified module. For example, 100 is the maximum clock frequency, 50 is half of the maximum clock frequency, and 0 represents clock off or clock gating. -1 denotes power gating.

In order to specify the target module such as in the Power Control API, the OSCAR-module description is introduced. The syntax of the OSCAR-module description is as follows[2]:

[1] Virtual Processor Core (VPC) in V1.0
[2] OSCAR_MODULE is FV_MODULE in V1.0



```
int syncflag;
int myversion;
#pragam omp threadprivate(myversion)
int main()
{
#pragma omp parallel sections
  {
#pragma omp section
    {    main_VC0();    }
#pragma omp section
    {    main_VC1();    }
....
#pragma omp secion
    {    main_VC4();    }
  }
 ...
  return 0;
}
```

```
/* for Virtual Core 0 (VC0) */
void main_VC0()
{
  /* MacroTask code */
  ...
  /* send sync flag to VC4 */
  syncflag = ++myversion;
#pragma omp flush
  /* MacroTask code */
  ...
}
...
/* for Virtual Core 4 (VC4) */
void main_VC4()
{
  /* MacroTask code */
  ...
  /* receive sync flag
     from VC0 */
  myversion++;
  do {
#pragma omp flush
  }
  while (syncflag != myversion);
  /* MacroTask code */
  ...
}
...
```

(a) Example in C

```
      COMMON /FLG/SYNCFLAG
      COMMON /FLG/MYVERSION
!$OMP THREADVRIVATE(MYVERSION)

!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL MAIN_VC0()
!$OMP SECTION
      CALL MAIN_VC1()
....
!$OMP SECTION
      CALL MAIN_VC4()
 ...
!$OMP END PARALLEL SECTIONS
      END
```

```
C for Virtual Core 0 (VC0)
      SUBROUTINE MAIN_VC0
      COMMON /FLG/SYNCFLAG
      COMMON /FLG/MYVERSION
!$OMP THREADPRIVATE(MYVERSION)
C MacroTask code
      ...
C send sync flag to VC4
      MYVERSION = MYVERSION+1
      SYNCFLAG = MYVERSION
!$OMP FLUSH
C MacroTask code
      ...
      END
...
C for Virtual Core 4 (VC4)
      SUBROUTINE MAIN_VC4
      COMMON /FLG/SYNCFLAG
      COMMON /FLG/MYVERSION
!$OMP THREADPRIVATE(MYVERSION)
C MacroTask code
      ...
C receive sync flag from VC0
      MYVERSION = MYVERSION+1
  10  CONTINUE
!$OMP FLUSH
      IF (SYNCFLAG .NE.
     $ MYVERSION) GOTO 10
C MacroTask code
      ...
      END
...
```

(a) Example in FORTRAN

Fig. 4. Example of Parallel Execution API

```
([[chip,]vc,](OSCAR_MODULE([submodule]),parameter_list))
```

Here, parameters enclosed in "[]" can be omitted.

"chip" and "vc" specify the chip number and the VC number, respectively. If -1 is specified, the specified module is a chip or a core shared module such as an on-chip centralized shared memory. If "chip" and "vc" are omitted, the specified module is owned by the VC that processes the directive. "submodule" specifies the sub-module inside a module by an integer value, if this parameter exists. This can be used for specifying memory banks inside a memory module. "parameter_list" is a parameter list, which is given to the specified module.

Fig.5-(a) shows a simple example of the Power Control API. In this example, VC4 sleeps until VC0 reaches synchronization points. Then, VC0 wakes VC4 up and sends a synchronization flag. Thus, VC4 can reduce power dissipation by busy-waiting for VC0.

A combination of Power API directives and the get_current_time directive from the Timer API can achieve power reduction when a program has a deadline, or for real-time processing. The get_current_time acquires the current elapsed time in the order of micro second. Fig.5-(b) shows an example of power reduction for a program with a deadline. When VC0 completes the required MTs, VC0 reduces its clock frequency by 25% of the maximum clock frequency. Then, VC0 waits for the specified deadline by monitoring the timer with the get_current_time directive. In this example, "0" in the get_current_time directives denotes the timer channel number.

```
/* for Virtual Core 0 (VC0) */
void main_VC0
{
  /* MacroTask code */
  ...
  /* wake-up VC4 */
#pragma oscar fvcontrol \
  (4,(OSCAR_CPU(),100))
  /* send sync flag to VC4 */
  syncflag = ++myversion;
#pragma omp flush
  /* MacroTask code */
  ...
}
...
```

```
/* for Virtual Core 4 (VC4) */
void main_VC4()
{
  /* MacroTask code */
  ...
  /* sleep until
     VC0 wakes me up */
#pragma oscar fvcontrol \
  ((OSCAR_CPU(),0))
  /* receive sync flag from VC0 */
  myversion++;
#pragma omp flush
  while (syncflag != myversion) {
#pragma omp flush
  }
  /* MacroTask code */
  ...
}
...
```

(a) Simple Power Control Example

```
/* for Virtual Core 0 (VC0) */
void main_VC0
{
  /* MacroTask code */
  ...
  /* wait until deadline
     with low-speed */
#pragma oscar fvcontrol \
  ((OSCAR_CPU()),25)
#pragma oscar get_current_time(time1,0)
  ...
  while (1) {
#pragma oscar get_current_time(time2,0)
    if (deadline<=time2-time1+overhead)
      break;
    /* wait loop */
    for (i=0;i<dummycount;i++);
  }
#pragma oscar fvcontrol \
  ((OSCAR_CPU()),100)
  /* wake-up other VCs */
#pragma oscar fvcontrol \
  (1,(OSCAR_CPU()),100)
...
}
...
```

(b) Power Control with Deadline Example

Fig. 5. Example of Power Control API

4 Experimental Evaluations

4.1 Evaluation Environment

The IBM p5 550Q workstation and the p6 595 UNIX server are used for the scalability evaluation. The p5 550Q has four Power5+ processors, each of which has two cores. C programs are evaluated on this machine with IBM XL C/C++ for AIX Compiler v10.1 as a backend compiler. The p6 595 has 16 Power6 processors, each of which has two cores. Fortran programs are evaluated on this machine with IBM XL Fortran for AIX v12.1. The purpose of this evaluation is to show that the combination of the OSCAR compiler and the OSCAR API works well for the ordinary combination of OpenMP compilers and server machines. The applicability of the OSCAR API for both C and Fortran is also shown.

In addition to these machines, the consumer electronics multicore RP2 is also used for the evaluation. RP2 is developed by Renesas Technology, Hitachi and Waseda University in the METI/NEDO project. RP2 has eight SH4A low-power cores driven at 600 MHz on a die. From the memory architecture point of view, RP2 is an implementation of the OSCAR memory architecture described in Section 3.1. This chip supports both the SMP mode and the AMP mode. In the SMP mode, RP2 can be used as two SMP clusters, each of which has four SH cores. Cache coherency is maintained at the L1 cache. In the scalability evaluation, only four cores, whose cache coherency are maintained, are used. In the AMP mode, the local memory, distributed shared memory, and DTU are used. With regard to the power reduction control mechanism of RP2, DVFS, clock gating, and power gating for each processor core can be controlled independently

by software. This mechanism also strongly supports low overhead and fine grain clock and ensures power and voltage control using the OSCAR compiler.

An API analyzer is developed for RP2. A parallelized C code generated by the OSCAR compiler is translated into a C code with memory allocation annotations and some runtime library calls for thread creation, data transfer, power control, and timer reading. Then, the translated C code is compiled into an executable binary code by the SH Compiler.

Specifications of these two systems are summarized in Table 4.1.

Table 1. Evaluation Environment

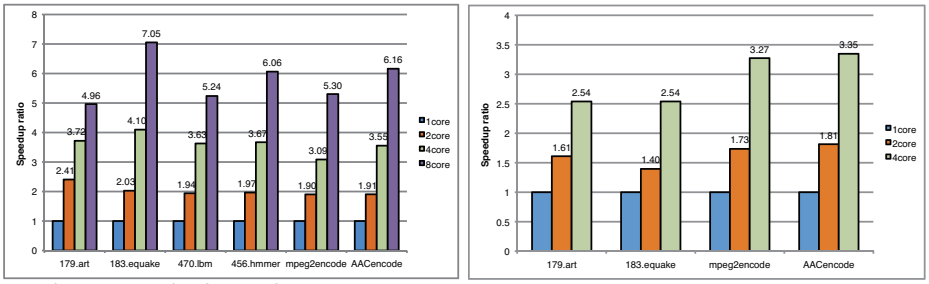
System	IBM p5 550Q	IBM p6 595	Renesas/Hitachi/Waseda RP2
CPU	Power5+ (1.5 GHz x 2 cores x 4 chips)	Power6 (4.2 GHz x 2 cores x 16 chips)	SH-4A (600 MHz x 8 cores, 4 cores are used)
L1D-Cache	32 KB/core	64 KB/core	16 KB/core
L1I-Cache	32 KB/core	64 KB/core	16 KB/core
L2 Cache	1.9 MB/2 cores	4 MB/2 cores	N/A
L3 Cache	36 MB/2 cores	32 MB/2 cores	N/A

4.2 Performances

For carrying out scalability evaluation using IBM p5 550Q and RP2, art and equake from SPEC 2000, lbm and hmmer from SPEC 2006, mpeg2encode from MediaBench, and an AAC encoder available on a market from Renesas Technology are used. The pointer usage is restricted for equake, lbm, hmmer, and mpeg2encode. In other words, they are modified in Parallelizable C. Some restructurings are also applied to them for the ease of parallelization by the compiler. These modified C programs are used for both IBM XL C/C++ compiler and the OSCAR compiler. For RP2, lbm and hmmer are not evaluated due to size limitation of the main memory. With regard to the IBM p6 595, 14 Fortran applications are used, such as tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fpppp, and wave5 from SPEC 95, and swim, mgrid, applu and apsi from SPEC 2000[19].

Fig.6-(a) and (b) show speedups over the sequential execution on IBM p5 550Q and RP2, respectively. As shown in these figures, both p5 550Q and RP2 achieve good scalability along with an increase in the number of processor cores. For example, 7.1 times speedup for equake and 6.2 times speedup for AACencoder can be achieved over the sequential execution on p5 550Q with eight cores, and 3.3 times speedup for mpeg2encode and 3.4 times speedup for AACencoder can be achieved on RP2 with four cores. On an average, p5 550Q achieves 5.8 times speedup with eight cores, and RP2 achieves 2.9 times speedup with four cores. These results show that the combination of the OSCAR compiler and

the OSCAR API can exploit parallelism in programs and ensure performance improvement for both the servers and the consumer electronics multicores.



(a) Evaluation Result Obtained Using IBM p5 550Q (b) Evaluation Result Obtained Using RP2

Fig. 6. Scalability Evaluation Results on IBM p5 550Q and RP2

Fig.7 shows the maximum speedup of p6 595 against sequential execution up to 32 cores by the OSCAR compiler and XL Fortran compiler. The compiler options of XL Fortran are "-O5 -qarch=pwr6 -q64" for sequential execution and "-O5 -qsmp=auto -qarch=pwr6 -q64" for automatic parallelization by XL Fortran, respectively. As shown in this figure, 26 times speedup for swim and 14 times speedup for mgrid, both of which are from SPEC 2000, can be achieved by the OSCAR compiler with OSCAR API. Similary, 11 times sppedup and 2.1 times spped for the same applications can be achieved by the IBM XL Fortran. On an average, the OSCAR compiler achieves 7.3 times speedup over the sequential execution, and the IBM XL Fortran achieves 2.3 times speedup, respectively. In summary, the OSCAR compiler can accelerate the XL Fortran compiler by up to 3.3 times by generating parallelized program in OpenMP or OSCAR API.

Fig.8 shows the evaluation result of power optimization in the real-time execution mode using RP2 with up to eight cores. An AAC encoder available on a market from Renesas Technology and an MPEG2 decoder from MediaBench are used for this evaluation. The MPEG2 decoder is also modified in Parallelizable C in this evaluation. An audio stream or movie data is processed per frame. The deadline of the AAC encoder is set to each frame process so that 44.1[frames/sec] can be achieved. Similarly, the deadline of the MPEG2 decoder is set to each frame process so that 30[frames/sec] can be achieved. Fig.8 shows the average power comparison between the minimum number of cores, which can satisfy the deadline restriction, and eight cores with and without low-power optimization. For example, the minimum number of cores required for the deadline satisfaction of the AAC encoder is one and that of the MPEG2 decoder is two.

Therefore, in the case of the AAC encoder, 66% of power consumption can be reduced by applying low-power optimization with one core, and 84% of power consumption can be reduced with eight cores. In this case, one core with low-power optimization and eight cores with low-power optimization consume almost

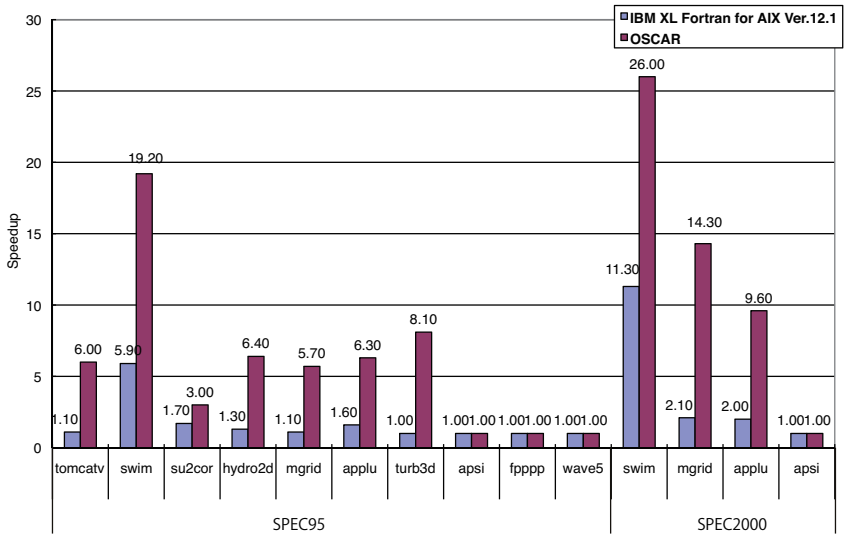


Fig. 7. Scalability Evaluation Results obtained using IBM p6 595

the same power since one core is sufficient for real-time encoding. Similarly, in the case of the MPEG2 decoder, 20% of power consumption can be reduced with two cores, and 76% of power consumption can be reduce with eight cores. Note that eight cores with low-power optimization consumed 33% lower power than two cores. This is because the power control duration becomes longer in the case of eight cores than in the case of two cores by the parallel execution of the MPEG2 decoder.

In summary, the OSCAR API can realize low overhead parallel execution and low-power optimization applied by the OSCAR compiler.

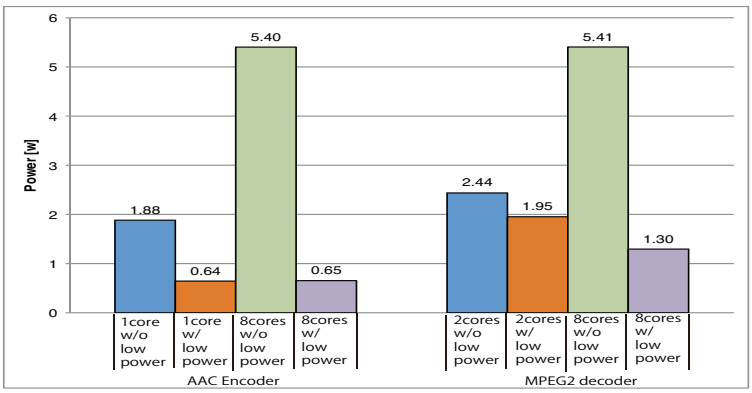


Fig. 8. Evaluation of Power Optimization in the Real-time Execution Mode

5 Conclusions

This paper has described the OSCAR API, which realizes multigrain parallel processing, power reduction, DMA transfer and real-time processing using the OSCAR compiler for various types of multicore or multiprocessors, such as from consumer electronics multicores to high-performance servers. The OSCAR API consists of Parallel Execution API, Memory Mapping API, Data Transfer API, Synchronization API, Power Control API and Timer API. The evaluation results show the OSCAR API can be applicable from consumer electronics multicores to SMP servers, and it allows us to achieve good scalability with the number of processor cores. The OSCAR API can also realize low-power optimization using OSCAR compiler with a low overhead. For example, on an average, the OSCAR compiler with the OSCAR API gives us 5.8 times speedup over the sequential execution on the 8-core Power5+ workstation and 2.9 times speedup on RP2 with four cores, respectively. The low-power optimization by the OSCAR compiler and the API on RP2 achieves a maximum power reduction of 84% for AAC encoder in the real-time execution mode. In future, we intended to evaluate local memory management and data transfer optimization. In addition, we also plan to extend the OSCAR API toward heterogeneous multicores and many cores.

Acknowledgement

This paper is supported by the METI/NEDO projects “Multicore Technology for Realtime Consumer Electronics”, “Heterogeneous Multicore for Consumer Electronics” and “Low Power Manycore Processor Systems Leading Research.” Specifications of OSCAR API are discussed at the Realtime Consumer Electronics Multicore Architecture and API Committee in these projects. The authors specially thanks to the members of the committee from Fujitsu Laboratory, Hitachi, NEC, Panasonic, Renesas Technology, and Toshiba.

References

1. Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation cell processor. In: Proc. of IEEE International Solid State Circuits Conference (ISSCC2005). (February 2005)
2. Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., Odaka, T., Takada, K., Kimura, K., Kasahara, H.: ”a 4320mips four-processor core smp/amp with individually managed clock frequency for low power consumption”. In: Proc. of IEEE International Solid State Circuits Conference (ISSCC2007). (February 2007)
3. Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Ito, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako,

J., Mase, M., Kimura, K., Kasahara, H.: An 8640 mips soc with independent power-off control of 8 cpu and 8 rams by an automatic parallelizing compiler. In: Proc. of IEEE International Solid State Circuits Conference (ISSCC2008). (February 2008)
4. Shiota, T., Kawasaki, K.I., Kawabe, Y., Shibamoto, W., Sato, A., Hashimoto, T., Hayakawa, F., Tago, S.I., Okano, H., Nakamura, Y., Miyake, H., Suga, A., Takahashi, H.: A 51.2gops, 1.0gb/s-dma single-chip multi-processor integrating quadruple 8-way vliw processors. In: Proc. of IEEE International Solid State Circuits Conference (ISSCC2005). (February 2005)
5. Torii, S., Suzuki, S., Tomonaga, H., Tokue, T., Sakai, J., Suzuki, N., Murakami, K., Hiraga, T., Shigemoto, K., Tatebe, Y., Obuchi, E., Kayama, N., Edahiro, M., Kusano, T., Nishi, N.: A 600mips 120mw 70ua leakage triple-cpu mobile application processor chip. In: Proc. of IEEE International Solid State Circuits Conference (ISSCC2005). (February 2005)
6. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., wei Liao, S., Bugnion, E., Lain, M.S., Benchmark, S.: Maximizing multiprocessor performance with the suif compiler. IEEE Computer **29** (1996) 84–89
7. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with polaris. IEEE Computer **29** (1996) 78–82
8. Association, T.M.: Multicore communication api specification
9. : http://www.khronos.org/opencl/
10. Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task parallel processing on smp using openmp. In: Proc. of 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC'00). (August 2000)
11. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical parallelism control for multigrain parallel processing. Lecture Notes in Computer Science **2481** (2005) 31–44
12. Kimura, K., Wada, Y., Nakano, H., Kodaka, T., Shirako, J., Ishizaka, K., Kasahara, H.: Multigrain parallel processing on compiler cooperative chip multiprocessor. In: Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9). (February 2005)
13. Yoshida, A., Koshizuka, K., Kasahara, H.: Data-localization for fortran macro-dataflow computation using partial static task assignment. In: Proc. of 10th ACM International Conference on Supercomputing. (May 1996)
14. Ishizaka, K., Obata, M., Kasahara, H.: 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 (LCPC2001). (August 2001)
15. Kasahara, H., Kogo, M., Tobita, T., Masuda, T., Tanaka, T.: An automatic coarse grain parallel processing scheme using multiprocessor scheduling algorithms considering overlap of task execution and data transfer. In: Proc. SCI99 and ISAS99. (August 1999)
16. Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K., Kasahara, H.: Compiler control power saving scheme for multi core processors. Lecture Notes in Computer Science **4339** (2007) 362–376
17. : http://www.openmp.org/
18. : http://www.kasahara.cs.waseda.ac.jp/
19. Mikami, H., Shirako, J., Mase, M., Miyamoto, T., Nakano, H., Takano, F., Hayashi, A., Wada, Y., Kimura, K., Kasahara, H.: Performance of oscar multigrain parallelizing compiler on multicore processors. In: Proc. of 14th Workshop on Compilers for Parallel Computing(CPC 2009). (January 2009)