OSCAR API v2.1: Extensions for an Advanced Accelerator Control Scheme to a Low-Power Multicore API

Keiji Kimura, Cecilia González-Álvarez, Akihiro Hayashi, Hiroki Mikami, Mamoru Shimaoka, Jun Shirako, and Hironori Kasahara

Department of Computer Science and Engineering, Waseda University,
27 Waseda-machi, Shinjuku-ku, Tokyo, Japan,
kimura@apal.cs.waseda.ac.jp,
{cecilia,ahayashi,hiroki,shimaoka,shirako}@kasahara.cs.waseda.ac.jp,
kasahara@waseda.jp
http://www.kasahara.cs.waseda.ac.jp/

Abstract. The number of cores in smartphones and tablet-PCs are rapidly increasing along with their required high computational power. However, almost all applications on those devices have not used multiple cores for their high speed and low power execution since the application development environments, which allow the application developers easy and prompt development of parallelized application, are not available. In addition to the development of parallelized applications, low-power consumption techniques and efficiently use of accelerators such as GPUs are required to application developers. In order to provide more productive application development environment for multicores, an automatic parallelizing compiler, OSCAR compiler, which parallelizes C and Fortran programs automatically by applying multi-grain parallelization, local-memory and cache optimization and low-power optimization, has been developed. Furthermore, the OSCAR API has been also developed as an interface between the OSCAR Compiler and various kinds of shared memory multicores including homogeneous and heterogeneous manycores with SMP, cc-NUMA and PGAS architectures from various vendors, such as ARM, Intel, IBM, AMD, Tilera, Fujitsu, Renesas Electronics, and so on. The OSCAR API v1.0 and v2.0 have been opened and their specifications are available from http://www.kasahara.cs.waseda.ac.jp/. In the OSCAR API v1.0, fundamental thread control, memory control, DMA-control, power control and flexible group barrier are supported. In the OSCAR API v2.0, various kinds of heterogeneous multicores are supported. In this paper, the API extensions in OSCAR API v2.1, which includes control schemes for asynchronously executable accelerators, and hint directives for low-power optimizations, are described in addition to brief review of the OSCAR API v1.0 and v2.0. A flexible and low-overhead accelerator control scheme that allows us overlapped execution of CPUs, accelerators and DMA controllers can be realized by newly added accelerator control APIs.

Key words: Multicore API, Parallelizing Compiler, Heterogeneous Computing

1 Introduction

Today's many computer systems are organized as multicores and accelerators to provide much computational power within limited power budget.

There are two popular approaches to add accelerators. The one is adding SIMD functional units inside a pipeline of the host CPU like Intel SSE, Intel AVX and ARM NEON[1–3]. The other is attaching accelerator modules, like NVIDIA GPUs[4, 5] and Intel Xeon Phi[6], to, or sometimes inside, the host CPU chip. Both approaches are used not only for desktop and server computers, but also for embedded computers.

The first approach is especially suitable for short vector computations since embedded SIMD functional units are driven by extended SIMD instructions in the host CPU. However, extension for the pipeline and the instruction set architecture is required to the host CPU. This extension requires expensive design cost both for hardware and system software including compilers.

The second approach is suitable for long vector computations since the accelerator module is separated from the pipeline of the host CPU. However, this approach has been suffered from expensive control and data transfer overhead between the host CPU and an accelerator module especially when they are connected by an external bus like PCI-express. In other words, the host CPU and the accelerator module are difficult to share the workload because of their large communication overhead.

Regarding to the software development environment, there have been several APIs and compiler frameworks especially for GPU, which is one of the most popular accelerator modules. For example, CUDA developed by NVIDIA has been widely used for GPGPU programing[7]. OpenCL and OpenACC try to support various accelerators in addition to GPUs[8, 9]. In order to mitigate expensive control and data transfer overhead between the host CPU and accelerators, these programing environments provide asynchronous accelerator execution model. Application programmers have been able to develop applications with more general programing style than before since those programing environments are extended from ordinarily C and Fortran specifications. However, there have been still difficulties to develop parallelized programs that can fully use of multiple cores and accelerators on a target computer system. Developers must still parallelize there own applications very carefully by their hands.

In order to overcome the difficulty on the development of parallelized applications for heterogeneous multicore systems in addition to homogeneous multicores, the OSCAR multigrain parallelizing compiler has been developed. The OSCAR compiler enables multigrain parallel processing[10–12], cache and local memory optimizations[13, 14], power reduction optimizations[15], and automatic parallelization considering accelerators[16]. The OSCAR API (Application Program Interface) has been also developed to apply these optimizations by the OSCAR compiler onto various multiprocessor and multicore systems, including servers, desktop computers and embedded systems. The first version of the OSCAR API (OSCAR API v1.0) supports thread creation, data allocation considering local data memory, distributed shared memory and on-chip/off-chip

shared memory, and it employs a user-level power control API[17]. From the second version, the OSCAR API (OSCAR API v2.0)[18] has supported accelerator modules with blocking execution model, and also cache control directives for future manycores.

This paper proposes a flag based accelerator control framework among CPUs, accelerators and DMACs. New three directives are also added to OSCAR API (OSCAR API v2.1) in order to realize that flag based accelerator control. This paper also describes this extension of the OSCAR API.

The rest of this paper is organized as follows: Section 2 provides an overview of the OSCAR compiler. Section 3 describes an overview of the OSCAR API v1.0 and v2.0. Section 4 introduces the flag based accelerator control framework. Section 5 describes the extension to the OSCAR API for the proposed flag based acceleration control. Finally, Section 6 summarizes the main conclusion of this paper.

2 OSCAR Compiler

This section provides an overview of the OSCAR multigrain 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. In this scheduling time, if the target architecture has accelerators, the OSCAR compiler assigns each MT on a processor core or an accelerator core depending on the MT's characteristics and availability of the processor cores and accelerator cores[18, 16].

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[19].

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[15]. 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.

3 OSCAR API

This section briefly introduces the OSCAR API v1.0 and v2.0, respectively. Then, the compilation flow of the OSCAR compiler using the OSCAR API is described. Evaluation results of the OSCAR Compiler and the OSCAR API are also shown in this section.

3.1 OSCAR API v1.0

The OSCAR API is designed on a subset of OpenMP[20] for preserving portability over a wide range of shared memory multicore architectures. An OpenMP-based design can support both C and Fortran programs.

The directives of the OSCAR API v1.0 are decided for the target multicore architecture, namely the OSCAR architecture shown in Fig.1. The OSCAR 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, data caches, instruction caches, an LPM (Local Program Memory), 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 kind of intelligent DMA controller), a Timer (Timer Unit), an FVR (Frequency and Voltage Control Register) and a Group Barrier (Group Barrier Synchronization module). Each module in the OSCAR architecture may have an FVR.

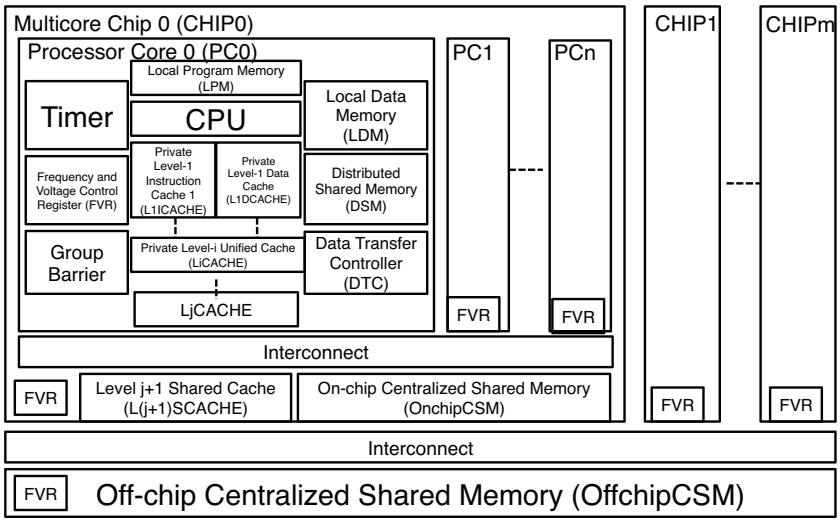

Fig. 1. OSCAR Multicore Architecture

Though the OSCAR API defines the OSCAR architecture as the target architecture, the API dose not require that the final target architecture has all of those modules except off-chip CSM. For example, the OSCAR API can be applied to the multicore, which has only instruction caches, data caches and off-chip CSM like an ordinary SMP multicore architecture, by ignoring directives related with LDM and DSM. Thus the OSCAR API can be applied to various kinds of shared memory type multicore systems.

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`". These three directives enables the parallel execution model of the OSCAR compiler, namely one-time single level thread creation[10, 14].

In the OSCAR API v1.0, one OpenMP directive (threadprivate) is extended, and 12 directives are newly added in addition to the previously mentioned three directives from the OpenMP. Fig.2 shows a list of these directives in the OSCAR API v1.0. 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. These APIs control modules in the OSCAR architecture. For example, in the case of Memory Mapping API, "threadprivate", "distributedshared" and "onchipshared" locate variables on LDM, DSM and on-chip CSM, respectively.

3.2 OSCAR API v2.0

From the OSCAR API v2.0, accelerator directives for heterogeneous multicores and cache control directives for non-coherent cache architecture have been sup-

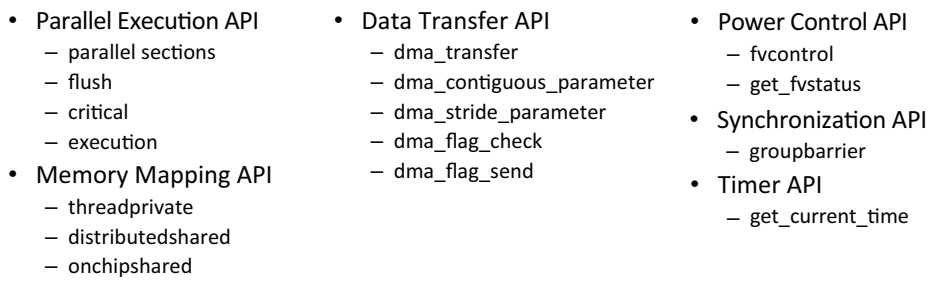

Fig. 2. List of Directives in OSCAR API v1.0

ported. In this paper, only accelerator related directives are described because of the scope of this paper.

Fig.3 shows a target heterogeneous multicore architecture in the OSCAR API v2.0. A processor core can have accelerator modules as shown in this figure in addition to the OSCAR architecture shown in Fig.1. Each accelerator must be coupled with a controller CPU, which manages initialization and data transfer for accelerators. It is recommended that an accelerator module and the corresponding controller CPU are equipped in the same processor core as in Fig.3 because of low communication overhead between them. However, an accelerator and the controller CPU can be located in different processor cores like ordinary heterogeneous system having GPUs.

Fig.4 shows added directives in the OSCAR API v2.0. One directive for accelerator control and five directives for cache control are added. In addition, two hint directives for the OSCAR compiler are added.

Regarding to the accelerator modules, an "`accelerator_task`" hint directive can be used for specifying a loop or a function call, which can be executed on accelerator modules. The OSCAR compiler uses this hint directive for its task scheduling considering accelerators[18, 16]. An "`accelerator_task_entry`" specifies an entry function that initiates the target accelerator and starts the program code for the accelerator. When a controller CPU calls an entry function specified by an `accelerator_task_entry`, this controller CPU waits for the completion of the function executed on an accelerator since the OSCAR API v2.0 supports a blocking execution model as an accelerator execution model.

3.3 Compilation Flow of OSCAR Compiler with OSCAR API

Fig.5 shows an image of the compile flow of the OSCAR compiler with the OSCAR API. In this figure, "Parallelizable C" stands for C with some restriction around pointer usages for ease of parallelization by the compiler.

Firstly, a sequential C or Fortran program is taken into the OSCAR compiler. For heterogeneous parallelization, accelerator compilers or application developers insert "`accelerator_task`" hint directives to specify the part of the program,

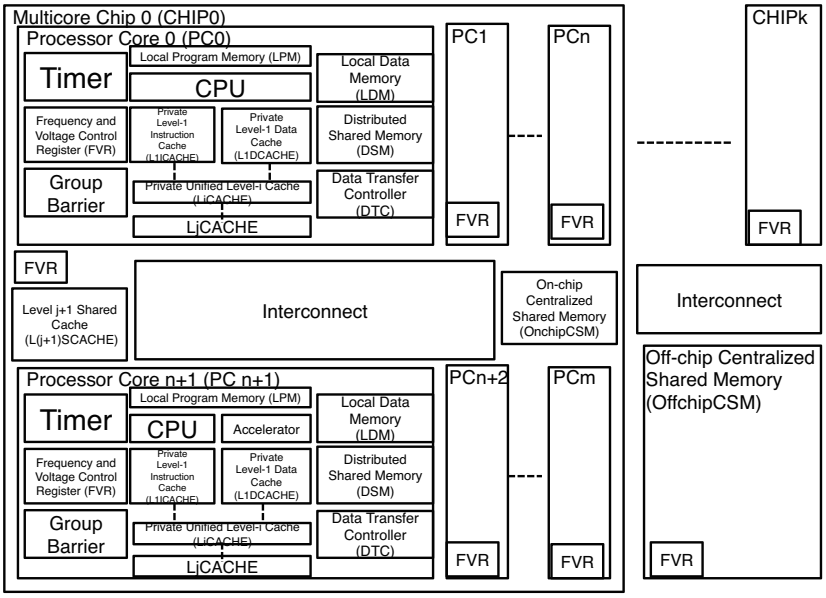

Fig. 3. OSCAR Heterogeneous Multicore Architecture

- Accelerator API
  - accelerator_task_entry
- Hint directives for OSCSR compiler
  - accelerator_task
  - oscar_comment
- Cache Control API
  - cache_writeback
  - cache_selfinvalidate
  - complete_memop
  - noncacheable
  - aligncache

Fig. 4. List of Added Directives in OSCAR API v2.0

which can be executed on accelerators, before the parallelizaiton by the OSCAR compiler.

Then, the source sequential program is parallelized by the OSCAR compiler. The OSCAR compiler generates a parallelized C or Fortran code by inserting directives in the OSCAR API described in Section3.1 and Section3.2. When the compiler assigns tasks onto accelerators, the generated codes for accelerator tasks are placed on separate files from those for the CPUs. The entry functions for the accelerator tasks are annotated by "`accelerator_task_entry`" directives.

This parallelized code is then compiled by an OpenMP compiler for server platform, or translated into a C or Fortran program with run-time library calls by an API translator in front of a backend compiler of the target multicore.

Finally, the backend compiler generates an executable object for the target multicore. Accelerator programs are compiled by accelerator compilers and liked with the CPU object files.

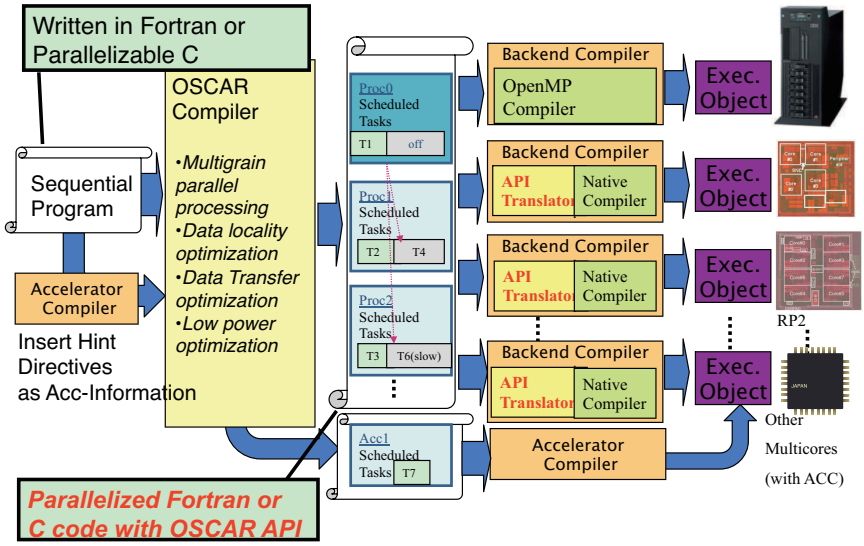

Fig. 5. Compilation Flow of OSCAR Compiler with OSCAR API

3.4 Evaluations

The evaluation results of the OSCAR compiler and the OSCAR API on a server machine and a heterogeneous embedded multicore are shown in this section.

For the evaluation on the server machine, Hitachi SMP RS440, which is equipped with Intel Xeon X7560 driven at 2.27GHz, is used. RS440 has four Intel Xeon processors and each of them has 8 cores. Therefore, this server machine has totally 32 cores. Fortran and C benchmark programs are used for the evaluation on RS440. tomcatv, swim2000 and mgrid2000 from SPEC benchmarks are used as Fortran benchmark programs. AAC encoder (aac), Optical Flow (optflow), MPEG2 encoder (m2enc) and Face Detect (fd) are used as C benchmark programs. These programs are parallelized by the OSCAR compiler, then the generated parallelized Fortran and C programs with the OSCAR API are compiled by gcc with "`-O3 -fopenmp`". The OSCAR API's compatibility with OpenMP allows developers to use gcc as an OSCAR API translator and a backend compiler.

Fig.6 shows the evaluation results on RS440. X-axis shows the number of cores and Y-axis shows the speedup against sequential execution time. Each bar shows the speedup for each evaluated program. These graphs show each

evaluated program achieves scalable performance improvement along with the increasing number of cores. Especially, mgrid2000 in Fortran benchmark programs achieves 14.20 times speedup, and aac in C benchmark programs achieves 19.06 times speedup with both on 32 cores, respectively.

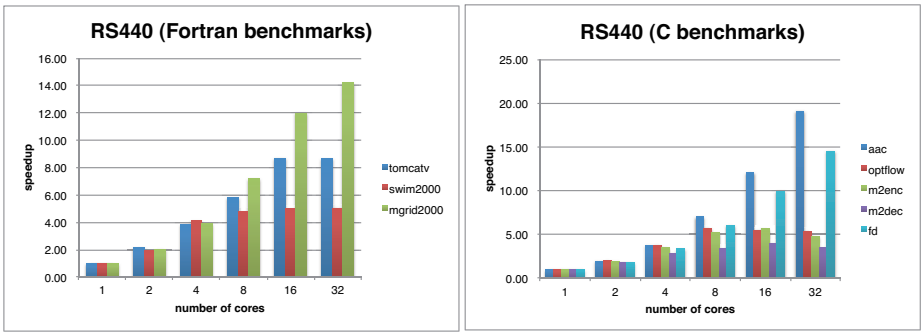

Fig. 6. Evaluation Result on Intel Xeon Server (RS440)

For the evaluation on the heterogeneous embedded multicore, RP-X developed by Hitachi, Renesas, Tokyo Institute of Technology, and Waseda University, is used[21]. RP-X has eight SH4A cores driven at 648MHz and four FE-GA reconfigurable accelerator cores driven at 324MHz. Different configurations of number of SH4A cores and FE-GA cores are evaluated. AAC encoder and Optical Flow are used for this evaluation. The OSCAR compiler parallelized these programs and generates parallelized C code with OSCAR API v2.0 including accelerator directives. These inserted OSCAR API directives are translated into runtime library calls for the evaluated RP-X system by the developed API translator.

Fig.7 shows the evaluation results on RP-X. Each bar shows speedup on each core configuration against the sequential execution on one SH4A core. For example, "8SH" bar shows the speedup using eight SH4A cores. Similarly, "4SH+2FE" bar shows the speedup using four SH4A cores and two FE-GA cores. This figure also shows the scalable performance improvement along with the increasing number of SH4A cores and FE-GA cores. Especially, Optical Flow program achieves 32.65 times speedup with eight SH4A cores and four FE-GA cores.

4 Flag Based Accelerator Control

The proposed CPU, DTC (Data Transfer Controller) and Accelerator execution model, Flag Based Accelerator Control, extended in the OSCAR API v2.1 is described in this section.

This execution model assumes there are CPU, DTC, ACC (Accelerator) and a local memory such as LDM or DSM in Fig.3 in a processor core of a multicore chip. CPU, DTC and ACC share the local memory. Program code for DTC and

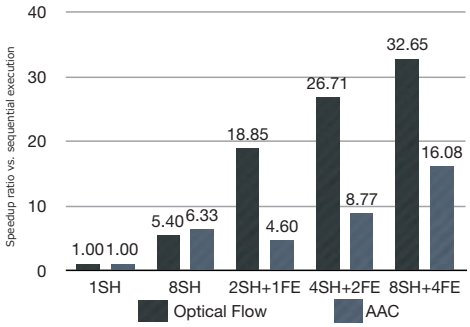

Fig. 7. Evaluation Result on Heterogeneous Embedded Multicore (RP-X)

ACC are located on the local memory by the compiler. When this system executes a parallelized program, CPU, DTC and ACC are executed independently.

Fig.8 shows a block diagram of such a processor core. This figure also shows an execution image of the CPU, DTC and ACC.

Flag variables shared among CPU, DTC and ACC are located on the local memory ((0) in the figure). When CPU starts the execution of DTC, CPU increments the flag variable "A" on the local memory (1). DTC checks the flag variable "A" (2). When DTC detects the update of flag "A", it starts data transfer between off-chip memory or a DSM on an other core (3). After data transfer, DTC increments the flag variable "B" (4). ACC also checks the flag variable "B" (5). When ACC detects the update of flag "B", it starts its execution (6). Thus, CPU, DTC and ACC can execute their own program simultaneously and the overhead for control and data transfer is hidden by such an overlap execution model.

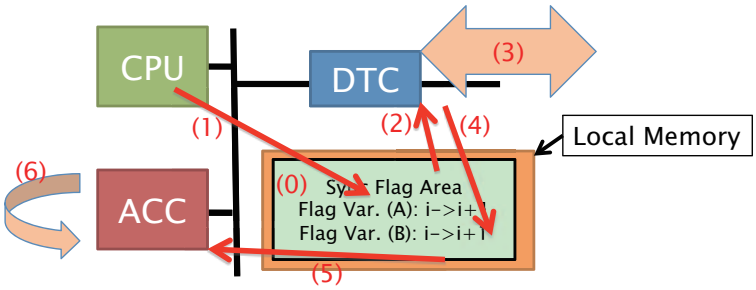

Fig. 8. Execution Image of Flag Based Accelerator Control

In order to give more concrete execution image, firstly, a sample code and its execution image on an ordinary non-overlap, or blocking, manner are shown in Fig.9-(a) and Fig9-(b), respectively. Then, sample codes of CPU, DTC, and

ACC for the proposed flag based accelerator control are shown in Fig.10-(a), and an execution image by those codes is shown in Fig.10-(b).

There is a doubly nested loop in the Fig.9-(a). The inner loop is to be executed on ACC. Before this inner loop, DTC loads the data from the off-chip memory to the local memory. Similarly, DTC stores the data from the local memory to the off-chip memory after the inner loop. This sample code is executed as shown in the Fig.9-(b). CPU starts the execution of DTC. This CPU waits for the end of DTC execution. Then, CPU starts the execution of ACC. CPU also waits for the end of ACC execution. Thus, the utilization of CPU, DTC and ACC becomes very low.

On the other hand, in the proposed execution model, each of program code for DTC and ACC are separately generated in addition to that of CPU, as shown in the Fig.10-(a). Flag-send and flag-check codes are also inserted into those program codes. The program codes for DTC and ACC are located on the local data memory. Fig.10-(b) shows an execution image of the sample code in Fig.10-(a), which is similar to double buffer execution image. In this example, the ACC execution with "Set Flag (G0)", DTC load for next iteration with "Set Flag (B1)", and CPU execution for preparing next iteration are simultaneously executed. The end of DTC load is notified by DTC's update of the flag variable "B". Each of CPU and ACC can start next execution by checking the flag variable "B" independently. Thus, such an overlap of execution among CPU, DTC and ACC can aggressively hide the data transfer and control overhead of DTC and ACC.

5 OSCAR API Extension for Flag Based Accelerator Control

The newly added directives in the OSCAR API v2.1 are introduced in this section.

In order to realize the proposed flag based accelerator control described in Section 4, the following three directives are added:

- `accelerator_task_entry_nonblocking`
- `acc_flag_send`
- `acc_flag_check`

`accelerator_task_entry_nonblocking` takes a list of function names. These functions are entry points of the accelerator cores. An entry function specified by this directive is located on a source file for the target accelerator core. Note that a source file for the target accelerator core includes other functions called by entry functions. Such an entry function is compiled by an accelerator compiler and combined with a start code for the accelerator. When a CPU calls an entry function, this CPU executes a start code of the accelerator, then this accelerator starts the body of the entry function. From this point, the accelerator core executes its own program independently from CPU cores.

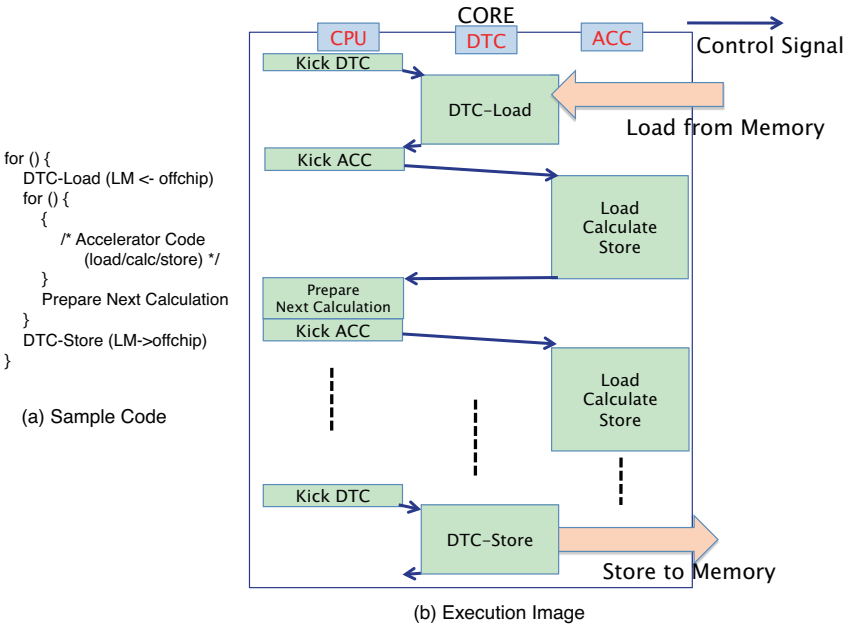

Fig. 9. Execution Execution Image of CPU, DTC and Accelerator with Ordinary Control Model

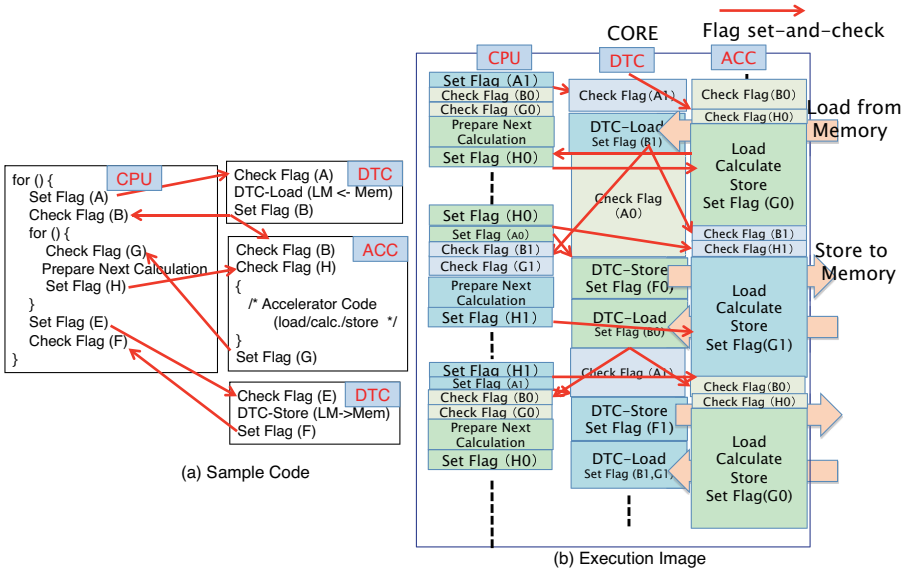

Fig. 10. Execution Image of CPU, DTC and Accelerator with Flag Based Control Model

`acc_flag_send` and `acc_flag_check` are placed in functions written in source files for target accelerators. An accelerator core sets a specified value into a flag variable at an `acc_flag_send` directive. Similarly, an accelerator core checks a flag variable and waits for that flag variable becoming a specified value at an `acc_flag_check` directive.

`dma_flag_send` and `dma_flag_check` have been also defined for controlling DTC from the OSCAR API v1.0. The usages of these directives are same as those of `acc_flag_send` and `acc_flag_check`, respectively. These directives enable fine grain and flexible communications between CPU cores, DTCs and accelerator cores as shown in Fig.10.

In addition to the accelerator directives, two hint directives for the OSCAR compiler are also added in the OSCAR API v2.1:

- `deadline`
- `upper_power_limit`

`deadline` specifies a deadline for the code fragment. This hint directive is used for task scheduling considering deadline in the OSCAR compiler. Similarly, `upper_power_limit` specifies a power limit for the code fragment. This hint directive is also used for task scheduling considering power capping in the compiler.

Fig.11 shows a sample accelerator program for `acc_flag_send`, `acc_flag_check` and `accelerator_task_entry_nonbloking` directives. In this sample program, "oscartask_CTRL2_loop1" function is listed in the `accelerator_task_entry` directive, which has been introduced from the OSCAR API v2.0, and "oscartask_loop2" function is listed in the `accelerator_task_entry_nonblocking` directive, respectively. "oscartask_CTRL2_loop1" is called from a controller CPU (controller(2)) and this controller CPU wait for the completion of this function on the accelerator core. On the other hand, a caller CPU resumes its execution just after calling "oscartask_loop2", then the CPU and the accelerator core can execute their own programs simultaneously. In order to synchronize between a CPU and an accelerator core, `acc_flag_check` and `acc_flag_send` are inserted before and after the for-loop in `oscartask_loop2`.

6 Conclusions

Hiding communication and data transfer overhead between CPU, DTC and accelerator cores enables more flexible, efficient and low-powered parallel program execution on future manycore processors for from embedded to super computers. For this purpose, a flag based accelerator control scheme has been proposed that allows us to use CPUs, DTCs and accelerators autonomously without the control by CPUs, in this paper. Then, the newly added directives in the OSCAR API v2.1 have been introduced in addition to the brief review of the OSCAR API v1.0 and v2.0. Development of manycore architecture and a parallelizing compiler, which enable this flexible execution model by cooperating between the architecture and the compiler, is our future work. The all

```
/* file: sample.VC2.c */
extern int flag1, flag2;
#pragma oscar distributedshared vpc(2) (flag1, flag2)
extern int y[10];

#pragma oscar accelerator_task_entry controller(2) \
    oscartask_CTRL2_loop1
#pragma oscar accelerator_task_entry_nonblocking \
    oscartask_loop2

void oscartask_CTRL2_loop1(int *x)
{
   int i;
   for (i=0; i < 10; i++)
      x[i]++;
}

void oscartask_loop2()
{
   int i;
#pragma oscar acc_flag_check(flag0, 1)
   while (flag0 != 1);
   for (i = 0; i < 10; i++)
      y[i]++;
#pragma oscar acc_flag_send(flag1, 2)
   flag1 = 2;
}
```

Fig. 11. Sample Code of OSCAR API v2.1

specifications for the OSCAR API v1.0, 2.0 and 2.1 will be available from `http://www.kasahara.cs.waseda.ac.jp/`.

References

1. S. Thakkar and T. Huff. Internet streaming simd extensions. *Computer*, 32(12):26–34, 1999.
2. N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and H. Kuo. Intel avx: New frontiers in performance improvement and energy efficiency, 2008.
3. Armv8 instruction set overview, 2012.
4. Nvidia's next generation cuda computer architecture: Kepler gk110, 2012.
5. Nvidia tegra 4 family gpu architecgure, 2013.
6. G. Chrysos. Knights corner, intel's first many integrated core (mic) architecture product. In *Proc. of Hot Chips 24 (HC24)*, August 2012.
7. http://www.nvidia.com/cuda/.
8. http://www.khronos.org/opencl/.
9. http://www.openacc-standard.org/.

10. H. Kasahara, M. Obata, and K. Ishizaka. 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. M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara. Hierarchical parallelism control for multigrain parallel processing. *Lecture Notes in Computer Science*, 2481:31–44, 2005.
12. K. Kimura, Y. Wada, H. Nakano, T. Kodaka, J. Shirako, K. Ishizaka, and H. Kasahara. 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. A. Yoshida, K. Koshizuka, and H. Kasahara. Data-localization for fortran macro-dataflow computation using partial static task assignment. In *Proc. of 10th ACM International Conference on Supercomputing*, May 1996.
14. K. Ishizaka, M. Obata, and H. Kasahara. Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In *Proc. of 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC2001)*, August 2001.
15. J. Shirako, N. Oshiyama, Y. Wada, H. Shikano, K. Kimura, and H. Kasahara. Compiler control power saving scheme for multi core processors. *Lecture Notes in Computer Science*, 4339:362–376, 2007.
16. Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara. A parallelizing compiler cooperative heterogeneous multicore processor architecture. *Lecture Notes in Computer Science*, 6760:215–233, 2011.
17. K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara. Oscar api for real-time low-power multicores and its performance on multicores and smp servers. *Lecture Notes in Computer Science*, 5898:188–202, 2010.
18. A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara. Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores. *Lecture Notes in Computer Science*, 6548:184–198, 2011.
19. H. Kasahara, M. Kogo, T. Tobita, T. Masuda, and T. Tanaka. 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.
20. http://www.openmp.org/.
21. Y. Yuyama, M. Ito, Y. Kiyoshige, Y. Nitta, S. Matsui, O. Nishii, A.Hasegawa, M. Ishikawa, T. Yamada, J. Miyakoshi, K. Terada, T. Nojiri, M. Satoh, H. Mizuno, K. Uchiyama, Y. Wada, K. Kimura, H. Kasahara, and H.Maejima. A 45nm 37.3gops/w heterogeneous multi-core soc. In *Proc. of IEEE International Solid State Circuits Conference (ISSCC2010)*, February 2010.