Design and Implementation of Scalable, Transparent Threads for Multi-Core Media Processor

Takeshi Kodaka, Shunsuke Sasaki, Takahiro Tokuyoshi, Ryuichiro Ohyama
Nobuhiro Nonogaki, Koji Kitayama, Tatsuya Mori, Yasuyuki Ueda, Hideho Arakida
Yuji Okuda, Toshiki Kizu, Yoshiro Tsuboi and Nobu Matsumoto
Toshiba Corporation, Semiconductor Company
Center for Semiconductor Research and Development
580-1, Horikawa-Cho, Saiwai-Ku, Kawasaki, Kanagawa, Japan

***Abstract*—In this paper, we propose a scalable and transparent parallelization scheme using threads for multi-core processor. The performance achieved by our scheme is scalable to the number of cores, and the application program is not affected by the actual number of cores.**

For the performance efficiency, we designed the threads so that they do not suspend and that they do not start their execution until the data necessary for them are available. We implemented our design using three modules: *the dependency controller*, which controls dependencies among threads, *the thread pool*, which manages the ready threads, and *the thread dispatcher*, which fetches threads from the pool and executes them on the core.

Our design and implementation provide efficient thread scheduling with low overhead. Moreover, by hiding the actual number of cores, it realizes transparency. We confirmed the transparency and scalability of our scheme by applying it to the H.264 decoder program. With this scheme, modification of application program is not necessary even if the number of cores changes due to disparate requirements. This feature makes the developing time shorter and contributes to the reduction of the developing cost.

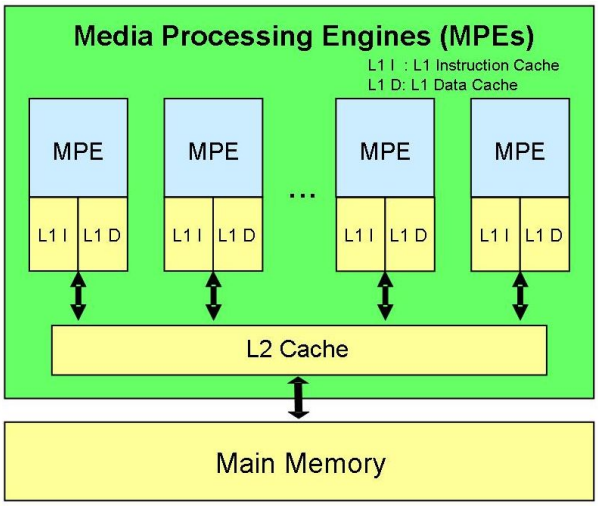

Fig. 1. Block Diagram of Target Architecture

I. Introduction

As portable devices prevail and network infrastructures become sophisticated, processing of multimedia data, such as image and audio, gets more and more divergent and complicated. In order to handle complicated and diverse processing of multimedia data, multi-core architectures have been widely adapted.

On multi-core architecture, parallelization of application program is necessary in order to process data efficiently. Parallelization of application program is usually realized by converting existing application program for single core into one for multi-core architecture.

It is sometime necessary to change the number of cores in order to satisfy different requirements. If application program has to be parallelized every time the number of core changes, developing time becomes longer and it makes developing cost more expensive.

To address these problems, some platforms are proposed [1], but they are for high performance computing and hence it is difficult to adapt them for portable devices due to high overhead.

In this paper, we propose a parallelization scheme, which enables us to achieve scalable performance without modifying application program even if the number of cores changes. In the next chapter, the hardware architecture of our target is described, followed by the details of our scheme. Then we show the application of our scheme to H.264 decoder and conclude this paper.

II. Target Hardware Architecture

Fig. 1 depicts the target architecture of the hardware [2], which consists of several cores, called MPE. Each MPE includes a 32-bit RISC core accompanied with a 64-bit SIMD 2-way VLIW coprocessor, L1 instruction and data cache. This VLIW coprocessor is the same as image recognition processor[3] except that it has extended instructions for audio processing. Outside MPEs exists L2 cache, which can be accessed from all the MPEs. Further, the L2 cache is connected to main memory.

This architecture is designed so that the performance improves in proportion to the number of MPEs. Therefore, by changing the number of MPEs, this architecture can handle a variety of applications.

Fig. 2 shows a chip micrograph of the processor implemented based on the architecture. The specifications of the chip are described in Table I.

978-3-9810801-5-5/DATE09 © 2009 EDAA

TABLE I
CHIP SPECIFICATION

Technology	65nm CMOS, triple-well, 8-layer-metal	
Die Size	5.06mm x 5.06mm	
Gate Counts	Logic	3.6M Gates
	SRAM	5.6Mb
Clock Frequency	MPE, L2 cache logic	333MHz
	L2 cache SRAM, Bus I/F	166MHz
L1 cache	8KB (Instruction) / 8KB (Data) 2-way set associative, FIFO 64B Line	
L2 cache	512KB (unified) 4-way set associative, LRU 256B Line	

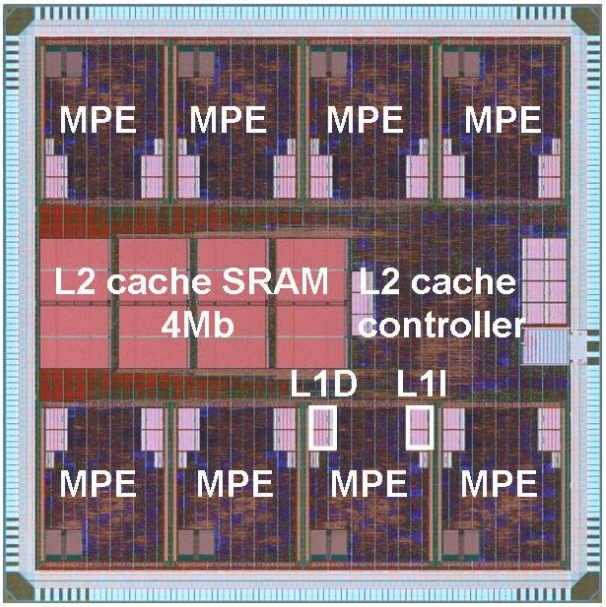

Fig. 2. Chip Micrograph

III. PARALLEL PROCESSING SCHEME FOR THE TARGET ARCHITECTURE

One of the most important factors in terms of performance is how to divide application program to smaller parts and how to assign them on each core.

Furthermore, for efficiency of application program development, the transparency, which hides the actual number of cores and makes the program independent from it, is another significant factor.

In this chapter, we propose the parallelization scheme which realizes both the transparency and the performance scalability.

A. The Proposed Parallel Processing Method

As explained below, there are two types of parallelization methods: pipelining and threading. We adopted the thread based parallelization as our fundamental principle.

1) Pipelining: One of the methods to develop parallel application program for multi-core processor is pipelining. As shown in Fig. 3, the application program is divided into some modules (func1() to func10()) so that data are passed among them. This method gives the best performance when the load is distributed equally among modules. However, if the number of cores changes, division and assignment have to be done again. Therefore, this method is not transparent and the application program is not reusable.

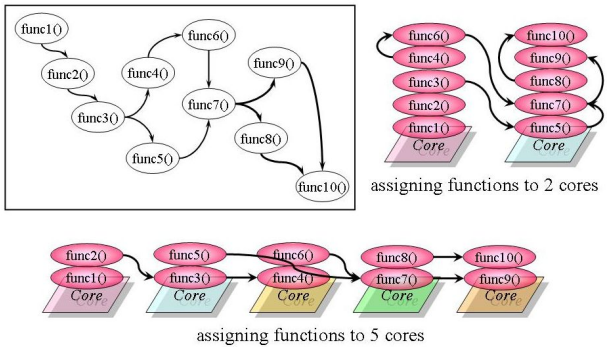

Fig. 3. Pipeline Based Parallel Processing

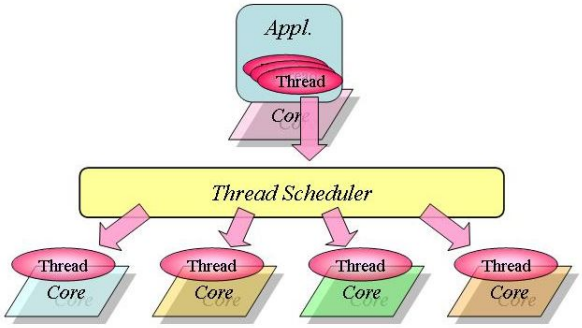

Fig. 4. Thread Based Parallel Processing

2) Threading: Another type of parallelization is threading. This method divides a application program into smaller parts, called threads, which are as small as a function in C language, and each thread is assigned to one core by thread scheduler on the fly as shown in Fig. 4. The data are passed through threads. As the number of threads increases, the performance improves. However, in order to assure the correct operation, the scheduling that assigns each thread to a core in appropriate order is necessary. In terms of application program reusability, the scheduler hides the number of cores and enables to develop application program independent from the actual number of cores.

B. The Proposed Thread Scheduler

In our threading concurrency model, the thread scheduler is the most important factor in order to realize the transparency and the performance scalability. For the performance scalability, we have developed the scheduling mechanism, aiming at reduction of scheduling overhead and improvement of efficiency. For the transparency, we have developed the interface that hides the actual number of cores from application

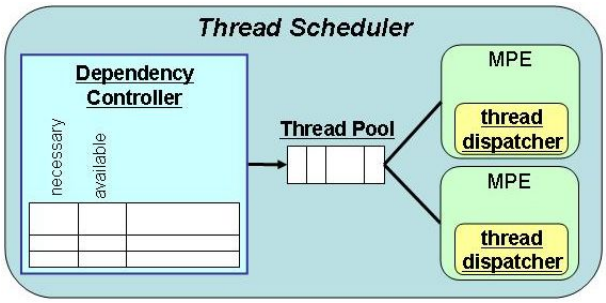

Fig. 5. Proposed Thread Scheduler

programs.

The thread scheduler has the following properties.

- A thread can not suspend during its execution
- A thread does not start until all the data necessary for the thread become ready.

Based on this design, we implemented the scheduler which consists of three modules as shown below.

- *The dependency controller*
- *The thread pool*
- *The thread dispatcher*

Our implementation is depicted in Fig. 5.

1) The synchronization mechanism of threads: An application program consists of threads which communicates with each other by sending and receiving data. The shared data must be read and written at proper timing so that the whole system runs correctly. When data are passed, a consumer thread must wait until a proper producer thread sends the data. If the consumer thread suspends until the producer thread generates data, the context, which includes registers, stack, and so on, must be saved. Likewise, the context must be restored when the thread resumes. These save and restore operations are costly, particularly for the architecture with many registers, such as VLIW coprocessor. Hence we designed so that threads do not suspend. Additionally, in our design, a thread starts its execution only when all the data necessary for the thread are available. This design eliminates the number of context switches, resulting in reduction of scheduling overhead.

2) Selecting ready threads: In our design, a thread becomes ready to run only when all the necessary data are available. In order to check if a thread is ready, the number of data necessary for the thread and the counter holding the number of data already available are used.

When an application program does calculation with threads, it registers the threads to the thread scheduler with the number of data necessary for each thread. Then, when the data necessary for the thread become ready, the counter for the thread is updated. If the number of data necessary for thread and the number of data which are already available are equal, the thread is deemed to be ready to run.

As shown above, by considering the number of data as information for a thread, the amount of information for checking the status of a thread and the cost of selecting a thread are reduced, resulting in decrease of scheduling overhead.

3) The thread execution mechanism: The overhead of scheduling heavily depends on operations to check which threads are available and to select the appropriate thread. We implemented the thread scheduler with three modules: *the dependency controller*, *the thread pool* and *the thread dispatcher*. This design makes overhead of scheduling smaller as described below.

The dependency controller keeps a record of the number of data necessary for the thread and the counter, and checks if a thread is ready to run. Moreover, *the dependency controller* provides interface between an application program and the thread scheduler. All the threads are registered to this module and managed by *the dependency controller.* A thread is transferred to *the thread pool* when it becomes ready to run.

The thread pool is a kind of buffer which stores threads transferred by *the dependency controller*. Threads are stored in this pool until they are fetched by *the thread dispatcher.*

The thread dispatcher fetches a ready thread and execute it. This module runs on every core and works independently. *The thread dispatcher* always monitors *the thread pool.* If it finds a thread in *the thread pool*, it fetches the thread and execute it. Fig. 6 illustrates an example of thread execution. When an application program does calculation with threads, it registers threads to *the dependency controller* along with the number of data necessary for the thread. (Fig. 6 (a)). When the data are created by the producer, the producer send a message to *the dependency controller* to tell that the data are ready. Receiving the message, *the dependency controller* increments the counter which holds actual number of ready data. If the number of data necessary for the thread and the counter which holds actual number of ready data are equal, the thread is deemed to be ready to run. The ready thread is transferred to *the thread pool*. (Fig. 6 (b)).

Meanwhile, *the thread dispatchers* run on cores for thread execution. *The thread dispatcher* monitors *the thread pool* and if a thread is transferred to there as described above, it fetches the thread from *the thread pool.* Finally, the thread is executed on the core that *the thread dispatcher* runs on. (Fig. 6 (c)).

This design of thread execution separates selecting ready threads from the execution of threads because *the dependency controller* and *the thread dispatcher* run independently. Therefore, each operation runs concurrently and scheduling is performed more effectively.

4) The transparency: In order to hide the number of cores, the interface to application program should be minimal. In our design, we decided to have one interface, *the dependency controller*. Application program sends message to the interface, such as requests of registering a thread and notifications of data generation. With this design, only *the dependency controller* is visible from application program, and it hides the number of cores.

IV. Application of the Proposed Scheme to H.264 Decoder

In this chapter, we show the result of application to H.264 decoder to prove the effectivity of our scheme.

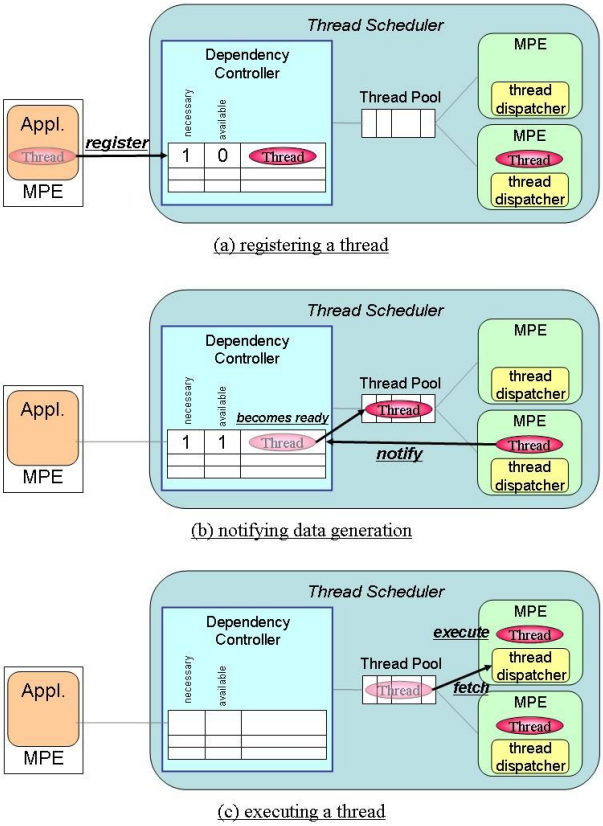

Fig. 6. Example of thread execution

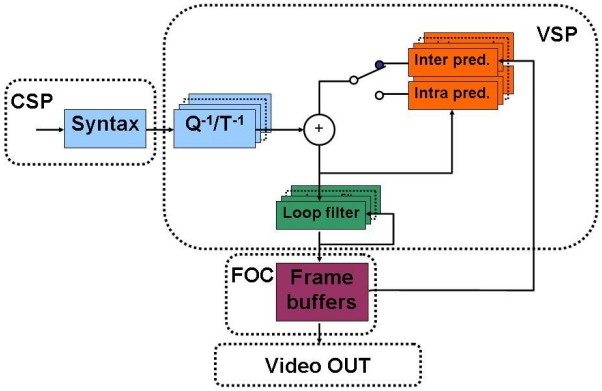

Fig. 7. System Block Diagram of H.264 Decoder

A. Exploit of Parallelism in H.264 Decoder

Fig. 7 illustrates typical H.264 decoder. We divided it into three components, CSP, the Coded Stream Processing component, VSP, the Video Signal Processing component, and FOC, Frame-buffer Output Controlling component. We applied our parallelization scheme to VSP.

First, we examined the parallelism of VSP. VSP has spatial parallelism and temporal parallelism. Temporal parallelism means that different operations can be done simultaneously. In the case of VSP, decoding of luma signal and chroma signal does not require data exchange. Hence these parts

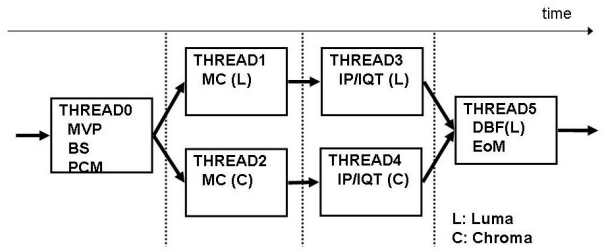

Fig. 8. Temporal Dependency of H.264 Decoder

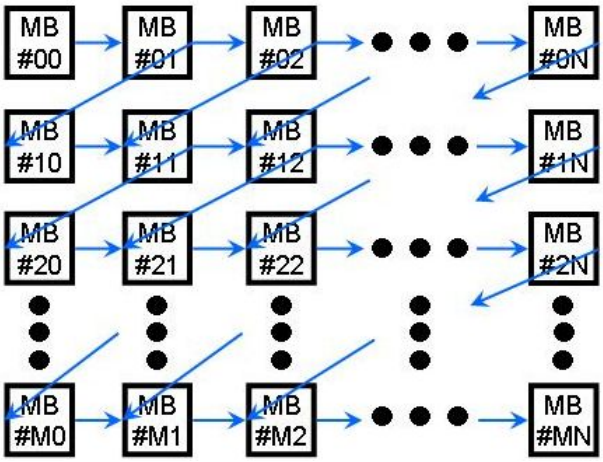

Fig. 9. Spatial Dependency of THREAD5

are considered to have temporal parallelism. Further, these parts can be divided into threads so that each thread has approximately the same load. As the result, the threads have dependencies, shown in Fig. 8. In this figure, each node indicates a thread after temporal parallelism is extracted and arrows correspond to data transmission among nodes.

Second, we analyzed the threads which were already divided in terms of temporal parallelization, in order to improve performance with spatial parallelization. Spatial parallelization means that a module which operates on different location can be executed simultaneously. The VSP, which operates in macro block level, can be divided so that they utilize spatial parallelism. Fig. 9 shows the extraction of spatial parallelism of THREAD5 in Fig. 8. Each node indicates operations in macro block level and arrows represent data transmission. Spatial parallelism is extracted from Fig 9. For example, MB#01 creates data necessary for MB#02 and MB#10, and each of them can be executed concurrently.

As shown above, temporal and spatial parallelism are extracted from H.264 decoder. Fig 10 shows the final result of concurrency.

B. Evaluation

Fig. 11 shows the result of performance of H.264 - 720p with the different number of cores. The x-axis indicates the number of MPEs for VSP processing, and the y-axis indicates the decoded frame rate. Because of the transparency, the same

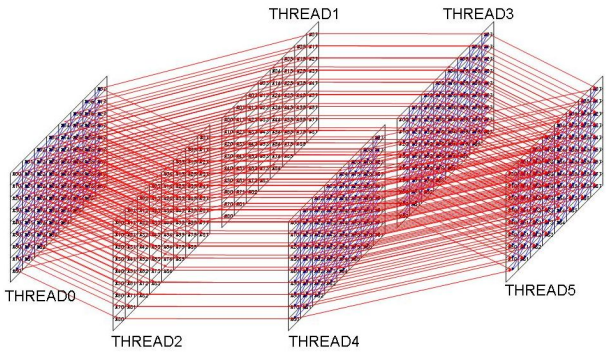

Fig. 10. Temporal and Spatial Dependency of H.264 Decoder

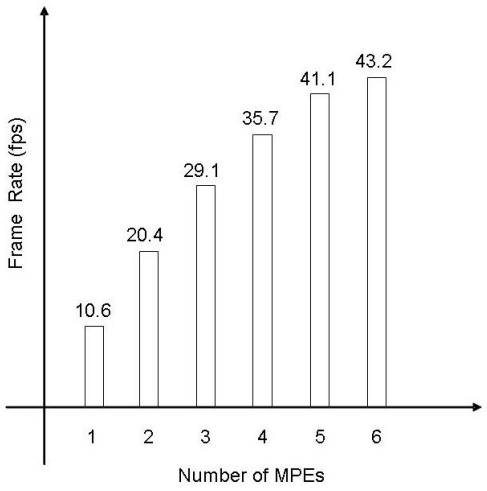

Fig. 11. Frame Rate of H.264 Decoder

application program is used on the evaluation, regardless of the number of cores.

The H.264 decoder, which is used for this evaluation, utilizes two MPEs for CSP and FOC. The VSP operation, which uses threads, runs on the different number of MPEs. The number of MPEs for VSP operation is specified when the decoder starts. We change the number of them from 1 to 6 for evaluation. The evaluation in this section is based on the in-house cycle accurate simulator, which is developed for the processor described in II.

As shown in Fig. 11, when the number of MPE changes from 1 to 4, the frame rates scale up to 10.6, 20.4, 29.1, and 35.7 respectively. This result proves that the performance improves in proportion to the number of MPEs and that our design and implementation are effective for multi-core architecture.

Meantime, the frame rate is 41.1 with 5 MPEs and 43.2 with 6 MPEs. The improvement is saturated at 5 MPEs. This saturation is caused by L2 cache which is shared by all the MPEs. As the number of MPEs increases, so does the number of accesses to L2 cache. Therefore, the hit rate of L2 cache decreases and the latency to access memory or cache becomes longer.

V. Conclusions

In this paper, we proposed the parallelization scheme which realizes scalable performance and transparency. We scrutinized the effectiveness and overhead of the scheduler, and the interface between application programs and the thread scheduler.

In our design threads have two properties: 1) a thread can not be suspended once it starts and 2) a thread becomes ready to run only when all the data necessary for the thread are available. The readiness of a thread depends only on the number of data necessary for the thread. Our scheduler consists of three modules, *the dependency controller*, *the thread pool* and *the thread dispatcher.*

In our design, as the number of cores increases, the performance improves. Also the application program developed by our scheme does not depend on the number of cores, which is hidden by our design. By applying our scheme to H.264 decoder, we confirmed that the performance improves in proportion to the number of cores without modifying the application program.

Yet, performance declines as the number of cores are large. To handle this issue, we are planing to develop a new scheduler which will improve cache efficiency.

References

[1] S. Maeda and et al., "A real-time software platform for the cell processor," *IEEE micro*, vol. 25, no. 5, pp. 20–29, Sept.-Oct. 2005.

[2] S. Nomura and et al., "A.9.7mw aac-decoding, 620mw h.264 720p 60fps decoding, 8-core media processor with embedded forward-body-biasing and power gating circuit in 65nm cmos technology," *ISSCC Dig. Tech. Papers*, pp. 262–263, Feb. 2008.

[3] J. Tanabe and et al., "Visconti: Multi-vliw image recognition processor based on configurable processor," *IEEE Custom Integrated Circuits Conference*, pp. 185–188, Sep. 2003.