Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2011, Article ID 373509, 20 pages
doi:10.1155/2011/373509

Research Article

A Programmable Video Platform and Its Application Mapping Framework Using the Target Application's SystemC Models

Daewoong Kim, Kilhyung Cha, Do-Sun Hong, Soonwoo Choi, and Soo-Ik Chae

School of Electrical Engineering and Computer Science, Seoul National University, Seoul 110-794, Republic of Korea

Correspondence should be addressed to Daewoong Kim, dwkim316@sdgroup.snu.ac.kr

Received 10 August 2010; Revised 15 December 2010; Accepted 17 January 2011

Academic Editor: Neil Bergmann

HD video applications can be represented with multiple tasks consisting of tightly coupled multiple threads. Each task requires massive computation, and their communication can be categorized as asynchronous distributed small data and large streaming data transfers. In this paper, we propose a high performance programmable video platform that consists of four processing element (PE) clusters. Each PE cluster runs a task in the video application with RISC cores, a hardware operating system kernel (HOSK), and task-specific accelerators. PE clusters are connected with two separate point-to-point networks: one for asynchronous distributed controls and the other for heavy streaming data transfers among the tasks. Furthermore, we developed an application mapping framework, with which parallel executable codes can be obtained from a manually developed SystemC model of the target application without knowing the detailed architecture of the video platform. To show the effectivity of the platform and its mapping framework, we also present mapping results for an H.264/AVC 720p decoder/encoder and a VC-1 720p decoder with 30 fps, assuming that the platform operates at 200 MHz.

1. Introduction

Recently, many different video codec formats have been introduced. For example, three video codec formats, H.264/AVC, VC-1, and MPEG-2, must be supported by all Blu-ray players. To access the huge amount of video contents available on the network, other formats such as MPEG-4 and H.263 must also be supported. Moreover, many consumer electronics, like digital video cameras, mobile handsets, and home entertainment equipments, require huge computation power to process HD contents in real time. For example, an H.264/AVC 720p encoder at 30 frames per second (fps) must process 108,000 macroblocks (MBs) per second, which corresponds to 3,600 giga-instructions according to profiling with RISC instructions [1]. With MB pipelining, video codec applications can be represented with a set of loosely coupled tasks, each of which is described with a set of tightly coupled threads while heavy streaming data and asynchronous distributed controls should be transferred among the tasks [2, 3].

For high-performance video systems, developing an application-specific integrated circuits (ASIC) can only be a short-term solution because new video formats, for example, high efficiency video coding (HEVC) [4], will be introduced in the market. A platform with multiple microprocessors is more preferable, which can easily support a new video codec format by simply loading its code into the platform [2]. However, it is not easy to provide enough computation power for high-performance video applications only with general-purpose programmable processors. Moreover, it is not a trivial task for programmers to obtain a good parallel executable code for the multicore platforms. Therefore, application mapping should maximally utilize all the available resource in the platform.

The main objective of this paper is to propose a high-performance programmable platform for video applications which can be represented with multiple tasks, each of which again consists of tightly coupled multiple threads. The proposed video platform consists of four PE clusters and their communication networks that can support control

and data transfers among the PE cluster as shown in Figure 1. Each PE cluster, which is allocated for a task that includes multiple tightly coupled threads, consists of one or more RISC cores, a HOSK, an accelerator for video task-specific operations, and network interfaces. Furthermore, a framework for effective mapping applications that utilize SystemC as shown in Figure 1 will also be described. The framework can hide the platform complexity and improve design quality in a shorter time.

First, we propose a high-performance programmable platform for HD video codec applications. Three programmable platforms for multimedia or video codec applications that have been published [2, 5, 6] will be compared with the proposed platform. The StepNP platform [5] is composed of multiple configurable RISC processors, each of which has multiple register sets to reduce the overhead of context switching between threads, and reconfigurable hardware accelerators that are loosely coupled with the RISC processors. In this platform, it takes about 50 instructions for each transfer between processing elements which are connected through network-on-chip (NoC) that includes a central buffer. The platform targets for multimedia applications including a VGA MPEG-4 encoder and also provides an application mapping framework that exploits two high-level parallel programming models such as distributed system object component (DSOC) message passing and symmetrical multiprocessing (SMP) using shared memory. For application mapping, an application is manually split into several parallel executable codes by using the communication primitives of the programming models. However, the footprint of the runtime for each communication primitive is roughly 1,000 instructions and its latency is too large for HD video applications. For example, the average cycle budget per macroblock should be less than 1,500 cycles in a 1080p 30 fps video encoder, assuming that the system operating frequency is 300 MHz. For an HD video application, therefore, it is necessary to support asynchronous distributed small data transfers among tasks with low latency and provide communication primitives with low latency. In other words, the overhead for context switching, thread scheduling, and synchronization between tightly coupled threads in a task as well as communication latency for efficient transfers among tasks should be reduced as much as possible. To minimize these overhead and latency, the proposed platform consists of the PE clusters, each of which includes a hardware operating system kernel (HOSK) that accelerates context switching of threads for a task allocated to the PE cluster, and schedules and synchronizes the threads. The HOSK also manages communication transfer with another PE cluster through its network interfaces.

The tightly coupled-thread (TCT) platform [6], which targets for multimedia applications including a 400×300 image JPEG encoder, consists of six customized RISC processors. In the platform, the RISC processors are connected with a FIFO-based point-to-point, full crossbar network with a simple hand-shake protocol for easy communication and synchronization among tightly coupled tasks. For the TCT platform parallel executable code is generated for an application by composing compiled code for each task, which is separated with task-partitioning delimiters in the application source program that is allocated to an RISC processor. The platform does not support fast context switching for multithreading and does not include hardware accelerators to enhance its computing power simply because its target applications does not require high-performance. Its communication network, which is suitable for distributed small data transfer, cannot efficiently transfer large streaming data between the tasks, which are required for the HD video codec applications. Therefore, the proposed platform has two separate point-to-point networks: one for transferring control data such as syntax elements and the other for transferring massive streaming data.

The programmable image processing element (PIPE) platform [2] consists of six PIPE processors, a stream processor, and six loosely coupled hardware accelerators. Each PIPE processor is composed of three tightly coupled processors which concurrently perform loading input data, processing them, and storing the results. The PIPE processor can be configured for a specific task; for example, calculating the summation of the absolute difference (SAD) for motion estimation. Similar to the proposed platform, its target application also focuses on HD video codec systems. To obtain high-performance for HD applications, the platform includes several hardware accelerators for intra prediction mode decision, coarse motion estimation and motion compensation (ME/MC), and motion vector (MV) calculation and all these functions have complex conditional branch operations. The platform's flexibility is substantially degraded because its PIPE processor is specialized for two-dimensional operations. In contrast, in the proposed platform, each PE cluster includes an accelerator dedicated for its task, and the accelerator is tightly coupled with the RISC processors in the PE cluster because it directly gets commands from the RISC processors that handle conditional branch operations.

Table 1 compares the proposed platform with the three platforms described in the above [2, 5, 6]. Note that the number of PEs and the number of processors in PE are flexible in the proposed platform.

Second, we provide an application mapping framework with which parallel executable codes can be obtained from a manually developed SystemC model of the target application without knowing the detailed architecture of the video platform. Several frameworks have been presented that exploit SystemC models to map applications into a programmable platform [7–10]. The SPACE [7] and other frameworks [8, 9] support SystemC-based software modeling for multicore platforms. In these approaches, which employ predefined (or available) SystemC primitives as application program interfaces (APIs) for communication and synchronization to encapsulate these RTOS functions, the compiled code of each SystemC module for a processor is linked with the RTOS kernel codes, each of which corresponds to each SystemC primitive for communication and synchronization. In the framework presented in [10], a sequential C/C++ code is first partitioned into multiple threads in a SystemC model, and then transformed into parallel executable code by overloading the SystemC class library with OS kernel functions

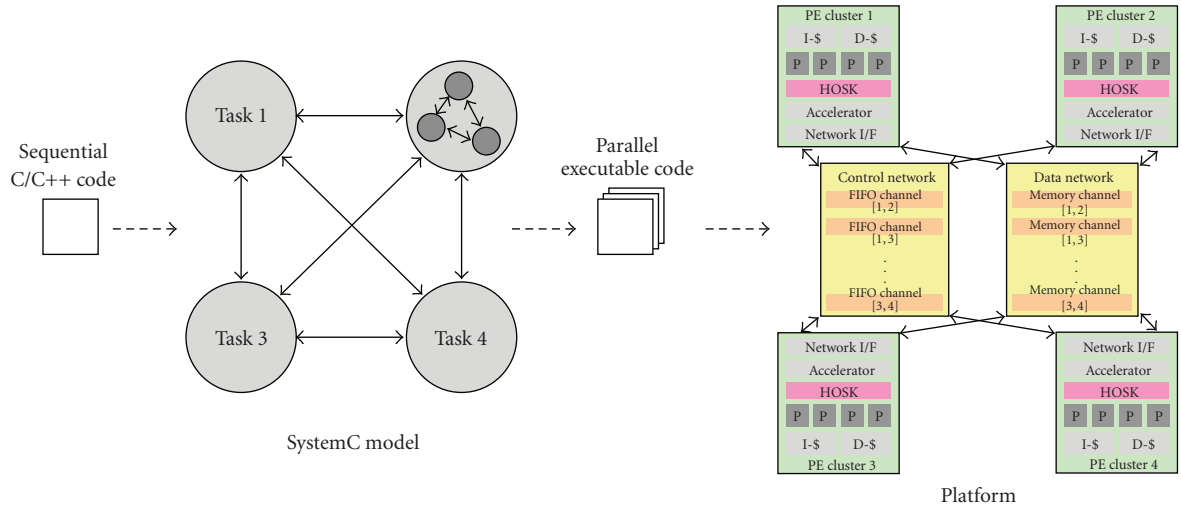


Figure 1: High-performance programmable video platform utilizing SystemC programming model for its application mapping.

Table 1: Comparison with other platforms.

	StepNP [5]	TCT [6]	PIPE [2]	This Work
Target Application	Multimedia (VGA MPEG-4)	Multimedia (400 × 300 JPEG)	High-Definition Video (1920 × 1080)	High-Definition Video (1280 × 720)
Processing Element (PE)	Configurable RISC Core, H/W Accelerator	Customized RISC Core	Tightly coupled Three Cores, Stream Processor, H/W Accelerator	Four RISC Cores + Tightly coupled H/W Accelerator + *HOSK*
Number of PE	14 PEs (5 Cores, 9 H/W Accelerators)	6 PEs (6 Cores)	13 PEs (18 Cores, 1 Stream Processor, 6 H/W Accelerators)	4 PEs (16 Cores, 4 H/W Accelerators)
Accelerator Connection Type	Loosely coupled	—	Loosely coupled	*Tightly coupled*
Fast Context Switching	Support	Not Support	Not Support	Support
Communication Subsystem	Network-On-Chip	One Point-to-Point Network	Shift-Register-Based Bus Network	*Two Separate Point-to-Point Networks*
Programming Model	DSOC, SMP	User-defined C Library	User-defined C Library	SystemC Constructs
Executable Code Generation	Manually	Compile with inserting Directives	Manually	Manually

and C++ structures like pthread library functions. All of the frameworks described above develop a multithreaded model independent of the architecture of the target platform. It implies that an architectural gap exists; in other words, a better parallel code can be obtained by using the architectural information of the target platform.

We need to reduce this architectural gap between a SystemC model and its mapping to the target platform as a parallel code. Therefore, we also provide a framework for mapping a SystemC model to the proposed platform. In SystemC modeling, the proposed framework facilitates application mapping by exploiting its basic constructs such as SC_MODULE, SC_THREAD, PUT, and GET to expose the parallelism of the target application and hide its communication details. For example, a task allocated to a PE cluster is represented with an SC_MODULE; each thread in a task with an SC_THREAD; a transfer between the threads in the same PE cluster with a pair of PUT and GET primitives; a control transfer between two PE clusters with a pair of PUT and GET primitives; a data transfer between two PE clusters with a pair of WRITE and READ primitives. From a SystemC model, we can obtain a parallel executable code for the platform by replacing each SystemC construct with its corresponding hardware operating system kernel (HOSK) API. Therefore, the platform users should be familiar with SystemC modeling to efficiently use the proposed SystemC framework for application mapping.

The rest of this paper is organized as follows. In Section 2 we describe the generic architecture of the platform that consists of multiple PE clusters and its communication subsystem together with the subcomponents of the PE clusters, which are RISC cores, a HOSK, and three different hardware accelerators for parsing, ME/MC, and filtering. In Section 3 we briefly describe how we developed the platform

targeted for 720p video codec. In Section 4 we explain an application mapping framework that utilizes SystemC modeling; how we get parallel executable code for an application from its SystemC model to the target platform. In Section 5 we present mapping results such as an H.264/AVC 720p decoder/encoder, and a VC-1 720p decoder, which is followed by the conclusions in Section 6.

2. High-Performance Programmable Video Platform

In this section we will describe architecture of the proposed platform that consists of multiple PE clusters and communication subsystem. The architecture of the proposed video platform consists of several PE clusters, which are connected with control and data communication networks as shown in Figure 1. Notice that the number of PE cluster is tuned into 4 for 720p video codec as will be explained in Section 3. Each PE cluster is composed of one or more RISC processors, a HOSK, one or more task-specific accelerators, and network interface, respectively. The PE cluster and its components are also explained in details.

2.1. Processing Element Cluster. Each PE cluster should have enough computation capability to handle a task allocated to it. Therefore, each PE cluster should include one or more accelerators that execute computation- or control-intensive operations specialized to its task because the task cannot be handled only with RISC processors for HD video applications. Their details will be presented in Section 2.3. To further exploit thread-level parallelism, each task is again partitioned into multiple threads relatively tightly coupled together, which justifies employing multiple RISC cores in the PE cluster. So the task is partitioned into two parts: one allocated to the accelerators and the other to the RISC cores. Because most of the computations for the task are offloaded to the accelerators, the RISC cores are responsible for the rest, which includes routines for initializing and controlling the accelerators, and communicating with other PE clusters. The RISC processors are attractive because each of them generally supports a readily available integrated design environment (IDE) including its compiler and debugger [5, 6].

Furthermore, it is very crucial to reduce overheads for both context switching and synchronization for HD video applications because the cycle budget is tight. For example, the average synchronization period is less than 2,500 clock cycles per macroblock for a 720p video codec application with 30 frames per second (fps) at 250 MHz. Moreover, the overhead of a software-only RTOS context switching is typically over 1,000 cycles [5], which is too large to handle context switching required for handling a task with multiple threads for HD video applications. Therefore, each PE cluster includes a HOSK to accelerate scheduling, context switching, and synchronization of its threads. In the proposed platform, each PE cluster consists of one or more RISC cores that shares instruction and data caches, and also includes a HOSK, a task-specific accelerator, and network interfaces to other PE cluster, as shown in Figure 2.

As shown in Figure 2, the HOSK in a PE cluster is tightly connected with its RISC cores through two local buses: a context bus and a command bus. The HOSK consists of a main controller, a context manager, and a thread and semaphore (TAS) manager [11]. A RISC core sends control commands through the command bus to the HOSK while the HOSK switches the context of a RISC core through the context bus. The HOSK commands include OS APIs such as creating and destroying threads, posting, and waiting semaphores.

The context manager stores the contexts of all threads in the context memory mapped into an off-chip memory due to its large size, and has an on-chip context buffer holding a context of a scheduled thread to be switched in or a context of a suspended thread to be switched out. Note that the size of the context memory should be greater than or equal to the context size of a RISC processor multiplied with the maximum allowable number of threads in a task.

After the TAS manager schedules a thread to be switched in, the context manager prefetches its context from the external context memory in background, and stores it to the on-chip context buffer. The TAS manager performs context switching when a thread running in a RISC core waits for input data or an external event, or when there is a ready thread with a priority higher than one of the thread running in the RISC cores. Context switching is started, only after context pre-fetching is completed, to hide the latency of accessing the external context memory. Therefore, the latency of context switching depends on the width of the context bus, for example, 16 cycles if the context bus is a duplex 32-bit bus. The TAS manager logs the descriptors for both threads and semaphores in the TAS control memory by implementing a queue for ready threads with a bit vector and a queue for waiting threads with a linked list [11].

The RISC processors in a PE cluster share both an instruction cache and a data cache. We employed cache sharing simply because a larger cache is more area-efficient than a smaller one and cache sharing eliminates the need for cache coherency support. However, the number of RISC cores in a PE cluster should be small enough to have a reasonably small degradation in performance due to cache sharing. To further reduce the degradation due to cache sharing, the instruction and data caches have multiple banks to resolve the conflicts of accessing the cache memory while the instruction cache supports pre-fetching and each data access reduces tag matching conflicts by checking before tag matching a tag history buffer that includes matched tags.

2.2. Communication Subsystem. The PE clusters are connected through the communication subsystem in the video platform. General transfers in the communication subsystem can be divided into two types: control transfers and data transfers [1–3, 12, 13]. After parsing syntax elements in the video codec applications, the parsing task sends them to other tasks and each of these transfers is a control transfer small in data size. In contrast, each data transfer between two tasks is large, which constitutes streaming data. For example, a task transfers to another task pixel data that is 384 bytes or more per macroblock. The average cycle

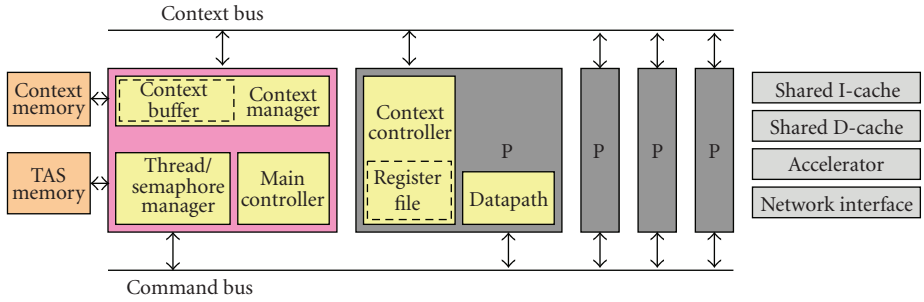


Figure 2: HOSK block diagram tightly coupled with RISC cores.

budget per macroblock gets tighter for HD video codec applications such as H.264/AVC 1080p, assuming that the operating frequency of the platform is constant. Therefore, it is very important to provide a programmable platform with efficient and flexible communication subsystem especially for HD video codec applications [2, 5, 6].

To implement efficient and flexible interconnects, we employed two separate point-to-point networks: one for control or small data transfers and the other for large data transfers. Although a point-to-point network requires more wires, which limits scalability [14], this is not an issue if the number of PE clusters is limited to a small number, for example, four to six in the proposed video platform, which will be discussed in Section 3.

Architecture of the control network is shown in Figure 3(a), where a group of FIFOs is allocated for a point-to-point control link between each pair of PE clusters. The control network also supports fully programmable connectivity between the PE clusters. A control link is established with a pair of identifiers (IDs): a PE cluster ID (PID) and a FIFO ID (FID). For example, a source PE cluster decodes a target PID to select a group of FIFOs for a target PE cluster and a FID to select a specific FIFO. An interface for the control network can establish a control link by simply decoding a PID and a FID, which also monitors the status of the link by checking whether it is full or empty.

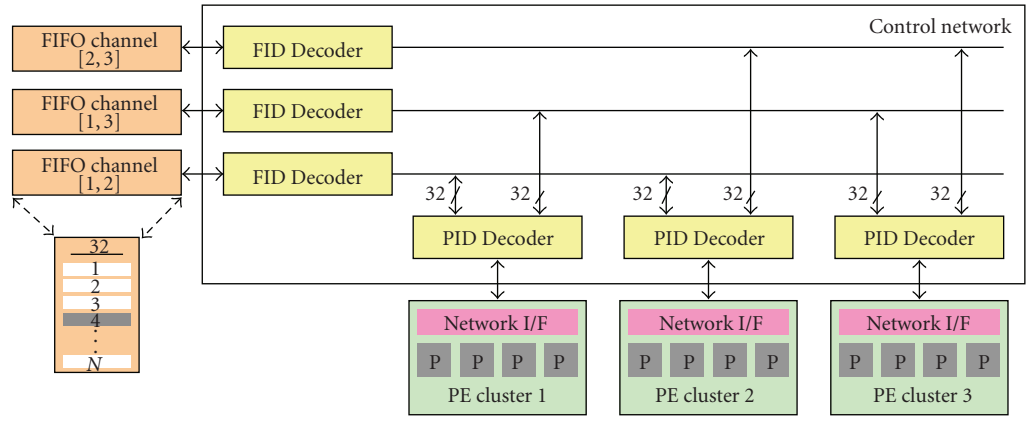
In an example of the control network shown in Figure 3(b), a control data is transferred from PE cluster 1 to PE cluster 2 through the fourth FIFO channel as follows. A RISC core in PE cluster 1 first finds an available FIFO that is not full and makes a link by issuing a PUT instruction with a pair of IDs, (2, 4) through its interface to the control network. Then, the control data is transferred into the FIFO. To get this control data, a RISC core in PE cluster 2 monitors through its network interface data availability of the same link, which is identified with (1, 4) in the target PE cluster and then issues a GET instruction to the link if not empty. In summary, a PUT primitive with (a target PID, a FID) and a GET primitive with (a source PID, a FID) are used for a control transfer, as shown in Figure 3(b). Note that a source PID, a target PID, and a FID should be predefined for each control link between two tasks in the proposed platform.

As shown in Figure 4(a), a data link is implemented with a shared memory in the data network which similarly supports a point-to-point link between each pair of PE clusters. Each shared memory for a data link is divided into two regions for double buffering. After completing a data transfer into a region, a source task forwards an explicit sync to a target task through its corresponding control link while performing its data transfer into the other region, if next data is ready. Receiving the sync, the target task gets the data from its corresponding region of the shared memory. Using shared memory as a communication link can reduce its latency. A shared memory can be identified with its base address, which is associated with a data link that is identified with a pair of PIDs, a source PID and a target PID. Therefore, a WRITE primitive with a base address and a READ primitive with a base address are used just for data transfer as shown in Figure 4(b). Note that a control transfer is required for synchronization of the data transfer with double buffering.

2.3. Task-Specific Accelerators. Assuming that the video platform is implemented with an ASIC approach, its operating frequency is somewhat limited well under 1 GHz. To run video codec applications, therefore, the video platform provides multiple PE clusters to concurrently execute multiple tasks. In addition to that, it is also important for each PE cluster to include one or more accelerators to effectively satisfy its computation requirement with its operating frequency, especially for HD video codec applications.

In the video codec application, the task for syntax parsing is control-intensive while other tasks require relatively large computation with large input/output data [2, 13]. The tasks for transform/quantization, intraframe prediction, and deblocking filtering commonly require weighted summation operations [3, 15, 16]. The task for motion estimation in the video encoder requires huge computation and large memory bandwidth, whose complexity can vary widely according to its algorithm [1, 2, 17]. Operations for fraction-pel motion estimation are very similar to those for the interframe prediction [3, 15, 16]. Based on these observations, we implemented three kinds of accelerators specialized for parsing, ME/MC, and filtering for the proposed video platform.

The PE cluster for parsing includes a parsing accelerator which performs iterative operations in the MB layer or below, which occupies dominant part of the computation for parsing. The RISC processors in the PE cluster perform the operations above the MB layer, which includes the network abstraction layer (NAL). To provide flexibility and performance of parsing, consequently, we implemented



(a) Control network block diagram

```
Void PE1_thread1_producer () {
    unsigned int value;
    unsigned int PID, FID;
                .
                .
     while (1) {
                .
        value = x;
        out->put (2, 4, value); //(target PID, FID, value)
    }
}
```

```
Void PE2_thread1_consumer () {
    unsigned int value;
    unsigned int PID, FID;
                .
                .
     while (1) {
        value = in->get (1, 4); //(source PID, FID)
                .
                .
    }
}
```

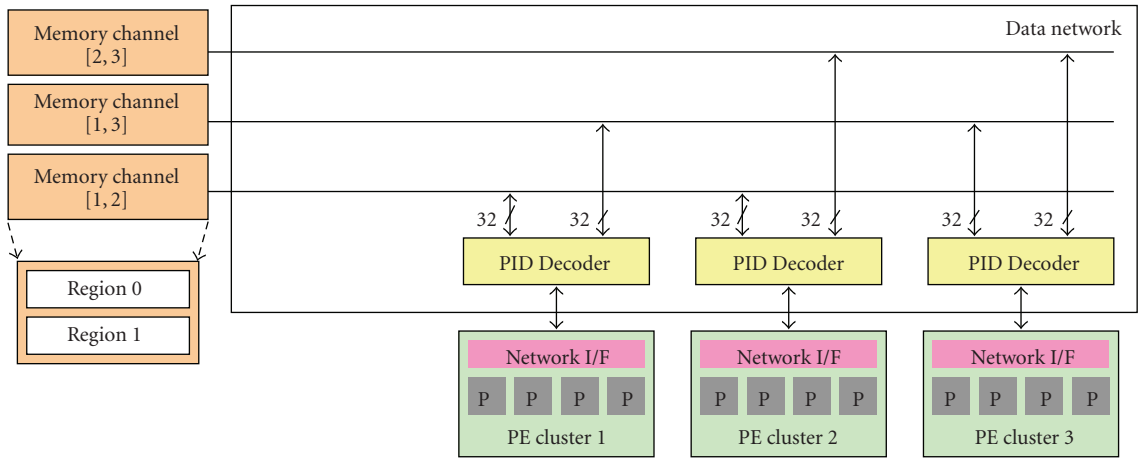
(b) Control transfer code example in producer and consumer tasks

Figure 3: Control network in the communication subsystem.

the accelerator with a VLIW processor with a 64-bit instruction format including several execution slots. From the profile results for parsing part of the H.264/AVC code, we found that 96% of the conditional statements such as if-then-else constructs include one or two conditions. Based on this observation, we decided to provide two instruction formats as shown in Figure 5. One instruction format is composed of two condition slots of 12 bits and two execution slots of 18 bits while the other is comprised of one condition slot of 12 bits and two execution slots of 24 bits. The leftmost 4 bits are used to encode the type of the instruction format and represent the relation among the condition and execution slots, for example, to determine which execution slot is activated according to the evaluation result of a condition slot. Therefore, this instruction format can easily implement many condition-controlled statements including if-then-else constructs with an instruction. Moreover, it also supports several customized instructions to accelerate bit operations, leading zero detection, table matching, and zigzag scanning. With these instructions, we manually developed an assembly code for a parsing accelerator, and its code size is about 1.8 KB for H.264/AVC CAVLC MB parsing.

The filtering accelerator performs weighted summation operations, which are common for transform/quantization, intraframe prediction, and deblocking filtering tasks. And the weighted summation operations occupy more than 80% of the total workload of each task according to profiling with RISC instructions. For their operations, we employ an accelerator that includes multiple copies of the 6-tap filtering datapath shown in Figure 6(a). For example, with four copies of it, integer transform for a 4×4 subblock can be performed in a clock cycle. In the PE cluster including a filtering accelerator, one of its RISC cores first configures the weight coefficients of the filtering accelerator before performing a specific filtering operation. The filtering accelerator also supports a customized instruction for reordering pixel-value.

According to profiling with RISC instructions, SAD and filtering operations occupy about 50% and 20% of the operation, respectively, in motion estimation. To accelerate their operations, therefore, we added several video datapaths for 16×16 SAD computation and 6-tap filtering in the ME/MC accelerator. The datapath for SAD computation, shown in Figure 6(b), can compute the SAD values for all 41 candidate blocks specified in the H.264/AVC standard by simply adding the SAD values for smaller blocks with an adder tree. The ME/MC accelerator also includes internal buffers for search range, interpolated half-pel and quarter-pel values, and motion vectors for a row of MBs. In the video platform, a PE cluster with the ME/MC accelerator should be provided that can perform motion estimation for a video encoder. The RISC cores in the PE cluster should perform all the control operations and data loading from off-chip memory for the ME algorithm and they should also reduce



(a) Data network block diagram

```
Void PE1_thread1_producer () {
    unsigned int value;
    unsigned int baseAdder;
                .
                .
    While (1) {
                .
    for (i) {
        value = x;
        out->write (target PID, baseAdder+i, value);
        }
                .
    }
}
```

```
Void PE2_thread1_consumer () {
    unsigned int value;
    unsigned int baseAdder;
                .
                .
    While (1) {
                .
    for (i) {
        value = in->read (source PID, baseAdder+i);
                .
                .
        }
    }
}
```

(b) Data transfer code example in producer and consumer tasks

Figure 4: Data network in the communication subsystem.

4 bits	12 bits	12 bits	18 bits	18 bits
Type	Cond 0	Cond 1	Exe 0	Exe 1

(a) 2 conditions, 2 base executions

4 bits	12 bits	18 bits	24 bits
Type	Cond 0	Exe 0	Exe 1

(b) 1 conditions, 2 extended executions

Figure 5: Examples of condition-controlled instruction format in the parsing accelerator.

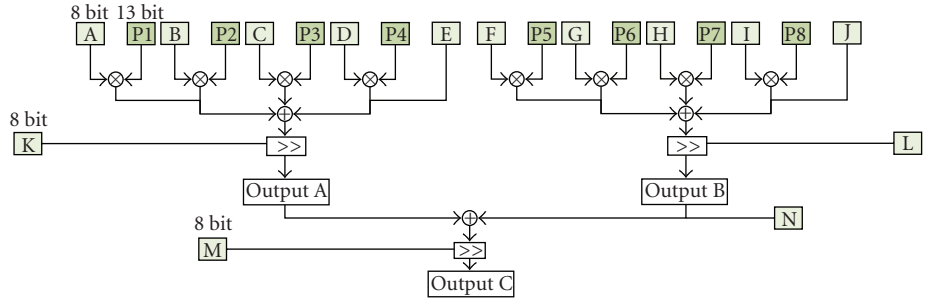
the memory bandwidth by reusing the loaded data as much as possible. Figure 7 shows the job partition between RISC cores and ME/MC accelerator for the ME algorithm where the part that is specific to motion vector searching schemes such as three-step search, directional gradient descent search, and full-search algorithms are executed in the RISC cores so that different ME algorithms can easily be implemented by simply changing the programs run on the RISC cores.

Table 2 lists some commands for task-specific accelerators specialized for parsing, filtering, and ME/MC, respectively, and also includes their execution cycles in the RISC processor.

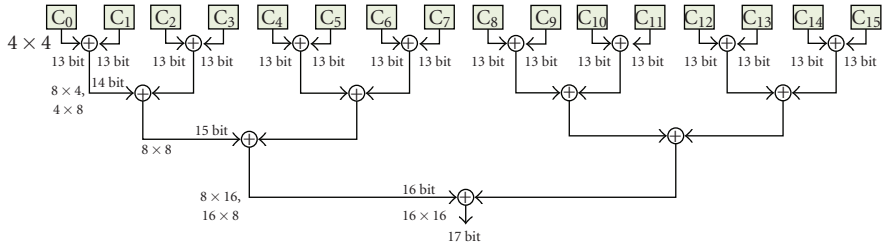
In each PE cluster, its accelerator is tightly coupled with its RISC cores for high-performance, as shown in Figure 8 [18]. Both the ME/MC accelerator and the filtering accelerator get coprocessor commands from the RISC cores as shown in Figure 8(a). In order to decouple the accelerator from the RISC codes, commands are buffered in a queue associated to the RISC core which issued it. In contrast, the parsing accelerator, which is a programmable processor, fetches its instructions from its code memory as shown in Figure 8(b). The accelerator reads input operands from its input memory and writes results to its output memory which can be the input memory of another PE cluster, as shown in Figure 8.

3. Platform Targeted for 720p Video Codec

In this section we will explain how we develop a multiformat video platform targeted for 1280 × 720 30 fps codec systems.



(a) 6-tap filtering engine in the filtering accelerator



(b) Variable block size cost adder tree engine in the ME/MC accelerator

Figure 6: Video datapaths in the filtering and ME/MC accelerators.

Table 2: Commands for task-specific accelerators.

	Commands	Description	Execution cycles on RISC
Parsing Accelerator	P_FBITOP	Fixed length bit operation	40
	P_EXPBITOP	Exp Golomb code bit operation	80
	P_CLZ	Count leading zeros	70
	P_VTMATCH	Variable length table matching operation	100
	P_RLREORDER	Reorder run-level register files	40
	P_T1DEC	Decode TrailingOnes of H.264 CAVLC decoding	60
Filtering Accelerator	F_6TAB	6-tab filtering operation	65
	F_QUANT	Special operation for quantizing operation of multiple register	120
ME/MC Accelerator	M_SAD 4 × 4	Calculate SAD for two pixel register, and store to one pixel registers	66
	M_CMPCUMV	Compare the costs of all 41 candidate blocks and update best motion vectors	4
	M_MVCE	Estimate the cost of motion vectors in pixel difference unit	80
	M_VBSAT	Adder trees for calculating all variable block sizes at once	40
	M_6TAB	6-tab filtering operation	65

For targeting the platform, we used profiling data for an H.264/AVC codec and a VC1 decoder using MB pipelining in the task level shown in Table 3. Note that two tasks for motion estimation of the encoder in Table 3 are much larger workload than other tasks. For integer motion estimation, we used the directional gradient descent search (DGDS) algorithm [19] with a search range of [−64, +64].

Each task is mapped to a PE cluster, which includes RISC processors and a task-specific accelerator. Since video tasks are too huge to be covered only with a small number of RISC processors, their workloads are partitioned into two groups: one for the accelerator and the other for the RISC processors. The size of workload to be mapped into the accelerator could be much larger than the latter, so that the latter should be small enough to be executed on the RISC processors. In the proposed video platform, each PE cluster is initially allocated for each task, which includes a task-specific accelerator: a parsing accelerator in the PARSING cluster, a filtering accelerator in the TQ/ITQ, INTRA, and DEBLOCK clusters respectively, and an ME/MC accelerator in the IME, FME, and INTER clusters, respectively. The workload partition between the accelerator and the RISC processors for each application is shown in Table 4, where its workload is represented with the profiled results in million instructions per sec (MIPS) for compiled code of the task for the RISC processor.

Table 3: Workload for each task in the several HD video codec applications.

Tasks	Required MIPS (720p, 30 fps)		
	H.264/AVC		VC-1
	Decoding	Encoding	Decoding
Syntax parsing (PARSING)	2100	1760	3340
Transform and quantization (TQ/ITQ)	2350	8050	5240
Intraframe prediction (INTRA)	1440	13160	—
Interframe prediction (INTER)	9880	1540	8700
Integer motion estimation (IME)	—	448620	—
Fractional motion estimation (FME)	—	55300	—
Deblocking filtering (DEBLOCK)	3240	4010	6820

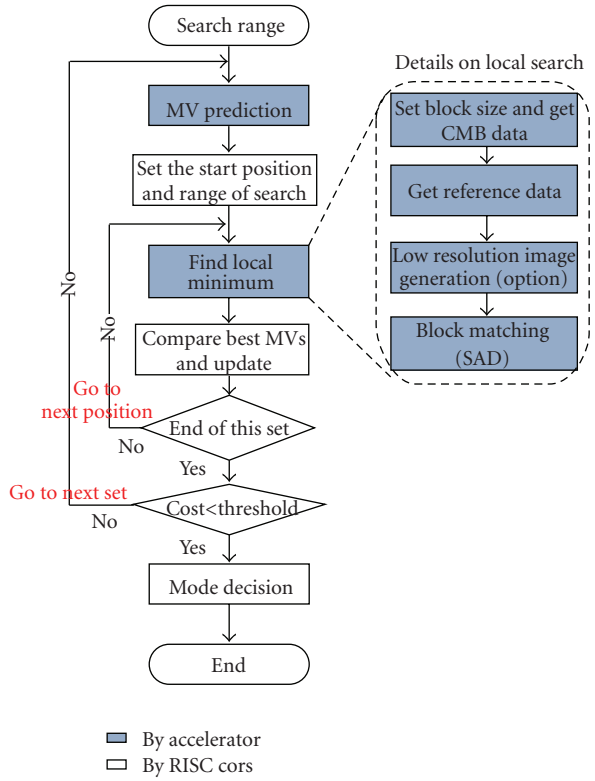


Figure 7: Job partition between RISC cores and ME/MC accelerator in ME algorithm.

For each task, we employ a PE cluster that initially includes four RISC cores, a HOSK, and a task-specific accelerator. Therefore, we represent the workload for the RISC processors with tightly coupled multiple threads to exploit thread-level parallelism. The number of PE clusters in the video platform as well as the number of RISC cores in a PE cluster can be adjusted during optimization process.

As shown in Figure 9, we decided that video decoders are initially partitioned into five tasks, namely PARSING, ITQ, INTRA, INTER, and DEBLOCK tasks. Based on averaged utilization of RISC cores and accelerator in a PE cluster allocated for each task, we decided to merge ITQ and INTRA clusters because their utilization are lower than 0.5 as shown in Figure 9 and because their accelerators are of the same kind. This merging is also helpful for improving performance of the decoder because it substantially reduces overhead of intercluster synchronization for neighboring reconstructed pixels for 4×4 subblock intraframe prediction mode [3, 15, 16]. Utilization after merging tasks is also shown in Figure 9.

We decided that the video encoder is initially partitioned into seven tasks, which include PARSING, TQ/ITQ, INTRA, IME, FME, INTER, and DEBLOCK tasks. Similar to the decoders, the TQ/ITQ and INTRA tasks are first merged together, which is represented with (a) in Figure 10. Second, IME and FME tasks are merged, which is followed by an additional merging of INTER task, represented with (b) in Figure 10.

Based on these architectural decisions, we decided that the video codec platform includes one PE cluster that includes a parsing accelerator, one PE cluster that includes an ME/MC accelerator, and two instances of the PE cluster that includes a filtering accelerator. To map a video application onto the platform, it should be properly partitioned into four tasks; for example, PARSING, ITQ+INTRA, INTER, and DEBLOCK tasks for a decoder while PARSING, TQ/ITQ+INTRA, IME+FME+INTER, and DEBLOCK tasks for an encoder.

To tune the platform's communication networks for 720p video applications, we used the profiling data for an H.264/AVC encoder. We need to determine the size of FIFOs for each link in the control network and the size of each shared memory for each link in the data network. These sizes can be easily determined based on simple calculations or communication traces from simulation because patterns of control and data transfers are relatively regular in the video codec applications. Due to the MB pipelining, the transfer size of data between PE clusters can be easily estimated. Because each pixel in residual, transform, and reconstruction data is less than 2 bytes, the size of the shared memory for each data link can be chosen to be equal to 1.5 KB, which is $8 \times 8 \times 6 \times 2$ bytes $\times$ 2 for luma and chroma pixels per macroblock when a double-buffered scheme is used [3, 15, 16]. We simply allocate a memory of 2 KB for each data link in the data network to provide about 25% margins.

Parsed syntax elements, which are outputs of the PARSING task, should be transferred to other tasks. For example,

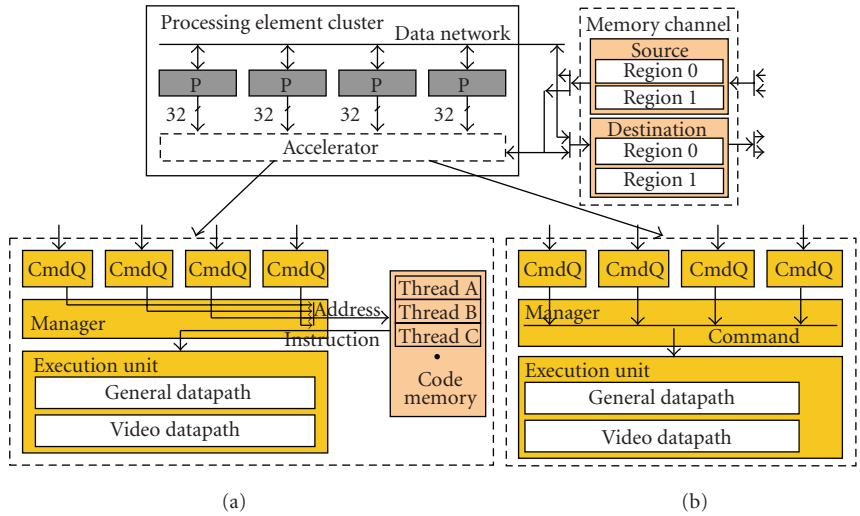


Figure 8: Task-specific accelerator block diagram tightly coupled with RISC cores. (a) Parsing accelerator. (b) Filtering, ME/MC accelerator.

Table 4: Job partitioning between the accelerators and the RISC cores.

	Total required MIPS	MIPS executed by accelerators		MIPS executed by RISC cores	
H.264 decoding	19010	18460	97.1%	550	2.9%
H.264 encoding	532450	531270	99.8%	1180	0.2%
VC-1 decoding	24100	23220	96.3%	880	3.7%

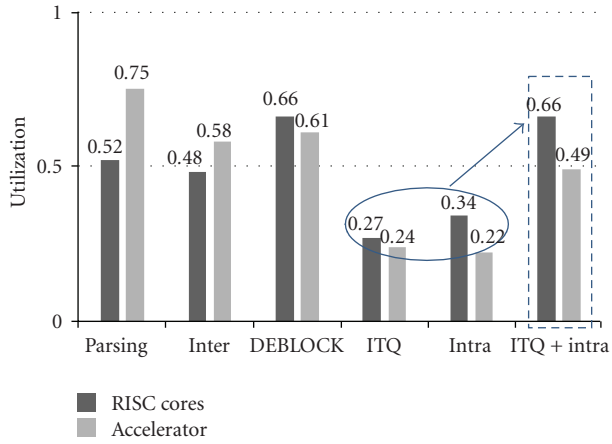


Figure 9: Utilization of RISC cores and accelerator for each task in the H.264/AVC HD decoder application.

a motion vector is transferred to the ME/INTER task, and a boundary strength to the DEBLOCK task. The largest syntax elements in the video codec applications are equal to 16 × 2 bytes × 2 and 16 × 4 bits × 2, respectively [3, 15, 16]. For each data link, its source and destination tasks need to exchange handshake signals for synchronization and a flag signal specifying a memory region in its shared memory using a double-buffered scheme. To transfer the syntax elements and these signals, the size of the group of FIFO for each control link can be chosen to be 32-bit FIFOs with a depth of 24 to provide about 50% margins.

Figure 11 shows reduction to the data transfer rates in the communication subsystem when the number of PE clusters is changed from five to four in the H.264/AVC decoder and from seven to four in the H.264/AVC encoder, respectively, as explained before, where the transfer rate reduction in the control and data networks are 4.2% and 33.3% for the decoder, and 26.8% and 51.8% for the encoder, respectively.

The platform includes a direct memory access control (DMAC) and a double-data-rate (DDR) memory controller in the 32-bit AMBA AHB bus for a global link to the off-chip for reference frames. For example, the memory bandwidth requirement for H.264/AVC 720p 30 fps encoding is up to 700 Mbytes/sec including about 80 Mbytes/sec for TFTLCD controller. The platform also includes a RISC processor as the host, a TFTLCD controller, and a UART controller. Each PE cluster is allocated with 16 KB instruction cache and 4 KB data cache, which are shared by four RISC cores. Note that a proper cache usage and the number of active RISC core in each PE cluster are chosen from the experiment that will be discussed in Section 5. Figure 12 shows the overall architecture of platform for the HD video codec applications.

We synthesized all components of the platform with a target clock frequency of 200 MHz for 65 nm CMOS technology. The gate complexity and memory size of the platform are summarized in Table 5, where P, M, and F represent the parsing, ME/MC, and filtering accelerators, respectively. Its total equivalent logic gate count and chip size are estimated approximately 3,820 K and around 9 mm^2, respectively.

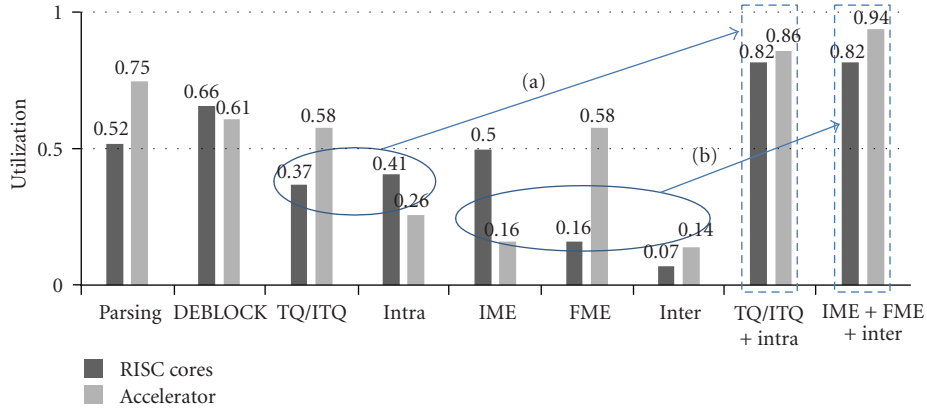


Figure 10: Utilization of RISC cores and accelerator for each task in the H.264/AVC HD encoder application.

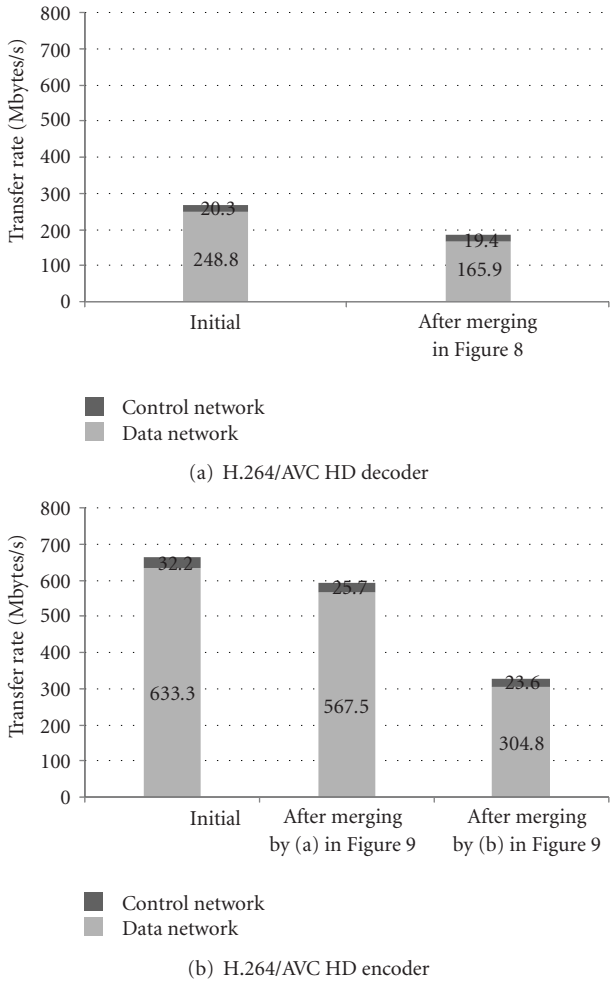


Figure 11: Transfer rate in the communication subsystem for H.264/AVC HD decoder/encoder.

To obtain the performance for higher resolution applications like 1080p, which requires processing 244,800 MBs per second, the cycle budget per macroblock is only 816 cycles when the clock frequency is assumed to be 200 MHz. This performance can be achieved by allocating two accelerators for each PE cluster, which should work cooperatively for dual MB-level parallelism. At the same time, the size for each link should be doubled in the communication network in order to support dual MB-based pipelines.

Table 5: Area breakdown for the platform.

	Platform Design
RISC cores, HOSK, Cache Controller, Network Interface	250 K × 4
I/D Cache Memory	64 KB/16 KB
Accelerator (Logic)	88 K × 1 (P) + 700 K × 1 (M) + 170 K × 2 (F)
Accelerator (Memory)	8 KB × 1 (P) + 20 KB × 1 (M) + 12 KB × 2 (F)
FIFO Channel	5 K × 6
Memory Channel	2 KB × 6
Others (Host Processor, DMAC, Memory Controller, TFTLCD Controller, Peripheral)	350 K + 20 KB
Total (Estimated)	2508 K gates + 164 KB (7.6 mm^2)

4. Mapping Framework Exploiting SystemC Modeling

In this section, we will explain a framework for mapping an application to the platform. In the application mapping framework, a sequential C/C++ code for the target application should manually be transformed into a SystemC model while satisfying a coding convention defined for automatic generation of parallel executable codes, as shown in Figure 13.

First, we should partition the sequential C/C++ code of the target application into several tasks, each of which is again divided into several threads in the transaction level in SystemC [8, 11, 20–22]. This transaction-level (TL) SystemC model can be incrementally developed by running a mixed-level model in the virtual prototype of the video platform.

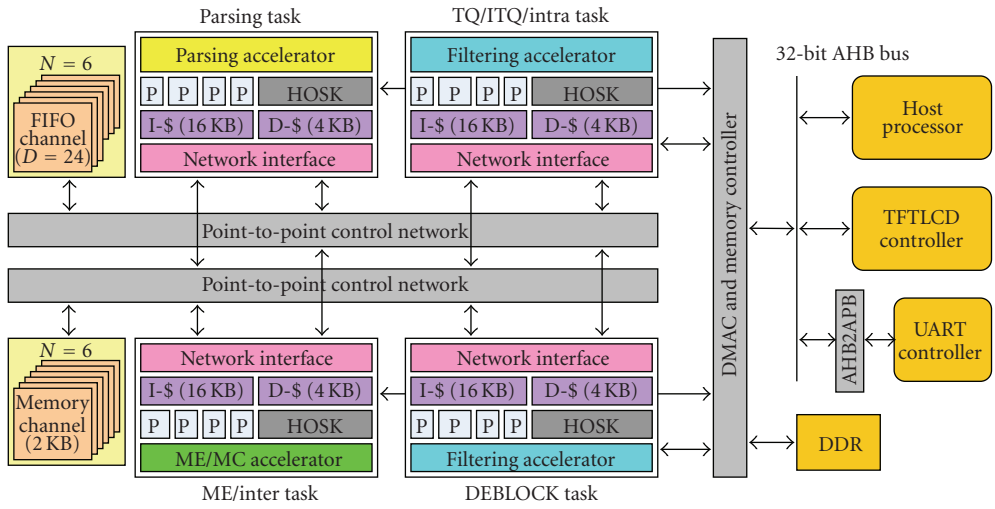


Figure 12: Overall architecture of platform for the HD video codec application.

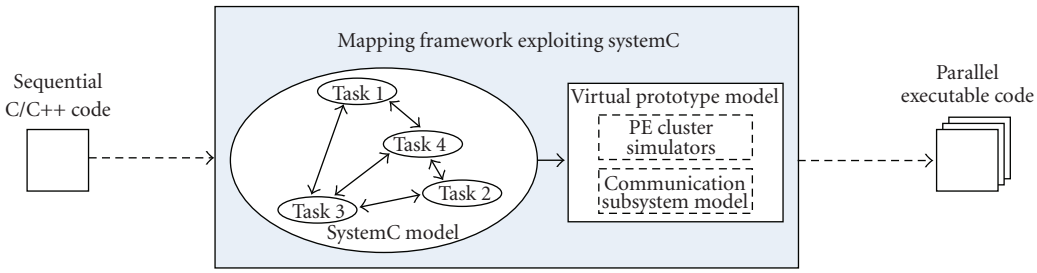


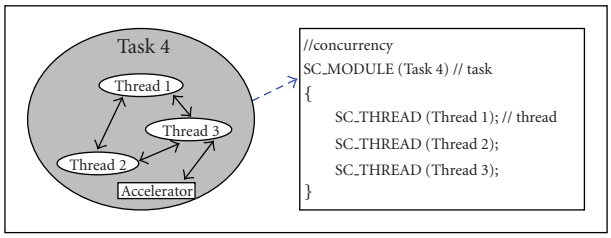
Figure 13: Mapping framework exploiting SystemC.

In the mixed-level simulation, some tasks of the target application represented in SystemC are simulated with the SystemC simulation engine while the others coded in C/C++ are executed with their ISS simulators. To generate parallel executable codes of the target application for the platform from the SystemC model, we employ a coding convention to define tasks and threads as well as an assembly-level library for communication and HOSK APIs to be replaced with SystemC primitives. We can estimate the performance of an application by running its parallel executable codes in the virtual prototype of the video platform. Based on this performance estimation, we can also configure the platform such as the number of RISC cores and the cache sizes for each of its PE clusters.

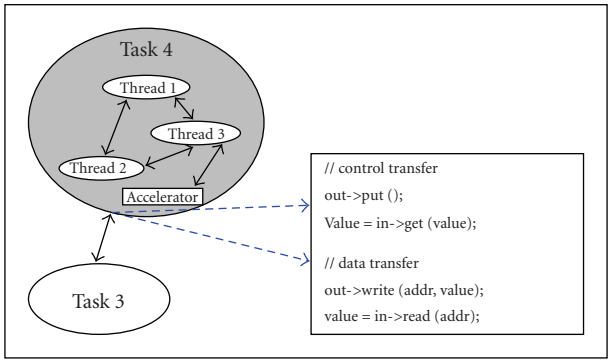
4.1. SystemC Model for the Video Platform. We represent concurrency in the platform by using SystemC constructs such as SC_MODULE and SC_THREAD in the SystemC models of the target application. As summarized in Figure 14(a), the coding convention is as follows. An SC_MODULE construct defines a task, which encapsulates a set of threads that are executed concurrently in a PE cluster. An SC_THREAD construct defines a thread, which is scheduled by the HOSK in the PE cluster. For intercluster communication transactions between the tasks, four sc_interface primitives such as PUT, GET, WRITE, and READ primitives are used as shown in Figure 14(b). PUT/GET primitives are used for control transfers while WRITE/READ primitives for data transfers.

In each PE cluster for a task, there is at least one accelerator that performs most of the computation for the task. The threads in the same PE cluster communicate with the accelerator by putting PUT and GET primitives into the command queue as shown in Figures 14(d) and 8. For intra-cluster communication transactions between a pair of threads within the same task, we can use either semaphores or PUT/GET primitives. The semaphores, which are managed by the HOSK, are used for synchronization between threads, as shown in Figure 14(c) while the PUT/GET primitives are used for control transfer between threads. The HOSK provides 11 *API calls* including communication primitives, which are summarized in Table 6. For example, change_thread_priority and set_active_core *API calls* are used for assigning a priority to the threads in the same task and specifying the number of active RISC cores in its PE cluster, respectively.

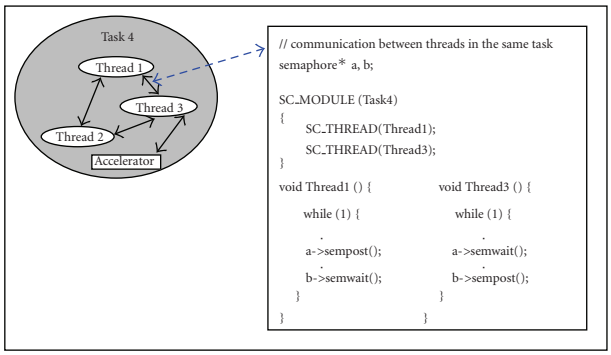
A SystemC model for the target application exposes parallelism of its tasks and threads and their communication on the target platform although the details of the communication are hidden with communication API. The application mapping framework generates parallel executable codes for the platform from the SystemC model.



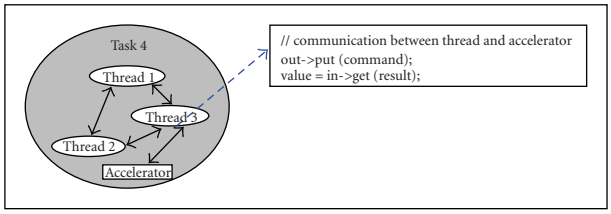
(a) Concurrency for task and thread



(b) Intercluster communication between different tasks



(c) intra-cluster communication between threads in the same task



(d) Communication between thread and accelerator in the same task

Figure 14: Programming model using SystemC's construct.

4.2. Parallel Executable Code Generation. To generate parallel executable codes from the SystemC model, we replace SystemC constructs with corresponding inline assembly codes of the HOSK APIs through a preprocessor before compilation as shown in Figure 15. Each SC_THREAD construct is replaced with the assembly code for HOSK API, thread_create, which receives a pointer of its entry function as an argument, allocates a context memory for the thread, and sets the program counter field in the context memory with the address of the entry function.

Table 6: SystemC API calls for HOSK and communication.

API calls	Description
Thread_create, thread_kill	Create new thread, Terminate running thread
Set_active_core	Notify the number of active RISC cores
Change_thread_priority	Change the priority of thread
Sem_init, sem_wait, Sem_post	Synchronization for semaphores
Put/get, write/read	Transfer for communication

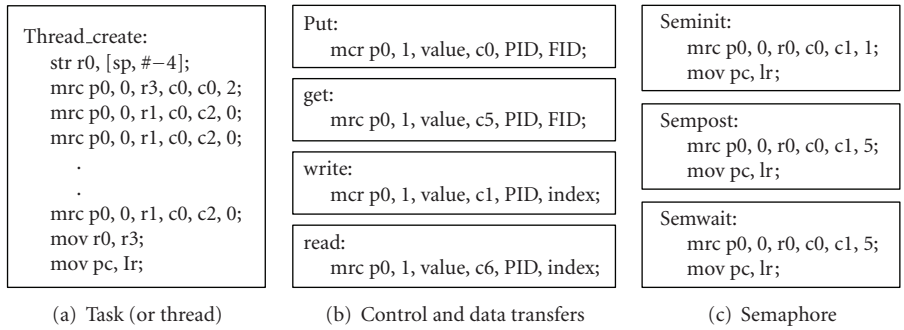
Each task in the TL SystemC model is separately compiled, linked together, and finally loaded by using the platform's memory map as shown in Figure 16. The assembly codes for HOSK APIs are linked to the compiled code for each task, and the size of HOSK API is about 1.5 KB including the HOSK kernel code. The HOSK for a task dynamically allocates the threads to the RISC cores in its PE cluster at run time. For thread scheduling, the HOSK manages a ready queue from which it selects the thread with the highest priority.

5. Experimental Mapping Results

To evaluate and confirm the video platform and its mapping framework, we implemented an H.264/AVC 720p codec that can encode 30.4 fps and decode 32.2 fps and a VC-1 720p decoder that can decode, 33.1 fps respectively, assuming that the operating frequency for the RISC processors is 200 MHz. In this experiment, we used "Parkrun" sequences of 1280×720 pixels resolution as an input image sequence [23]. Table 7 summarizes several mapping results into the platform including the number of active RISC cores, the number of threads, the number of context switching per macroblock, code size, and I/D cache usage for each PE cluster corresponding to a task, respectively. Table 8 lists thread names for the applications summarized in Table 7, where each name also implies its function.

The average performance curve of each task in the H.264/AVC 720p encoder when the number of active RISC cores in its PE cluster is changed from one to four is depicted in Figure 17(a). For this evaluation for a different number of the active RISC cores in each PE cluster, the SystemC model does not need to be modified except for just configuring the number of active RISC cores through a HOSK *API call* of set_active_core (N), which means that we can easily obtain parallel executable codes. Utilization of each active RISC cores in the video platform is also depicted in Figure 17(b) for four tasks of the H.264/AVC 720p encoder.

The performance curve of each PE cluster in the H.264/AVC 720p 30 fps encoder is shown in Figure 18 for various cache configurations, where an encircled node annotated with its cache miss rate in percent associates the chosen cache size. After selecting an instruction cache size assuming a perfect data cache first, we select a data cache size assuming the chosen instruction cache size. These results are



(a) Task (or thread) (b) Control and data transfers (c) Semaphore

Figure 15: Assembly code examples of the HOSK and communication APIs for an executable code generation.

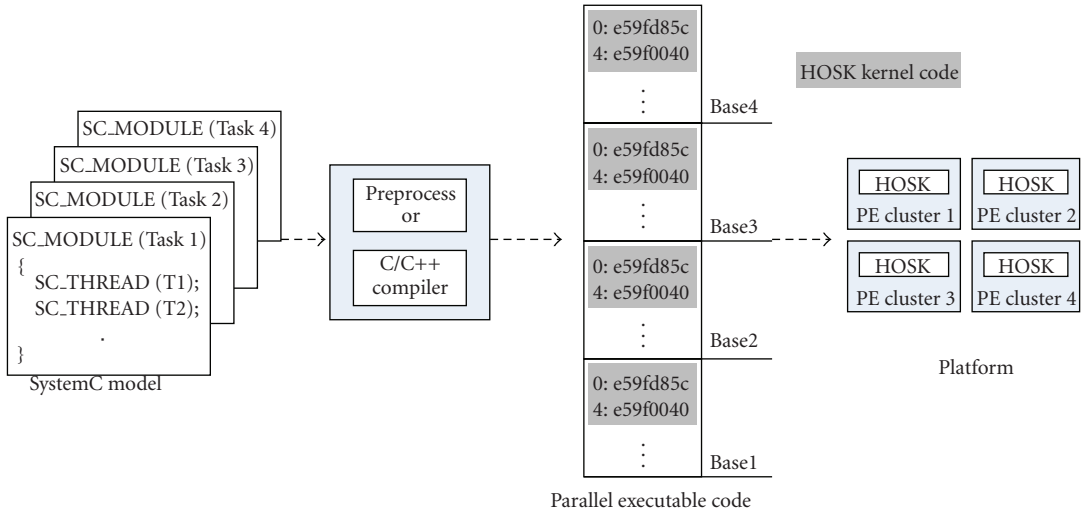


Figure 16: Loading parallel executable code into the platform.

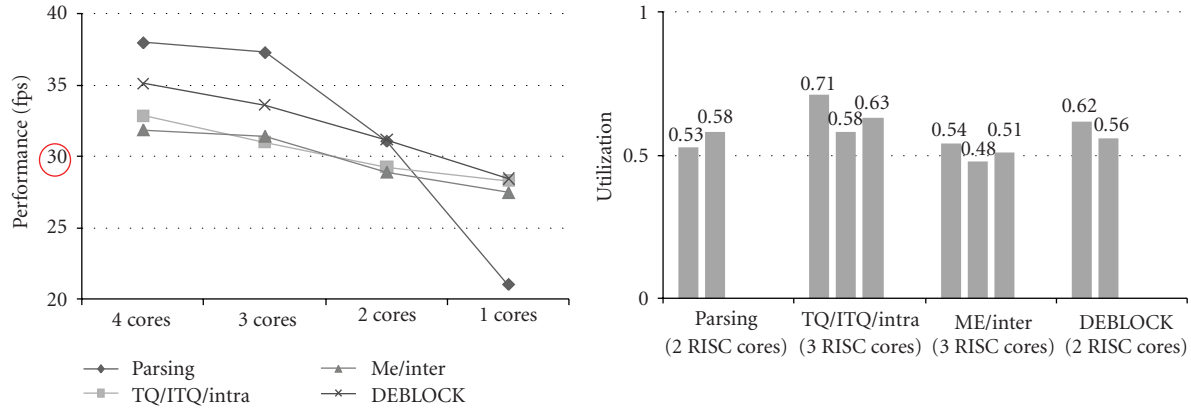
obtained, assuming that the cache miss penalty is about 20 clock cycles. The cache miss rate for various cache sizes of each task is shown in Figure 19.

The averaged overheads of context switching and thread scheduling per macroblock are summarized respectively in Figure 20 where each of them is normalized by the number of its active RISC cores is shown. The maximum overhead in the H.264/AVC decoder is 7.6% for ITQ/INTRA task, that in the H.264/AVC encoder is 6.5% for TQ/ITQ/INTRA task and that in the VC-1 decoder is 3.3% for DEBLOCK task. Thanks to the accelerated HOSK, the overhead of context switching and scheduling is below 10% of the overall execution time.

In Figure 21, the performances for the PARSING, ITQ/INTRA, and DEBLOCK tasks in the H.264/AVC decoder are compared for two cases: with HOSK and with software-only RTOS. With the HOSK, the performance of each task is substantially improved by reducing the overhead of a context switching to 20 cycles from about one thousand cycles [5]. Therefore, the hardware support for context switching with the HOSK is highly justified for the tasks with tightly coupled multiple threads just as the video platform for HD applications. The overall size of HOSK kernel codes for the H.264/AVC decoder, H.264/AVC encoder, and the VC-1 decoder, which are summarized in Table 7, are approximately 10%, 13%, and 12%, respectively, in the code size while a copy of the HOSK kernel code is approximately 1.5 KB including its APIs assembly code.

The utilization of the active RISC cores is depicted in Figure 22 for each task in the H.264/AVC 720p decoder and the VC-1 720p decoder. Note that the utilization of the active RISC cores for the H.264/AVC encoder is shown in Figure 17(b). The numbers of the active RISC cores for DEBLOCK task in both H.264/AVC decoder and encoder are not the same because the boundary strength for deblocking filtering is calculated in the DEBLOCK task for the encoder and in the syntax parsing task for the decoder, respectively. Note that the boundary strength for each 4-pel edge, which value is inclusively between 0 and 4, is used to select one of filtering types in deblocking filtering in H.264/AVC [15]. Compared to the DEBLOCK in the H.264/AVC standard, its task in VC-1 requires more computation because it performs both overlap smoothing and in-loop filtering together [3, 16]. Therefore, the utilization of the RISC cores in DEBLOCK task is relatively high.

Figure 23 shows the estimated power consumption for the H.264/AVC 720p encoder in Table 7. These values are measured through the RTL simulation using Synopsys Power Compiler [24]. In the power estimation, we used "Parkrun"



(a) Performance curve of each PE cluster for different active RISC cores

(b) Utilization of active RISC cores for each task

Figure 17: Performance curve of each PE cluster for different active RISC cores and utilization of active RISC cores for each task in the H.264/AVC 720p encoder.

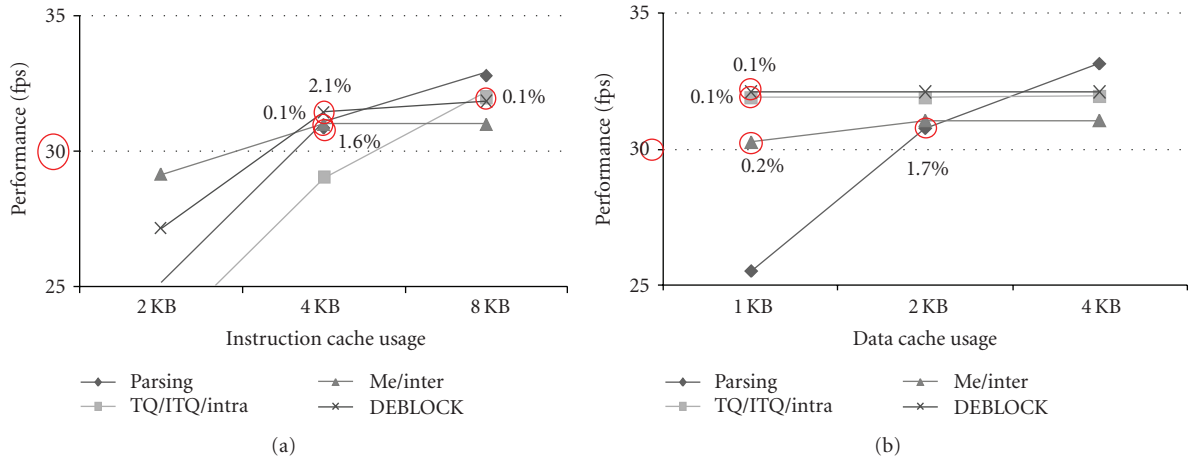
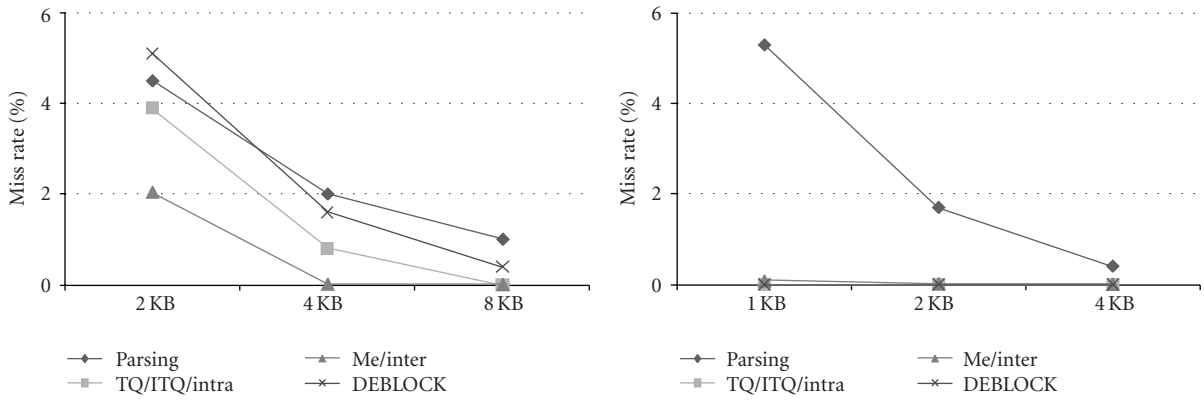


Figure 18: Performance curve of each PE cluster for different cache usage in the H.264/AVC 720p encoder.



(a) Instruction Cache Usage (KB)

(b) Data Cache Usage (KB)

Figure 19: Cache miss rate for various cache sizes of each task in the H.264/AVC 720p encoder.

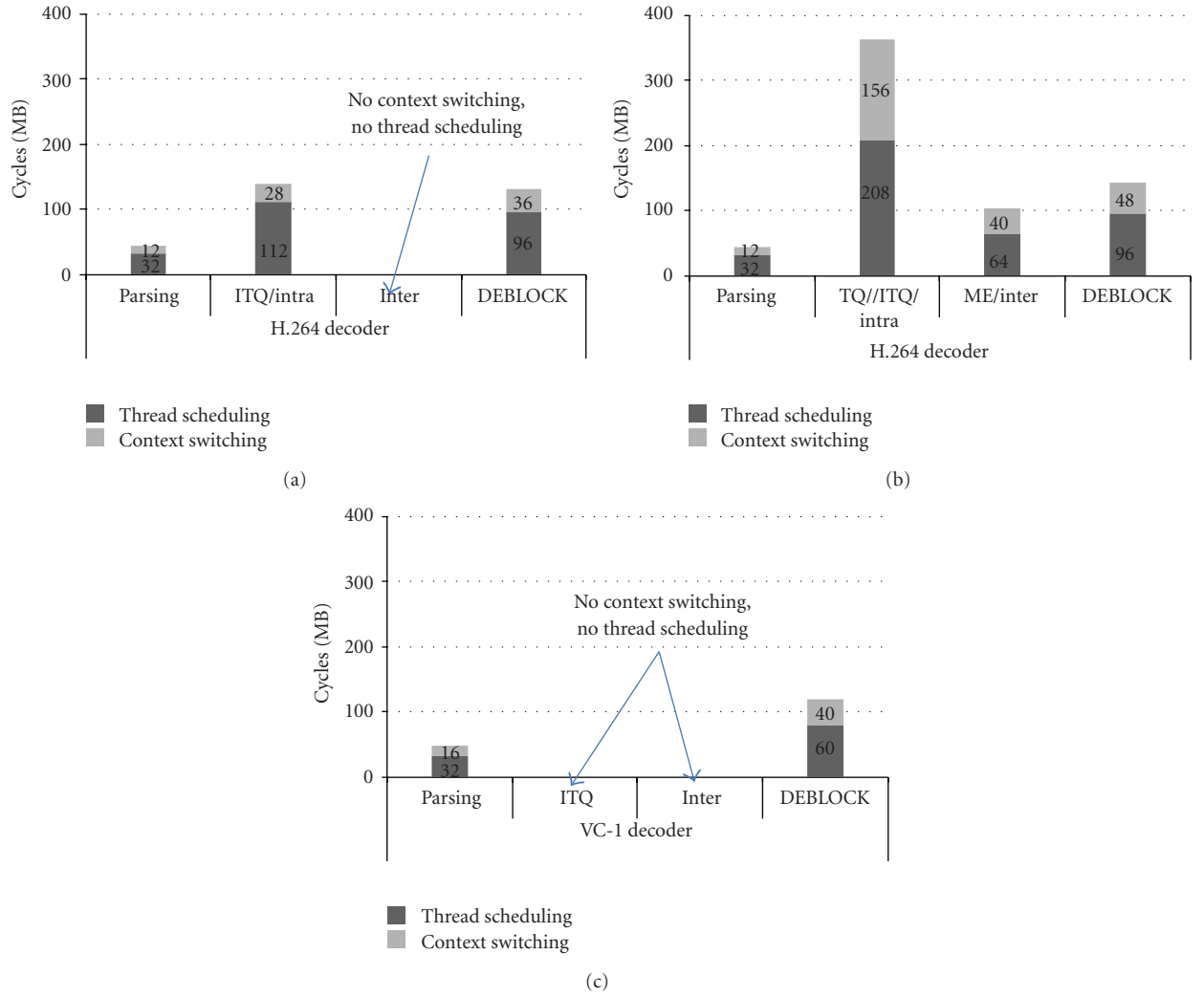


Figure 20: Context switching and thread scheduling cycle of each task for each application in Table 7.

Table 7: Mapping results for several applications.

	Task	Number of active RISC cores	Number of threads	Number of context switching	Code size (KB)	(I, D) Cache usage (KB)
H.264 Decoder	PARSING	2	4	2	26	(4, 2)
	ITQ/INTRA	1	3	7	11	(4, 1)
	INTER	1	1	0	15	(2, 1)
	DEBLOCK	1	2	6	7	(4, 1)
H.264 Encoder	PARSING	2	3	2	26	(4, 2)
	TQ/ITQ/INTRA	3	6	13	7	(8, 2)
	ME/INTER	3	5	4	6	(4, 2)
	DEBLOCK	2	3	6	7	(4, 1)
VC-1 Decoder	PARSING	2	3	2	18	(4, 2)
	ITQ	2	2	0	6	(2, 1)
	INTER	1	1	0	16	(1, 1)
	DEBLOCK	2	4	5	10	(8, 1)

Table 8: Threads in each task for several applications in Table 7.

H.264 Decoder		H.264 Encoder		VC-1 Decoder	
PARSING	Control	PARSING	Control	PARSING	Control
	MVD calculation		MVD calculation		MVD calculation
	NAL decoding		NAL encoding		NAL decoding
	bS calculation				
ITQ/INTRA	ITQ	TQ/ITQ/INTRA	Luma TQ/ITQ	ITQ	ITQ
	INTRA		Chroma TQ/ITQ		AC/DC prediction
	Reconstruction		Cost compare and prediction decision		
			Luma INTRA		
			Chroma INTRA		
			Reconstruction		
INTER	INTER	ME/INTER	Control	INTER	INTER
			IME I		
			IME II		
			FME		
			INTER		
DEBLOCK	Luma DF	DEBLOCK	bS calculation	DEBLOCK	In-loop Luma DF
	Chroma DF		Luma DF		In-loop Chroma DF
			Chroma DF		Overlap Luma DF
					Overlap Chroma DF

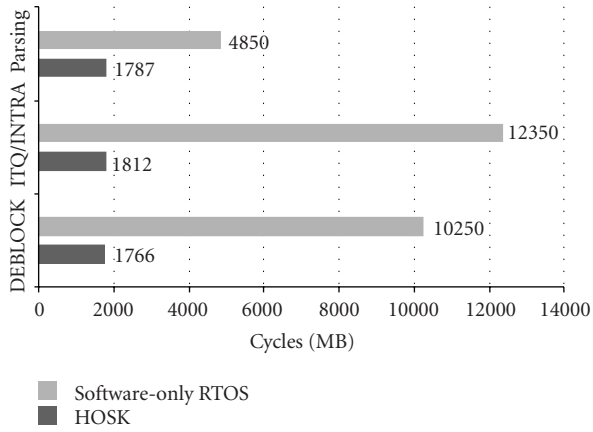
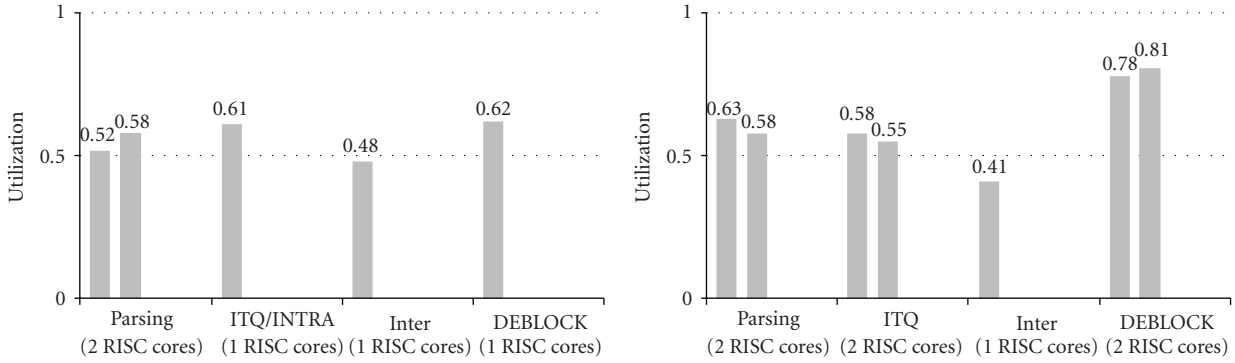


Figure 21: Performance comparison with HOSK and software-only RTOS in the H.264/AVC 720p decoder.

sequence of 1280 × 720 pixels resolution as an input image sequence to the encoder [23] and employed the DGDS motion estimation algorithm [19] for a search range of [−64, +64] to find integer motion vectors and the quarter-pixel fractional motion vectors. When the number of PE clusters is reduced from seven to four, as explained in Section 3, the total power consumption is reduced 14.4% as shown in Figure 23(a). On the average, the platform consumes about 32.7 mW for RISC processors, 9.1 mW for HOSKs, 40.5 mW for accelerators, 60.3 mW for instruction caches, 7.1 mW for data caches, and 26.0 mW for communication network, respectively, as shown in Figure 23(b). Its total power consumption is about 175.8 mW when estimated before placement and routing in 65 nm CMOS technology. Deduced from the previous figures, each task consumes about 35.5 mW for PARSING, 30.7 mW for TQ/ITQ/INTRA, 79.0 mW for ME/INTER, and 30.6 mW for DEBLOCK, respectively. We found that the instruction caches occupy about 35% of the total power consumption of the encoder implementation. Instead of the instruction cache, therefore, we should try to scratch pad memories to further reduce the power consumption because the video codec applications typically have regular memory access patterns that can be statically analyzed and predicted at compile time [25].

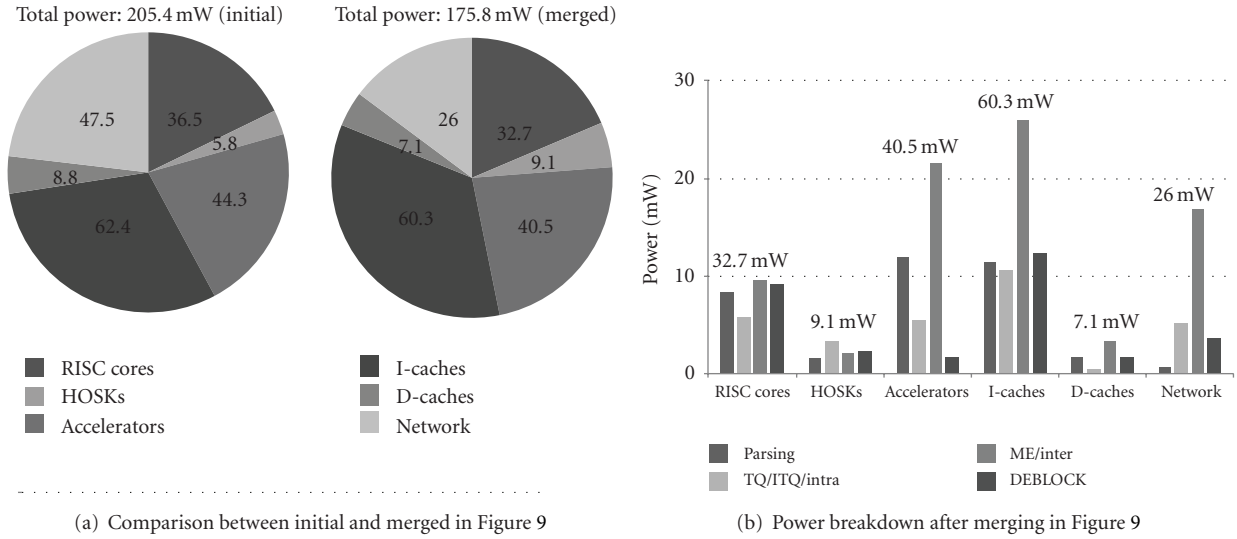
The performance of the parsing accelerator is evaluated as shown in Table 9. An H.264/AVC reference software code and its optimized C code are profiled on the RISC processor, excluding NAL decoding part for fair performance comparison. The parsing accelerator can parse the bitstream in less than 4.8 cycles per bit on the average, which is about 16 times faster than the optimized C code executed in the RISC processor. This result implies that an H.264/AVC CAVLC bitstream of 40 Mbps can be parsed with the parsing accelerator at 192 MHz.

The utilization of the active RISC cores and the accelerator is depicted in Figure 24 for different motion estimation algorithms with a search range of [−64, +64]: four-step search (4SS) [26], block-based gradient descent search (BBGDS) [27], and directional gradient descent search (DGDS) [19] where they employ two, two, and three RISC



(a) H.264/AVC 720p decoder (b) VC-1 720p decoder

Figure 22: Utilization of active RISC cores for each task in the H.264/AVC decoder and VC-1 720p decoder.



(a) Comparison between initial and merged in Figure 9 (b) Power breakdown after merging in Figure 9

Figure 23: Estimated power consumption for an H.264/AVC 720p encoder.

Table 9: Parsing accelerator performance for several test sequences of 30 frames.

	Foreman	Earth	Birds
Image size	CIF	D1	720p
Bit rate (Mbps)	1.27	7.25	15.70
Reference SW on RISC (Mcycles)	411.1	2099.2	4302.0
Optimzed Code on RISC (Mcycles)	100.6	515.8	1059.7
Proposed parsing accelerator (Mcycles)	6.1	32.6	67.5
Cycles/bit	4.8	4.5	4.3
Speed up [Proposed/Optimized code]	16.5	15.8	15.7

cores, respectively, and most of the computation for the motion estimation task is offloaded to the accelerators such as SAD and filtering operations. Therefore, we can easily implement a different ME algorithm by simply changing RISC codes which correspond to part of the ME algorithm that are specific to motion vector searching scheme. In this experiment, we used "Park run" sequences of 1280 × 720 pixel resolution as an input image sequence [23].

6. Conclusions

We proposed a high-performance programmable video platform that utilizes SystemC programming model for its application mapping. The platform has four PE clusters, which are connected with two separate point-to-point control and data networks together with a DMAC and DDR controller. With the application mapping framework that exposes the architectural details of the target platform, we can get a good executable code utilizing all the resources easily at earlier time.

From several mapping results, we found that the platform with its mapping framework is easy and suitable for implementing the HD video codec applications at a reasonable operating frequency. In the 65 nm CMOS technology, the complexity of the platform is around 3,820 K equivalent gates in total. With the platform running at the operating frequency of 200 MHz, we have implemented an H.264/AVC 720p codec that encodes 30.4 fps and decodes 32.2 fps and a VC-1 720p decoder that decodes 33.1 fps, respectively.

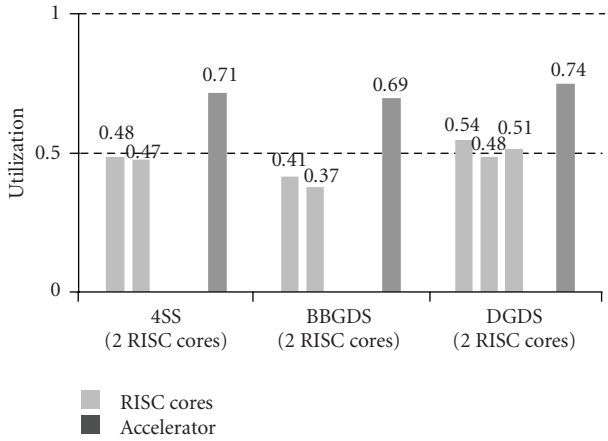


Figure 24: Utilization of active RISC cores and accelerator for different motion estimation algorithm implementations.

For the future works, we plan to map MPEG-4 and other video standards into our platform, and to improve the platform to achieve the performance for higher resolution applications like 1080p after adjusting the number of PE clusters in the platform and newly allocating a task-specific accelerator to each PE cluster. Moreover, we will compare the power consumption of the platform after replacing the instruction caches in each PE cluster with scratch pad memories to find out how much the power consumption can be reduced. Furthermore, we need to try different topologies for interconnect networks to find an efficient solution when the number of PE clusters gets larger in the video platform for higher resolution video codecs. We will also develop a tool for the timed transaction-level simulation to estimate the performance in the earlier time.

References

[1] T. C. Chen, S. Y. Chien, Y. W. Huang et al., "Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 6, pp. 673–688, 2006.

[2] K. Iwata, S. Mochizuki, M. Kimura et al., "A 256 mW 40 Mbps full-HD H.264 high-profile codec featuring a dual-macroblock pipeline architecture in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1184–1191, 2009.

[3] Y. S. Tung, S. W. Wang, C. W. Tsai, Y. T. Yang, and J. L. Wu, "DSP-based multi-format video decoding engine for media adapter applications," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 1, pp. 273–280, 2005.

[4] Joint Collaborative Team of ITU-T VCEG, ISO/IEC JTC 1/SC 29/WG 11, Joint call for proposals for next-generation video coding standardization, 2010.

[5] P. G. Paulin, C. Pilkington, M. Langevin et al., "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 667–679, 2006.

[6] M. Z. Urfianto, T. Isshiki, A. U. Khan, D. Li, and H. Kunieda, "A multiprocessor SoC architecture with efficient communication infrastructure and advanced compiler support for easy application development," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 91, no. 4, pp. 1185–1196, 2008.

[7] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. Aboulhamid, and F. Boyer, "SPACE: a hardware/software systemC modeling platform including an RTOS," in *Proceedings of the International Forum on Specification and Design Languages (FDL '03)*, Frankfurt, Germany, September 2003.

[8] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic embedded software generation from SystemC,," in *Proceedings of the Design, Automation, and Test in Europe*, 2003.

[9] M. Besana and M. Borgatti, "Application mapping to a hardware platform through automated code generation targeting a RTOS," in *Proceedings of the Design, Automation, and Test in Europe*, pp. 41–44, 2003.

[10] N. Pazos, P. Ienne, Y. Leblebici, and A. Maxiaguine, "Parallel modeling paradigm in multimedia applications: mapping and scheduling onto a multi-processor system-on-chip platform," in *Proceedings of the International Global Signal Processing Conference*, 2004.

[11] S. Park, D. S. Hong, and S. I. Chae, "A hardware operating system kernel for multi-processor systems," *IEICE Electronics Express*, vol. 5, no. 9, pp. 296–302, 2008.

[12] J. Cho, D. Lee, S. Yoon, S. Park, and S. I. Chae, "VLSI implementation of a VC-1 main profile decoder for HD video applications," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 92, no. 1, pp. 279–290, 2009.

[13] Y. Yi and B. C. Song, "A novel CAVLC architecture for H.264 video encoding at high bit-rate," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '08)*, pp. 484–487, May 2008.

[14] S. V. Tota, M. R. Casu, M. R. Roch, L. MacChiarulo, and M. Zamboni, "A case study for NoC-based homogeneous MPSoC architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 3, pp. 384–388, 2009.

[15] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, Draft ITU-T Recommendation and Final Draft International Standard Joint Video Specification ITU-T Rec. H.264—ISO/IEC, 14496-10 AVC JVT_G050, 2003.

[16] "VC-1 Compressed Video Bitstream Format and Decoding Process (SMPTE 421M-2006)," SMPTE Standard, 2006.

[17] L. K. Liu and E. Feig, "A block-based gradient descent search algorithm for block motion estimation in video coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 4, pp. 419–422, 1996.

[18] I. Gelado, J. H. Kelm, S. Ryoo, S. S. Lumetta, N. Navarro, and W. M. W. Hwu, "CUBA: an architecture for efficient CPU/Co-processor data communication," in *Proceedings of the 22nd ACM International Conference on Supercomputing (ICS '08)*, pp. 299–308, June 2008.

[19] L. M. Po, K. H. Ng, K. W. Cheung, K. M. Wong, Y. M. Salah Uddin, and C. W. Ting, "Novel directional gradient descent searches for fast block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 8, pp. 1189–1195, 2009.

[20] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, Norwell, Mass, USA, 2002.

[21] F. Herrera and E. Villar, "A framework for embedded system specification under different models of computation in SystemC," in *Proceedings of the Design Automation Conference*, pp. 911–914, 2006.

[22] S. Park, S. Yoon, and S. I. Chae, "A mixed-level virtual prototyping environment for SystemC-based design methodology," *Microelectronics Journal*, vol. 40, no. 7, pp. 1082–1093, 2009.
[23] ftp://ftp.ldv.e-technik.tu-muenchen.de/ .
[24] Synopsys Power Compiler User Guide, Release V, 2004, http://www.synopsys.com/.
[25] I. Issenin, E. Brockmeyer, B. Durinck, and N. Dutt, "Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies," in *Proceedings of the Design Automation Conference*, pp. 49–52, 2006.
[26] L. M. Po and W. C. Ma, "A novel four-step search algorithm for fast block motion estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 3, pp. 313–317, 1996.
[27] L. K. Liu and E. Feig, "A block-based gradient descent search algorithm for block motion estimation in video coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, no. 4, pp. 419–422, 1996.