Pipelined Scheduling of Functional HW/SW Modules for Platform-Based SoC Design

Wonjong Kim, June-Young Chang, and Hanjin Cho

We developed a pipelined scheduling technique of functional hardware and software modules for platform-based system-on-a-chip (SoC) designs. It is based on a modified list scheduling algorithm. We used the pipelined scheduling technique for a performance analysis of an MPEG4 video encoder application. Then, we applied it for architecture exploration to achieve a better performance. In our experiments, the modified SoC platform with 6 pipelines for the 32-bit dual layer architecture shows a 118% improvement in performance compared to the given basic SoC platform with 4 pipelines for the 16-bit single-layer architecture.

Keywords: Pipelined scheduling, platform-based SoC design, MPEG4.

Manuscript received Jan. 12, 2005; revised May 20, 2005.

The material in this work was presented in part at IT-SoC 2004, Seoul, Korea, Oct. 2004.

Wonjong Kim (phone: +82 42 860 6890, email: wjkim@etri.re.kr), June-Young Chang (email: jychang@etri.re.kr), and Hanjin Cho (email: hjcho@etri.re.kr) are with Basic Research Laboratory, ETRI, Daejeon, Korea.

I. Introduction

System-on-a-chip (SoC) can be defined as a complex IC that integrates the major functional elements of a complete end-product into a single chip or chipset. In general, SoC design incorporates at least one programmable processor, on-chip memory, and accelerating functional modules implemented in hardware. It also interfaces with peripheral devices, and/or the real world, and encompass both hardware and software components [1].

The short life cycle and diversification of consumer electronics have placed a premium on getting products to market as quickly as possible. Therefore, it is now more important to design a system that meets the target specifications on time than to design a solution with better performance at a cost of delaying the introduction of a product to the marketplace.

Platform-based design (PBD) is the best-validated industrial approach for achieving high reuse in SoC design and the lowest risk in derivative design. Beyond the reuse of individual IP blocks, PBD reuses complex architectures of hardware and software components organized for a specific application [2]. PBD can decrease the overall time-to-market for the first products and expand the considerably early-delivering opportunities of derivative products.

PBD is a hierarchical design methodology that starts at the system level. PBD achieves its high productivity through extensive, planned design reuse. Productivity is increased by using predictable, pre-verified blocks that have standardized interfaces. The better planned the design re-use, the less changes are made to the functional blocks [3], [4].

Several platform types have emerged nowadays as a result of the evolution of platform-based design. Table 1 summarizes

four types of platforms [5]. Note, however, that the boundaries between these types can blur as providers expand their reach. In this paper, we focus on the processor-centric and communication-centric platforms that require adding specific hardware elements to model each of the applications using them.

Table 1. Platform types.

Platform type	Example
Full-application platforms	- Nexperia: Philips Semiconductors - Open Multimedia Applications Platform (OMAP): TI
Processor-centric platforms	- Micropack: ARM
Communication-centric platforms	- uNetwork: Sonics - AMBA bus architecture: ARM
Fully programmable platforms	- Virtex-II Pro: Xilinx

Figure 1 gives a general platform architecture for processor-centric and communication-centric platforms. It has two master modules, a processor and direct memory access controller (DMAC), and three slave modules (shared memory and hardware modules) connected via the communication network. The processor performs software functions, initiates hardware modules (HW setup) and controls DMAC (DMA setup) for data transfer between the shared memory and hardware modules. The communication network can be a single-layer or multi-layer on-chip bus, or a packet or circuit switch network.

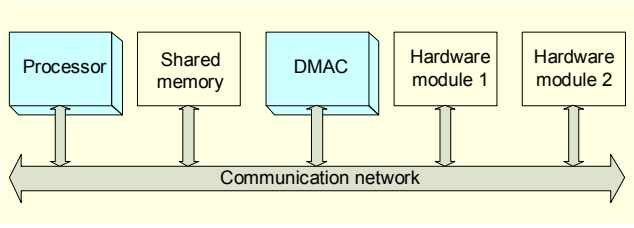

Fig. 1. A general platform architecture for SoCs.

Transformative applications such as JPEG images and MPEG video compression-decompression algorithms should be cost effective, have high performance, and be flexible in order to succeed in the market. As a result, most of them are implemented by an SoC platform that utilizes an off-the-shelf software (SW) processor core and custom hardware (HW) coprocessors. The SW processors reduce the cost of the system and provide flexibility. The custom HW coprocessors implement the computation-intensive components of the application and enhance the performance of the system [6], [7].

HW-SW co-design techniques can be used for designing such SoCs. In HW-SW co-design, the application specification is transformed into communicating HW and SW components, which comprise a platform that exhibits the desired behavior and satisfies the performance constraints. HW-SW co-design consists of two basic design stages: partitioning the application specification into HW and SW components, and scheduling the execution order of these components.

Figure 2 shows a block diagram of functional modules for an MPEG-4 video encoder [8], [9]. The encoder has two-step motion estimation (MEC for coarse, and MEF for fine), motion compensation (MC), motion vector to motion vector difference (MVMVD) calculation, DCT and quantization (DCTQ), inverse quantization and inverse DCT (IQIDCT), reconstruction (REC), header/texture variable length coding (HVLC/TVLC), and stream production (SP) modules. It encodes video frames coming from the "current frame" and outputs the encoded stream through SP. The "reconstructed frame" is generated to exploit temporal redundancy between frames. The encoding procedure is performed based on macro block data of 16 × 16 pixels.

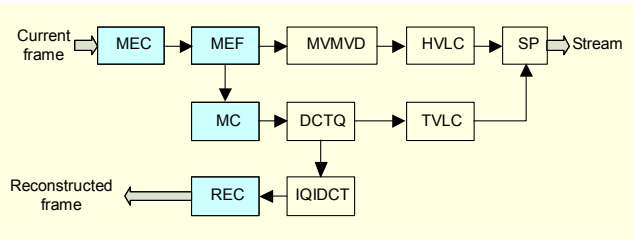

Fig. 2. Block diagram of MPEG4 encoder.

Table 2 shows the execution cycles for major functional HW and SW modules. We used a register-transfer level (RTL) simulator for HW cycles and an ARMulator with ARM7TDMI model for SW cycles. In this table, the 'cycles' column indicates the maximum number of cycles required to process a macro block during the simulation of 300 frames of the CIF-size (352 × 288 pixels) foreman stream.

To encode fifteen frames of CIF size (22 × 18 MBs) per

Table 2. Cycles for major functional modules.

Task	Cycles	Task	Cycles
MEC (HW)	2,500	MC (HW)	1,250
MEF (HW)	1,250	DCTQ (HW)	1,200
MVMVD (HW)	192	TVLC (HW)	1,300
HVLC (SW)	130	IQIDCT (HW)	1,100
SP (HW)	114	REC (HW)	800

second with 27 MHz, it should process an MB in 4,500 cycles. However, based on Table 2 the longest data path requires about 8,300 cycles at 27 MHz for execution without counting the data transfer cycles between functional modules. To implement this application on a platform as shown in Fig. 1 satisfying the performance specification, we have to implement it in a pipelined architecture.

Although a lot of work has been done for the fine-grained synchronous pipeline design, little has been done for a coarse-grained asynchronous pipeline design. More detailed descriptions of previous works on coarse grained and fine grained pipeline designs can be found in [6]. For efficient implementation of the pipelined architecture and architecture exploration, we developed a pipelined scheduling technique.

In this paper, we developed a pipelined scheduling technique of hardware and software modules for platform-based SoC design. Then, we applied it to an MPEG4 video encoder application for performance evaluation and architecture exploration.

II. Pipelined Hardware and Software Scheduling

Transformative applications are dominated by dataflow operations with few control-flow operations. Also, they can be easily broken down into distinct functional tasks at a coarse level of granularity. Each task is computation-intensive and internally strongly interconnected, having a sparse external communication. Therefore, transformative applications can be specified by a data dependency-based task-graph format. Note that these applications are iterative in nature and execute repeatedly over different sets of input data. Hence, they are good candidates for pipelined designs.

1. Platform Architecture

We implement the application on an SoC platform that consists of one single SW processor, one shared memory, one DMAC, and several dedicated HW modules, as shown in Fig. 3. The SW processor is a uniprocessing system and has a local memory for SW execution. Each HW module has its own buffer memory for efficient pipelined operation. HW modules support the concurrent execution of multiple HW tasks. The DMAC is controlled by the SW processor and controls the data transfer between the shared memory and HW buffer memories. The shared memory and SW local memory are single port memories. HW and SW tasks communicate with each other through the shared bus. We consider single-layer and multi-layer shared buses as the communication network in this paper.

2. Modeling Task Graphs and Resource-Conflict Graphs

A given application can be specified as a directed acyclic graph G(V, E), where V is the set of tasks with the execution cycles and E is the set of dependency arcs. Major tasks are functional HW and SW tasks. For bus-based platforms, data transfers controlled by DMAC (DMA transfer), HW setup, and DMA setup can also be modeled as tasks. This will give the scheduler further flexibility to improve the performance of the scheduling result.

Execution cycles of tasks can be estimated by simulation, but it cannot cover all the input data. For SW tasks, computation cycles can be estimated from a complexity analysis of the algorithm. Because HWsetup or DMAsetup tasks performed by the SW processor consist of a register setting and calculation of the register values, their computation cycles can be computed by the number of registers and bus characteristics. DMA transfer cycles can be estimated with the number of data to be transferred and the specifications of the DMAC and memories. Table 3 summarizes the task types according to the usage of platform resources. Since HW modules support concurrent operations, they can be performed any time when all the registers are set by the HW setup. All the task types that use a common resource cannot be performed at the same time. Any tasks that have checks in common in a column cannot be performed at the same time. For example, SW tasks and DMA setup schedules cannot be overlapped even though they are assigned different pipelines.

These relations of task types can be represented as a resource-conflict graph C(T, R), where T is a set of vertices representing task types and R is a set of edges representing a resource conflict. Figure 3 shows a resource-conflict graph of Table 3. In this graph, tasks which have an edge between them cannot share the scheduling time.

Table 3. Task types for a bus-based platform.

Task type	Processor	Bus	DMAC
SW	X		
HW setup	X	X	
DMA setup	X	X	X
DMA transfer		X	X

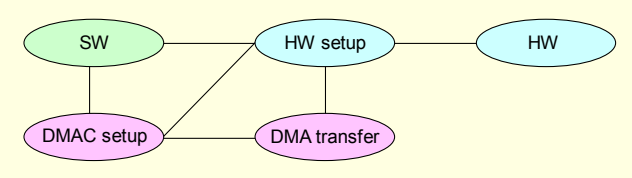

Fig. 3. Resource-conflict graph for Table 3.

3. Problem Definition

Given an application specified as a task graph G(V, E) and resource-conflict graph C(T, R) with a pipeline initiation interval as the performance constraint, find a feasible pipelined schedule and the minimum number of pipelines for executing the task graph.

The pipeline initiation interval is the time difference between the start of two successive iterations of the steady state of the pipeline. Usually, this value is calculated from the specification of the application.

4. Pipelined Scheduling Algorithm

Since resource constrained scheduling is a non-polynomial (NP) complete problem, pipelined scheduling is also NP complete [10]. To achieve optimal solutions of the pipelined scheduling problem in polynomial time, we developed a pipelined scheduling algorithm as shown in Fig. 4 by modifying a list scheduling algorithm. In this figure, 'head' and 'tail' are virtual start and end modules with 0 execution cycles.

```
Pipelined_Scheduling (G(V, E) and C(T, R))
{
   Set start cycles and pipeline number of all modules to 0;
   Calculate_Slack(head, tail);
   Initialize(queue);
   Add_Candidates(queue, source);
   while ((m = Pop(queue) )!= NULL) {
         Find_Schedule(m);
         Set_Schedule(m);
         Calculate_Slack(m, tail);
         Add_Candidates(queue, m);
   }
}
```

Fig. 4. Modified list scheduling algorithm.

Calculate_Slack(**m**) calculates the slacks of all the successor vertices by using as-soon-as-possible scheduling (ASAP) and as-late-as-possible scheduling (ALAP). The slack of **m** is defined as the difference between the scheduling results of ASAP and ALAP.

Add_Candidates(queue, **m**) adds candidate vertices to the queue. Candidate vertices are vertices whose predecessor vertices are all scheduled. When it adds a candidate, it sorts the candidate vertices in descending order of priority. The priority is calculated from a combination of slack, task type, and user-defined priority.

Pop(queue) returns the first vertex from the queue. It has the most priority among the candidates in the queue.

Find_Schedule(**m**) finds a start cycle of **m** such that no resource-conflict violation occurs. Each **m** has three types of information for its scheduling:

1. start cycle: absolute start cycle of scheduling
2. pipeline cycle = (start cycle) % (initiation interval)
3. pipeline number = (start cycle) / (initiation interval)

Set_Schedule(**m**) marks the scheduled information of its resource type using its "pipeline cycle" and "execution cycle" so that scheduling other modules may not generate resource conflicts.

This scheduling technique is flexible in that the scheduling results can be controlled by giving a user-defined priority of tasks and pre-scheduling of some tasks with Set_Schedule(**m**).

III. Experimental Results

We used the pipelined hardware and software scheduling technique to the application given in Fig. 2. First, we scheduled the application for a single-layer 16-bit bus-based platform as shown in Fig. 5. In this case, the resource conflict-graph in Fig. 3 can be used.

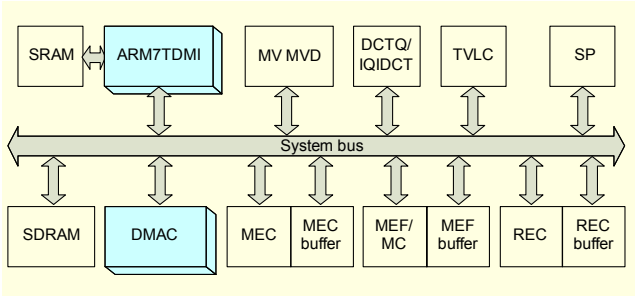

Fig. 5. Single-layer bus architecture.

Figure 6 shows a scheduling result for the single-layer bus-based architecture in Fig. 5. The scheduling result includes hardware modules (HW), software modules (SW), DMA transfer (DMA), and HW/DMA setup (FW). SW modules are header variable length coding (HVLC), intra refresh (IR) decision, rate control operations (PreRC and PostRC), pre-calculations for DMA transfers, and post processing for HW modules. FW modules are named with HW modules or DMA transfers followed by "Init." MEC has two buffers named SWC0 and SWC1. Also, MEF/MC has two buffers named SWF0 and SWF1, and SWF1 has three regions for luminance (Y) and chrominance components (U/V). In this case, the bus usage is about 75%.

Then, we explored the platform architecture to improve the performance by using the developed scheduling technique. Because the bus usage is very high, we tried two variations of the architecture: bus-width expansion and bus partitioning.

Bus-width expansion can reduce FW (HW setup and DMA setup) cycles and DMA transfer cycles. FW cycles can be

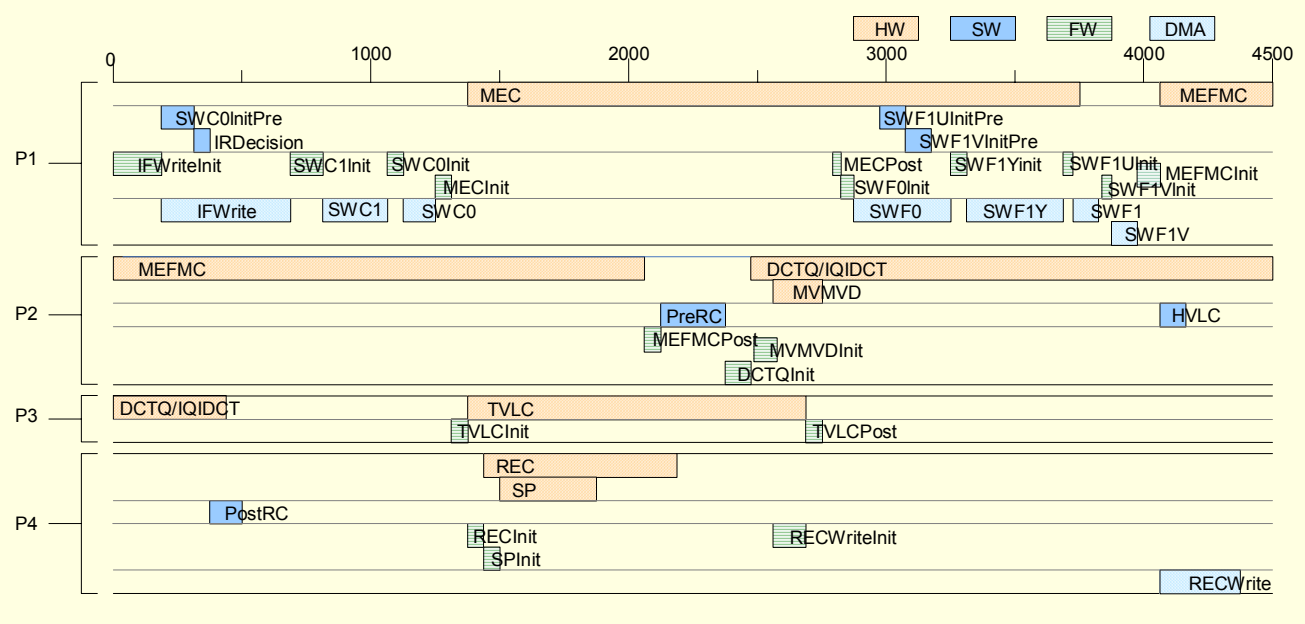

Fig. 6. A scheduling result for the architecture shown in Fig. 5.

reduced as much as the bus-width expands. However, DMA transfer cycles are dependent on the SDRAM features and DMAC characteristics. By analyzing the two characteristics, we obtained the reduction factor of DMA transfer cycles. In our case, it is 0.67 for doubling the bus-width.

By analyzing the data transfer within the bus system, we partitioned the bus into two buses. One is to control the HW modules and DMAC and the other is to transfer data between HW modules and SDRAM.

Figure 7 shows a dual-layer bus-based platform, which is implemented by partitioning the shared bus given in Fig. 5. In this case, the resource-conflict graph should be slightly modified because the DMA transfer and HW setup can be performed concurrently.

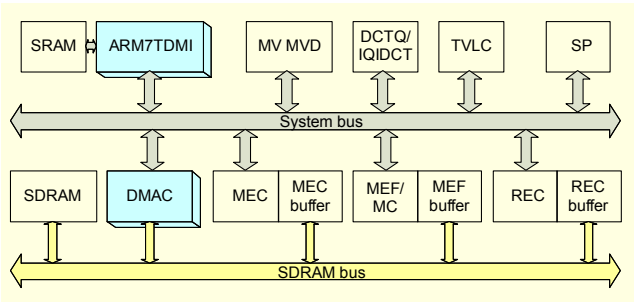

Fig. 7. Dual-layer bus architecture.

Table 4 summarizes the scheduling results for the variable bus architectures. When four pipelines are used, we could improve the frame rate performance by 45% for the 32-bit dual-layer architecture compared to the 16-bit single-layer architecture. We achieved the best performance with seven pipelines for the 32-bit single-layer architecture and six pipelines for the 32-bit dual-layer architecture. The 32-bit dual-layer architecture with six pipelines has a 118% better performance than the 16-bit single-layer architecture with four pipelines and can process over 30 frames per second. Note that when the number of pipelines increases, more buffers will be required for the boundaries of the pipelines, which will increase the area. As a rule-of-thumb, a 6-pipeline architecture may require 50% more buffers compared to a 4-pipeline architecture. Also, note that if the pipeline cycle is less than the HW module cycles, those modules should be modified to support multi-pipeline processing.

Table 4. Scheduling results for various architecture.

Bus width	Bus layer	Pipelines	MB cycles	Frame rate
16-bit	Single layer	4	4,500	15.2 (100%)
16-bit	Dual layer	4	4,150	16.4 (108%)
32-bit	Single layer	4	3,410	20.0 (132%)
32-bit	Dual layer	4	3,090	22.1 (145%)
32-bit	Single layer	7	2,290	29.8 (196%)
32-bit	Dual layer	6	2,060	33.1 (218%)

IV. Conclusions

In this paper, we described a pipelined scheduling of

hardware and software modules for platform-based SoC designs. We applied it to the architecture exploration of platforms for a performance analysis. We could achieve a 118% performance improvement in the frame rate by exploring various architectures. The techniques used in this paper can be applied to a decoder, codec, or other multimedia processing applications such as JPEG or H.264 codec. The scheduling results can also be used for firmware coding of embedded processors.

References

[1] Grant Martin and Henry Chang, *Winning the SoC Revolution: Experiences in Real Design*, Kluwer Academic Publishers, 2003.

[2] G. Martin, "The Reuse of Complex Architectures, Guest Editor's Introduction," *IEEE Design & Test of Computers*, Nov.-Dec. 2002.

[3] H. Chang, Ll Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SoC Revolution*, Kluwer Academic Publishers, 1999.

[4] W. Kim, S. Kim, Y. Bae, S. Jun, Y. Park, and H. Cho, "A Platform-Based SoC Design of 32-bit Smart Card," *ETRI J.*, vol. 25, no. 6, Dec. 2003, pp. 510-516.

[5] G. Martin and F. Schirrmeister, "A Design Chain for Embedded Systems," *IEEE Computer*, vol. 35, issue. 3, March 2002, pp. 100-103.

[6] K. S. Chatha and R. Vemuri, "Hardware-Software Partitioning and Pipelined Scheduling of Transformative Applications," *IEEE Trans. VLSI Systems*, vol. 10, no. 3, June 2002, pp. 193-208.

[7] S. Lee, "Pipelined Macroblock Processing to Reduce Internal Buffer Size of Motion Estimation in Multimedia SoCs," *ETRI J.*, vol. 25, no. 5, Oct. 2003, pp. 297-304.

[8] S.-M. Kim, J.-H. Park, S.-M. Park, B.-T. Koo, K.-S. Shin, K.-Bum. Sun, I.-G. Kim, N.-W. Eum, and K.-S. Kim, "Hardware-Software Implementation of MPEG-4 Video Codec," *ETRI J.*, vol. 25, no. 6, Dec. 2003, pp. 489-502.

[9] ISO/IEC 14496-2, *Information Technology – Coding of Audio-Visual Objects – Part 2: Visual*, 1999.

[10] K. Melhorn, *Graph Algorithms and NP-Completeness*, New York, Springer-Verlag, 1977.

Wonjong Kim received the BS degree in electronics engineering from Chonnam National University in 1989. He received the MS and PhD degrees in electronics engineering from Hanyang University in 1992 and 1999. He joined Electronics and Telecommunications Research Institute (ETRI) in 2000 as a Senior Member. His research interests include CAD for VLSI, SoC design methodology, and multimedia SoC design.

June-Young Chang received the BS degree in computer science from Chonnam National University in Gwangju, Korea, in 1985, the MS degree in computer science from Chungang University in Seoul, Korea, in 1987, and the PhD degree in computer science from Chonnam National University in 1996. He joined ETRI in 1999 in the area of electronic design automation responsible for developing logic synthesis tools. His current research interests include SoC platform design for multimedia applications and SoC design methodology.

Hanjin Cho was born in Seoul, Korea on July 8, 1960. He received the BS degree in electronic engineering from Hanyang University in 1982. He received the MS degree and PhD degrees in electrical engineering from New Jersey Institute of Technology in 1987, and the University of Florida in 1992. He joined ETRI in 1992, where he currently works in SoC design methodology development and wireless multimedia SoC design as a project manager.