Performance Evaluation of Inter-Processor Communication for an Embedded Heterogeneous Multi-Core Processor[*]

SHIAO-LI TSAO AND SUNG-YUAN LEE
Department of Computer Science
National Chiao Tung University
Hsinchu, 300 Taiwan

Embedded systems often use a heterogeneous multi-core processor to improve performance and energy efficiency. This multi-core processor is composed of a general purpose processor (GPP), which manages the program flow and I/O, and a digital signal processor (DSP), which processes mass data. An inter-processor communication (IPC) mechanism is thus required to exchange data between a GPP and a DSP. This paper uses comprehensive experiments to evaluate the IPC performance of an embedded heterogeneous multi-core processor under different design strategies. We further develop the IPC performance model and suggest dynamic adjustment of IPC strategies under environmental parameters and system resource constraints. Based on the results and findings, we improve the IPC performance of a voice over IP (VoIP) phone. Experimental results demonstrate that the GPP workload decreases significantly by 35% without sacrificing the functionalities and voice quality of the VoIP system. Moreover, we apply the concept of dynamic adjustment of IPC strategies to an embedded media gateway. The simulation results demonstrate that the dynamic IPC strategy can considerably improve the system performance of the media gateway compared with the static IPC design approach.

Keywords: multi-core, inter-processor communication, performance evaluation, embedded system, heterogeneous multi-core processor

1. INTRODUCTION

Embedded systems handle both I/O and computation jobs. Using a general purpose processor (GPP), which provides better I/O controls, or a digital signal processor (DSP), which offers rich computational resources, to handle both I/O and computational jobs is usually inefficient [1]. To improve energy and performance efficiency, many embedded systems use a heterogeneous multi-core processor composed of GPPs and DSPs [2]. A heterogeneous multi-core processor requires an inter-processor communication (IPC) mechanism for exchanging data and control messages between the GPPs and DSPs. According to previous studies, this IPC mechanism is critical, especially for embedded systems involving frequent interactions between processors [3-5].

A number of studies have evaluated the IPC performance of a heterogeneous multi-core processor. Gorgonio *et al.* [3] and Chiu *et al.* [4] examined IPC overhead and performance of a task running on a GPP or DSP. Their studies assist the designers to map the tasks to the GPP and DSP efficiently. For example, it is not always efficient to assign computation-intensive tasks to the DSP through IPC, as this introduces a considerable overhead. Luiz *et al.* [5] further proposed a formal model based on timed automata for

Received June 17, 2010; revised December 21, 2010 & March 23 & May 18, 2011; accepted July 13, 2011.
Communicated by Tei-Wei Kuo.
[*] This work was supported by MediaTek Inc. and the National Science Council of Taiwan for financially under Contracts No. 100-2219-E-009-022, 100-2220-E-009-038, NSC 99-2220-E-009-045-, NSC 99-3113-P-006-004, and NSC 99-2915-I-009-064.

the IPC of a heterogeneous multi-core processor. Their model helps people understand the details of the IPC mechanism. Other researchers have proposed several hardware and software improvements for the IPC mechanism. Chen *et al.* [6] proposed a new bus architecture to speed up IPC between GPPs and DSPs. For the software improvements, Kluter *et al.* [7] suggested using scratchpad memory, *i.e.*, internal memory, instead of external memory as the shared memory for IPC data exchanges. This approach significantly improves IPC performance, especially for streaming applications which involve frequent IPCs. Brisolara *et al.* [8] presented a method for aggregating several IPC requests into a single request, reducing the number of IPCs between the GPP and DSP. This technique also eliminates certain IPC overheads.

Unfortunately, previous studies fail to consider combining the design factors above, or compare IPC performance under different design strategies. Therefore, this paper presents a test-bed and evaluates the IPC mechanism through comprehensive experiments. Based on results and findings, we further develop the IPC performance model and suggest dynamic adjustment of IPC strategies under environmental parameters and system resource constraints. The main contributions of this paper are (1) to establish a precise evaluation environment for IPC procedures and compare the performance of different IPC design strategies and their combinations, and (2) to develop the IPC performance model and suggest dynamic adjustment of IPC strategies under environmental parameters and system resource constraints. The study provides designers with a reference for choosing appropriate IPC designs for embedded systems.

The rest of the paper is organized as follows. Section 2 describes the IPC mechanism and presents different IPC design strategies. Section 3 introduces the performance evaluation test-bed, methodologies, and results. Section 4 suggests dynamic adjustment of IPC strategies based on environmental parameters and system resource constraints. Based on the experimental results and findings, we apply appropriate IPC designs to a VoIP phone. Moreover, we apply the concept of dynamic adjustment of IPC strategies to an embedded media gateway. Section 5 presents and discusses these results. Finally, section 6 offers conclusions.

2. INTER-PROCESSOR COMMUNICATION MECHANISM AND DESIGN PARAMETERS

2.1 IPC Mechanism

The generic procedures of the IPC mechanism from the GPP to DSP include five steps illustrated in steps 1 to 5 of Fig. 1. Before the IPC process starts, the GPP program first downloads the DSP programs to the DSP internal memory so that they can be executed when the GPP program invokes the DSP functions. Once the GPP program has a job to assign to the DSP, the GPP program makes a DSP function call in the DSP library. The DSP library then initiates an IPC, which transfers the control message and data to the DSP.

During the IPC process, the GPP first copies the data to the shared memory, which can be accessed by both the DSP and GPP in step 1. Then, in step 2, the GPP prepares a control message that specifies the request information, such as the address, length of the

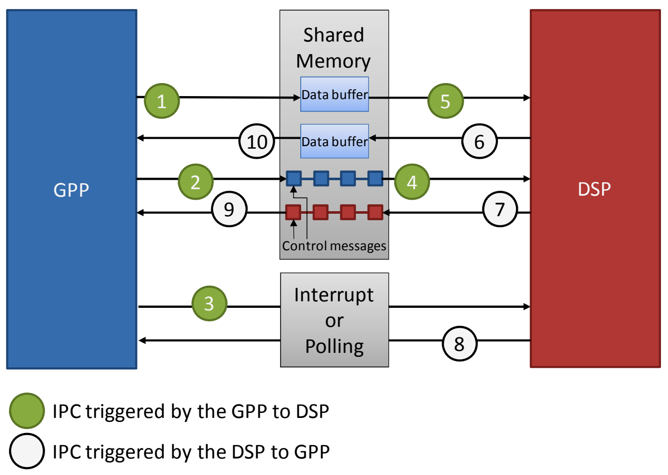


Fig. 1. Generic procedures of IPC triggered by the GPP to DSP, and the DSP to GPP.

data, and the DSP function invoked by the GPP program, and stores this control message in the shared memory. The control messages are structured as a list to allow the GPP program to continuously send new requests to the DSP, and the DSP can process the DSP function requests one by one. Both the GPP and DSP have to know the format and address of the control message in the shared memory before the IPC process begins. In step 3, the GPP uses an interrupt or other mechanism to notify the DSP that the GPP program has a request for the DSP. Platforms such as Texas Instruments (TI) DaVinci and The Industrial Technology Research Institute (ITRI) PAC (Parallel Architecture Core) [14] provide the mailbox hardware. When the GPP writes data to the mailbox, an interrupt signal is automatically generated and sent to the DSP. Therefore, steps 2 and 3, steps 7 and 8 might be combined. When the DSP is notified, it reads the control messages from the list in step 4. In step 5, the DSP finds the data to be processed based on the control message and starts to process the request. Steps 1 to 5 illustrates the generic procedures of an IPC triggered by the GPP.

After the DSP finishes the job, the DSP can initiate another IPC process and reports the results and/or status to the GPP. The DSP copies the processed data to the shared memory in step 6 so that the GPP can access the results. Like the procedure of the GPP notifying the DSP, the DSP must prepare a control message indicating the address and length of the processed data in the shared memory in step 7. In step 8, the DSP uses an interrupt or other means to notify the GPP that the request has been processed. After notification, the GPP reads the response message in step 9. Finally, the GPP obtains the processed data in step 10. Steps 6 to 10 illustrates the generic procedures of an IPC from the DSP to GPP. The IPC introduces extra GPP and DSP workload to handle memory copies, request messages, response messages, and interrupts. The following subsection discusses the IPC design strategies that may influence IPC performance.

2.2 IPC Design Strategies

The first IPC design consideration is the granularity of the IPC request, *i.e.* the size of the data block to be passed from the GPP to the DSP. A possible design choice here is to merge several DSP function calls and their associated data blocks into one request, invoking only one IPC [8]. This approach reduces both the number of IPCs and the IPC overhead. However, in this case, more shared memory is required to store the data blocks, and the response time of the DSP function call may increase since function calls are deferred, merged, and sent to the DSP together.

Another design strategy is to utilize different types of shared memory for the IPC data exchanges. A common approach is to use the external SDRAM attached to the system bus as shared memory. Another possible choice is to allow the GPP internal memory to be accessed by the DSP, or vice versa. Using the internal memory as shared memory can significantly speed up memory access for both the GPP and DSP [7], speeding up IPC as a result. However, internal memory is much smaller and more expensive than external memory, and internal memory should be carefully managed to maximize cost-efficiency of an embedded system.

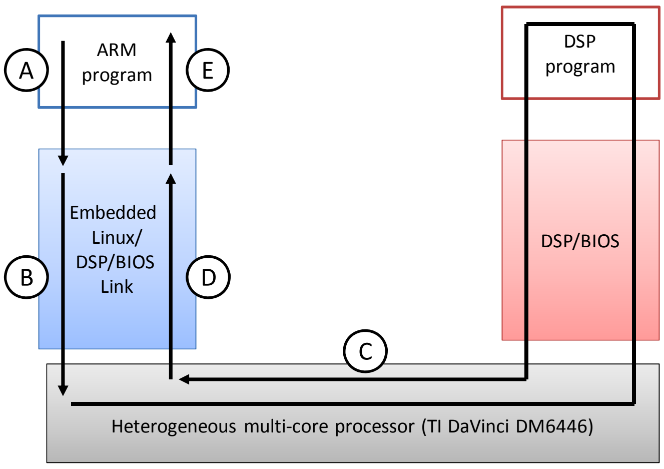
In another design, the GPP and DSP use an interrupt or polling mechanism to notify the other processor when there is an IPC request or response. A common approach is to use an interrupt for this notification, but the interrupt involves an interrupt handling procedure and introduces a considerable overhead. Another approach is to use a polling mechanism for this notification. The polling mechanism simply sets and resets a flag in the shared memory when there is a request or response. For example, after steps 1 and 2, the GPP sets a flag in a particular address in the shared memory. The DSP checks the flag by reading the memory address to see if there is a new request from the GPP. If the flag is set, the DSP reads the request message and processes the data. Reading an internal memory address produces much less overhead than an interrupt service routine. However, knowing when the DSP and GPP should poll the flag is a challenge of the polling approach. If the DSP or GPP polls the flag frequently, the overhead increases. On the other hand, the function call latency increases if the GPP polls the flag infrequently. Therefore, the timing of polling IPC requests or responses should be carefully managed.

3. EVALUATION OF THE IPC MECHANISM

3.1 Evaluation Test-Bed and Methodologies

To examine IPC performance under different design parameters, this study first presents an evaluation test-bed. A Texas Instruments (TI) DaVinci DM6446 [9], which has an ARM926-EJS processor and TI C64 DSP, was used for the experiments in this study. Embedded Linux and DSP/BIOS were run on the ARM processor and DSP as the operating systems. ARM programs on Embedded Linux call DSP functions through the DSP library. The DSP library further utilizes the DSP/BIOS Link [10], which implements IPC, to transfer requests from the ARM processor to the DSP. The DSP/BIOS Link is a Linux kernel driver that can communicate with the DSP/BIOS to realize IPC.

To evaluate the IPC mechanism, we developed user-space testing programs on the ARM processor and DSP. We also modified the Linux kernel and DSP/BIOS Link to track the IPC procedures and measure the latency for each IPC phase. This study only investigates IPC performance from the ARM processor point of view. This is mainly be-


ARM program
A
E
Embedded Linux/ DSP/BIOS Link
B
D
DSP program
DSP/BIOS
C
Heterogeneous multi-core processor (TI DaVinci DM6446)
Fig. 2. Measurements of IPC phases.

cause that the DSP usually serves as a slave co-processor that is dedicated to processing the data. The ARM processor, on the other hand, usually handles multiple tasks. Therefore, designers are more concerned about the IPC delay and the ARM resources occupied by the IPC.

The IPC procedures carried out by a GPP include the manipulation of requested and response data in the memory, notification of an IPC request to the DSP and handling procedures of an IPC response from the DSP. If we consider the DSP workload, the IPC procedures on a DSP are the handling procedures of an IPC request, manipulation of the requested and response data in the memory, and notification to the GPP for the task complete. The same evaluation environment presented in this paper can be also applied to the DSP and can examine the DSP workload for handling IPC requests.

To examine the detail performance of the IPC, we divided the IPC process into the five major phases shown in Fig. 2 and measured the latency of each phase. In phase (A), the ARM program copies the data block to be processed from a user space memory to the kernel space memory. The kernel space memory is directly mapped to the shared memory space that both the ARM processor and DSP can access to avoid additional memory copies in the kernel space. Phase (A) corresponds to step 1 shown in Fig. 1. Phase (B) prepares the control message and sends an interrupt to the DSP. Phase (B) corresponds to steps 2 and 3 in Fig. 1. Phase (C) includes all the DSP procedures during an IPC, and corresponds to steps 4 to 8 in Fig. 1. Phase (D) is the interrupt handling procedure for receiving the results, and corresponds to step 9 of Fig. 1. Finally, phase (E), *i.e.* step 10 of Fig. 1, involves a memory copy of the processed data from the kernel space to the user space.

To precisely measure the latency of each IPC phase and minimize the instrument overhead, we implemented our own timer driver instead of using the standard `gettimeofday()` library in the Linux. The standard `gettimeofday()` provides microsecond (μs)-level accuracy, but our own driver, which utilizes the TI DaVinci 64-bit general-purpose timer, offers $1/27\mu s$ precision. This approach also significantly reduces the overhead of retrieving the timer. Experimental results shows that the user programs and kernel programs take 11.85μs and 5.87μs, respectively, to retrieve the current time using the standard `gettimeofday()` library. On the other hand, this process only takes 3.63μs and 0.44μs using our driver. Since we inserted timer retrieval codes in the testing program and kernel to gather the latency of a specific procedure, the instrument overheads must be deducted from the measurement results.

3.2 Experimental Results

The first experiment considers different granularities of data blocks passed from the ARM processor to the DSP. Since we are only concerned about IPC performance, the DSP testing program in this experiment does not perform any tasks, and simply sends the same data block back to the ARM processor. In this experiment, an interrupt mechanism notifies the ARM processor and DSP when there is an IPC request or response. Further, external SDRAM served as the shared memory for exchanging IPC data. Fig. 3 illustrates the latency of each IPC phase for different block sizes. This figure shows that the IPC spends a lot of time in phases (A) and (E), which perform data copies between the user space memory and kernel space memory. The latency of phases (A) and (E) depends on the block size. Phases (B) and (D) both process control messages and handle interrupt notifications. Since the DSP program does not perform any task in this experiment, phase (C) consumes only 70μs – 80μs. These experimental results indicate that if the block size is small, such as 128bytes, the control message process and notification overheads, *i.e.*, phases (B) and (D), are significant. The overhead for notifying processors through interrupts is similar to the overhead of the memory copy for small data blocks. Therefore, merging multiple small-block requests and performing fewer IPCs could improve the overall IPC performance. For example, it takes 34ms, 10ms, and 8ms to exchange a total of 32K between the ARM processor and DSP through thirty-two 1Kbyes IPCs, two 16Kbytes IPCs, and one 32Kbytes IPC, respectively.

The second experiment considers different types of shared memory, *e.g.*, internal memory or external memory, for exchanging IPC data. Specifically, this experiment compares the IPC performance using the ARM internal SRAM and the external SDRAM. The latency of the memory accesses is the primary factor influencing the data copies in phases

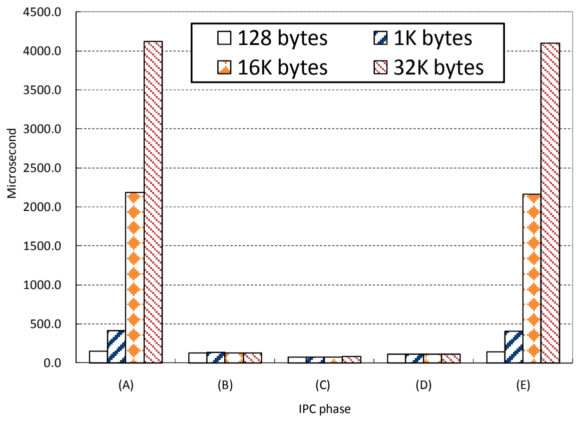


Fig. 3. Latency of each IPC phase.

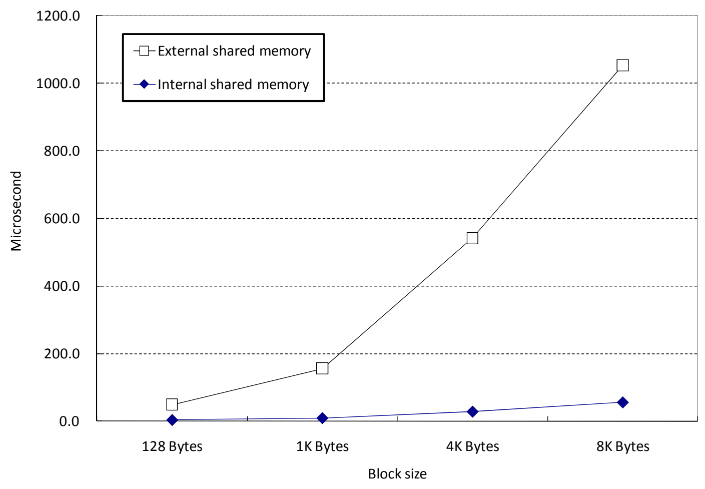


Fig. 4. Comparison of IPC data copies through internal and external shared memory.

(A) and (E). This experiment only measures the delay in the kernel space caused by copying different sizes of data blocks from the user space to the internal or external shared memory. The results illustrated in Fig. 4 are less than those shown in Fig. 3 since the delay shown in Fig. 3 includes the latency for executing the user library, system call, and kernel code to process the memory copy. Obviously, memory copies through the internal memory are much faster than those through the external memory. For example, the latency to perform an 8Kbytes IPC data copy using the ARM internal SRAM is 94% less than that using the external SDRAM. This experiment indicates that using internal memory as the shared memory considerably improves IPC performance.

The interrupt notification mechanism introduces serious overheads for small-block and frequent IPCs. Therefore, the next experiment uses the polling mechanism to notify the ARM processor instead of using an interrupt approach. Experimental results show that the notification latencies of a DSP function call introduced by the interrupt and polling mechanisms are $76.3\mu s$ and $2.3\mu s$, respectively. The polling notification approach significantly reduces latency by 95% compared with the interrupt notification approach. For embedded applications that generate frequent and periodical IPCs with small block sizes, the polling notification approach is a better design choice, and can minimize IPC overhead.

4. RUN-TIME ADJUSTMENT OF IPC STRATEGIES

For an embedded system with multiple tasks, these tasks, called IPC tasks, may generate IPC requests simultaneously and compete the GPP, DSP and memory resources with each other. Moreover, the embedded system may suspend and resume some of the tasks and functionalities at run-time due to environmental changes. Therefore, IPC strategies and parameters should be adjusted dynamically to accommodate different IPC requests under the system resource constraints. We use an embedded video conference system as an example. A multi-party video conference system may have different number of concurrent video sessions and may change the settings of audio and video codecs dynamically according to the available network bandwidth. The changes of the number of concurrent video sessions and codec settings such as audio frame length, video frame rate, and video frame size, influence the decisions of IPC strategies and parameters. One possible scenario is that a video conference system begins with an audio-only communication session due to insufficient network bandwidth. The audio processing task generates IPC requests with small amount of audio data so that it can use the internal memory for its IPC requests. While the network throughput improves, the video conference system enables another two video communication sessions. However, the internal memory may not be sufficient to accommodate the data generated by the audio and video processing tasks if all tasks use the internal memory for the IPC requests. Therefore, the video conference system may rearrange the IPC strategies and parameters such as the granularity of IPC requests, IPC notification mechanisms, and use of the internal or external memory for each audio and video processing task to minimize the GPP workload. We thus propose below performance models for an embedded system so that the system can dynamically adjust the IPC strategies under environmental parameters and system resource constraints. In the below model, we only consider periodical IPC tasks. Also, we develop a simplified model based on average cases. To precisely evaluate the GPP resources, delay and memory requirements for IPC, the probability models and/or queuing theory are required. The analytic model of the IPC overheads and the comparison of the model with experimental results are out of scope of this paper and will be our future work.

In this paper, we only consider periodical IPC requests. For an embedded system with k tasks which generate periodical IPC requests, denoted as IPC_i, and IPC_i requests occur every PI_i seconds. For each IPC request, the GPP sends a block with the size of B_i^I bytes to the DSP. The DSP processes it and returns the result at B_i^O bytes. An embedded system may choose the type of the IPC shared memory, the IPC notification mechanism for each IPC task, and the granularity of an IPC request, which also influences the frequency of IPC requests. We define $M_i = 1$ if the external memory is used for exchanging data between the GPP and DSP. Otherwise, $M_i = 0$ implies that the internal memory is used for exchanging data between the GPP and DSP. R_E and R_I denote the access rates of the external and internal memory, respectively. $I_i = 1$ denotes that the interrupt-based mechanism is used to notify the GPP when the DSP finishes the IPC request. Otherwise, $I_i = 0$ implies that the polling-based mechanism is used for the IPC notification, and the GPP performs the polling procedure every PP_i seconds. Although, using the internal memory as the shared memory significantly improve IPC performance, the internal memory is a limited resource and shared by IPC requests. Moreover, using polling-based notification achieves a better performance than interrupt-based notification. Polling-based notification requires more memory to temporarily buffer the data between the GPP and DSP. We should carefully decide the granularities of IPC requests, IPC notification mechanisms, and allocate the internal or external memory to IPC requests under the system resource constraints. To evaluate the GPP resources that IPC_i occupies, we define

$$U_i = \frac{C_{IPC} + I_i \times C_I + (I - I_i) \times \dfrac{PI_i}{PP_i} \times C_P + \dfrac{B_i^I + B_i^O}{M_i \times R_E + (1 - M_i) \times R_I}}{PI_i}$$

as the overhead for processing IPC_i requests. In the above equation, C_I is the overhead for the interrupt-based IPC notification. C_P is the overhead for performing the IPC status polling. C_{IPC} is the overhead for handling an IPC request, and it includes the procedures for initializing and completing an IPC request. The embedded system may have hardware and software requirements which impose constraints for the selection of the IPC strategies. For example, an embedded system has a limited internal memory so that the size of the internal memory that IPC requests can use for exchanging data must not exceed the size of the internal memory. We assume that the DSP uses the same memory area that the GPP stores the requested data to return the results after completing the task. Hence, for an IPC request using interrupt-based notification, it requires $\text{Max}(B_i^I, B_i^O)$ memory to temporarily buffer the requested and returned data. By using polling-based notification, the system has to allocate more memory to the IPC since the IPC requests are continually generated and processed. The polling process clears and returns the shared memory periodically. Therefore, we reserve the maximal buffer at

$$\left(1+\left\lfloor \frac{PP_i}{PI_i} \right\rfloor\right) \times \text{Max}(B_i^I, B_i^O)$$

bytes for each IPC request using polling-based notification. The internal memory that all IPC tasks use must not exceed the total size of the internal memory, defined as B_{intl}. The buffer constraint must be satisfied. That is:

$$\sum_{i=1}^{k} \{M_i \times \text{Max}(B_i^I, B_i^O) + (1-M_i) \times (1+\left\lfloor \frac{PP_i}{PI_i} \right\rfloor) \times \text{Max}(B_i^I, B_i^O)\} \leq B_{intl}. \quad (1)$$

On the other hand, each IPC request may have its own delay constraint. For an IPC request using interrupt-based notification, the maximal latency, say d_i, for an IPC request is the time to process the request. That is:

$$d_i = C_{IPC} + C_I + C_{DSP} + \frac{B_i^I + B_i^O}{M_i \times R_E + (1-M_i) \times R_I}, \text{ if } I_i = 1,$$

where C_{DSP} is the latency that the DSP processes the request. If the system employs polling-based notification and the polling frequency is faster than the IPC request frequency, the maximal latency for an IPC request becomes

$$C_{IPC} + C_P + C_{DSP} + \frac{B_i^I + B_i^O}{M_i \times R_E + (1-M_i) \times R_I}.$$

If the polling frequency is slower than the IPC request frequency, the latency for an IPC request increases to

$$PP_i + C_P + \frac{B_i^O}{M_i \times R_E + (1-M_i) \times R_I}.$$

Therefore, we could model the maximal latency for an IPC request if polling-based notification is applied as:

$$d_i = \text{Max}(PP_i + C_P + \frac{B_i^O}{M_i \times R_E + (1-M_i)\times R_I'} C_{IPC} + C_P + C_{DSP} + \frac{B_i^I + B_i^O}{M_i \times R_E + (1-M_i)\times R_I}),$$
$$\text{if } I_i = 0.$$

The latency constraint for IPC_i, denoted as D_i, should be satisfied. That is:

$$I_i \times d_i + (1 - I_i) \times d_i \leq D_i, \forall i. \tag{2}$$

Based on the models mentioned above, we can dynamically adjust the parameters of the k periodical IPC tasks in order to minimize GPP resources in performing IPC procedures, *i.e.* to minimize $\sum_{i=1}^{k} U_i$, without violating the delay and internal memory constraints. Assume IPC_i has IPC_i^g design choices, *i.e.* IPC_i^g IPC granularities. IPC_i^g IPC granularities imply that an IPC requested block and returned block for IPC_i also have IPC_i^g sizes. For each IPC task, it can use either polling-based notification or interrupt-based notification for its IPC requests. Also, each IPC request can use either the internal or external memory as its IPC shared memory. Therefore, a system can generate a total of $\prod_{i=1}^{k} IPC_i^g \times 2^k \times 2^k$ design combinations. When the environmental parameters change, the system can perform internal memory and delay constraint checks based on Eqs. (1) and (2), evaluate different IPC strategies and parameters for each IPC task, and dynamically adjust the strategies and parameters of every IPC task. We further apply this dynamic adjustment concept to an embedded media gateway and evaluate its performance in section 5.2.

5. IPC IMPROVEMENT FOR TWO CASE STUDIES

5.1 Case Study 1: VoIP Phone

The experiments above show that depending on the characteristics of an embedded system, designers may prefer different IPC strategies. This section applies different IPC strategies to a VoIP phone. The GPP runs a VoIP client and the DSP compresses and decompresses voice packets. The two processors frequently conduct IPC, *e.g.*, every 20 ms for the G.711 voice codec, and the size of the IPC data block is usually small, *e.g.*, 160 bytes for the G.711 voice codec.

The VoIP test-bed in this study consists of a public SIP server which handles VoIP calls and two client nodes which can establish a VoIP communication. One VoIP client runs Linphone [11], an SIP user agent, on a PC, and the other VoIP client uses the TI DaVinci DM6446 evaluation board. We ported and modified Linphone [11] on the embedded evaluation board to evaluate its performance. The two VoIP clients first registered with the public SIP server, and then established a VoIP communication through the SIP server. These experiments evaluated the ARM processor workload of the TI DaVinci board after the VoIP call was established and voice packets were transmitted between the two clients. We used a popular CPU workload monitor tool, called Top, on Linux, to measure the ARM processor workload.

The VoIP client running on the ARM processor periodically calls the DSP function to encode and decode voice frames. For example, if the G.711 codec is used, the VoIP client calls the G.711 library every 20ms for decoding and encoding voice packets. According to our findings, the polling notification approach is preferred for the VoIP phone since the application generates periodical IPC requests with small block sizes. Therefore, we modified the DSP/BIOS Link to support polling notification and compared its performance with the conventional interrupt approach. To implement polling notification on the ARM processor, the ARM processor first prepared the control messages and data blocks in the shared memory. Since the ARM and DSP programs perform their own tasks asynchronously, it is necessary to prepare a number of blocks in the shared memory to allow the ARM program to continually generate requests for the DSP program to process. The blocks in the shared memory are implemented as a circular buffer. The ARM program sends a request to the DSP by setting the flag. After the DSP finishes the task, it resets the flag in the shared memory.

The conventional VoIP application has a periodical software timer which is used to trigger the audio capture, audio process, packetization, and packet transmission. The ARM sends an audio process request to the DSP via IPC periodically. If interrupt-based notification for an IPC request is applied, an additional hardware interrupt and interrupt service handling routine are introduced. On the other hand, if polling-based notification is employed, we check the IPC status flag when the original software timer expires. If the flag is reset, the ARM program understands that the task has finished, and then accesses the results in the shared memory. Therefore, neither an additional hardware interrupt nor an additional software timer is introduced if polling-based notification for an IPC request is applied. We could increase the polling frequency by shortening the interval of the software timer, but the ARM workload increases. Table 2 shows our experimental results by applying different polling frequencies, *i.e.* different software timer intervals. Experimental results show that the workload only increases by 0.5% when the polling frequency is doubled on the ARM processor.

Small IPC block size is another important characteristic of VoIP applications. For example, the G.711 voice frame is 1280 bits, or 160 bytes. The IPC block size becomes smaller when using a low-bit-rate codec, such as G.723. This small data block size makes it affordable to use the internal memory to exchange the IPC data. Thus, we used the internal memory rather than the external memory for the IPC data exchanges.

Table 1 illustrates the ARM processor workload under different IPC strategies. The first design strategy only uses the ARM processor to handle both the VoIP client and the voice codec. In other words, the DSP is not involved in this case. This approach consumes 47% ARM processor's resources. The DSP handles the voice codec in the second design configuration, requiring IPC. In this case, IPC uses external SDRAM as the shared memory and uses interrupt mechanism for notifications. This approach significantly reduces the ARM workload by 50%. The third configuration uses the polling mechanism to replace the interrupt mechanism. The ARM program initiates a 20ms polling timer to periodically check the completion of the DSP task. These experiments indicate that the ARM workload can be further reduced by 8% using polling notification. The fourth configuration uses internal memory as the shared memory. A comparison of design configurations 2 and 4 shows that using internal memory for IPC data exchanges reduces the

Table 1. ARM processor workload under different design configurations.

Design Configurations	ARM Workload (%)
1. ARM	46.53
2. ARM + DSP + Intr + Extl	22.73
3. ARM + DSP + Poll + Extl	20.92
4. ARM + DSP + Intr + Intl	16.37
5. ARM + DSP + Poll + Intl	14.84

Table 2. The ARM processor workload and the maximal latency of the DSP function call under interrupt and polling notification approaches.

	Interrupt	Polling (5ms)	Polling (10ms)	Polling (15ms)	Polling (20ms)
ARM workload (%)	16.71	16.02	15.35	15.18	14.87
Maximal latency of the DSP function calls (ms)	6.83	10.28	10.31	17.35	20.42

ARM processor workload by 28%. For an embedded system with small IPC blocks, using internal memory can significantly improve IPC performance. Finally, the fifth design configuration uses both internal memory and polling technologies. Experimental results indicate that this approach can reduce the ARM processor workload by 35% compared with the second design configuration.

When using the polling notification mechanism, the time that ARM can detect the completeness of the DSP function may increase slightly compared with the interrupt approach. This is because the interrupt notification can interrupt the ARM processor as soon as the DSP function is complete. However, the detection of the DSP function complete on the ARM processor may be delayed for one polling period if the DSP resets the flag just after the ARM program polls the flag.

One concern with using the polling approach is that the VoIP application may have extra packet delay and/or packet delay jitters. Therefore, we measured the packet delay and delay jitters using WireShark [12] in the test-bed. Experimental results show that the polling approach results in additional 6ms to 10ms delays for each voice packet, but does not introduce additional delay jitters. A VoIP phone using the polling and interrupt notification approaches offers the same voice quality. This is mainly because most current VoIP clients maintain a receiving buffer with two to three voice packets, *i.e.*, a 40ms to 60ms jitter buffer, to accommodate network delays and jitter. Therefore, the receiving buffer can easily accommodate IPC notification latencies. The proposed approach of using internal memory and the polling approach can reduce the ARM processor workload by 35% without sacrificing VoIP functionalities or voice quality.

Although the polling mechanism does not influence applications that can tolerate a slight DSP function call delay, some real-time embedded systems may require a fast and guaranteed response time for each DSP task. Therefore, we evaluated the maximal latency of the DSP function calls by applying the polling and interrupt approach using the same VoIP program. Table 2 shows the maximal latency of the DSP function calls and corresponding ARM processor workload for polling intervals of 5ms, 10ms, 15ms, and 20ms, respectively. This table shows that the ARM workload increases with polling timer frequency. On the other hand, the complete of a DSP function call can be detected earlier using a frequent polling timer, which reduces the maximal latency of DSP function calls. Table 2 also shows that when using a 5ms polling interval, the ARM processor workload increases, but the latency of the DSP function calls cannot decrease. This is because the DSP programs spend about 6ms to process the task in this VoIP example. If we poll the flag every 5ms, it may be necessary to poll the flag twice to determine if the task is complete or not. In this case, the maximal latency of the DSP function call becomes 10ms using a 5ms polling interval.

With asynchronous software pipelining design, a stream application such as VoIP can continue generating IPC requests to the DSP, and does not have to wait for the DSP process of IPC requests and IPC responses. The DSP processing latencies of the IPC requests might be hidden. The IPC overhead can be further reduced by exploiting both DLP (data-level parallelism) and TLP (thread-level parallelism). For example, a complicated DSP task may require large data buffer for IPC. The available resources may not be sufficient to accommodate this request if a programmer wants to use the internal memory as the Ping-Pong buffer and apply the asynchronous software pipelining design to hide the IPC overheads. On the other hand, the programmer may carefully adjust the jobs, modify the algorithms, and then may divide the complicated DSP task into a number of small DSP sub-tasks. In that case, the internal memory may become sufficient to accommodate data requests from these small DSP sub-tasks. More efficient IPC mechanisms and asynchronous software pipelining design can thus be applied, and the IPC overheads may be further hidden and/or reduced.

5.2 Case Study 2: Media Gateway

Based on the above experimental results, we can obtain the parameters of the IPC performance models such as C_{IPC}, C_I, C_P, R_E, and R_I for TI DaVinci platform. We apply the dynamic adjustment strategy to a media gateway and simulate the ARM workload for handling different IPC requests. A media gateway is an embedded device and can be used in a VoIP communication system. It decodes, encodes, and/or converts multiple audio and video streams simultaneously. We assume that the media gateway encodes G.711 audio streams at 64Kbps, and H.263 video streams at 10 frames/second, SQCIF (128×96) resolution, and 64Kbps. The conventional media gateway forks an encoding thread when it receives an encoding request of an audio or video session. Here, we assume that the IPC strategy for an audio encoding thread can be different from that for a video encoding thread, but the IPC strategies for all audio encoding threads and for all video encoding threads should be the same, respectively. The IPC strategies for all encoding threads are determined during the system design phase and they are fixed no matter how many tasks that the media gateway currently handles. We call this conventional approach as the static IPC design. Therefore, the static approach must consider the worst-case situation, and decides the IPC strategy for an audio and video encoding thread. Different from the static approach, our suggestion is to modify the audio and video encoding thread to dynamically adjust the IPC strategies based on the current running tasks, environmental parameters and system resource constraints. The dynamic adjustment approach first generates all design combinations of IPC strategies for audio and video encoding threads, and then decides the optimal IPC solutions for these threads under system resource constraints. With the dynamic adjustment approach, IPC strategies for every audio

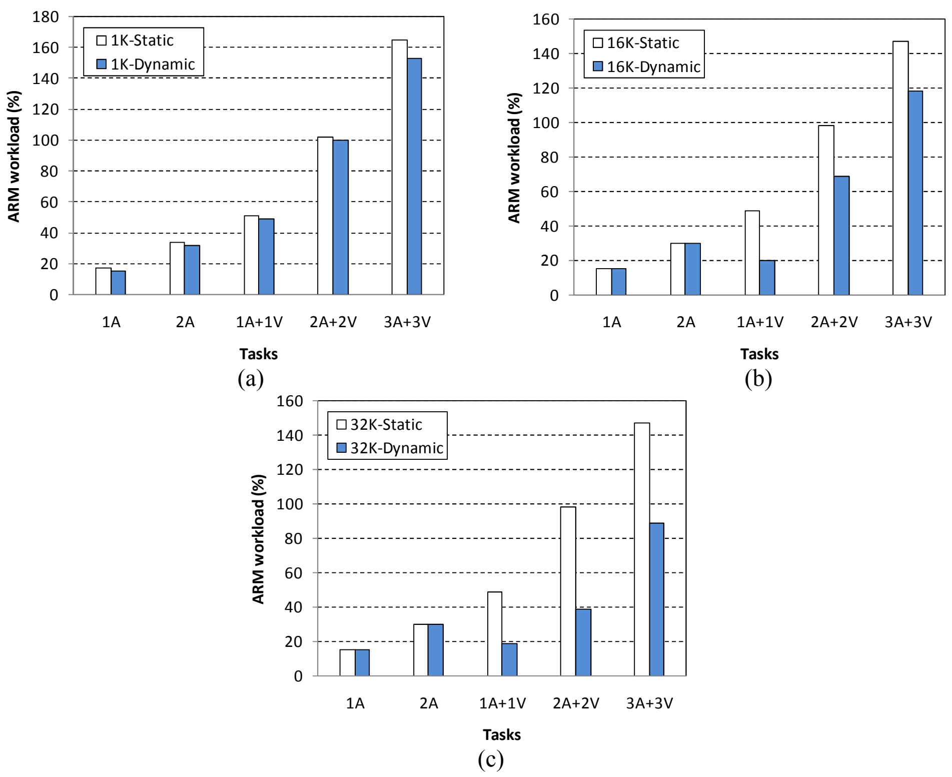


Fig. 5. ARM workload under different running tasks and internal memory budgets.

and video encoding thread can be different, and the audio and video encoding thread can change their IPC strategies during run-time.

We simulate the ARM workload under different running tasks and system resource constraints. Fig. 5 illustrates the simulation results under three system configurations. Figs. 5 (a)-(c) consider the media gateway with 1K, 16K and 32K internal memory budget, respectively. The ARM workload is the overhead for handling IPC requests. We target the system to support up to three audio and three video encoding sessions. 1A, 2A, 1A + 1V, 2A + 2V, and 3A + 3V mean that the media gateway currently handles one audio encoding, two audio encoding, one audio and one video encoding, two audio and two video encoding, and three audio and three video encoding sessions, respectively. If the ARM workload is over 100%, the current ARM is not able to handle the tasks and a more powerful ARM is required. In Fig. 5 (a), the system only has 1K internal memory. For the static IPC strategy, the memory is only sufficient if three audio encoding threads all apply interrupt-based notification and use the internal memory, and video encoding threads use the external memory and polling-based notification. By applying the static IPC strategy, the current system cannot support two audio and two video encoding sessions. The dynamic adjustment approach can dynamically adjust the IPC strategies for each individual encoding thread depending on workload and the remaining internal memory. When the system only handles one audio encoding session, the internal memory is sufficient to support polling-based notification and the use of the internal memory as the shared memory. The dynamic approach can reduces 12% ARM workload if the system handles only one audio session. If the system handles two audio encoding sessions and the dynamic IPC strategy is employed, the internal memory is sufficient for one audio session using the internal memory and polling-based notification, the other audio session using the external memory and polling-based notification. The dynamic approach can reduces 6% ARM workload if the system handles two audio sessions.

While the system increases the internal memory budget to 16K, the internal memory budget is sufficient for the static IPC approach to support three audio encoding sessions using polling-based notification and internal memory. However, the internal memory is not sufficient to provide all three video encoding sessions for using the internal memory. On the other hand, our proposed dynamic adjustment of IPC strategy allows individual video encoding session to change IPC strategies and parameters during run-time, and the IPC strategies for three different video encoding sessions can be different. In Fig. 5 (b), the dynamic approach allows one video encoding session using the internal memory and polling-based notification, the other video encoding session using external memory and polling-based notification, and two audio encoding sessions using the internal memory and polling-based notification. The dynamic approach significantly reduces the ARM workload by 30%. Moreover, the static IPC approach cannot support three audio and three video encoding sessions, but the dynamic approach can handle all six tasks. While the system increases the internal memory budget to 32K, the dynamic approach can reduce the ARM workload by 40% to 61% for handling both audio and video encoding sessions compared with the static approach. It is important to note that the dynamic approach optimizes the IPC strategies under system resource constraints. Fig. 5 already shows the best performance by employing the dynamic adjustment of IPC strategies. If there is unlimited internal memory budget, the optimal solution is to use the internal memory and polling-based notification for all periodical IPC tasks. For example, the ARM workload reduces to 57% but the system requires 76K internal memory for the 3A + 3V case. Since the GPP workload can be reduced by employing the dynamic adjustment approach, we could consider a low-end GPP for the media gateway at the system design phase, support more number of audio and video streams, and/or adjust the GPP voltage and frequency to save the energy at run-time.

We further applied the dynamic adjustment approach to an embedded system with one GPP and two DSPs. We assume that each DSP has 32K internal memory, and the embedded system aims to support four audio and four video encoding sessions. For the static approach, each DSP handles two audio and two video sessions. The audio sessions can use the DSP internal memory and polling-based notification, but the video sessions have to use the external memory and polling-based notification due to insufficient DSP internal memory. The GPP workload is almost 200% when it handles four audio and four video sessions. If the dynamic approach is employed, for each DSP, the audio sessions can use the DSP internal memory and polling-based notification, one video session can use the DSP internal memory and push-based notification, and the other video session can use the DSP internal memory and polling-based notification. The GPP workload is only 80% and can handle all IPC tasks. The dynamic approach can also significantly reduce the GPP workload for handling IPC tasks on an embedded system with multiple DSPs.

6. CONCLUSIONS

This paper investigates different IPC design strategies for a heterogeneous multi-core processor using an experimental test-bed, and measures precise IPC latencies under different configurations. Experimental results reveal that different IPC strategies significantly influence IPC performance, and that designers should select IPC strategies based on the characteristics of the embedded system.

Based on the findings above, we suggest dynamic adjustment of IPC strategies for an embedded heterogeneous multi-core processor. With the proposed IPC performance models, the system can estimate the IPC performance at run-time and dynamically adjusts the IPC strategies under environmental parameters and system resource constraints.

We provided two case studies. In the first case study, we applied the internal shared memory and polling notification techniques to improve the IPC performance of a VoIP phone. Experimental results show that we could reduce the GPP workload by 35% compared with the conventional approach without sacrificing VoIP functionalities and voice quality. In the second case study, we employed the concept of dynamic adjustment of IPC strategies to an embedded media gateway. The simulation results reveal that the dynamic approach outperforms the static IPC design approach.

We outlined the future work. First, the simplified performance model of IPC overhead presented in section 4 only considers average cases and it is mainly to assist system designers in adjusting run-time IPC strategies. A detail analytic model of IPC overhead based on probability models and/or queuing theory are required to precisely evaluate the GPP resources, delay and memory requirements for IPC. Second, the partition of DSP tasks and the asynchronous software pipelining design for the IPC can further improve the IPC performance. These application- and algorithm-specific optimizations may very well be our next goals. Third, cell processor composes of a GPP, a number of DSPs, local memory of the GPP and DSPs, and external memory [13]. The IPC procedures of the GPP and DSPs are similar to IPC procedures mentioned in this paper. The evaluation environment of the IPC procedures and IPC design strategies can be applied to Cell processor. The proposed dynamic adjustment scheme of IPC strategies presented in this paper can be extended and applied to Cell processor. For example, the system can further consider the current workload of every DSP and the use of the DSP internal memory as the shared memory, and thus dynamically determine the strategies and parameters for IPC requests under system resource constraints. More advanced technologies for applying dynamic adjustment of the IPC strategies to the architectures with multiple DSPs such as Cell processor are research issues for our future work.

REFERENCES

1. D. Talla and J. Gobton, "Using DaVinci technology for digital video devices," *IEEE Computer*, Vol. 40, 2007, pp. 53-61.
2. H. Y. Hsieh, S. F. Liang, L. W. Ko, M. Lin, and C. T. Lin, "Development of a real-time wireless embedded brain signal acquisition/processing system and its application on driver's drowsiness estimation," in *Proceedings of IEEE International Conference on Systems*, *Man and Cybernetics*, Vol. 5, 2006, pp. 4374-4379.

3. U. S. Gorgonio, H. R. B. Cunha, E. X. L. de Filho, S. O. D. Luiz, and A. Perkusich, "Application profiling in a dual-core platform," in *Proceedings of International Conference on Consumer Electronics*, 2008, pp. 1-2.
4. C. N. Chiu, C. T. Tseng, and C. J. Tsai, "Tightly-coupled MPEG-4 video encoder framework on asymmetric dual-core platforms," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Vol. 3, 2005, pp. 2132-2135.
5. S. O. D. Luiz, G. de M. Vasconcelos, and L. D. da Silva, "Formal specification of DSP gateway for data transmission between processor cores of OMAP platform," in *Proceedings of ACM Symposium on Applied Computing*, 2008, pp. 1545-1549.
6. T. Chen, G. Chen, H. Dai, and Q. Shi "A function-based on-chip communication design in the heterogeneous multi-core architecture," in *Proceedings of International Conference on Multimedia and Ubiquitous Engineering*, 2007, pp. 1086-1092.
7. T. Kluter, P. Brisk, E. Charbon, and P. Ienne "MPSoC design using application-specific architecturally visible communication," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009, pp. 183-197.
8. L. Brisolara, S. I. Han, X. Guerin, L. Carro, R. Reis, S. I. Chae, and A. Jerraya, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC," in *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems*, 2007, pp. 81-89.
9. Texas Instruments, TMS320DM6446 Digital Media System-on-Chip, March, 2007.
10. Texas Instruments, DSP/BIOS Link User Guide, April, 2006.
11. Linphone, http://www.linphone.org/index.php/eng.
12. WireShark, http://www.wireshark.org/.
13. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, Vol. 49, 2005, pp. 589-604.
14. The Industrial Technology Research Institute (ITRI) PAC (Parallel Architecture Core), http://pac.itri.org.tw/.

ACKNOWLEDGMENT

The authors would like to thank Industrial Technology Research Institute, MediaTek Inc. and the National Science Council of Taiwan.

Shiao-Li Tsao (曹孝櫟) earned his Ph.D. degree in Engineering Science from National Cheng Kung University in 1999. His research interests include embedded software and system, and mobile communication and wireless network. From 1999 to 2003, Dr. Tsao joined Computers and Communications Research Labs (CCL) of Industrial Technology Research Institute (ITRI) as a researcher and a section manager. He was a visiting scholar at Bell Labs, Lucent technologies, U.S.A., in the summer of 1998, a visiting professor at Department of Electrical and Computer Engineering, University of Waterloo, Canada, in the summer of 2007, and Department of Computer Science, ETH Zurich, Switzerland, in the summer of 2010 and 2011. Dr. Tsao is currently an Associate Professor of Department of Computer Science of National Chiao Tung University. Prof. Tsao has published more than 75 international journal and conference papers, and has held or applied 18 U.S. patents. Prof. Tsao received the Young Researcher Award of Pan Wen-Yuan Foundation, Young Engineer Award from the Chinese Institute of Electrical Engineering in 2007, Outstanding Teaching Award of National Chiao Tung University, and K. T. Li Outstanding Young Scholar Award from ACM Taipei/Taiwan chapter in 2008.

Sung-Yuan Lee (李松遠) received his M.S. degree in Computer Science from National Chiao Tung University, Taiwan, in 2009. His research interests include wireless networks and embedded systems.