# Enabling Streaming Remoting on Embedded Dual-core Processors

Kun-Yuan Hsieh, Yen-Chih Liu, Po-Wen Wu, Shou-Wei Chang, Jenq Kuen Lee
Department of Computer Science
National Tsing-Hua University, Hsin-Chu, Taiwan
{kyshieh, ycliu, pwwu, swchang, jklee}@pllab.cs.nthu.edu.tw

## Abstract

*Dual-core processors (and, to an extent, multicore processors) have been adopted in recent years to provide platforms that satisfy the performance requirements of popular multimedia applications. This architecture comprises groups of processing units connected by various interprocess communication mechanisms such as shared memory, memory mapping interrupts, mailboxes, and channel-based protocols. The associated challenges include how to provide programming models and environments for developing streaming applications for such platforms. In this paper, we present middleware called streaming RPC for supporting a streaming-function remoting mechanism on asymmetric dual-core architectures. This middleware has been implemented both on an experimental platform known as the PAC dual-core platform and in TI OMAP dual-core environments. We also present an analytic model of streaming equations to optimize the internal handshaking for our proposed streaming RPC. The usage and efficiency of the proposed methodology are demonstrated in a JPEG decoder, MP3 decoder, and QCIF H.264 decoder. The experimental results show that our approach improves the performance of the decoders of JPEG, MP3, and H.264 by 24%, 38%, and 32% on PAC, respectively. The communication load of internal handshaking has also been reduced compared to the naive use of RPC over embedded dual-core systems. The experiments also show that the performance improvement can also be achieved on OMAP dual-core platforms.*

## 1 Introduction

Dual-core processors are increasingly used to provide platforms that satisfy the ever-increasing performance requirements of popular multimedia applications. This architecture comprises groups of processing units connected by various interprocess communication mechanisms such as shared memory, memory mapping interrupts, mailboxes, and channel-based protocols. The TI OMAP [1] platform is a typical example in this category. The associated challenges include how to provide programming models and environments for developing applications for such platforms. One of the most important issues is to provide streaming functionality in the application domain of multimedia applications. Applications such as video encoding and decoding, image processing, data mining, and graphic rendering naturally include data streaming. With application characteristics in mind, it is important to explore the programming flow and environments of streaming when attempting to support middleware for embedded dual-core processors. In this paper, we present methods to enable streaming RPC (remote procedure call) flow to support streaming-function off-loading on asymmetric dual-core architectures.

One of the most promising programming models on distributed systems is the Java RMI, which is a form of RPC. Improving the efficiency of such layers has been widely investigated. For example, the ARMI [2] and Manta [3] systems overcome various drawbacks of RMI and provide new RMI-style systems with extended functionality. KaRMI [4] improves the implementation of RMI by exploiting Myrinet hardware features to reduce latencies. Adopting this layer for wireless environments has also been investigated [5]. Although remoting mechanisms represent a promising and easy way to model applications on distributed systems, how to enable streaming data flow to control and program embedded dual-core or multicore systems remains challenging. Moreover, the requirement to have programming models on asymmetric dual-core architectures to utilize data streaming flow makes it difficult to model and interface interprocess communications and data streaming on such architectures [6, 7].

Research work such as StreamIt [8] and Brook [9] provide language supports for streaming programming. There are research tried to apply the stream language on multicore system [10] or to provide a runtime system that automatically maps stream program onto multiple processors [11]. However, few research discussed about supporting such programming paradigm in the layer of remote procedure invocation. Yang et.al [12] proposed a streaming style of RMI programming model based on Java RMI which provides a good indicator that this layer of model is a possible direction for embedded dual-core programming.

In this paper, we present methods to enable streaming RPC flow to support streaming-function off-loading on asymmetric dual-core architectures, which is applicable to both on PAC [13, 14] dual-core platforms and TI OMAP dual-core environments. The concept of streaming RPC provides several advantages. First, it provides RPC-style abstraction to provide programmers with a programming environment to develop function off-loading programs, with the abstraction being at a higher level than

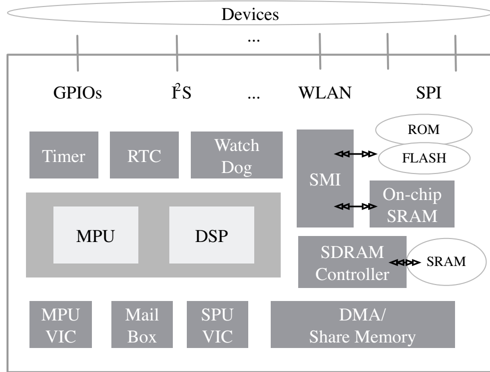


**Figure 1. Conventional dual-core architecture**

message passing. Second, streaming RPC overlaps communication and computation to improve performance in a parallel execution environment. Third, the APIs and programming model provided by streaming RPC guide the programmer to exploit the potential control and data parallelism in a intuitive way. Finally, streaming RPC reduces the amount of hand-shaking between the client and server by automatically streaming data from the server to the client. We present a software architecture to enable such communication flow for embedded dual-core processors. In addition, to manage the buffers for our proposed streaming RPC, an analytic model of streaming equations is proposed to reduce the amount of redundant communications caused by the asymmetry of the two processing units due to their different streaming rates in accessing the streamed data elements. The discrepancy in the streaming rate between two processors produces frequent remote function invocations between the server and client, which leads to a large amount of redundant communications. We present a streaming equation that suggests an appropriate threshold parameter to control the streaming rate in the streaming RPC to further reduce interactions between the client and server. A JPEG decoder, MP3 decoder, and H.264 decoder are presented to demonstrate both the use and efficiency of the proposed methodology. The experimental results show that our approach improves the performance of the decoders of JPEG, MP3, and H.264 by24%, 38%, and 32% on PAC, respectively and reduces the internal handshaking compared to the naive use of RPC over embedded dual-core systems.

The remainder of the paper is organized as follows. Section 2 provides an overview of the software framework and programming model of streaming RPC. Section 3 presents the systematic methodology and procedures of the communication mechanism. Section 4 details the streaming equations to prune redundant communications when performing streaming RPC. Section 5 provides the experimental results. Finally, Section 6 concludes this paper.

# 2 Overview

## 2.1 Software Design Flow on Dual-core Architecture

As shown in Figure 1, to achieve the processing requirements of high performance, multifunction, and low power-consumption, a conventional dual-core architecture comprises a dedicated processor(e.g. a digital signal processor), a microprocessor(MPU), and integrated peripheral controllers. Compared to the design of a MPU, a digital signal processor(DSP) is optimized for processing high computational tasks with different architectural features of memory system, data operation, and instruction sets. The asymmetric design of such architectures raise challenges of programming in language support, algorithm design and standard communication protocols, various instruction-sets, new parallel computing scenario, and debugging challenges [15]. One of the architectural issues affecting the performance and modeling of applications on dual-core architectures is the design of DSP. Take PACDSP for example, the design of multibank and distributed register files are adopted to provide a low power consumption and cost architecture. To support high computation power, PACDSP is constructed as five-issue VLIW processor. Such design brings challenges to the programmers of how to exploit potential parallelism provided by architecture and utilize the computation power of DSP.

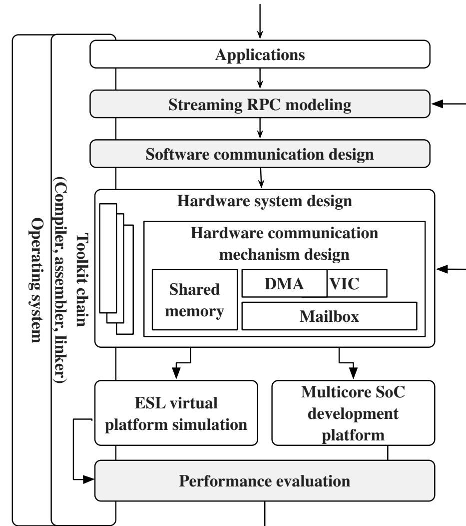


**Figure 2. Dual-core application design flow based on streaming RPC**

Figure 2 presents an overview of the application design flow for embedded dual-core systems conforming with our design. It includes toolkit chains, operating systems, ESL (electronic system level design) for dual-core simulations, and interprocess communication mechanisms. The modeling of applications involves the underlying interprocessor communication mechanisms provided by the architecture [16]. The basic communication protocol for interacting the MPU and DSP is both processor polling through shared memory [17] or sending events by triggering interrupts. When polling is used, both processors agree with a specific memory partition scheme. When a task asks resources from tasks resided in the other processor, it writes a flag in the shared memory and then polls the status until the flag is polled by the corresponding task. Alternative approach for inter-processor communication mechanism is sending events by invoking interrupts which provides a basic hardware mechanism for implementation software communications.

## 2.2 Programming Model

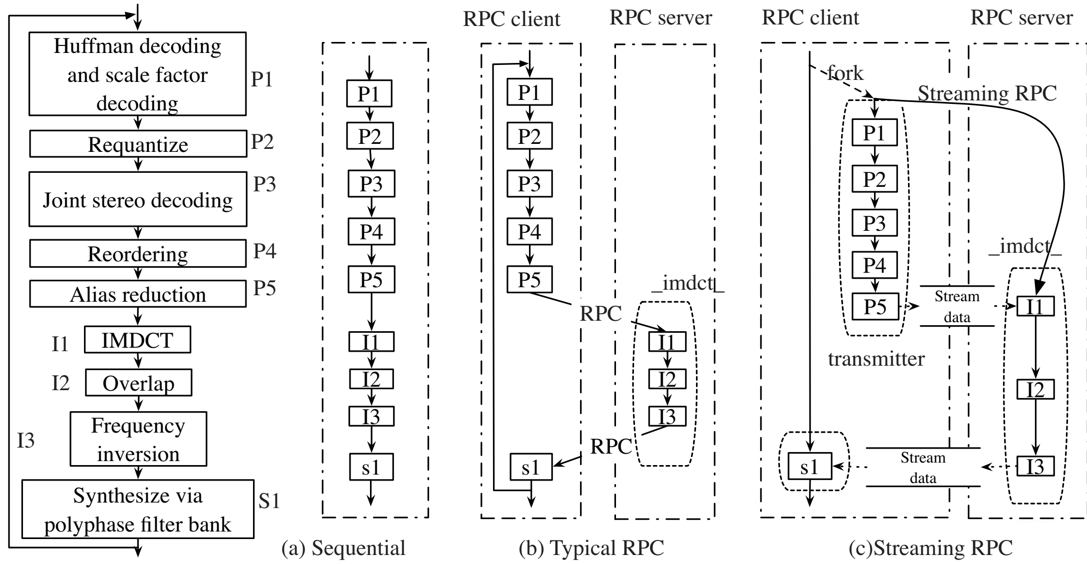


**Figure 3. Flow control in a simplified MP3 decoder**

**Program 1. Sample code of an MP3 decoder**

```
/* Streaming RPC client */
void MP3_decoder(){
        stream_rpc(_imdct_, _transmitter_);
}
void _transmitter_(){
 STREAM_ID id = 4;
  /* Initializing streaming channel */
 stream_create(id);
 /* Pushing data to streaming channel */
 stream_put(id, DATA);
 stream_push(id);
 ...
}
...
/* Streaming RPC server */
void _imdct_(){
 STREAM_ID id = 4;
  /* Initializing streaming channel */
 stream_create(id);
 /* Aggregating data from streaming
    channel */
 stream_get(id, DATA);
 stream_pop(id);
 ...
}
...
```

The proposed streaming RPC is based on a middleware called pCore Bridge which provides basic communication modules on dual-core architectures. pCore Bridge is built based on dual-OS environment with Linux running on MPU and pCore [18] running on DSP. pCore is a multithreaded, priority-based, preemptive, and dual-core/multicore supportive kernel designed for the DSP on dual-core architectures. With transparent kernel modules and well-defined design patterns, pCore cooperates well with the OS on the MPU. Moreover, as a highly flexible and configurable system, pCore supports the developers to easily adopt various programming models according to different needs to provide a high-productivity runtime environment with efficient execution model on the dual-core architecture of PAC.

The software APIs used for the streaming RPC are based on a conventional C-function call with Linux system library and pCore Bridge support. The programming model for streaming RPC is illustrated using an MP3 decoder. Figure 3(a) shows the decoding flow, and Figure 3(b) shows the execution flow of the applications on dual-core SoCs, which represents about 25% of the workload on the DSP. The client invokes the remote module *_imdct_* by calling *rpc_invoke(_imdct_)*. The procedure invocation is synchronized in the paradigm of a typical RPC. This represents a simple programming model of a dual-core application, and it does not exploit the potential architectural parallelism. Moreover, it fails to model the data streaming of the multimedia application. Figure 3(c) shows the execution model of streaming RPC the maps the application level parallelism to the hardware communication mechanism by providing simple programming APIs. A streaming RPC client is invoked as follows:

```
stream_rpc(_imdct_, _transmitter_);
```

Program 1 lists sample code of an MP3 decoder implemented by a streaming RPC. In the mechanism, the thread in which data are pushed from a sender is called the transmitter, while the thread in which data are aggregated on the receiver is called the aggregator. When invoking a streaming RPC, the data required are transmitted to the RPC server by the transmitter that is monitored by the stream controller. After invoking *stream_rpc()*, the client is allowed to continue computation concurrently. The stream controller then schedules the transmitter specified in the argument for data transmission. To initialize the streaming communication, the transmitter and aggregator first create a streaming channel by flagging the same identifier to the API *stream_create*. Processes with the same *stream_id* are bound to the same streaming channel.

The transmitter uses *stream_put* to put data into the stream buffer and *stream_push* to send it to the streaming channel. The APIs allow the developer to shape the streaming kernel by assembling the available stream buffers.

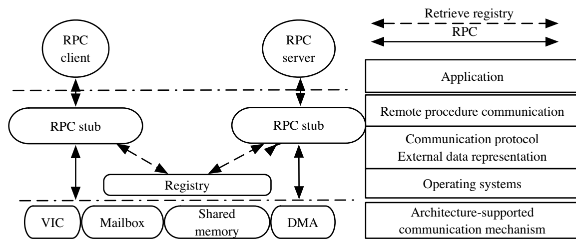


**Figure 4. Software architecture of RPC on dual-core architectures**

Once the transmitter has successfully transmitted the data required, the stream controller invokes *_imdct_* for computation. To retrieve the data from the streaming channel, the aggregator in the server first creates a streaming channel with the same identifier, then gets the data from the streaming channel by invoking *stream_get*. The aggregator is required to free the streaming buffer by applying *stream_pop* to the data that is not needed.

As shown in Figure 3, streaming RPC not only provides a mechanism of data streaming function off-loading, but also exploits the parallelism in the applications. The underlining mechanism of streaming RPC execute the application while concurrently running aggregator and transmitter for data transferring. In addition to provide a middleware for streaming programming, streaming RPC exploit the performance of dual-core applications by aggressively utilizing the hardware resources and exploiting the potential parallelism in the architecture.

# 3 Supporting Data Streaming for Remote Function Invocation

## 3.1 Software Architecture of Streaming RPC

Remote procedure call(RPC) is a promising technique in distributed systems that allows the invocation of a remote procedure on a different processor. Figure 4 shows the software architecture of RPC in the dual-core architectures that have been adopted in programming dual-core system-on-chip(SoC) in recent years [19]. Applications running on different processors communicates by invoking commands provided by the software communication protocol. The software communication is built on top of the hardware communication modules. The lower layer of the communication design employs a standard RPC design for function off-loading.

As shown in Figure 4, the client and the server communicate with each other through the RPC stub to perform proper communication strategies. The stub queries a shared structure called *registry* that resided in the operating systems to keep the descriptions of each remote application.

Our proposed streaming RPC is based on the software framework of RPC and is implemented in the upper layer of design flow of communication layers. The novel programming paradigm allows developers to model streaming applications by exploiting potential parallelism for multimedia applications by streaming data required for the RPC calls. In the proposed model, a stream is a data pipe associated with an RPC invocation. To date we have enabled streaming RPC flow for two dual-core platforms, PAC and TI OMAP 5912, both of which comprise a general processing unit as the main processor and a DSP as a specialized slave processor for accelerating multimedia applications. Figure 5 depicts the software architecture of streaming RPC, which comprises the three key components described below:

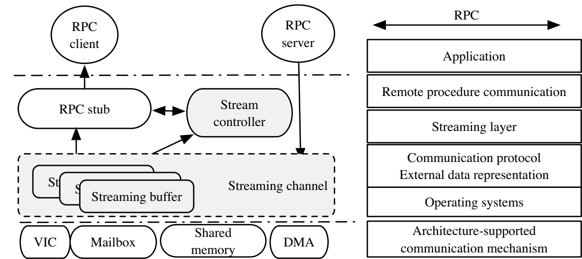


**Figure 5. Software architecture of streaming RPC**

- **Streaming channel** A streaming channel is associated with an RPC request for transmitting data by setting the predefined stream identifier. The streaming channel provides a communication channel between the RPC client and server.
- **Streaming buffer** The streaming operations allow the client and server to overlap communications and computations by continuously receiving and sending data through the streaming channel. To support such data streaming, multiple *streaming buffers* are associated with a streaming channel for providing data buffering while performing streaming operations.
- **Stream controller** The discrepancies in processing speed and I/O latency between processors mean that the production rate of the sender and the consumption rate of the receiver are not equal. This results in the computation unit with a faster streaming rate having to wait for data communications. To avoid the associated blocking overhead, the stream controller monitors and manages the streaming channel to implement data-driven streaming operations.

## 3.2 Pushing and Aggregating Mechanism

Streaming RPC is based on the typical RPC communication mechanism that allows the invocation of a remote procedure on a different processor. For this mechanism to support efficient data streaming, we introduce a pushing and aggregating mechanism for data transmission to avoid the typical call-and-wait mechanism of typical RPC. The pushing and aggregating mechanism automatically transfers data from the sender to the receiver by pushing data into the streaming channel using an asynchronous communication protocol. In the mechanism, the thread in which data are pushed from a sender is called the transmitter, while the thread in which data are aggregated on the receiver is called the aggregator. When a transmitter/aggregator queries the streaming channel for a streaming buffer to perform data transfer, the corresponding stream controller first checks if a streaming buffer is ready; if it is not, the transmitter/aggregator is suspended until a free streaming buffer is available.

Frequent suspending and waking up is avoided by assigning a threshold to a streaming channel, with the stream

**Algorithm 1** : Pushing mechanism

**Require:** $Q$ streaming channel
**Require:** $P$ next empty streaming buffer from $Q$
**Require:** $n$ rate control threshold
**Require:** *DATA* data to be transmitted
  **while** $P \neq \phi$ **do**
    $P \Leftarrow$ *DATA*
    Push $P$ to $Q$
    $Buffer_{ready}$++
    **if** $Buffer_{ready} \geq n$ and $aggregator$ is suspended **then**
      Trigger $aggregator$ by passing messages
    **end if**
  **end while**
  **if** $P = \phi$ **then**
    Suspend the $transmitter$
  **end if**

**Algorithm 2** : Aggregating mechanism

**Require:** $Q$ streaming channel
**Require:** $P$ next ready streaming buffer from $Q$
**Require:** *DATA* data to be aggregated to
  **while** $P \neq \phi$ **do**
    *DATA* $\Leftarrow P$
    Pop $P$ from $Q$
    $Buffer_{ready}$−−
    **if** $Buffer_{ready} \leq n$ and $transmitter$ is suspended **then**
      Trigger $transmitter$ by passing messages
    **end if**
  **end while**
  **if** $P = \phi$ **then**
    Suspend the $aggregator$
  **end if**

controller only waking up procedures when a streaming channel satisfies the threshold criterion. For example, if the data streaming speed of the transmitter is faster than that of the aggregator, the stream controller blocks the transmitter when there is no streaming buffer available. When there is an empty buffer, the transmitter is woken up immediately. However, such a mechanism increases the redundant communication overhead caused by frequent message passing. Thus, we proposed a new rate-controlling parameter $n$ to reduce such overhead. By setting the threshold of $n$ to be any number $k$ larger than 1, the stream controller wakes up the transmitter when there are $k$ streaming buffers available, rather than waking up the transmitter immediately. Controlling the threshold allows the programmer to improve the performance by reducing the frequency of system call invocations for internal RPC hand-shaking.

Algorithm 1 illustrates the pushing mechanism. Once invoked, the transmitter first queries the corresponding streaming channel $Q$ for a empty streaming buffer $P$ to transmit the data to the receiver. The stream controller records the number of the ready buffer in the $Buffer_{ready}$ parameter for implementing the streaming flow. When $Buffer_{ready}$ is greater than $n$, this indicates that the buffer is large enough to support data streaming. After the data are successfully pushed to the streaming channel, the stream controller checks the state of the aggregator, and if the aggregator is suspended and $Buffer_{ready}$ is greater than $n$, the transmitter then wakes up the aggregator so that it receives data. If there is no empty stream buffer available, the transmitter suspends itself instead of actively waiting for a buffer to be released. Such a paradigm allows the processor to be fully utilized by making more computation power available to other applications.

The data aggregating mechanism in streaming RPC is illustrated in Algorithm 2. The aggregator first queries the streaming channel $Q$ for a ready buffer $P$. After successfully retrieving data from streaming buffer to *DATA*, the streaming buffer is then released for the transmitter to keep transferring more data. When there is no ready buffer in the streaming channel, the aggregator checks the state of the transmitter. If the transmitter is suspended, the aggregator wakes it up and then suspends itself to wait for the transmitter to transfer data. The proposed pushing and aggregating mechanism improves the efficiency of data transmission by using an asynchronous communication protocol without waiting during busy periods.

## 4 Optimizing Internal Handshaking

Whilst the proposed streaming RPC provides an efficient data streaming mechanism, differences in the processing speed and I/O latency between processors results in discrepancies in the streaming rates between the transmitter and aggregator. The streaming rate refers to the amount of stream data accessed by the transmitter or the aggregator per unit of time. The asymmetry in the streaming rates between application client and server could result in a large amount of implicit internal RPC communication for hand-shaking. In this section, we propose a governing equation to avoid redundant communication using a threshold parameter in the streaming channel without invalidating the real-time requirement of a multimedia application. Moreover, the proposed streaming RPC provides an option to estimate an appropriate threshold value by profiling the architectural parameters using a proposed slack-less analysis.

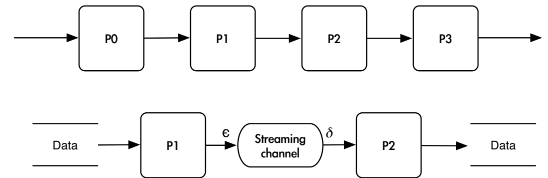


**Figure 6. Pipelined streaming application**

Figure 6 shows a conventional streaming application model with data-pipelined execution flow where the sender and receiver access the streaming channel at average rates of $\epsilon$ and $\delta$, respectively. Assume that the application transfers $k$ bytes of data through the streaming channel between each pipeline stage, where the data are processed by the sender on processor 1 (P1) in $\beta$ seconds. If P1 can access $\frac{1}{T_{P1}}$ bytes in the memory per second, a time of $k \times T_{P1}$ seconds is required to write the data to the stream-

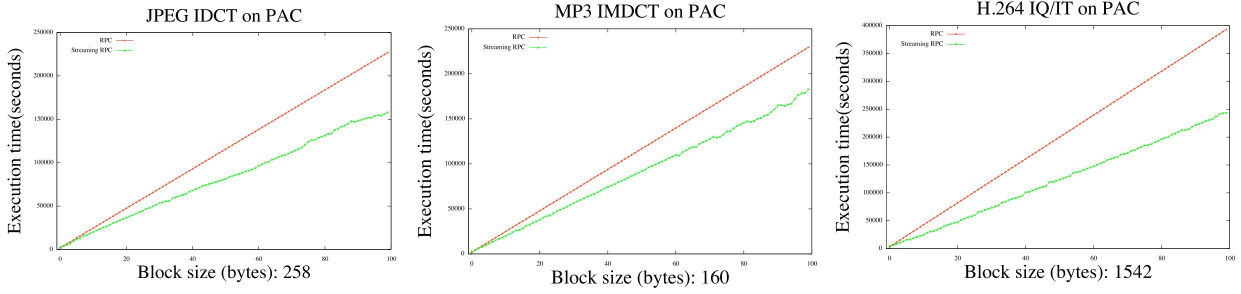


**Figure 7. Performance evaluation of different application kernels**

ing channel. Transferring $k$ bytes of data from one pipeline stage to the next stage through the streaming channel takes $k \times T_{P1} + \beta$ seconds. Thus, the streaming rate of the sender is

$$\epsilon = \frac{1}{k \times T_{P1} + \beta}$$

Assume the receiver on processor 2 (P2) reads the $k$ bytes of data in $k \times T_{P2}$ seconds and processes it in $\alpha$ seconds. The streaming rate of receiver $\delta$ can be determined in the same way:

$$\delta = \frac{1}{k \times T_{P2} + \alpha}$$

If $\delta > \epsilon$, the receiver is blocked when there is no stream element available in the streaming channel. The stream controller will wake the receiver once the sender pushes the next element into the streaming channel. This behavior represents a huge overhead to the system. To avoid frequent triggering of the receiver, the proposed streaming operations allow the programmer to set up a threshold $n$, whereby the stream controller waits until there are $n$ stream data elements available for processing.

One of the problems is to decide the most appropriate value for threshold parameter $n$. Ignoring memory consumption and the response-time requirement, the best solution to this problem is to keep the receiver from waiting once it is triggered by setting $n$ to $\infty$. However, the memory is always limited and expensive in embedded dual-core architectures, and keeping the receiver waiting for a long period of time conflicts with the requirement for real-time operation. Let us assume that the response-time requirement of the receiver for the next stage of computation is $T_r$, and that the overhead of triggering the receiver is $T_{trigger}$. Once a receiver is blocked, it has to wait for the sender to transfer $n$ data elements to the streaming channel. Then the stream controller triggers the receiver to process these data elements. The response time of the receiver after being blocked with threshold parameter $n$ is determined by the summation of the times required to transfer $n$ data elements $\frac{n}{\epsilon}$, for the overhead of triggering the receiver $T_{trigger}$, and to process the first data element $\alpha$. The response time must satisfy the following response-time requirement:

$$T_r \geq T_{trigger} + \frac{n}{\epsilon} + \alpha \qquad (1)$$

The upper bound of threshold parameter $n$ is obtained by solving the inequality in Equation 1 for $n$:

$$n \leq (T_r - T_{trigger} - \alpha) \times \epsilon \qquad (2)$$

Setting the threshold to an appropriate value should reduce the frequency of signal passing for internal RPC hand-shaking. Moreover, threshold analysis should guide the developer for future optimizations by adjusting the algorithms to fit the streaming rate or to scale the power status when attempting to minimize the power consumption. To guide the remodeling for enhancing the performance as mentioned in Section 2, the proposed streaming RPC provides an option for estimating an appropriate threshold value. The effects of applying different threshold values in different applications are demonstrated in Section 5.

# 5 Experiments

We now describe the methods we used to evaluate the proposed streaming RPC and the results obtained. The experiments were performed on PAC. The PAC development board that is built on the ARM Versatile PB926, which contains a 300-MHz ARM 926EJ-S processor with a 32-KB cache. It is extended with an evaluation board (EVB) comprising a 250-MHz PAC DSP, 64 Kbyte of data memory, and 32 KB of instruction cache. The two processors can exchange data and signals via 128 MB of SDRAM residing on the Versatile and 512 MB of SDRAM on the EVB. Early evaluation were also performed on TI OMAP 5912 to reveal that the mechanism can also be applied on different architectures. The OMAP starter kits comprise an ARM 9 core and a DSP core, both operating at 192 MHz and having 32 MB of SDRAM and 32 MB of ROM. The evaluation was implemented in Linux on the ARM core and using a runtime system called pCore on the DSP.

The application kernel, inverse discrete cosine transformation (IDCT) of the JPEG decoder, inverse modified discrete cosine transformation (IMDCT) of the MP3 decoder, and inverse quantization and inverse transformation (IQ/IT) were evaluated to assess the performance and overhead of streaming RPC with different computational features for various amounts of data. Figure 7 compares the performances of the evaluated application kernels running with different numbers of data elements. The IDCT of the JPEG decoder required 258 bytes data to accomplish a complete round-trip transformation, while the IMDCT and IQ/IT required 160 bytes and 1542 bytes respectively. There was a performance reduction when processing only a few data elements due to the overhead of maintaining the streaming channel of the stream controller. However, the performance of streaming RPC improved as the number of data elements increased. The data operations benefit from the pushing and aggregating mechanisms, which improve the total performance of kernels employing streaming RPC. Moreover, an increasing amount of processed

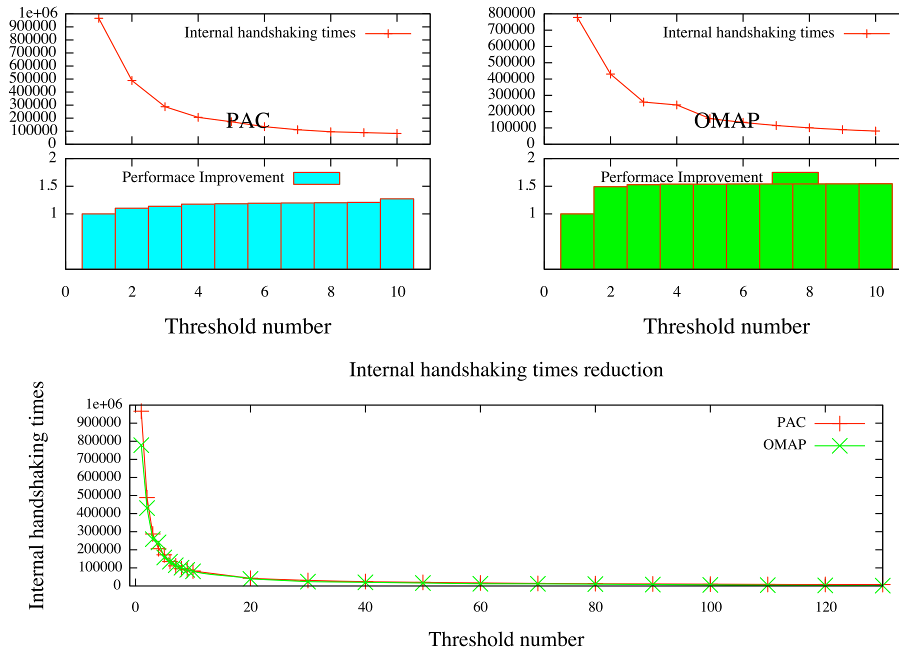


**Figure 8. Performance improvement and corresponding internal RPC hand-shaking times for different threshold values on the MP3 decoder**

data provides more opportunities for the streaming RPC to overlap computations and communications. In summary, the performance of each kernel was improved after applying streaming on PAC.

In order to assess the performance of streaming RPC relative to typical RPC in multimedia applications, we implemented three common decoders: a JPEG decoder with a resolution of 317×255, an MP3 decoder, and a QCIF H.264 decoder. Figure 9 shows the performance improvement of streaming RPC for these multimedia decoders. The features of the three decoders are quite different. The partitioning of the JPEG and MP3 decoders resulted in about 25% of the program being executed on the DSP with data of fine granularity and high data parallelism, and without cross referencing between frames. The performance improvement for the JPEG decoder was 24% on PAC and the performance of MP3 decoder was improved by 38% on PAC compared to RPC implementation. The H.264 decoder also had 25% of the workload off-loaded to the DSP, but its operation is much more complex. Decoding a frame of an H.264 video involves many references to the previous frame, which potentially reduces the degree of data parallelism. Moreover, the data are difficult to partition with good granularity. However, although with coarse-grain data partitioning, the performance of H.264 was still improved by 32% on PAC when using streaming RPC. The early evaluation of two decoders of JPEG and MP3 with streaming RPC support also had been implemented on OMAP to demonstrate that the proposed mechanism is applicable to different architectures. The performance improvement of JPEG decoder is 26% and the performance of MP3 decoder is improved by 40% on OMAP. The improvement shows that streaming RPC is able to attain high performance improvement on different architectures compared to naive use of RPC.

To demonstrate the effects of the choice of threshold for the streaming channel, we measured the execution time of the MP3 decoder without synchronizing the time ticks for different threshold values. Figure 8 shows that the performance of the MP3 decoder improved dramatically when the threshold was changed from 1 to 10, and improved less for larger values. The performance line gives the performance improvement of streaming RPC with different number of thresholds from Equation 2 shown earlier in Section 4. The base version is the one set threshold as *one*. The LHS of the figure is for MP3 on PAC platform, and the RHS of the figure is for TI OMAP platform. The line for RPC times is the communication amount. Significant amount of reductions on communications were achieved. Both lines are given with the improvement ratio. Figure 8 shows the corresponding internal RPC hand-shaking times for different threshold values on the MP3 decoder. Although the performance was not greatly affected by the threshold value, the amount of communication improved dramatically as the threshold increased. We also measured response time $T_{trigger} + \frac{n}{\epsilon} + \alpha$ (see Section 4) for different thresholds in the MP3 decoder. The decoder played 40 frames per second, where each frame comprised 128 sub-banks. The program was partitioned to process a sub-bank as a data unit on the DSP. The average response time $T_r$ for the streaming RPC was $125,000$ microseconds to satisfy the real-time requirement. Figure 10 shows the distribution of the response times for different thresholds of numbers of execution. The results of the response time of each threshold varies in a range that could be affected by the Linux scheduler and the on-board I/O latency, however, it illustrates an average soft-realtime requirement of the program. The profiler of streaming RPC conservatively suggests a threshold value of 80 on PAC and 100 on OMAP, which represents a light communication workload (as shown in Figure 8) and satisfies the real-time requirement.

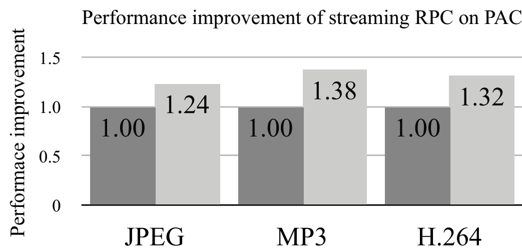


**Figure 9. Performance evaluation of different decoders**

## 6 Conclusion

Streaming programming can benefit multimedia applications by increasing the efficiency of data transmission and utilizing the potential parallelism in dual-core embedded systems. Here we presented a mechanism called streaming RPC that supported data streaming. Streaming RPC employed a pushing and aggregating mechanism for data transmission with an asynchronous communication protocol. This paradigm improved the performance of the application and increased the utilization of the processors by avoiding waiting for the streaming application during busy periods. We also presented a systematic way of analyzing the streaming rate for slack-less data streaming by setting a threshold value for a streaming channel. The results showed that the proposed mechanism provided an efficient way of supporting streaming programming on

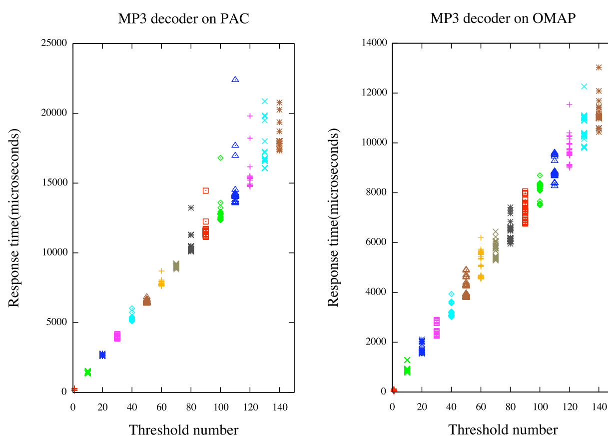


**Figure 10. Effect of threshold on response time in the MP3 decoder**

## Acknowledgment

This research was supported in part by the NSC under grant nos. NSC 95-2220-E-007-001 and NSC 95-2220-E-007-002, and by the MOEA research project under grant nos. 95-EC-17-A-01-S1-034 and 96-EC-17-A-01-S1-034 in Taiwan.

## References

[1] Texas Instuments. *OMAP5912 Application Processor*.

[2] Rajeev R. Raje, Joseph I. William, and Michael Boyles. An asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, 1997.

[3] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. In *Proceedings of Principles Practice of Parallel Programming*, pages 173–182, 1999.

[4] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for java. In *Proceedings of the ACM Java Grande Conference*, pages 152–157, 1999.

[5] Cheng-Wei Chen, Chung-Kai Chen, Jyh-Cheng Chen, Chien-Tan Ko, Jenq-Kuen Lee, Hong-Wei Lin, and Wang-Jer Wu. Efficient support of Java RMI over heterogeneous wireless networks. In *Proceedings of the International Conference on Communication*, pages 1391–1395, 2004.

[6] Ahmed A. Jerraya, Aimen bouchhima, and Frédéric Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Proceedings of Design Automation Conference*, pages 280–255, 2006.

[7] Grant Martin. Overview of the MPSoC design challenge. In *Proceedings of Design Automation Conference*, pages 274–279, 2006.

[8] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of Computational Complexity*, pages 179–196, 2002.

[9] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[10] David Chang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multiore execution. In *Proceedings of International Conference on Prarallel Processing (ICPP)*, 2007.

[11] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: Programming gerneral-purpose multicore processor using streams. *ACM SIGOPS Operating System Review*, 42(2):297–307, Mar 2008.

[12] Chih-Chieh Yang, Chung-Kai Chen, Yu-Hao Chang, Kai-Hsin Chung, and Jenq-Kuen Lee. Streaming support for Java RMI in distributed environment. In *Proceedings of ACM International Conference on Principles and Practices of Programming In Java*, pages 53–61, 2006.

[13] David Chih-Wei Chang. PAC digital signal processor. In *Proceedings of Fall Microprocessor Forum*, 2006.

[14] David Chih-Wei Chang, I-Tao Liao, Jenq-Kuen Lee, Wen-Feng Chen, Shau-Yin Tseng, and Chein-Wei Jen. PAC DSP core and application processors. In *Proceedings of IEEE International Conference on Multimedia and Expo*, pages 289–292, 2006.

[15] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual conference on Design automation*, pages 681–685, 2004.

[16] Dana S. Henry. *Hardware mechanisms for efficient interprocessor communication*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1996.

[17] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, 1991.

[18] Kun-Yuan Hsieh, Yung-Chia Lin, Chien-Ching Huang, and Jenq Kuen Lee. Enhancing microkernel performance on VLIW DSP processors via multiset context switch. *Journal of VLSI Signal Processing Systems*, 51(3):257–268, June 2008.

[19] Steve Preissig. *Programming details of codec enging for DaVinci Techonology whitepaper*. Texas Instuments.