THE SUPPORT OF SOFTWARE DESIGN PATTERNS FOR STREAMING RPC ON EMBEDDED MULTICORE PROCESSORS

Kun-Yuan Hsieh, Yen-Chih Liu, Chi-Hua Lai, and Jenq Kuen Lee

Department of Computer Science
National Tsing-Hua University, Hsin-Chu, Taiwan
{kyshieh, ycliu, chlai, jklee}@pllab.cs.nthu.edu.tw

ABSTRACT

The development of embedded system has been toward the multicore architectures in the recent years. It raises concerns in the community of supporting programming models and languages to derive maximal performance from the architectures. Among the diversity of models for programming multicore processors, remote procedure call (RPC) is one of the most relevant programming techniques for supporting an explicit parallel programming model. Although such promising programming technique provides an easy way of modeling the applications on multiple processors, it remains an interesting and challenging problem of how to provide an effective system of programming data-intensive applications under the programming scenario of RPC. In this paper, we propose a streaming mechanism called streaming RPC to provide a system for modeling data-intensive and stream-based applications to efficiently utilize the constituents of the multicore processors. Streaming RPC is based on the framework of RPC and implemented as a middleware support to provide a library-based programming model with parallelism by mandatory. We also propose design patterns for the streaming mechanism and present experiences of developing high performance multimedia applications. Experimental results show that our streaming RPC framework is efficient to support multicore programming for multimedia applications.

Index Terms— Parallel processing, Multiprocessor, Remote procedure calls

1. INTRODUCTION

The development of embedded system has been toward the multicore architectures that combine processors connected through bus-level interfaces into a multi-processor system-on-chip(MPSoC) package in recent years. As MPSoCs are coming into the mainstreams of commercial products such as hand-held devices, the number of processors to be used are also predicted to grow rapidly for supporting increasingly added application features. It raises concerns in the community for supporting programming models and languages to drive performance from the architecture [1, 2].

Among the diversity of models for programming multicore processors, remote procedure call (RPC) is one of the most relevant programming techniques for supporting an explicit parallel programming model. Such mechanism has been a promising programming paradigm in distributed computing to support ubiquitous components communications in heterogeneous environments. There are several well-known software layers in the area such as Java remote method invocation (RMI), .NET Remoting, and CCA remoting. Research has been proposed for provide features and optimization in this layer. Thiruvathukal et al. [3] proposed an open RMI implementation to provide a better use of object-oriented features of Java; Raje et al. [4], and Maassen et al. [5] improved the basic RMI mechanism to provide a system with features of extended RMI functionality. In 1999, Nester et al. implemented KaRMI to exploit the hardware features of Myrinet that improves the RMI performance by reducing communication latency. The possibility of using remote-invocation as universal programming model is evidenced by the work in supporting Java RMI over heterogeneous wireless network [6].

Although such promising programming technique provides an easy way of modeling the applications on distributed environments, it remains an interesting and challenging problem of how to provide the programming scenario of RPC for more tightly multicore systems compared to distributed environments. In recent years, streaming programming has been proposed to provide optimizations and specifications controlling data flow of for multimedia applications. Research work such as StreamIt [7] and Brook [8] provide language supports for this application space. It then looks an interesting issue to investigate how to combine two promising programming paradigms, remoting and streaming, together. Yang et.al [9] earlier successfully proposed a streaming programming model based on Java RMI over the remoting layer for component software on distributed programming and provided a good indicator that this layer of model is a possible direction for embedded multicore programming.

In this paper, we propose a streaming mechanism called streaming RPC to provide a system for modeling stream-based applications based on the RPC programming paradigm.

Streaming RPC is implemented as a middleware support to provide a library-based programming model with parallelism and to efficiently deploy the constituents of the multicore processors. To make it easy to write efficient and reusable programs on multicore processors, we also propose design patterns for the streaming mechanism with the aspects and sample implementations. The design patterns provide structural suggestion of writing streaming RPC programs. The design patterns are based on experiences of developing high performance multimedia applications on multicore processors using RPC with streaming mechanism.

This paper is organized as follows. Section 2 presents the background information of and the programming model of streaming RPC. Section 3 describes motivation and aspects of the proposed design patterns in streaming RPC. Section 4 discusses the effect and efficiency of streaming RPC and the proposed programming model on multicore processors. Finally, section 5 concludes this paper.

2. OVERVIEW OF STREAMING RPC

2.1. Operating Systems and Middleware Support

The runtime system of the proposed streaming RPC is based on the communication model of RPC. The early version of streaming RPC is implemented with the support of a middleware called pCore Bridge which provides basic RPC communication modules on multicore architectures. pCore Bridge is running on an environment of multiple operating system with Linux running on main processing unit (MPU) and pCore [10] running on specialized processing units (SPUs). pCore is a multi-threaded, priority-based, preemptive, and multi-core supportive kernel designed for the SPUs on heterogeneous multicore 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 multicore processors.

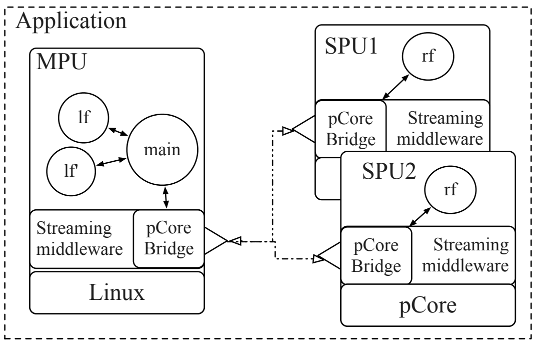

Fig. 1. Software framework of streaming RPC.

The basic execution unit of streaming RPC is a thread. The client and server of streaming RPC are executable threads implemented with C-code running on different processors. Figure 1 shows the software framework of streaming RPC which illustrates the application software stack on multicore processors. As shown in Figure 1, a multicore application is partitioned into several parts: main thread, local functions (lf, in MPU's view point), and remote functions (rf). Each part of the program is modeled as a executable thread which is under the control of main thread. The main thread invokes remote threads by using the application interfaces(APIs) provided by pCore Bridge and streaming RPC. The remote invocation of pCore Bridge is asynchronous RPC. Thus, the execution of the executable threads can be sequential or parallel which is determined by the main thread.

The middleware support of pCore Bridge is designed to provide light-weighted communication modules for multicore processors. The APIs comprise several key categories: environment initialization, RPC invocation, signal exchanging, data communication, and resource releasing. The following lists several important APIs provided by pCore Bridge.

- *void* **pb_load**(*char *exec_name*) Load pCore and user programs to SPU. This API should be invoked before any further action.
- *void* **pb_boot**() Boot SPU and start execution of pCore.
- *TASKID* **pb_create**(*const char * p_task, int prio*) Create a remote procedure *p_task* with priority *prio* on SPU.
- *void* **pb_rpc**(*TASKID tid*) Asynchronously invoke a remote procedure of task identifier *tid*. The procedure to be invoked must be created by using *pb_create()* first.
- *void* **pb_wait**(*TASKID tid*) Wait event from previous issued remote procedure of task identifier *tid*. Issuing *pb_wait()* after *pb_rpc* to implement a synchronous remote procedure call,

2.2. Communication Model

Streaming RPC is based on the communication model of RPC which is a promising technique in distributed system. The simple, and ease-to-use mechanism of RPC provides an efficient way of programming multicore processors which allows a remote procedure to be invoked to execute on a different processor. The process that invokes a remote procedure is the client whereas the remote process is called the server. Based on the communication mechanism, streaming RPC is implemented as a middleware to provide a design flow for the multicore applications.

Figure 2 shows the typical streaming RPC operations. Associated with the streaming channels, an RPC request is allowed to transmit data between the client and server by flagging a predefined stream identifier sID to the API

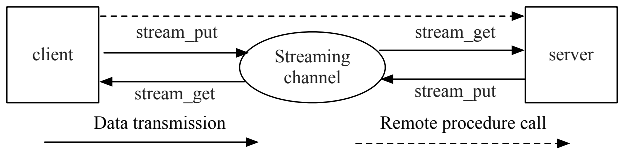

Fig. 2. Typical streaming RPC operations.

$stream\_create(sID)$. Once the channel is established, the sender is able to use $stream\_put()$ to gather the data for streaming communication. The gathered data is then transmitted to the receiver through streaming channel by using $stream\_push()$. To retrieve data from the sender, the receiver uses $stream\_get()$ to gather data from streaming channel. When the gathered data is not needed for further computation, the receiver frees the stream data by using $stream\_pop()$.

2.3. Pushing and Aggregating

Once the streaming channel is initialized, the RPC client and server are then allowed to transmit data through the streaming channel. To support efficient data transmitting, streaming RPC introduces the pushing and aggregating mechanism which precludes call-and-wait of typical RPC. Moreover, the mechanism exploits the underling architectural benefits in data communication, such as direct memory access(DMA).

The thread in which a sender pushes data to the streaming channel is called the transmitter, while the thread in which a receiver aggregates data is called the aggregator. In the communication process, the data streaming is initialized by giving the handler of the transmitter when invoking RPC. For example, to transmit data described in the $transmitter$ when invoking a remote process rf_spu1, the main thread uses $stream\_rpc(rf\_spu1, transmitter)$ which first initializes a streaming channel for the $transmitter$ to perform data streaming, then it invokes the remote process rf_sup1 that is the corresponding receiver (aggregator) of the data streaming.

For example, to transmit s to the receiver through streaming channel of sID, the transmitter uses $stream\_put(sID, s)$ to gather the data elements for transmission. The gathered data is then transmitted to the receiver by pushing the data to streaming line using $stream\_push(sID)$. On the other hand, the aggregator uses $stream\_get(sID, r)$ to retrieve the data element from the streaming channel and store it to r. When the transmitted data is not needed for further computation, the aggregator frees the stream data by using $stream\_pop(sID)$.

Such API design allows the developers to aggregate the stream element before the transmission really takes place. For example, to transmit an data array $list[m]$ from the sender to the receiver, the programmer can transmit the data element one by one if the receiver only requires one or few data elements for computing.

```
for(i = 0; i < m; i++)
{
  stream_put(sID, list[i]);
  stream_push(sID);
}
```

The corresponding aggregator code is as follows:

```
for(i = 0; i < m; i++)
{
  stream_get(sID, list[i]);
  stream_pop(sID);
}
```

On the other hand, if the receiver requires a large portion or even the whole array to proceed computing, the developers can gather all the data required to transmit it to the receiver as shown in the following code:

```
for(i = 0; i < m; i++)
{
  stream_put(sID, list[i]);
}
stream_push(sID);
```

The corresponding aggregator code is listed in the following:

```
for(i = 0; i < m; i++)
{
  stream_get(sID, list[i]);
}
stream_pop(sID);
```

3. SOFTWARE DESIGN PATTERNS

In this section, we describe the streaming RPC programming model with three design patterns: source, pipe, and sink. Figure 3 shows the simplified streaming RPC system components. An application is composed of three structural components: a $source$ that retrieves data from the data source to dispatch it to the remote processes, a $pipe$ that serves as a computation unit, and a $sink$ that aggregates the data for integration. In the following of this section, we follow the description proposed in $Design\ Patterns$ [11] but in more paper format to describe the proposed design patterns.

3.1. Source

A $source$ presents as a data originator to transmit streaming data from a non-streaming RPC data source, e.g. file or device input, to a remote process which requires the data for computation. It is motivated by the requirements for data source while processing multimedia applications, for example, to read data input from the camera device for image encoding or to open a video file for displaying. To achieve this goal, the developers have various way of the implementation and abstraction. A simple way to implement the data source is to transmit data to the remote process by using streaming

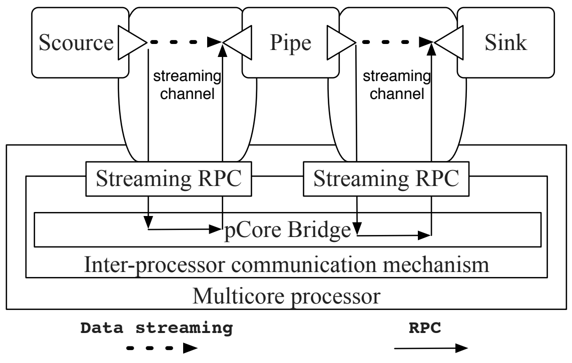

Fig. 3. Streaming RPC system components.

APIs without further partitioning or modifying the application. It is possible to exploit performance in this way, but not much. One of the important features of streaming RPC is to overlap communication and computation in a efficient and flexible way. A better approach to solve the problem is to separate the operation of data transmission into a specialized function called *transmitter*. Figure 4 depicts the structure of a *source*.

The pattern is applicable when:

- A thread needs to transmit data to a remote process as a dispatcher or just a data transmitter.
- To retrieve data from a source file or input device for dispatching/transmitting it to remote processes.
- The source has no input data streaming from a remote process.

The following sample code illustrates the implementation of a *source*. A *source* transmits data element from $a[]$ to a remote function rf_1. A *source* first builds a function *transmitter* that is responsible for transmitting data stored in $a[]$ to the receiver.

```
void transmitter(void * p){
 STREAM_ID sID = 0;
 stream_create(sID);
 int a[] ;
 /* Computation, setting data source */
 for(i){
  stream_put(sID, a[i]);
  stream_push(sID);
 }
}
```

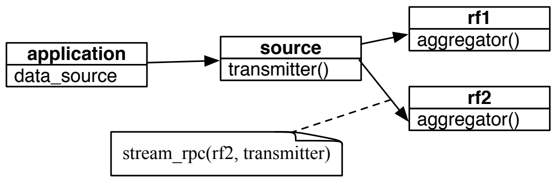

Fig. 4. Design pattern: source.

The source then uses *stream_rpc* to invoke the corresponding remote process rf_1 by giving the handle of *transmitter* and the corresponding remote process rf_1. The *transmitter* is then scheduled as a single concurrent executable unit which transmits the data defined in the function body.

```
void source( )
{
 stream_rpc(rf1, transmitter);
 /* Computation */
}
```

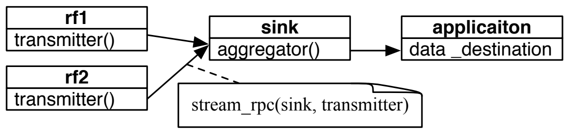

Fig. 5. Design pattern: sink.

3.2. Sink

A *sink* gathers streaming data from remote processes by forming a aggregated data stream to a non-streaming RPC or local destination, e.g. a file, a output device. It is with the motivation for displaying the result of streaming multimedia application and for the applications that requires gathering data from multiple remote processes. The aggregated data is sent to a local file or display devices, such as a monitor, after aggregation. The design of a *sink* is to build a single aggregator for gathering data from other transmitters. Figure 5 depicts the structure of a *sink*.

The pattern is applicable when:

- A thread aggregates streaming data into a file or output to a device, e.g. VGA output.
- The sink does not invoke any streaming RPC for transmitting data to a remote process.

To illustrate, the following sample code lists the usage of sink pattern. A *sink* is a running thread that acts as a data aggregator that get a 32-bits integer from the streaming channel and write the data to a device.

```
void sink( ){
 STREAM_ID rID = 0;
 u32 a;
 stream_create(rID);
 for(i){
  stream_get(rID, a);
  stream_pop(rID);
  iowrite32(a, io_addr)
 }
}
```

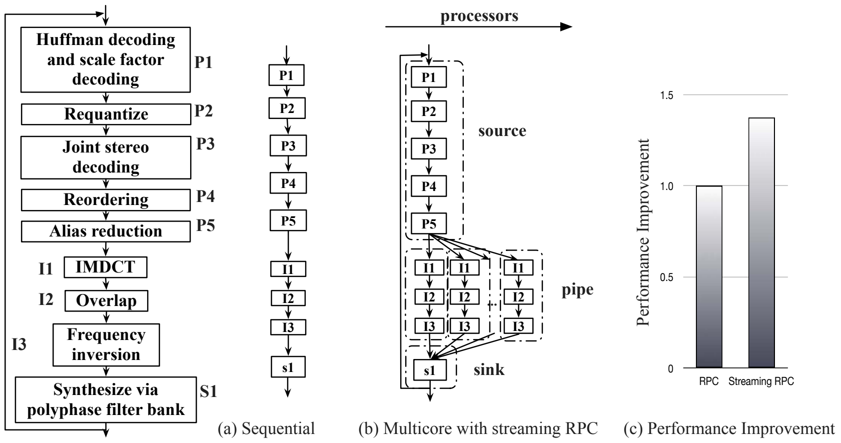

Fig. 6. Simplified control flow of MP3 decoder and evaluated performance.

3.3. Pipe

A *pipe* first gathers streaming data from remote processes to form an aggregated data stream for computation or dispatching. It then transmits the aggregated data stream to remote processes require the data. It is motivated with the pipelined execution which is a promising technique in programming streaming applications on multicore processor. It is often complex to schedule the program for pipelined communication. This pattern provides a simple and ease-to-use way of implementing pipelined parallelism by using streaming RPC. Figure 7 depicts the structure of a *pipe*.

Pipe is applicable when:

- A thread that acts as a pipe or filter that aggregating data streaming from remote processes and then transmits the data to remote processes requires it.

The sample code illustrated a pipe that creates two streaming channel with sID and rID for transmitting and aggregating streaming data.

```
void pipe( ){
 STREAM_ID sID = 0;
 STREAM_ID rID = 1;
 stream_create(sID);
 stream_create(rID);
 u32 a;
 while(true){
  stream_get(sID, a);
  stream_pop(sID);
  /* Compute a */
  stream_put(sID, a);
  stream_push(sID);
 }
}
```

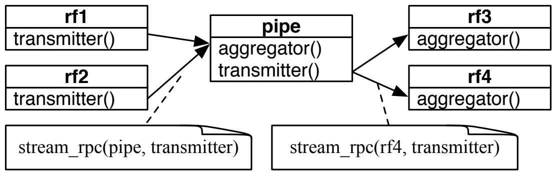

Fig. 7. Design pattern: pipe.

4. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of streaming RPC. The early evaluation was performed on the parallel architecture core (PAC) [12] which comprises a 300-MHz ARM 926EJ-S processor with 32-KB cache as MPU, and a novel digital signal processor(DSP) running at 250-MHz as SPU. An MP3 decoder is implemented to evaluate the performance of streaming RPC over conventional RPC in multimedia applications. Figure 6 (a) shows the simplified flow of a conventional MP3 decoder. As shown in Figure 6 (b), about 45% of the decoder is partitioned to run on DSP including IMDCT, overlap, and frequency inversion. In the stage, the input data are divided into 32 subbands for further processing. The decoding processes of subbands are independent, thus reveals a potential parallelism in the program. Thus, we partitioned the MP3 decoder for the dual-core processor where MPU and DSP are responsible equally 16 subbands for the decoding. The performance of the MP3 decoder was improved by 38% over conventional RPC implementation as shown in Figure 6 (c).

Although the streaming RPC is able to attain high performance improvement on multimedia application compared to naive use of RPC, observing from the implementation, the discrepancies in processing speed and I/O latency between processors affects the performance of the applications. Such discrepancies results in the differences between the production rate of the sender and the consumption rate of the receiver. Thus, the thread with faster processing speed has to wait for data communication. To avoid the blocking overhead, the streaming RPC adopts a data-driven operations. Instead of blocking, thread that waits for data is suspended by a streaming monitor associated to a streaming channel. Although the data-driven mechanism solved the problem of blocking, it increases the internal handshaking while performing streaming RPC. A thread with fast processing speed suffers from frequent triggering for suspension and waking-up. To avoid such unnecessary overhead, the streaming channel is associated with a threshold number. Instead of waking up the suspended thread immediately, the streaming monitor waits until the number of transmitted streaming data is over the threshold value. Figure 8 shows the effects of threshold number in internal handshaking times by running an MP3 decoder with different threshold values. As the figure depicted, by setting a larger number of threshold, the performance of the application is improved as the internal handshaking reduces.

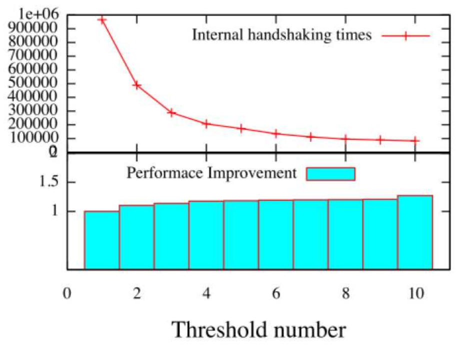

Fig. 8. Internal handshaking times reduction.

5. CONCLUSION

Supporting streaming programming for the remoting programming paradigms is a novel way to achieve performance on multicore architectures and it benefits data-intensive applications by utilizing the potential parallelism with increasing the efficiency of data transmission. In this paper, we presented a mechanism called streaming RPC to support data streaming on remoting with a novel programing model. We also presented the design patterns of streaming RPC on multicore processors. The evaluation showed that streaming RPC is able to attain great improvement on multimedia applications.

Acknowledgment

This research was supported in part by the NSC under grant nos. NSC 97-2218-E-007-009-, NSC 96-2220-E-007-031- and NSC 96-2218-E-007-013-, 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.

6. REFERENCES

[1] Wayne Wolf, "The future of multiprocessor systems-on-chips," in *Proceedings of the 41st annual conference on Design automation*, 2004, pp. 681–685.

[2] Grant Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd annual conference on Design automation*, 2006, pp. 274–279.

[3] G. K. Thiruvathukal, L.S. Thomas, and A. T. Korczynski, "Reflective remote method invocation," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 911–925, 1998.

[4] Rajeev R. Raje, Joseph I. William, and Michael Boyles, "An asynchronous remote method invocation (ARMI) mechanism for Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1207–1211, 1997.

[5] 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*, 1999, pp. 173–182.

[6] 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*, 2004, pp. 1391–1395.

[7] William Thies, Michal Karczmarek, and Saman Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of Computational Complexity*, 2002, pp. 179–196.

[8] 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*, vol. 23, no. 3, pp. 777–786, 2004.

[9] 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*, 2006, pp. 53–61.

[10] Kun-Yuan Hsieh, Yung-Chia Lin, and Jenq Kuen Lee, "Enhancing microkernel performance on VLIW DSP processors via multiset context switch," *Journal of VLSI Signal Processing Systems*, 2007.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Deisng Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[12] 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*, 2006, pp. 289–292.