Architectural Support for Reducing Parallel Processing Overhead in an Embedded Multiprocessor

Jian Wang, Joar Sohl and Dake Liu
 Department of Electrical Engineering
 Linköping University, Sweden
 Email: {jianw, joar, dake}@isy.liu.se

Abstract—The host-multi-SIMD chip multiprocessor (CMP) architecture has been proved to be an efficient architecture for high performance signal processing which explores both task level parallelism by multi-core processing and data level parallelism by SIMD processors. Different from the cache-based memory subsystem in most general purpose processors, this architecture uses on-chip scratchpad memory (SPM) as processor local data buffer and allows software to explicitly control the data movements in the memory hierarchy. This SPM-based solution is more efficient for predictable signal processing in embedded systems where data access patterns are known at design time. The predictable performance is especially important for real-time signal processing. According to Amdahl's law, the non-parallelizable part of an algorithm has critical impact on the overall performance. Implementing an algorithm in a parallel platform usually produces control and communication overhead which is not parallelizable. This paper presents the architectural support in an embedded multiprocessor platform to maximally reduce the parallel processing overhead. The effectiveness of these architecture designs in boosting parallel performance is evaluated by an implementation example of 64×64 complex matrix multiplication. The result shows that the parallel processing overhead is reduced from 369% to 28%.

Index Terms—Parallel DSP; Multiprocessor; Control overhead; Communication overhead; Matrix multiplication

I. INTRODUCTION

Parallel processing has emerged as an area to provide satisfactory performance to meet the ever-increasing computation requirements in various applications. Different levels of parallelism are explored to design high performance embedded DSP processors. The Very Long Instruction Word (VLIW) and superscalar architectures take advantage of instruction level parallelism (ILP) by executing multiple instructions in parallel using different hardware resources. For multimedia applications, data level parallelism (DLP) is used by Single Instruction Multiple Data (SIMD) extension architecture, which applies the same arithmetic operation on a group of data by a single vector instruction. The task level parallelism (TLP) is widely used in heterogeneous system-on-chip (SoC) architectures by designing function specific hardware modules and integrating those modules into a system by a non-standard or standard interconnection architecture such as AMBA [1]. The SoC approach is good for a domain-specific applications and there have been many industrial successes based on this architecture, for example TI OMAP platform [2] and STi7100

HDTV set-top-box SoC [3] from ST Microelectronics. However, when the computing complexity of application algorithm increases and more functional units are needed in one SoC system, the high implementation cost and power consumption become the disadvantage of the heterogeneous SoC solution. Recently, another trend of task level parallel processing is towards a multiprocessor architecture with multiple processor cores on a single chip. For embedded DSP computing, the chip multiprocessor architecture is further optimized by decoupling system control and data processing and running them in two kinds of processors, which leads to a host-multi-SIMD architecture. This architecture consists of one host controller and multi SIMD coprocessors. The host controller focuses on sequential task execution and the SIMD processors, with a parallel data path, focus on data processing. A well-known example of this architecture is the IBM, Sony and Toshiba Cell broadband engine [4]. For embedded signal processing, especially in real-time systems, the host-multi-SIMD architecture usually uses scratchpad memory (SPM) instead of data cache as the processor local data buffer. The SPM allows software to explicitly control data movements from external main memory to processor local memories and data communications between SIMD processors. The use of SPM makes the performance of memory subsystem predictable and enables the software developers to control the data communications to hide the memory access latency.

The speedup of multiprocessor parallel processing is limited by the time required for the execution of sequential fraction of the program. The basic concept of parallel processing is to complete a large task quicker by dividing it into several small tasks and executing them simultaneously using more resources. However, the maximum benefit is not easily attainable from a parallel solution. It requires various issues to be considered. A significant factor that affects parallel processing performance is the amount of parallelism presented in the algorithm. Amdahl's law [5], in its simplest form, models the relationship between execution time versus number of processors N :

$$T(N) = t_s + \frac{t_p}{N} \quad (1)$$

where t_s is the strictly sequential time section (non-parallelizable part) and t_p is the perfectly parallelized time

section. This simple model assumes that the problem size remains the same when parallelized. Let η be the fraction of non-parallelizable part in an algorithm:

$$\eta = \frac{t_s}{t_s + t_p} \quad (2)$$

The speedup of using N processors is:

$$S(N) = \frac{T(1)}{T(N)} = \frac{1}{\eta + \frac{1-\eta}{N}} = \frac{N}{\eta(N-1) + 1} \quad (3)$$

$$\lim_{N \rightarrow \infty} S(N) = \frac{1}{\eta} \quad (4)$$

As shown in Equation 4, the parallel speedup is limit by the sequential fraction η . For example, if an algorithm is with a sequential part of 10% and parallel part 90%, the speedup of implementing this algorithm on a parallel architecture of 8, 16, and 32 processors is 4.71, 6.40 and 7.80. Double the number of processors could not give a double performance.

Parallel processing overhead has critical impact on the performance of host-multi-SIMD multiprocessor. This overhead coming from architecture design and hardware implementation contributes to the non-parallelizable part which is to be executed sequentially. In practice, Equation 1 is an incomplete model for evaluating parallel processing performance because it ignores several significant effects such as control and configuration, memory access, network contention and others. To include these practical effects and evaluate parallel processing performance on different architectures, the Amdahl's law may be modified by adding an overhead function $T_o(N)$:

$$T(N) = t_s + \frac{t_p}{N} + T_o(N) \quad (5)$$

The modified Amdahl's law is used for evaluating parallel processors in practice [6]. The overhead function $T_o(N)$ for different parallel architectures and memory models are described in [7]. We use the same method to analyze parallel processing performance on host-multi-SIMD multiprocessor architecture. In this architecture, the overhead function $T_o(N)$ is affected by the hardware design for coprocessor control and synchronization, the data communication mechanism, as well as the task distribution algorithm used to control the assignment of processors. When a task is dispatched on more processors, usually more control, synchronization, and data communications will be introduced by parallelization. We classify these overheads in two major groups, the control overhead and the communication overhead:

$$T_o(N) = T_{ctrl} + T_{comm} \quad (6)$$

The main task of host-multi-SIMD parallel architecture design is to minimize these two kinds of overhead.

The ePUMA (embedded Parallel DSP platform with Unique Memory Access) project at Linköping University Sweden is developing an embedded parallel DSP platform targeting on low-power (sub1-watt) and high performance (100 GOPS) signal processing for future media and communication applications. The ePUMA platform uses the host-multi-SIMD with architectural optimizations to minimize parallel processing

overheads. We are working on the following three design challenges to make ePUMA better than other similar architectures on performance over silicon cost, power consumption, and tool-chain support:

- Memory subsystem design to maximally reduce data access latency. This includes an enhanced Direct Memory Access (DMA) controller design and a power efficient interconnection architecture.
- Conflict free multi-bank memory access using P3RMA (Predictable Parallel memory architecture for Predictable Random Memory Access) technology [8] to achieve high SIMD processing efficiency.
- User-friendly parallel programming model using kernel based programming and separated data access kernels and computing kernels.

A. Related Works

Many industrial and academic efforts have been spent on reducing parallel processing overhead in multiprocessors [4], [9], [10], [11]. Two widely used techniques are double buffering and DMA controller. One way of implementing double buffer is by using a dual port SRAM, which supports two independent memory accesses at the same time. However, large size dual port memory is hard to achieve high speed, plus commercial considerations, most distributed memory systems use conventional single port SRAM as the local data buffer. There are mainly two approaches of using single port SRAM for double buffering, one way is to implement a hardware memory controller which does arbitration of two memory accesses, for example in the Cell broadband engine [4]. Usually one memory access is issued by the processor core which has a higher priority and another memory access is from the data communication interface such as DMA controller. In this way, DMA controller transfers data when the processor is not accessing the same memory, otherwise the DMA data transaction is blocked. Another approach of realizing double buffering is by using memory swapping, for example in the BBP baseband processor [9]. The connection between processing units and memory modules is switched during ping-pong swapping. The memory controller approach has an advantage that the software developers do not need to take care of ping-pong swapping. They use different memory space to allocate ping-pong buffers. The second approach is more power efficient. Once a connection is setup, no access arbitration is required. Several design optimizations have been developed to enhance the DMA controller [10], [11]. Most of them are designed to enrich data access patterns for various applications.

The rest part of this paper presents the ePUMA architectural support for reducing control and communication overhead in parallel processing. It is organized as follows. First, an overview of the ePUMA platform is given in Section II. Section III describes hardware design to reduce the control path overhead. Section IV is about ePUMA architectural support for reducing data communication overhead. Then we use a large complex matrix multiplication example in Section V to

Fig. 1. ePUMA chip multiprocessor architecture

illustrate parallel implementation of an application program on ePUMA platform. Section VI analyzes the parallel processing overhead in the matrix multiplication algorithm and evaluates the effectiveness of the ePUMA architecture design techniques in reducing the overhead. Section VII at the end concludes the paper.

II. AN OVERVIEW OF EPUMA

As shown in Figure 1, the ePUMA platform uses a heterogeneous single chip multiprocessor architecture consisting of one host controller and eight SIMD coprocessors. It decouples program control and data processing to maximize the parallel processing performance. The host controller uses a 16-bit RISC processor to control the memory subsystem and the SIMD coprocessors to execute independent data transfer and computation. The eight 8-way 16-bit SIMD coprocessors deliver the majority of the system's computing power. The ePUMA memory subsystem, including the centralized DMA controller, the combined *Star* and *Ring* interconnection network and the SIMD local memory system, is designed to maximally reduce the data access overhead for parallel processing. Each SIMD processor has a local 128-bit program memory (PM), one 128-bit constant memory (CM), and three 8×16 -bit vector memories (VM) that support double buffering. Eight network nodes (N0 to N7) are used for on-chip interconnection. The network node is circuit switch based and the switching is controlled by host processor.

ePUMA uses a kernel based parallel programming model. The host processor executes the main program stream, which schedules and controls the kernel execution on SIMD coprocessors. The SIMD program is loaded to SIMD local PM by a DMA subroutine in host program. The host program implements DMA subroutines for data transactions of program binary code, constant values, and vector data blocks. ePUMA separates computing kernels and communications kernels. The SIMD kernel program which focuses on data processing only views one CM and two VMs with its local address mapping. The SIMD processor implements very rich vector computing instructions, but simple flow control instructions for branch, interrupt and synchronization. The communication kernels are executed on host processor to control DMA data transfer, ping-pong buffer swapping and network configuration.

Fig. 2. Host controller general IO interface and parallel IO peripheral

III. REDUCING CONTROL PATH OVERHEAD

This section introduces ePUMA architectural design to reduce the control path overhead, the T_{ctrl} in Equation 6. Figure 2 shows ePUMA host processor control link. Through its general IO interface, the host processor can control and configure the SIMD coprocessors, the DMA controllers and the on-chip network. Each SIMD coprocessor has a dedicated IO connection to the host controller. It is a common case in parallel computing that one task is dispatched to multiple SIMD processors which run the same program but on different data blocks. The execution control of all the SIMD processors and configuration of their local memory systems (e.g. ping-pong swapping) are the same at a certain execution point. The repeated IO operations on multiple IO ports increase the sequential execution time.

A. Configurable Parallel IO

To reduce control path overhead, a configurable parallel IO module is designed to support simultaneous IO write to selected SIMDs. This module supports multiple configurations of SIMD sets. Each set of SIMD selection is programmed by software. This parallel IO module has its own host IO interface, as shown in Figure 2. The use of parallel IO for reducing control path overhead is illustrated by the example in Figure 3, where the parallel IO module is configured to select all eight SIMDs. In this example, one *out* instruction in the left side equals to eight instructions to the right.

	out	0x0701, r0	; SIMD 0
	out	0x0801, r0	; SIMD 1
	out	0x0901, r0	; SIMD 2
	out	0x0a01, r0	; SIMD 3
	out	0x0b01, r0	; SIMD 4
	out	0x0c01, r0	; SIMD 5
	out	0x0d01, r0	; SIMD 6
	out	0x0e01, r0	; SIMD 7
<hr/>			
<i>;Parallel I/O</i>	out	0x0601, r0	=

Fig. 3. Parallel IO instruction accelerates SIMD coprocessor control and configuration

IV. REDUCING DATA COMMUNICATION OVERHEAD

Data communication overhead is a vital issue when designing multiprocessor systems. Different memory architectures

have been developed and corresponding programming models are used to support on-chip data communications. In shared memory system using uniform memory access (UMA), the *access arbitration* and *process scheduling* are the major data communication overheads. In partially shared memory model using non-uniform memory access (NUMA), *barrier synchronization* and *remote memory access* are two main contributors to the overhead function. In distributed memory system, the overhead is mainly contributed by the *DMA communication cost*, which includes the transfer setup prolog, the data transaction payload, and the task finish epilog.

ePUMA multiprocessor architecture uses a distributed memory system. The data communications from off-chip main memory to SIMD local memory and between SIMD local memories are through the DMA controller. DMA data transfer latency is the major communication overhead on ePUMA parallel platform. This section introduces ePUMA architecture design techniques to minimize data communication overhead.

A. Ping-Pong Buffer

Ping-pong buffer is used to hide memory access latency by overlapping data transaction with processor execution. While the processor is working on the current data in the *ping* buffer, the next block of data are transferred to the *pong* buffer at the same time. A fast ping-pong swap is executed when the computation switches to the next data block. In ePUMA system, SIMD processor local memory uses three VMs to support ping-pong buffering. At runtime, two VMs are connected to SIMD core for kernel computing. Another VM is connected to the network interface for data transaction either to other SIMD processor or to the off-chip main memory. A control register is used to configure the connections of the three VMs to the three data access ports.

Fig. 4. SIMD local vector memory with ping-pong buffer

B. Overlapped DMA Task Configuration and Data Transaction

The control process of DMA communication includes three steps: setup a task, transfer data, and finish the task. All these steps produce communication overhead in time. The setup of DMA task includes configurations of source memory and its address, target memory and its address, transfer size, bit manipulation, source and target addressing patterns, etc. The setup latency depends on the task size and data width of the host IO interface. DMA data transfer payload decides the overhead time consumed on data movement. The factors that affect this transaction latency include the source and target memory data width and the bus width. DMA controller finishes one transaction by sending interrupt to the host processor or

changing the value of its flag register. The host processor has overhead in finishing a DMA task either from responding to DMA interrupt or from iteratively checking the flag register.

To minimize the task setup overhead, we design a DMA controller that supports overlapped task configuration and data transaction. This is achieved by implementing a task memory which can hold multiple pre-configured tasks in DMA controller. The host processor starts a DMA transaction by sending a start signal together with a task ID. Then the DMA controller looks into the task memory using the provided ID for the current task and starts executing the data transaction immediately. The task configuration can be done during a previous DMA data transaction as shown in Figure 5.

Fig. 5. Overlapping DMA task configuration with data transaction

C. DMA Auto-initialization

DMA auto-initialization is implemented to reduce the host processing overhead on finishing a DMA task and triggering the next one. The auto-initialization of next task is configured by setting the corresponding parameters in DMA task configuration, which includes the auto-initialization enable flag and the next task ID. It is configurable to choose whether or not to send an interrupt when DMA finish. Also the task self-update of some parameters, for instance the source/target base-address and source/target ports, are supported. As shown in Figure 6, with DMA auto-initialization, the next data transaction starts immediately after the current one finishes.

Fig. 6. DMA auto-initialization

D. Data Broadcasting

DMA broadcasting is used when same data are required to be transferred to multiple destinations for computing on different processors. We design to support data broadcasting in ePUMA platform to save redundant data transactions so as to reduce communication overhead. The source memory for broadcasting can be the off-chip main memory or any one SIMD local VM. The data block is transferred to the selected SIMDs in a broadcast way by issuing one DMA task. Figure 7 illustrates the overhead saving by a DMA broadcasting to four destination memories.

Fig. 7. DMA broadcasting

Fig. 8. Task distribution of 64×64 complex matrix multiplication on eight SIMD processors

V. 64×64 COMPLEX MATRIX MULTIPLICATION

Matrix-matrix multiplication is the kernel of many mathematical algorithms. Complex value matrix multiplication is widely used in baseband processing in wireless communication and radar systems. Hence a fast matrix-matrix multiply benefits these applications that require real-time processing performance. In this section we present a parallel implementation of matrix multiplication of two 64×64 complex matrices on ePUMA platform, using all eight SIMD coprocessors. The goal of ePUMA architecture design is to maximally reduce control and communication overhead for parallel processing. The main difficulty with this large matrix multiplication implementation is the data access latency. The following two subsections describe the parallel task distribution and scheduling. In the next section, we analyze the effectiveness of ePUMA architectural support on reducing parallel processing overheads.

A. Task Distribution on Multiprocessors

The 64×64 complex matrix multiplication is implemented on ePUMA platform using eight SIMD processors. The computing task is divided equally and assigned to eight SIMD processors, as shown in Figure 8. Each SIMD processor works on a subset of result matrix C, an 8×64 row block. The input data used by each SIMD processor for computing includes a row block of matrix A and the full matrix B. Since there is no data dependency between SIMD processors, they can run in parallel.

B. Task Scheduling

The host processor executes the top level program for coprocessor configuration, program loading, data communication, and execution control. The pseudo-code in Figure 9 shows the task scheduling of parallel matrix multiplication on ePUMA multiprocessor. It consists of three parts: the prolog part, the kernel iterations, and the epilog part. Each SIMD kernel computes one 8×4 result block in C, using SIMD instruction for complex vector-vector multiplication and

```

/* Prolog */
DMA broadcast kernel program to SIMD[0..7];

for i in range (0,7){
    DMA input matrix A[i] to SIMD[i];
}

DMA broadcast matrix B[0] to SIMD[0..7];

do SIMD[0..7] kernel execution;

/* Iteration */
for iter in range (0,13){
    do SIMD[0..7] kernel execution;
    DMA broadcast next column block from matrix B to SIMD[0..7];
    for i in range (0,7){
        DMA a result block of C from SIMD[i] to main memory;
    }
}

/* Epilog */
do SIMD[0..7] kernel execution;
for i in range (0,7){
    DMA a result block of C from SIMD[i] to main memory;
}

for i in range (0,7){
    DMA the last result block of C from SIMD[i] to main memory;
}

/* Finish */

```

Fig. 9. Pseudo-code of parallel matrix multiplication on ePUMA

accumulation. Eight SIMD processors run in parallel in every iteration. So 16 iterations are required to compute the result. It can be seen from the pseudo-code that the software also programs the DMA tasks for data transactions. To minimize the memory access latency, we use DMA broadcasting to reduce redundant data transactions and ping-pong buffering to hide data communications behind kernel computing. Other architecture supports are not shown in the pseudo-code, but they are applied in implementation, for example, the loop of DMA tasks in the prolog uses auto-initialization to save DMA task switching time.

VI. EXPERIMENTAL RESULTS

We have built a cycle true and pipeline accurate ePUMA simulator using C++ and python wrapper. The simulator implements all the architecture features introduced in this paper that reduce parallel processing overhead. The simulator assumes that the off-chip memory access has a throughput of 128-bit per cycle. The host processor is programmed in C with extension functions and inline assembly. The SIMD kernel program is developed in assembly to get the best data processing performance. The 64×64 complex matrix multiplication is implemented in the simulator with applying the architecture supports step by step, as shown in Figure 10. The kernel computing time, 8544 cycles, is the same for all the test cases as listed in Table I. It equals to the time of running one SIMD kernel for 16 iterations; each kernel execution takes 534 cycles. The table also lists the overhead in cycle counts and percentage, from which we can see that ePUMA architecture design reduces the parallel processing overhead from 369% to 28% for the matrix multiplication example. From the step-by-step optimization, it can be seen that the ping-pong buffer is the most effective approach. The DMA broadcasting also reduces a certain amount of overhead in this example. According to Amdahl's law, a small sequential part

Fig. 10. Parallel processing overhead for matrix multiplication on ePUMA

TABLE I
EVALUATION RESULTS

Tests	Kernel computing (cycles)	Overhead (cycles)	Overhead (%)
No optimization	8544	31555	369.3%
Ping-pong buffer		13962	163.4%
DMA broadcasting		8460	99.0%
DMA task memory		5557	65.0%
Parallel IO		3190	37.3%
DMA auto-init		2429	28.4%

of a parallel algorithm has significant impact on the overall performance, so the other approaches are also important for improving parallel performance.

VII. CONCLUSION

The ePUMA project at Linköping University aims at designing a low power and high performance embedded multi-processor for future media and communication applications. This paper presents the ePUMA architectural support for reducing control and communication overhead in parallel processing. The architecture designs are described in detail including the configurable parallel IO, ping-pong buffering, overlapping DMA configuration with data transaction, DMA auto-initialization, and DMA broadcasting. These architecture designs are evaluated by an implementation of a large matrix multiplication on eight SIMD coprocessors. The evaluation result shows that the ePUMA architectural design effectively reduces the parallel processing overhead for the matrix multiplication example. Several projects in cooperation with other algorithm groups at the university are in progress to implement more computing kernels and applications on the ePUMA platform to further improve the architecture design.

ACKNOWLEDGEMENTS

The authors would like to thank SSF, Swedish Foundation for Strategic Research, for the support of this project.

REFERENCES

- [1] "AMBA system IP & design tools," <http://www.arm.com/products/system-ip/amba/>.
- [2] *Wireless Terminals Solution Guide*, Texas Instruments, 2007.
- [3] *Low cost HDTV set-top box decoder for H.264/AVC and MPEG-2*, ST Microelectronics, 12 2006.
- [4] M. Gschwind, "The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor," *Int. J. Parallel Program.*, vol. 35, no. 3, pp. 233–262, 2007.

- [5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485.
- [6] A. H. Karp and H. P. Flatt, "Measuring parallel processor performance," *Commun. ACM*, vol. 33, no. 5, pp. 539–543, 1990.
- [7] X. Zhang and X. Qin, "Performance prediction and evaluation of parallel processing on a numa multiprocessor," *IEEE Trans. Softw. Eng.*, vol. 17, no. 10, pp. 1059–1068, 1991.
- [8] D. Liu, *Embedded DSP Processor Design, Volume 2: Application Specific Instruction Set Processors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ch. 20.
- [9] A. Nilsson, E. Tell, and D. Liu, "An 11 mm², 70 mw fully programmable baseband processor for mobile WiMAX and DVB-T/H in 0.12μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 44, pp. 90–97, February 2008.
- [10] *TMS320VC5501/5502 DSP Direct Memory Access (DMA) Controller Reference Guide*, Texas Instruments, 2005.
- [11] *PrimeCell DMA Controller (PL080) Technical Reference Manual*, ARM, 2005.