Int J Parallel Prog (2014) 42:875–899
DOI 10.1007/s10766-013-0256-7

Parallelizing Complex Streaming Applications on Distributed Scratchpad Memory Multicore Architecture

Shin-Kai Chen · Cheng-Yu Hung · Ching-Chih Chen · Chih-Wei Liu

Received: 17 November 2010 / Accepted: 22 June 2013 / Published online: 4 July 2013
© Springer Science+Business Media New York 2013

Abstract Multicore processors can provide sufficient computing power and flexibility for complex streaming applications, such as high-definition video processing. For less hardware complexity and power consumption, the distributed scratchpad memory architecture is considered, instead of the cache memory architecture. However, the distributed design poses new challenges to programming. It is difficult to exploit all available capabilities and achieve maximal throughput, due to the combined complexity of inter-processor communication, synchronization, and workload balancing. In this study, we developed an efficient design flow for parallelizing multimedia applications on a distributed scratchpad memory multicore architecture. An application is first partitioned into streaming components and then mapped onto multicore processors. Various hardware-dependent factors and application-specific characteristics are involved in generating efficient task partitions and allocating resources appropriately. To test and verify the proposed design flow, three popular multimedia applications were implemented: a full-HD motion JPEG decoder, an object detector, and a full-HD H.264/AVC decoder. For demonstration purposes, SONY PlayStation®3 was selected as the target platform. Simulation results show that, on PS3, the full-HD motion JPEG decoder with the proposed design flow can decode about 108.9 frames per second (fps) in the 1080p format. The object detection application can perform real-time object detection at 2.84 fps at 1280 × 960 resolution, 11.75 fps at 640 × 480 resolution, and 62.52 fps at 320 × 240 resolution. The full-HD H.264/AVC decoder applications can achieve nearly 50 fps.

S.-K. Chen (✉) · C.-Y. Hung · C.-C. Chen · C.-W. Liu
Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan
e-mail: skchen@twins.ee.nctu.edu.tw

Keywords Parallel programming · Streaming application · Multicore architecture · Distributed scratchpad memory architecture

1 Introduction

Owing to the so-called brick wall [8], i.e., the memory wall, the instruction-level parallelism (ILP) wall, and the power wall, performance gains achieved by increased operating frequency have been greatly diminished. This limitation brings a dramatic change to microprocessor design. To meet the ever-increasing demand in computing power, modern systems on a chip adopt multicore solutions. Intuitively, a multicore processor can be considered as a group of well-organized cores. The cores may be homogeneous or heterogeneous, and they can be coupled together tightly or loosely. The most common interconnection network designs include the bus, ring, mesh, and crossbar. Polling, mailbox, or shared memory techniques are used to implement inter-processor communication (IPC). The performance achieved by using a multicore processor is strongly dependent on the target software and its implementation. However, multicore programming poses new challenges.

Conventional software programming was typically sequential, based on C and similar languages, resulting in sequential execution, flattened shared memory, and implicit data communications. Thus, if a conventional program is not recoded, it will run on a single core, and its performance can only be improved by taking advantage of ILP and sub-word data-level parallelism (DLP), e.g., single-instruction multiple-data (SIMD). If the resulting latency cannot meet the real-time constraints, we can achieve increased performance and efficiency from the multicore processor, by re-coding the program as a parallel, multithreaded version. Instead of sequential execution, tasks are allocated to different cores and are processed concurrently.

Flattened shared memory with cache support is utilized in today's dual-core and quad-core platforms. Legacy software programming follows the shared memory multiprocessor programming model, which is rather simple. The centralized flattened shared memory is ideal for inter-core data exchange. Unfortunately, the bandwidth of the shared memory and the cache coherence logic can cause bottlenecks. As a result, hardware complexity grows exponentially as the number of cores increases. Consequently, distributed or hybrid memory systems now rely on special memory transfer mechanisms, e.g., direct memory access (DMA) [8]. Similar performance can be attained if memory transfer operations are inserted in the right place.

The programmer is now responsible for generating concurrent tasks, managing non-uniform memory access explicitly, and considering coherence and synchronization between cores. In addition to the application-specific characteristics, hardware-dependent factors make programming difficult, time-consuming, and error-prone. Workload balancing, synchronization, and IPC overhead complicate software programming and seriously degrade performance. These factors are always critical in achieving the peak performance of the distributed scratchpad memory multicore processor.

Data partitioning and function partitioning are two general approaches normally used to analyze an application [28]. The data partition scheme exploits DLP to create concurrent tasks. To achieve DLP, the original data stream is partitioned into disjoint or parallel segments. These segments are served by different cores simultaneously. Usually, the cores share the same program. In the case of an N-core processor, if the N-parallel DLP technique is applied on an application, then the workload of each core is reduced to $1/N$, and the latency becomes $1/N$. The data partition method usually pursues extra data parallelism by raising the granularity of the data segments. Consequently, each core requires a larger buffer or memory to hold the temporary data. If the local memory is not large enough, performance could be degraded significantly due to the additional movement of data.

On the other hand, function partitioning decomposes the application into tasks (or processes) according to its functionality. The core is then in charge of serving one or more tasks. With step-by-step processing, the program could be executed and completed in a software pipeline. However, this may introduce IPC overhead between cores. The IPC overhead is frequently the most significant factor in the performance of the multicore processor. In data partitioning, cores exchange information only if true data dependencies exist among the different segments; however, function partitioning always suffers from IPC overhead between tasks. Different agglomeration and allocation of tasks result in different IPC overhead. By reducing the IPC overhead or by hiding it in the function partition scheme, higher multicore performance can be achieved. Furthermore, when the number of cores change from N to $2N$, the designer can easily readjust the segments in the data partition scheme. However, repartitioning and reallocation of tasks may be necessary in the function partition scheme if better performance and efficiency is required.

Workload balancing and IPC overhead are important issues regarding multicore performance. Unfortunately, these primary factors are usually at odds with each other and must be balanced. If we simply make tasks small and apply dynamic task dispatching, workload balancing is conceptually fulfilled as long as the application provides enough concurrency. On the other hand, to minimize IPC overhead, we would attempt to run the program on a single core, as long as the necessary data fits in the local memory. The balancing process, which depends on software programming and hardware porting, is rather complicated and time-consuming.

Programming models facilitate multicore programming, because they hide numerous hardware details and serve as an abstraction layer to bridge software and hardware. Using a programming model, programmers can easily develop a parallel program on a simpler virtual platform. The Kahn process network [15] or streaming programming model [17] is used with streaming multimedia applications. In this model, programmers are responsible for decomposing the application into a data-flow process network, which is a graphical structure containing data and processes. The data primitive is a stream, i.e., a continuously ordered set of data, and the FIFO-communicated processes are expressed in terms of the operations applied on the stream. The operations include data load/store, mathematical computations, and data communication through the interconnection network. Depending on its input queue, the process performs its functions when a stream is available and places the resulting stream in

its output queue, typically the input queue of another process. In this manner, the processes in the stream programming model can be activated simultaneously. Finally, the processes are allocated to the physical cores, and the FIFO queues between the processes are managed by the on-chip interconnection network and memory subsystem. The streaming programming model simply defines the task partition and the interactions between tasks, which is a general programming model suitable for porting streaming multimedia applications on all kinds of multicore processors. We rely on the streaming programming model to analyze general task decomposition and allocation. To perform specific optimizations, advanced programming models are required.

The streaming programming model simplifies the programming into a partition problem and an allocation problem. An application is partitioned into numerous components, which are then allocated to physical cores. However, the process to gain efficiency in partition and allocation is still complicated. The performance of a parallelized streaming application on a multicore platform T_{tot} can be modeled as follows:

$$\begin{aligned} \mathsf{T_{tot}} &= \mathsf{T_{seq}} + \mathsf{T_{par}} \\ &\quad \mathsf{T_{seq}} + \mathsf{MAX}\left[\mathsf{T_{proc}}\right] \\ &\quad \mathsf{T_{seq}} + \mathsf{MAX}\left[\mathsf{T_{cmp}} + \mathsf{T_{comm}}\right] \\ &\quad \mathsf{T_{seq}} + \mathsf{MAX}\left[\Sigma(\mathsf{T_{comp}} + \mathsf{T_{overhead}}) + \mathsf{T_{comm}}\right] \end{aligned}$$

An application may include a sequential part T_{seq} and a parallel part T_{par}. Only a parallel part can be partitioned and accelerated by a multicore. When porting to multicore processing, the time spent processing the parallel part is reduced, depending on the processor with the heaviest workload, which is T_{proc}. The performance is degraded by both the computation time T_{cmp} and the data communication T_{comm}. The computation time is the summation of the execution time of each component T_{comp} allocated to the critical processor. Unfortunately, the partition process creates computation overhead. Different components are unable to share intermediate data, which can cause extra execution time $T_{overhead}$.

In order to raise the performance, T_{seq}, $T_{overhead}$, and T_{comm} must be reduced and the load must be balanced among processors. The T_{seq} is strongly dependent on the application, and algorithmic transform can help reduce T_{seq}. Communication overhead T_{comm} is usually the critical obstacle to parallel programming. To address the problem, most multicore architectures provide hardware support, such as high-speed interconnection networks and DMA chips. High-speed interconnection networks utilize a complex topology and a strong arbiter to allow high-volume concurrent data transfer, which is usually supported to further raise the bandwidth. DMA is a standalone data transfer engine, which can perform data transfer while processors take care of computation. Both these devices prefer massive continuous memory access and can highly reduce T_{comm}. The load balancing and $T_{overhead}$ are managed by the partition-and-allocation process. The number of components and the level of granularity resulting from partition are the most crucial factors in load balancing. The granularity of each component depends on the application, which is

difficult to control; therefore, the number of components is usually raised to improve load balancing. However, large numbers of components complicate the allocation process, which may lead to unnecessary data communication and increased computation overhead $T_{overhead}$. On a multicore platform with n cores, the theoretical n processing time reduction is possible only if most of the application is parallelizable and can be perfectly partitioned into n equal and independent components. However, porting most applications may not be practical, especially when n is high.

In this paper, based on the streaming programming model, we parallelize complex streaming applications on a distributed scratchpad memory multicore architecture. Various hardware-dependent factors and application-specific characteristics are involved in generating an efficient task partition. A workload-balancing process-allocation algorithm is applied to guarantee the best performance for multicore processing. For our experiment, we choose the commercial SONY PlayStation®3 (PS3) as the target platform. Three popular complex streaming applications were implemented: a full-HD motion JPEG decoder, an object detector, and a full-HD H.264/AVC decoder. The CPU on the SONY PS3 is a Cell processor [18], which is a distributed scratchpad memory multicore processor.

On the PS3, the motion full-HD JPEG decoder with the proposed algorithm can decode at about 109 frames per second (fps) in the 1080p format. The object detection application with the proposed algorithm improved the performance by about 16 %, compared with that of the parallel detector from CellCV [23]. Recently, an object recognition system for Cell was proposed in [27]. At the QVGA resolution, it can detect objects in real-time at 22 fps. Using scaling factors similar to those in [27], the object detector with the proposed algorithm can process a QVGA source at a maximum of 63 fps. Finally, a state-of-the-art H.264/AVC decoder on a Cell processor has been proposed [2,20], and its best frame-rate performance on the PS3 is about 35.1 fps with full-HD quality. With our efficient design flow, the H.264/AVC decoder can achieve almost 50 fps with full-HD quality on the PS3.

The rest of this paper is organized as follows: Sect. 2 demonstrates the design flow that parallelizes complex streaming applications on the distributed scratchpad memory multicore architecture. Section 3 provides a brief introduction to the multicore architecture of PS3. The porting of two DLP streaming applications to PS3 (frame-based motion JPEG decoder and slice-search-window-based object detection) is presented in Sect. 4. Section 5 describes a real-time high-definition H.264/AVC decoder using mixed data and function partitioning. Finally, Sect. 6 concludes this paper.

2 Parallelizing Streaming Applications on Distributed Scratchpad Memory Multicore Architecture

Multicore programming introduces additional partitioning and allocation steps, which are unfamiliar to most programmers used to working on single-core architectures. The goal of these steps is to reduce processing time by balancing the workload of the cores

as much as possible while minimizing the resulting IPC overhead. In general, the finer the task granularity, the better workload balancing can be achieved. Unfortunately, large amounts of tasks complicate the allocation step, which may lead to unnecessary data communication, thereby increasing IPC overhead dramatically. In this section, we briefly introduce the design flow we adopted to parallelize multimedia applications on distributed scratchpad memory multicore architectures, to attain the best design that balances conflicting demands.

A few research studies focused on multicore programming for distributed scratchpad memory architectures. Some [2,13,20,26,27] parallelized specific applications according to their detailed dataflow, and efficient porting results were obtained. However, their proposed solutions were difficult to apply to different applications, and new programming models [21,22,25] were proposed. These new programming models provided libraries and compiling tools that hid the tedious tradeoffs. Other research studies [5,16] developed automatic code generation based on existing programming models.

Since the capabilities of compiling tools are still limited, these programming models are rather complicated. They rely on fine-grained partitions to ensure load balancing. For complex applications, such as H.264, using these programming models can be difficult for programmers. The limited capacity of scratchpad memory brings another challenge, especially when the program codes, the stack, and the heap all reside in the scratchpad memory. Some studies [1,3,14] focused on the automatic handling of memory allocation, by concentrating on accelerating the execution of a single thread on a single core with scratchpad memory and hiding the details from programmers.

Our research attempts to develop a general design flow that starts from a simplified programming model and attains a high-performance parallel program for distributed scratchpad memory multicore architectures using coarse-grained partitioning. We started with an optimized single-core program, assuming that all functionalities have gone through a conventional single-core optimization, such as SIMD and loop unrolling. In other words, we assumed that the computations would not change drastically during the optimization. To deal with the limited scratchpad memory, we applied dynamic code mapping [14] and circular stack management [3] to the program code and the stack, respectively. All memory spaces were allocated statically to prevent the use of heap variables, as suggested by [10].

The design flow is shown in Algorithm 1. Data partitioning normally benefits the workload balance, because workloads resulting from data partitioning are more predictable and flexible than task partitioning, even with data-dependent computations. Multiple levels of DLP exist in streaming applications, and relying on DLP allows us to completely avoid the IPC overhead.

Algorithm-I Parallelizing Application on Multicore

Input: A dataflow process network of the application

Output: A parallel application on multicore

1: Determine largest data segment by system specification;
2: Find data segment with minimum ($T_{overflow} + T_{ipc}$);
3: **While** $T_{comm} > T_{comp}$ **do:**
4: Slice processes evenly;
5: Re-order processes and Allocate processes;
6: **If** extra scratchpad memory space left
7: Adopt dual/multiple buffer
8: **End if**
9: **End while**
10: **If** not meet the performance constraint **then**
11: **While** $T_{comm} > T_{comp}$ **do:**
12: Pipeline processes exceed $WL_{threshold}$; or
DLP-enhanced function partition below $WL_{threshold}$;
13: Re-order processes and Allocate processes by **Algorithm II**;
14: Update $WL_{threshold}$
15: **End while**

The selection of the data segment granularity is critical in data partitioning. To reduce data movements and unnecessary IPC operations, larger data segments are preferred; however, oversized data segments may violate system specifications, because large data segments usually result in longer latencies, which some applications might not endure. In addition, the required temporary storage could exceed system memory. Even when a large data segment granularity meets the system specifications, it could still suffer from memory overflow. Memory allocation techniques, such as dynamic code mapping and circular stack management, allow us to assign a large data segment to a single core; however, this technique could create overhead associated with switching memory blocks between the local memory and the global memory. If we assume that a balanced workload could be achieved, the sum of $T_{overflow}$ and T_{ipc} could be used as the ideal data segment granularity. $T_{overflow}$ indicates the time wasted due to insufficient local memory, while T_{ipc} refers to the time spent in IPC. Both overheads can be estimated from the benchmarks taken from a single core implementation. The use of double buffering or even multiple buffering may influence the choice of granularity level. Double buffering can overlap time spent in computation and communication, while multiple buffering can overlap operations from multiple iterations. Both techniques require additional local memory, which can be an important concern when selecting the granularity of the data segments. On the contrary, we can take full advantage of the local memory when dealing with computation-intensive applications. Buffering techniques have limited results, and thus can be considered at the end of the optimization process.

After selecting the proper granularity for the data segment, we allocate the processes to evenly distribute the workload. Compared to process allocation, data partitioning can more easily attain workload balance, because the workload of each partition is more predictable. If a longer process blocks a single core, further partitioning is usually available with a small penalty. When dealing with data-dependent applications, we have to rely on dynamic allocation mechanisms to slice the processes into finer pieces based on the current workload. Finally, hardware-dependent techniques are activated to hide the IPC overhead and data movements. Hardware resources of the multicore are fully-utilized to achieve the best possible performance.

Data partitioning is broadly adopted in multicore programming. However, it does not produce the desired performance all the time. In some applications, enormous data dependencies exist among data segments, which impose constraints to concurrency. Simply applying data partitioning may result in a large overhead in the synchronization. In this situation, the multi-stage (or software pipelined) function partition scheme must be applied.

The application is first decomposed into processes by applying function partitioning. The initial partition usually starts from basic function blocks. After investigating the characteristics of each process, the process can be properly allocated, as we will discuss later. From the results of the initial process allocation, we can also estimate the benefit of further partitioning. In distributed scratchpad memory architecture, communication can be overlapped with computation. No additional IPC overhead is generated if all communications are covered by computation. If the total amount of communication time T_{comm} approaches or even exceeds computation time T_{comp}, we define it as communication saturation, which indicates that we are no longer able to hide the communication. Any increase in communication time will transform into IPC overhead. In order to achieve the best work balance without introducing considerable IPC overhead, the algorithm gradually partitions and allocates the processes until load balance is attained or communication is saturated.

If performance could still be improved, critical processes are identified and further partitioned. A critical process is one whose workload exceeds a predefined threshold. We define the total workload of the application as T_{total}. If the number of cores available for data acceleration is N, then we define $WL_{threshold}$ for each process as

$$WL_{threshold} = \frac{T_{total}}{\alpha N},$$

where α is the granularity factor controlling the number and granularity of processes. For an N-core architecture, the threshold is initially set to $1/N$ of the total workload T_{total}. During each re-partition, the granularity factor α is doubled to produce a finer partition.

The function re-partitioning process is far more complex than that of data partitioning. The ideal cut points are at positions that have minimum live contexts, i.e., temporary data, to help limit IPC overhead. Based on the partition threshold, potential cut points are set, and physical cut points are explored around these potential cut points. Finally, we review the granularity of new partitions to decide if additional cut points are required to hold the partition threshold. After that, the processes are re-ordered

and re-allocated. We can also consider the DLP-enhanced parallel function partitioning scheme, where the process network is unfolded and the number of processes is doubled. By applying concurrency on the different iterations, the workload-balance can be improved, while reducing the IPC overhead.

The resource allocation procedure assigns the processes in the process network to physical cores. Before the allocation, processes are categorized into three types: controlling tasks, computation tasks, and communication tasks. Controlling tasks might contain more branch operations than computations, and part of the decision-making can be replaced by a look-up table. On heterogeneous platforms, it is better to allocate controlling tasks to the multicore processor rather than the digital signal processors (DSPs). Computation tasks calculate data, while communication tasks transfer data. Distributed scratchpad memory multicore environments typically provide DMA support; therefore, explicitly specified inter-core data transfers can overlap with computation tasks. Memory optimization with the help of DMA is critical. In some applications, IPC overhead can even be eliminated. This phenomenon should be taken into account during the allocation of processes.

In general, it is best to assign adjacent connected subtasks to the same core to prevent IPC overhead and reduce data movements, and it is important to disperse communication tasks to different cores. For simplicity, the latencies associated with serving the process allocated to a specific core are denoted by $Cost_W$ and $Cost_{IPC}$, which represent the computation time and the total waiting time, respectively. We can measure both latencies during the execution; however, in this subsection, we only consider the problem of allocating the resources of the DSPs.

Algorithm 2 describes the proposed workload-balancing process-allocation algorithm. Depending on the process network, the processes are allocated as follows: (1) the process closest to the output node of the process network is processed first, so that the computation flow of the process network is unhindered. (2) Only active processes are examined for allocation at each time epoch. In addition, exactly one active process is chosen to be allocated at a specific time. A process is "active", if all its successor processes are allocated. If two or more DSP cores are available, a simple round-robin selection policy is applied. We assume that, unless a task is small, most of the computation-intensive processes are essentially equivalent. Otherwise, the critical processes will be sliced appropriately. Consequently, off-loading processes to a specified core will have a latency that is initially equal to or less than T_{total}/N. The active process P_i will be allocated to the lth core, if the lth core is available and the cost function $Cost(l, P_i)$ is minimized. In Algorithm 2, the *for*-loop in lines 4–16 searches for the $\mathrm{P_{min}}$ with the minimum cost. During the evaluation of a candidate process P_i on the lth core, the total computation time $Cost_W(l, P_i)$ and the communication time $Cost_{IPC}(l, P_i)$ are calculated. $Cost_W(l, P_i)$ refers to the summation of workloads that were already allocated on the lth core, denoted by $Cost_W(l)$, and the computation time of P_i, denoted by $Cost_W(P_i)$. Similarly, the term $Cost_{IPC}(l, P_i)$ is equal to the summation of IPC overhead previously allocated on the lth core, denoted by $Cost_{IPC}(l)$, and the communication time required by P_i, denoted by $Cost_{IPC}(P_i)$. However, if P_i is connected to some process P_j that was already allocated on the l-th core, the IPC between P_i and P_j, denoted by $Cost_{IPC}(P_i, P_j)$, can be removed from $Cost_{IPC}(l, P_i)$. The elimination of the unnecessary IPC is achieved by the *for*-loop in lines 7–11 in

Algorithm 2. With respect to memory optimization, only the longer of the two time values—the total workload allocated or the total waiting time—is considered in the cost function.

Algorithm-II Workload-Balancing Process Allocation

Input: A process network of detail workload distribution

Output: Multicore workload-balancing process allocation

```
1:  Process Allocation {
2:      While there is available core, says l th core, do
3:          P_min = P_0;
4:          For all P_i in the active process list do
5:              Cost_W(l , P_i) : Cost_W(l ) + Cost_W(P_i);
6:              Cost_IPC(l , P_i) : Cost_IPC(l ) + Cost_IPC(P_i);
7:              For all P_j already scheduled on l th core do
8:                  If P_i is connected to P_j
9:                      Cost_IPC(l , P_i) : Cost_IPC(l , P_i) - Cost_IPC(P_i , P_j);
10:                 End If
11:             End For
12:             Cost (l , P_i) = max( Cost_W(l , P_i), Cost_IPC(l , P_i) );
13:             If Cost (l , P_i) < Cost (l , P_min)
14:                 P_min = P_i;
15:             End If
16:         End For
17:         Allocate P_min to l th core
18:     End While
19: }
```

After the allocation, we can estimate the benefits of further partitioning, in terms of core utilization and computation/communication ratio. With the ideal allocation, each computing resource is efficiently utilized at 100 %, which demands both load-balancing and IPC elimination. Finer partitioning leads to a more balanced workload, but it can increase IPC overhead at the same time. Through appropriate allocation, memory optimization can hide the IPC overhead if the total computation time is less than the total communication time. Otherwise, extra communication time should be considered. Assuming that re-partitioning does not introduce computation overhead, the performance can be estimated using a synthetic workload.

3 Cell Broadband Engine Architecture

For demonstration and validation purposes, the commercial SONY PlayStation®3 was selected as the target platform. PS3 uses a Cell processor as its CPU.

The Cell processor, which is a heterogeneous distributed scratchpad memory multicore processor, was designed for high performance multimedia and entertainment applications. It is comprised of two kinds of processing elements: (a) 1 PowerPC processor element (PPE), and (b) 8 synergistic processor elements (SPEs), connected by a ring-based inter-core network called the element interconnection bus (EIB). The

PPE and SPEs share a similar instruction set architecture, but their actual behavior is very different. The PPE is a general-purpose, dual-threaded, 64-bit PowerPC processor, supporting vector/SIMD multimedia extension instructions. The operating system can run on the PPE and take full control of the system; whereas, the SPE is designed for data-rich computation-intensive SIMD and scalar applications. Using a similar SIMD intrinsic design, each SPE provides 128 quad-word SIMD registers, thus allowing compilers or programmers to deploy high-performance applications. To allow programmers to port applications easily on the PPE, the software on the PPE follows a conventional programming model. Moreover, the PPE can access the whole memory space, while the SPE can only access its own 256KB scratchpad memory, called the local store (LS). Its own memory flow controller unit exchanges data with other elements or peripherals in the EIB. The memory flow controller (MFC) is simply a DMA with a 16-entry instruction queue. With the memory flow controller, the SPE can simultaneously perform computation and communication tasks.

Synchronization between elements in the Cell is done by mailbox and special signals. Signals perform the same functionality as mailboxes. Normally, data is exchanged between PPE and SPEs using mailboxes, while signals are utilized in inter-SPE synchronizations. When the SPE needs to use the memory flow controller specifically, mailboxes or signals are used to communicate with other elements in the EIB. Theoretically, every element in the EIB can fill their mailboxes or signals with data by accessing corresponding memory-mapped addresses. The interconnection network EIB consists of four token rings, two clockwise and two counterclockwise. Each ring can simultaneously provide three non-overlapping 128-bit data communications. Because the EIB runs at 1.6 GHz, the potential peak bandwidth achieved is 204.8 GB/s.

An SPE program must be initialized from the PPE. During the initialization, the PPE creates a service thread for an SPE, loads the program code on the SPE, and activates the SPE. Controlling information is passed between the PPE and SPEs through 32-bit mailboxes. The service thread on the PPE is in charge of providing system services for the SPEs. As a result, an SPE can ask for system services, like file access, directly. However, the system service request from the SPE is terribly slow, and its use is discouraged [18]. Even though the PS3 uses the Cell as its CPU, only six SPEs are accessible to the programmers.

A few official and third-party programming models [21,22,25] are available for the Cell processor. Most of the programming models aim at hiding data communications from the programmer to reduce programming complexity caused by managing the scratchpad memory. These models show promising performance for applications using data partition; however, we chose to handle task partition by using the asymmetric-thread runtime model [11] in the following case studies. The task management and data communication required in streaming programming model are handled manually using functions in the Cell software design kit.

More detailed information about the Cell processor can be found in [7,18,19].

4 DLP Streaming Applications

In DLP streaming applications, the workload balance is usually accomplished naturally. The key design issue is how to determine the ideal data segmentation in

order to fully utilize the hardware resources. In this section, we discuss two different DLP streaming applications that were implemented on the PS3: (a) the motion JPEG decoder, and (b) the object detector. Several design issues related to hardware architectures were carefully evaluated in the attempt to obtain the best performance.

4.1 Frame-Based Motion JPEG Decoder

JPEG, a well-known standard [24], has been broadly used around the world for image compression. In multimedia, motion JPEG is an informal class of video formats, which is simply a series of digital compressed JPEG frames. Due to its low computational power demand, motion JPEG is used in portable devices with video-capture capability, such as digital cameras and cell phones.

As suggested by Algorithm 1 in Sect. 2, we chose to apply our algorithm to the frame-based motion JPEG decoder. First, we simplified the open source program and ported it to PS3 by using the released Cell software design kit [9,12]. The parallel code structure of the motion JPEG decoder is described in Fig. 1. In the beginning, the PPE extracts encoded frames from the file system, en-queues the frame sequences, and dispatches the frames to SPEs in a round-robin manner. If input data is available, the SPE applies the JPEG decoding algorithm and uses the memory flow controller to pass the decoded result directly back to the frame buffer, which also resides in shared memory. Finally, the PPE displays the decoded frames on the monitor in the proper order. As a result, the SPEs accelerate all the computation-intensive parts, while the PPE manages the memory and controls the system tasks.

To utilize the Cell processor fully, the planning and optimization of the dataflow are carefully investigated. In typical JPEG decoding, the compressed input frames are of variable length, but the decoded output frames are fixed-length; therefore, the storage required by the output frames is enormously larger than that of the input frames. Note that, in the SPE, both programs and data share a single 256 KB local memory.

Program memory and data memory are further divided into input and output buffers. With an I/O buffer smaller than 256 KB, the entire full-HD JPEG frame cannot be stored in the data memory. Thus, additional data transfer operations are unavoidable. We further noted that the size of each data transfer from the SPEs depends on the I/O buffer configuration. In our implementation, we applied a simple, but efficient, circular input buffer to accommodate the variable-length encoded frames. On the other hand, we adopted a static output buffer for constant-sized output frames, as shown in Fig. 2. For a 16×16 macroblock workload, the output buffer requires at least 768 bytes. When decoding in the single I/O buffer configuration, several data transfers and SPE computations are interleaved to complete the work. Consequently, SPE utilization is low and influenced by the DMA transfer. The Ping-Pong buffering or multiple buffering approach can help to address this problem, because it behaves like an efficient pipeline process, where the SPE utilization approaches 100 %. The I/O buffer configuration and the data-transfer size for each I/O call from the memory flow controller to the SPE are important design parameters.

On distributed scratchpad memory architectures, like the Cell, SPEs cannot directly access the shared memory but must call the DMA function, i.e., the memory flow

PPE

```
// Initiate SPEs
Foreach SPE do
    Initiate_SPE()
End For

// Issue active tasks
Foreach active task T_k do
    preprocess ( T_k )
    enqueue_task ( SPE_Tk, tcb_Tk )
End For

// Post-process tasks
Foreach issued task T_k do
    ack_Tk = dequeue_ack ()
    postprocess ( T_k )
End For
```

SPE

```
// Setup SPE
Initiate()

While (true) do
    // Fetch task control block
    tcb_ptr = dequeue_task ()
    mfc_get ( tcb, tcb_ptr )

    // Perform computation
    Foreach data segment do
        mfc_get ( i_buffer, i_addr )
        computation ( i_buffer, o_buffer )
        mfc_put ( o_buffer, o_addr )
    End For

    // Return ack
    enqueue_ack ( ack_Tk )
End While
```

Fig. 1 Parallel code structure for motion JPEG decoder

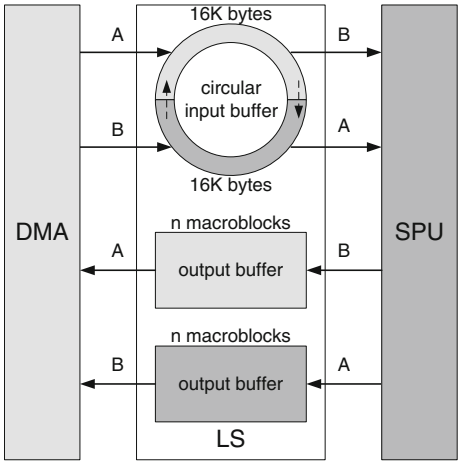


Fig. 2 I/O buffer allocation of motion JPEG decoder on PS3

controller. The total transfer size of a single call by the memory flow controller ranges from 1 byte to 16 KB, whereas a single bus transaction carries 16 bytes per cycle. The SPE Runtime Management Library [12] suggested that both source and destination addresses of memory flow controller calls be maintained at a length of 128 bytes and that the transfer size be an even multiple of 128 bytes.

Figure 3 summarizes the performances of the frame-based motion JPEG decoders on PS3 when using different data transfer sizes per memory flow controller call. Six SPEs are utilized in the experiment. Each bar in Fig. 3 contains two effects: the effects of $Cost_W$ (with the dark color) and the effects of $Cost_{IPC}$ (with the light color). When

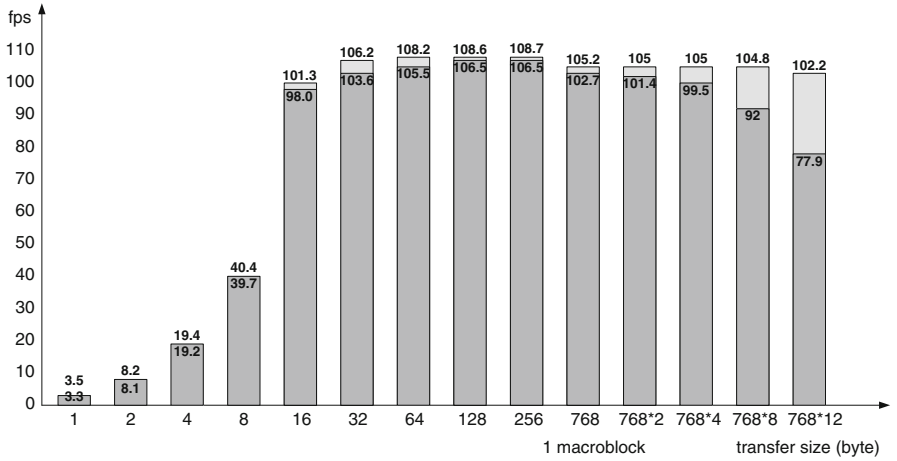


Fig. 3 Decoding performance using different transfer sizes per memory flow controller instruction

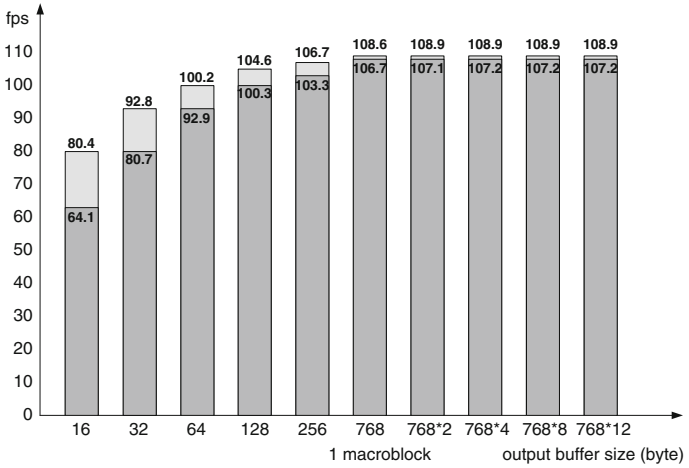


Fig. 4 Decoding performance using different output buffer sizes

the transfer size is less than 128 bytes, a portion of bandwidth is wasted. Even when the transfer size exceeds 128 bytes but it is not a power of two, the performance also degrades. Figure 4 shows the effect of output buffer size when using 6 SPEs and setting the transfer size to 256 bytes. Due to the small data dependency between successive macroblocks, the performance is saturated if the size of the output buffer is above 768 bytes. The motion JPEG decoder can accommodate exactly 16×16 macroblocks.

Figure 5 shows the decoding performance for a full-HD motion JPEG, using different numbers of SPEs. The transfer size was set to 256 bytes and the output buffer was set to 1,536 bytes. The performance of the original decoder was highly degraded due to communication overhead and could not be scaled up linearly. When the input/output buffers were properly allocated, the dataflow of the optimized decoder showed excellent performance and achieved a linear increase in speed. The double

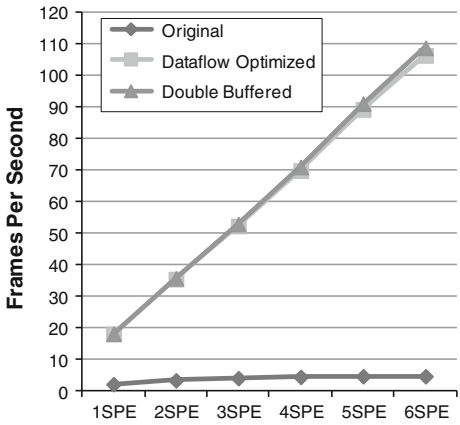


Fig. 5 Performance of parallel JPEG decoder

buffering technique can provide only a small increase in performance. With frame-based parallelization, most inter-core data transfers, except for native ones, are eliminated. With dataflow optimization, the time spent in data communications is low compared to actual JPEG decoding. The simulation results reveal that we have implemented an efficient full-HD motion JPEG decoder on PS3, which can decode full-HD motion JPEG at 108.9 fp/s.

4.2 Slice-Search-Window-Based Object Detection

Object detection is an important computer vision and image processing application. It identifies and locates semantic objects in digital images or videos. These objects could be human faces, buildings, or cars and could be found regardless of their position and scale. Based on Haar-like features and the AdaBoost method, Viola–Jones's algorithm is the first fast and robust object detection algorithm for real-time applications [29]. Fig. 6 illustrates the algorithm. Haar-like features, which are masks comprised of 2–3 rectangles, form the basis of the algorithm. Pixels masked by white and black are weighted and summed up respectively as a feature sum. A weak classifier h_m is set if the feature sum exceeds a threshold. Weak classifiers, which are produced by features, are further weighted and averaged to produce strong classifiers H_M, which indicate whether the examined area is an object or a non-object. Normally, thousands of features are required to precisely distinguish an object from a non-object. The features are ordered and grouped in stages, and the earlier stages could drop detections that were highly uncorrelated. This early-termination fast algorithm reduces computations and makes the detection data-dependent and communication-intensive. When detecting an image, a detection window is deployed with an initially small window size. It traverses the image in a raster-scan manner and scales up until the window size exceeds the size of the image. The Viola–Jones object detection framework was implemented in OpenCV, an open source computer vision library originally developed by Intel. A parallel OpenCV decoder developed for the Cell can be found in [4,23,27].

Figure 7a demonstrates the parallel code structure of the conventional object detector. Object detection within the same search window size is treated as a process (or

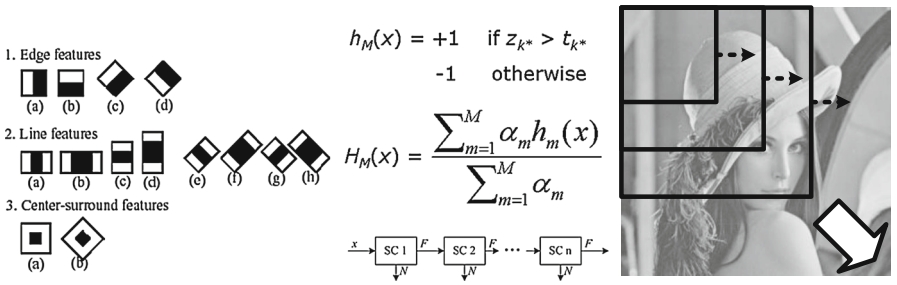


Fig. 6 Viola and jones object detection algorithm [29]

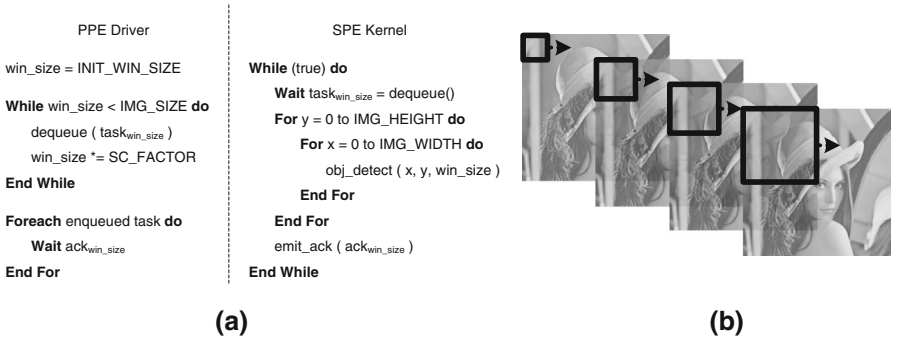


Fig. 7 Conventional parallel object detector: **a** Code structure. **b** Task Partition

task) as shown in Fig. 7b. These tasks can then be accelerated by SPEs. When porting objects on distributed scratchpad memory architectures, discontinuous and irregular shared memory access becomes a bottleneck. As the search window is stretched, we calculate the feature sum using discontinuous pixel values. Because the memory flow controller unit is designed mainly for large amounts of data communication, looking for an 8-bit pixel value from the main memory is an expensive operation. Worse, we have to take care of the memory alignment issue, which creates enormous overhead in this case. When dealing with small search window sizes, several adjacent integral values can be summed by the same memory flow controller call function for efficiency.

The local store is treated as a software-controlled cache. Due to the limitation of local memory capacity, the tasks are categorized into two implementations, one for search window sizes smaller than 88×84 bytes and another for larger window sizes. The buffer is set at 7,392 bytes. The resulting tasks and their execution time are shown in Fig. 8a, where each bar represents a task ordered by detection window size. The height of bars indicates the actual execution time of each task on the SPE. Time wasted in communication is marked using a light color, while tasks in different implementations are drawn using different colors. The resulting tasks from window-size partitioning are very different from those in motion JPEG. Even after memory optimization, the tasks still spend more time dealing with memory data transfer, especially when using large window sizes. In addition, the granularity of the resulting tasks varies, which makes workload balancing more difficult. The actual

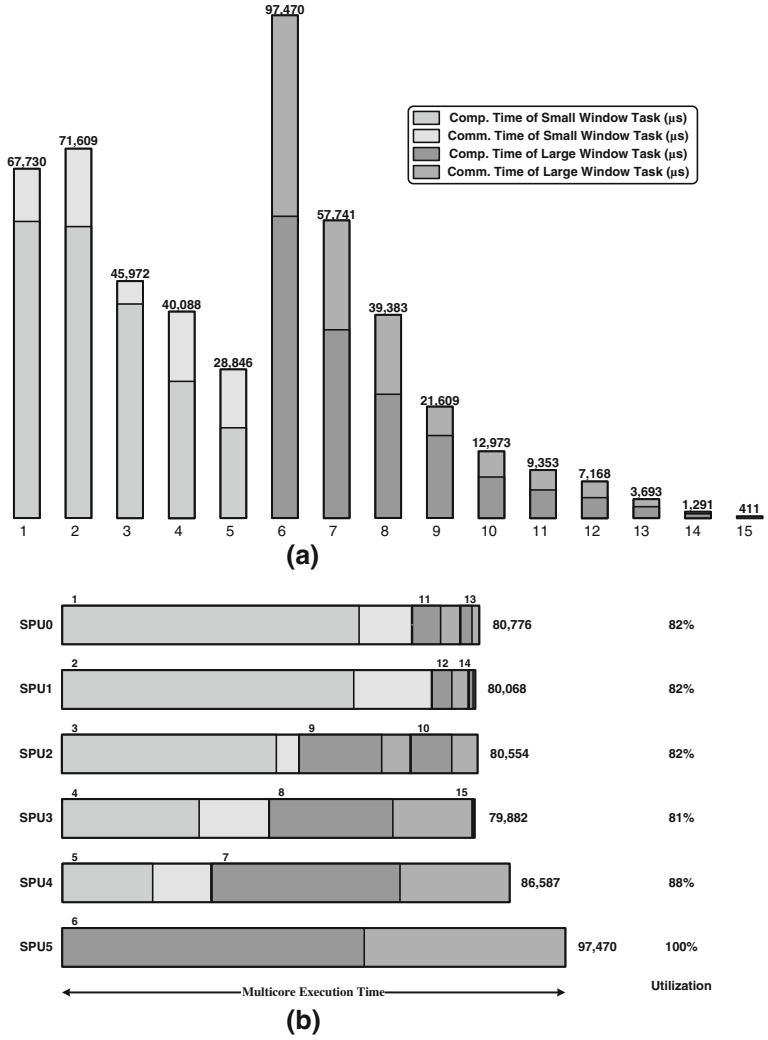
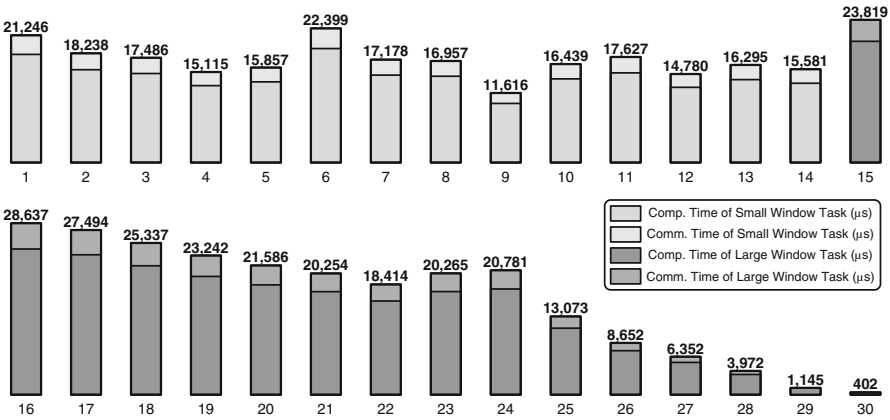


Fig. 8 Workload analysis of conventional parallel detector: **a** Workload distribution. **b** Task scheduling

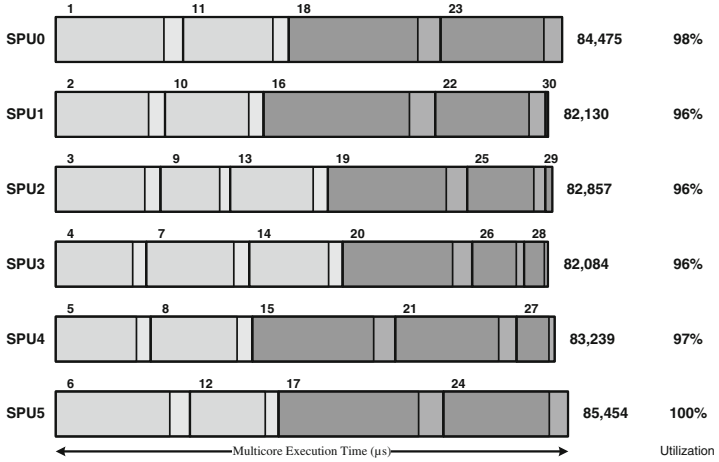
allocation results are shown in Fig. 8b. Although a scheduler tries to balance workload among cores, a considerable overhead is created when waiting for the critical process to complete.

From Algorithm 1, we conclude that the re-partition should be revisited for workload balancing by decreasing the granularity of the data segments. An efficient object detection process using a row-and-slice search window is proposed in [4]. The process splits a task into numerous subtasks (or slices). As shown in Fig. 8a, each task is evenly partitioned into several slices depending on the threshold. Because of DLP, this partition technique does not introduce any IPC overhead. Although the partition slightly raises the computation complexity, it is negligible, and all tasks spend more

(a)



(b)



(c)

Fig. 9 An efficient implementation of a real-time object detector: **a** Task partition. **b** Workload distribution. **c** Task scheduling

time transferring memory data. Figure 9b, c show the resulting tasks and allocation results, respectively. The utilization is very close to 100 %, and the minimal wasted communication time implies very little potential for performance improvement.

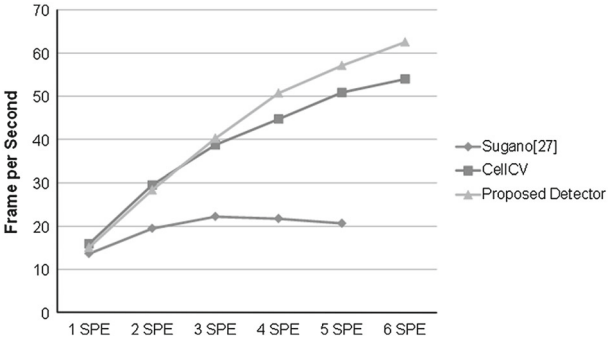


Fig. 10 Performance comparison of parallel object detectors

Performances of parallel object detectors are summarized in Fig. 10, where a 256-byte transfer is deployed while the buffer is set at 7,392 bytes. The simulation results show that the performance of the slice-search-window-based object detector on PS3 improves by about 16 %, compared to the original CellCV detector. Sugano and Miyamoto [27] proposed a real-time object-recognition system on the Cell for QVGA video. Using 3 SPEs, a rate of 22 fps was achieved. However, the frame rate performance of their implementation degrades when using 4–6 SPEs on the PS3. Using similar scaling factors as in [27], the slice-search-window-based object detector achieves better performance on PS3. Further evaluations on numerous test images of different sizes were also made. With 6 SPEs, the proposed object detector can perform object detection at 2.84 fps at $1{,}280 \times 960$ resolution, 11.75 fps at 640×480 resolution, or 62.52 fps at 320×240 resolution, which outperforms the algorithm presented in [27].

5 Multi-Stage Pipeline H.264/AVC Decoder

We also implement a full-HD H.264/AVC decoder on the PS3. The decoder displays 1080p I and P frames, using all inter- and intra-prediction modes. The motion compensation makes use of a single reference frame applying all block sizes, with a search range of ± 16 pixels at 1/4-pixel accuracy.

H.264 [6] adopts various aggressive coding techniques to accommodate high-quality video content with low bit rate. To attain a high compression rate, redundancies within the data are removed as much as possible. It establishes solid data dependencies between pixels, blocks, macroblocks, and even frames. As suggested by the proposed design flow, we should adopt a group of pictures (GOPs) as a basic data segment to avoid all IPC overhead. GOP is a series of successive frames that starts with an I-frame. Each frame in the GOP can be reconstructed without referencing any other frames outside the GOP. However, the storage limitation of the 256 KB LS memory of the SPE leads to heavy data movements from the PPE to the SPE, and vice versa. Thus, enormous $Cost_{IPC}$ overhead arises, which degrades the performance dramatically. Based on our proposed design flow, the function partition scheme should be considered.

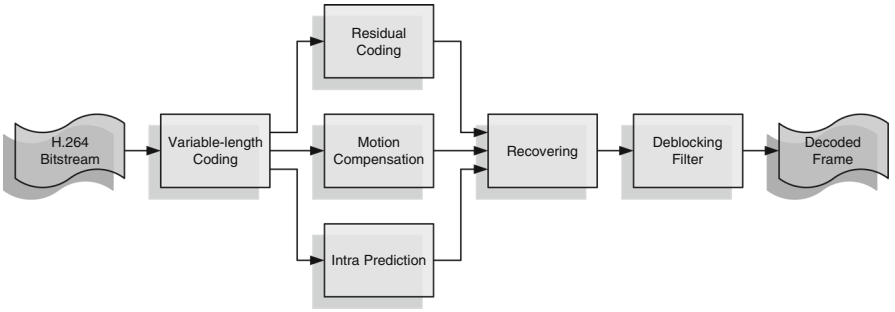


Fig. 11 Functional block diagram of H.264 decoder

Figure 11 illustrates the generic block diagram of the H.264 decoder. It includes six function blocks, namely: variable-length coding (VLC), residual coding (RC), motion compensation (MC), intra-prediction (InP), recovering, and deblocking filter (DF). VLC transforms a compressed context-based adaptive variable-length code (CAVLC) bitstream into a fixed-length macroblock-based stream, where prediction information and residuals are encoded. If the current macroblock is being intra-predicted, InP will be performed to rebuild the macroblock, according to pixel information from neighboring macroblocks. Otherwise, it is reconstructed by MC. Motion compensation, which is the most complicated function, locates prediction values from previously decoded frames. Motion-vector predictors and motion shifts are calculated in advance to direct the compensation procedure. At the same time, RC performs de-quantization and inverse transformation to resolve residual values. Then, recovering sums up the predicted and residual values, while ensuring that the decoded values remain within a feasible range. Finally, DF removes edge effects across block boundaries. Before allocating processes, we first check and implement the program behavior of each block. Because the PPE and SPEs share similar SIMD and intrinsic functions, the optimized program is evaluated separately on both types of cores, as shown in Fig. 12. The VLC is a controlling task with many flow control statements and bit manipulations. Because the SPE deals with these operations inefficiently, a table look-up and predications are applied to reduce the overhead. However, utilizing SIMD on the VLC is difficult, because all variable-length inputs come from a single bitstream. As a result, the VLC runs faster on the PPE, even if the communication time is ignored. Therefore, we prefer to leave the VLC on the PPE. If the MC looks for predictive values from other frames, it could consume considerable bandwidth from the interconnection network EIB. To attain 1/4-pixel accuracy, we might have to grab 9×9 pixels in order to recover a 4×4 block. Moreover, the transfer of reference data suffers from memory alignment issues. The MC is the most difficult function block to parallelize in the H.264 decoder, due to its large communication resource requirements. The remaining function blocks are computation tasks, and, among them, the DF has the heaviest workload. InP and recovering are mutually dependent; therefore, they are they are agglomerated, because both blocks do not require long computation times.

The basic process network of the H.264 decoder is shown in Fig. 13a. Workload distribution without considering IPC overhead is illustrated in Fig. 13b. MC could

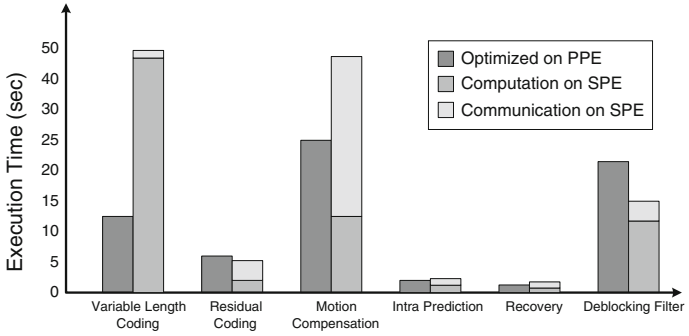


Fig. 12 Workload analysis of H.264 function blocks on PPE and SPE

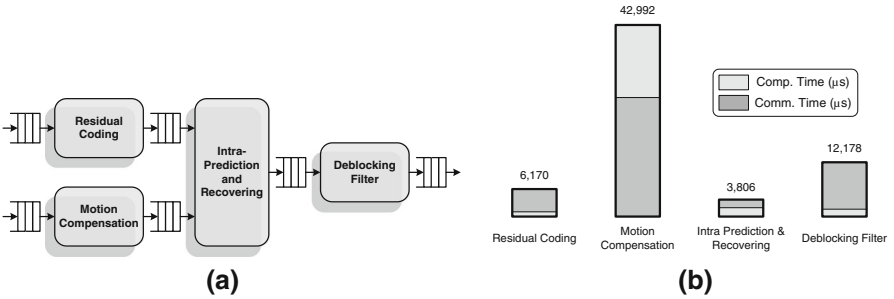


Fig. 13 **a** Basic process network of H.264 decoder. **b** Workload distribution of each process

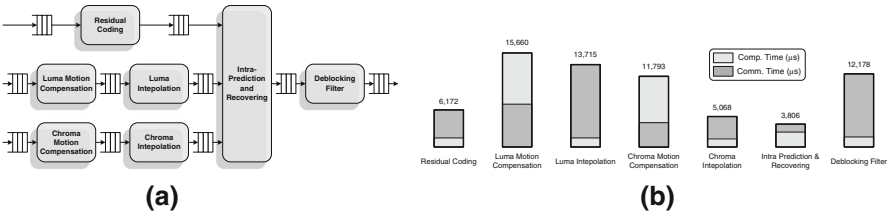


Fig. 14 **a** Advance process network of H.264 decoder. **b** Workload distribution of each process

easily become the critical process that blocks the parallelization. The H.264 decoder on the PS3 can only decode about 15.58 fps in the 1080p format.

Due to the existence of the critical process MC, the performance is limited and cannot be linearly scaled up as the number of SPEs increases. We considered slicing MC by its functionality, and we extracted the interpolation part from the MC as a new process to reduce IPC overhead. The two processes can be decomposed into a luminance part and a chrominance part, as shown in Fig. 14. As a result, the improved process network of the H.264 decoder consists of more processes with similar granularity. The performance factors are listed in Table 1. Utilization indicates the time ratio that the cores are calculating or transferring data. The communication and computing ratios imply the potential for further partition. While communication and computation can be overlapped, the quantity of communication is not equal to the actual IPC

Table 1 Performance evaluation of improved H.264 decoder

No. of SPE	Utilization (%)	Comm./comp. ratio (%)	IPC over head (%)	Performance (FPS)
2	89.80	53.64	0.00	20.42
3	77.50	66.14	5.75	24.59
4	89.12	68.28	10.12	35.09
5	84.91	76.31	17.38	37.11
6	70.89	81.39	18.29	37.27

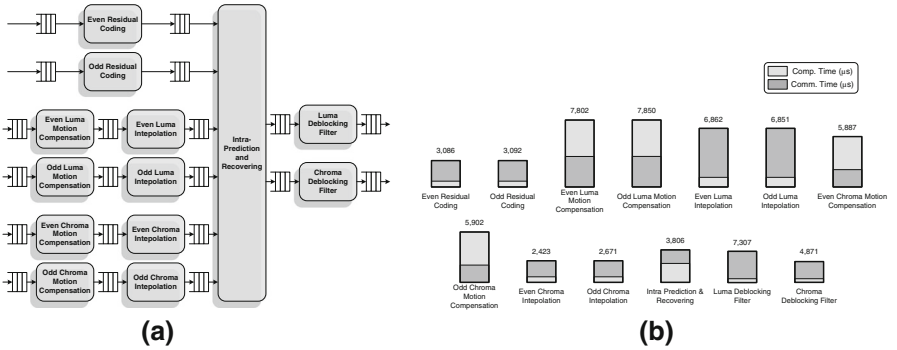


Fig. 15 **a** 2-parallel process network of H.264 decoder. **b** Workload distribution of each process

overhead observed. The IPC overhead field stores the cumulative time that the cores spend to perform communications only. The performance field summarizes the frame rate.

After the allocation process, the improved H.264 decoder on the PS3 can decode about 35.06 fps in the 1080p format using 4 SPEs. A small increase in performance can be achieved if 5–6 SPEs are used. The performance of the improved H.264 decoder on PS3 is comparable to others presented in [2,20].

As the design flow suggests, further partitioning for 5–6 SPEs may help raise utilization without introducing much IPC overhead. However, further partitioning with the improved H.264 decoder is difficult. Instead, we considered the DLP-enhanced function partition scheme, which was implemented as the 2-parallel process network of the H.264 decoder, as shown in Fig. 15. More processes of even granularity are available. The 2-parallel H.264 decoding operations are accomplished in a pipeline manner.

As shown in Fig. 16, the PPE is in charge of variable-length decoding. Two macroblocks are decoded in a single iteration. SPE0 and SPE1 receive the macro-block and locate the required luminance pixels. Then the interpolation process is performed in the next iteration. Another motion compensation workload is inserted to hide the long data transfer latency. SPE2 and SPE3 are responsible for simultaneously compensating and interpolating the chrominance motion. Residual coding takes place after compensating for the chrominance motion to hide the data transfer latency. After the previous SPEs complete their tasks, SPE4 performs intra-prediction and recovering. A chrominance deblocking filter is allocated right after the recovering. Finally, a luminance deblocking filter residing in SPE5 completes the decoding. The performance of

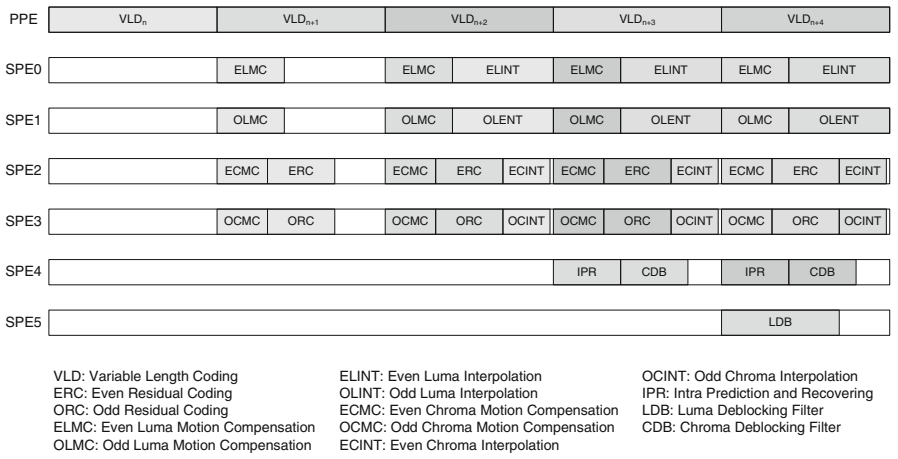
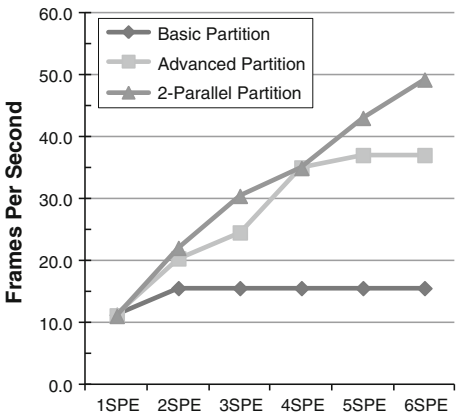


Fig. 16 Scheduling of 2-parallel H.264 decoder on PS3

Table 2 Performance evaluation of 2-parallel H.264 decoder

No. of SPE	Utilization (%)	Comm./comp. ratio (%)	IPC over head (%)	Performance (FPS)
2	92.12	53.96	0.00	22.09
3	91.85	67.48	7.61	30.53
4	89.23	68.94	10.74	35.06
5	87.75	81.30	18.38	43.06
6	86.23	96.33	19.77	49.29

Fig. 17 Performance comparison of different H.264 decoder on the PS3



the decoding is shown in Table 2. The increase in the number of processes effectively raises utilization in all cases, especially for 5 and 6 SPEs. The burden of communication also becomes heavier. Fortunately, most of the communication is still hidden behind computation, and only a small increase in IPC penalty is observed. Based on these results, the 6-SPE decoder has little chance to exceed the 90 % utilization

obtained by function partitioning. Furthermore, the high communication and computation ratio implies that the concealment of the IPC overhead from DMA is already saturated. More processes will certainly introduce IPC overhead.

Figure 17 summarizes the performance evaluations of different parallelized H.264 decoders on the PS3 in different partitions. The decoding speed of the proposed DLP-enhanced multi-stage pipeline H.264 decoder on the PS3 comes close to 50 fps by using 6 SPEs.

6 Conclusion

Multicore programming is considered time-consuming and error-prone. The partition and allocation procedure remains unclear, and it cannot be handled by automation tools, especially when function partitioning is involved. In this study, we explored the efficient parallelization of streaming applications based on the streaming programming model to achieve the best design trade-off between conflicting demands on the distributed scratchpad memory multicore architecture. Three streaming applications (full-HD motion JPEG, object detector, and 1080p H.264/AVC decoder) were implemented on the SONY PlayStation® 3. On the PS3, the motion JPEG decoder based on the proposed algorithm can decode about 109fps in the 1080p format. For object detection, the performance of the proposed algorithm improved by about 16 % compared to the conventional parallel CellCV detector. The simulation results show that the proposed object detector on PS3 can detect objects in real-time at 2.84 fps at $1{,}280 \times 960$ resolution, 11.75 fps at 640×480 resolution, or 62.52 fps at 320×240 resolution. Parallel computing on the Cell using the parallel full-HD H.264/AVC decoder is much more complicated; however, our efficient design flow enables the proposed H.264/AVC decoder to achieve almost 50 fps at full-HD on the PS3.

Acknowledgments This work was supported in part by the Nation Science Council, Taiwan, under Grant NSC-102-2220-E-009-013- and Ministry of Economic Affairs, Taiwan, under Grant MOEA-101-EC-17-A-02-S1-202.

References

1. Bai, K., Shrivastava, A.: Heap data management for limited local memory (LLM) multi-core processors. In: Proceedings of the CODES+ISSS, pp. 317–325 (2010)
2. Baik, H., Sihn, K., Kim, Y., Bae, S., Han, N., Song, H.J.: Analysis and parallelization of H.264 decoder on cell broadband engine architecture. In: Proceedings of the IEEE Symposium Signal Processing and Information Technology, pp. 791–795 (2007)
3. Bai, K., Shrivastava, A., Kudchadker, S.: Stack data management for limited local memory (LLM) multi-core processors. In: Proceedings of the ASAP, pp. 231–234 (2011)
4. Chen, S.-K., Lin, T.-J., Liu, C.-W.: Parallel object detection on multicore platforms. In: IEEE Workshop on Signal Processing Systems, pp. 75–80 (2007)
5. Che, W., Panda, A., Chatha, K.S.: Compilation of stream programs for multicore processors that incorporate scratchpad memories. In: Proceedings of the DATE, pp. 1118–1123 (2011)
6. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, ITU-T Rec. H.264 and ISO/IEC 14496–10 AVC (2003)
7. Gschwind, M.: The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. Int. J. Parallel Program. **35**(3), 233–262 (2007)

8. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann Publishers, California (2007)
9. IBM Corp.: C/C++ Language Extensions for Cell Broadband Engine Architecture. User Guide (2008)
10. IBM Corp.: Cell Programming Guide. User Guide, (2008)
11. IBM Corp.: Cell Programming Tutorial. User Guide, (2008)
12. IBM Corp.: SPE Runtime Management Library. User Guide, (2008)
13. Ismail, L., Guerchi, D.: Performance evaluation of convolution of the cell broadband engine processor. IEEE Trans. Parallel Distrib. Syst. **22**(2), 337–351 (2011)
14. Jung, S.C., Shrivastava, S., Bai, K.: Dynamic code mapping for limited local memory systems. In: Proceedings of the ASAP, pp. 13–20 (2010)
15. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of the IFIP Congress, pp. 471–475 (1974)
16. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. In: Proceedings of the PLDI, pp. 114–124 (2008)
17. Kapasi, U., Rixner, S., Dally, W., Khailany, B., Ahn, J., Mattson, P., Owens, J.: Programmable stream processors. IEEE Comput. **36**(8), 54–62 (2003)
18. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4/5), 589–604 (2005)
19. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: built for speed. IEEE Micro. **26**(3), 10–23 (2006)
20. Kim, Y., Kim, J., Bae, S., Baik, H., Song, H. J.: H.264/AVC decoder parallelization and optimization on asymmetric multicore platform using dynamic load balancing. In: IEEE International Conference on Multimedia and Expo., pp. 1001–1004 (2008)
21. McCool, M.: Data-parallel programming on the cell BE and the GPU using the RapidMind development platform. In: GSPx Multicore Applications Conference (2006)
22. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the cell broadband engine^{TM} processor. IBM Syst. J. **45**(1), 85–102 (2006)
23. OpenCV on the cell. http://cell.fixstars.com/opencv/index.php/OpenCV_on_the_Cell (2010)
24. Pennebarker, W.B., Mitchell, J.L.: JPEG: Still Image Data Compression Standard. Kluwer, Massachusetts (1993)
25. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: making it easier to program the cell broadband engine processor. IBM J. Res. Dev. **51**(5), 593–604 (2007)
26. Sarje, A., Zola, J., Aluru, S.: Accelerating pairwise computations on cell processors. IEEE Trans. Parallel Distrib. Syst. **22**(1), 69–77 (2011)
27. Sugano, H., Miyamoto, R.: A real-time object recognition system on cell broadband engine. In: Mery, D., Rueda, L. (eds.) Advances in Image and Video Technology, LNCS Series 4872, pp. 932–943. Springer, Berlin (2007)
28. Tol, E. van der, Jaspers, E., Gelderblom, R.: Mapping of H.264 decoding on multiprocessor architecture. In: Proceedings of the SPIE Conference on Image and Video Communications and Processing, pp. 707–718 (2003)
29. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features. In: Proceedings of the IEEE Symposium Computer Vision and Pattern Recognition, pp. 511–518 (2001)