The Implementation of Dual Operating System Collaboration Computing on Multi-core Platform *

Li Gang, Liao Yong and Lei Hang
University of Electronic Science and Technology of China (UESTC)
Embedded Real Time Computing Lab
Chengdu, China
igangly@gmail.com
{liaoyong, hlei}@uestc.edu.cn

ABSTRACT

We implement a DualOSCCF (Dual Operating System Collaborative Computing Framework) on ARM PB11MPCore four-core embedded platform [3], in which Linux and aCoral[1] are configured to run. aCoral is a small size real-time operating system (RTOS), which is capable of improving the efficiency of computation-sensitive applications. Linux is mainly responsible for I/O and user interaction, while aCoral is only for computation jobs, such as large-scaled matrix computing. DualOSCCF provides inter-core interrupt mechanism, communication mechanism between Linux and aCoral (Dual-OS), task scheduling and load balancing strategies in multi-core environment. When evaluated over matrix multiplication, DualOSCCF is capable of improving the parallelism and throughput for the computation-intensive applications by task decomposition and effective load balancing. Moreover, the collaborative mechanisms provided by DualOSCCF specifications can simplify the application development.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling, multiprocessing*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Design, Performance

Keywords

Operating system, collaborative computing, task scheduling, load balancing

*This work is partially supported by the National Natural Science Foundation of China (Grant No. 61103041), the Fundamental Research Funds for the Central Universities of China (Grant No. ZYGX2012J070), the Huawei Technology Foundation (Grant No. ZYGX2012J070), and the National High-tech R&D Program of China (Grant No. SQ2011GX02D03708).

[1]aCoral is an open source embedded multi-core RTOS developed by Embedded Real Time Computing Laboratory of UESTC (University of Electronic Science and Technology of China).

1. INTRODUCTION

Recently, the focus of modern processor architectures has shifted from the increase of clock speeds to the amount of parallelism [4]. Multi-core processors have replaced single-core processers and have become the main route of enhancing processor performance. Heterogeneous multi-core processors have been the current topic in research field, for it provides more flexible and efficient mechanisms for applications with different requirements by assigning different kinds of tasks onto different kinds of processors or cores and processing in parallel [5].

Currently, the heterogeneous multi-core architecture, such as CPU+DSP, CPU+GPU, IBM CELL [19], etc, is widely used in industry and consumer electronics [12]. In such architecture, operating system (OS) and applications usually run on the host processor, while the computation oriented jobs execute on the coprocessor. TI proposes a heterogeneous multi-core running platform for OMAP processor, which consists of an ARM core and a DSP core. In the platform, Linux runs on ARM and an embedded RTOS, DSP/BIOS [8], runs on DSP. In order to implement the collaborative computing between the heterogeneous cores, a DSPLink is designed to implement the communication and memory share between Linux and DSP/BIOS. This platform is capable of distributing traditional tasks on ARM and DSP-oriented task to DSP, which implements the concurrent execution of multi-task on DSP, and further makes the original application run in parallel among heterogeneous cores. Thus the total utilization is improved greatly [20].

IBM CELL processor consists of a main processor (PPE) and eight coprocessors (SPE). OS and applications run on PPE, while computation-oriented tasks run on SPEs. PPE is responsible for task allocation and resource management among SPEs, and a SPE creates threads to execute tasks allocated by PPE. Thus nine-core collaborative computing enables CELL processor strong computing power [1]. NVidia's Compute Unified Device Architecture (CUDA) [11] provides a collaborative computation solution for CPU+GPU architecture [18, 9]. However, CUDA supports only for NVidi-

a's GPU. Then Open Computing Language (OpenCL) [16] proposed by Khronos Group is platform-independent programming standard [14, 10]. Typically the computationally intensive functions of an application are written in OpenCL and are called kernels in OpenCL terminology. The kernels are executed in parallel by a large number of GPU's processors. This architecture improves task's parallelism between CPU and GPU, but once the kernel is enqueued for execution, it is not possible to preempt kernel execution [4].

Although heterogeneous multi-core processor archives great performance [13, 7], the complex processor structure and difficult programming model influent its application. The difficulties for collaborative computing in multi-processor consists of task allocation, threads scheduling, resource management and real-time. In this paper, we implement a DualOSCCF on the ARM PB11MPCore four-core embedded board. The main contributions of this paper are as following: 1) rebuilding Linux and aCoral, and running Dual-OS stably on four-core ARE11 MPCore (Linux runs on CPU0, aCoral runs on CPU1, CPU2 and CPU3 in SMP); 2) implementing a set of multi-core mechanisms for task decomposition, synchronization, mutex, memory share and communication; and 3) improving computation capability and task throughput for multi-core processors.

The rest of the paper is organized as follows: part 2 introduces DualOSCCF development environment and presents the design of DualOSCCF, part 3 is the implementation of DualOSCCF; part 4 provides the application development process on the DualOSCCF and gives examples of matrix multiplication, and part 5 is the performance test and analysis.

2. DUALOSCCF DESIGN

DualOSCCF is developed on the Platform Baseboard for ARM11 MPCore (PB11MPCore). PB11MPCore is a highly integrated software and hardware development system based on the ARM SMP architecture and is equipped with four ARM11 MPCore processors. Based on this four core platform, we run Linux on one core, and run aCoral on the left three cores (Figure 1). aCoral is an open-source embedded multi-core RTOS developed by Embedded Real Time Computing Laboratory of UESTC, supporting multi-core (SMP), with high configuration and high scalability. Besides, aCoral supports the classical real time scheduling strategy, e.g., Rate Monotonic (RM) [17] and Earliest Deadline First (EDF) [6] scheduling strategy.

The architecture of DualOSCCF is designed as Figure 1, which is composed of Linux, aCoral and acoral_link:
1) *Linux*: Linux runs on CPU0 and controls most hardware resources, such as network card, Timer0, serial port and so on. Linux is mainly responsible for user interaction and delivering the computation tasks to aCoral.
2) *aCoral*: aCoral runs on slave CPUs(CPU1, CPU2 and CPU3) in SMP way and has a small amount of hard resources such asTimer2. It is mainly responsible for receiving tasks (aCoral Tasks) sent by Linux and creating threads to execute task and then sending back the computation result to Linux.
3) *acoral_link*: acoral_link is a driving module in charge of booting aCoral, managing shared memory, transmitting

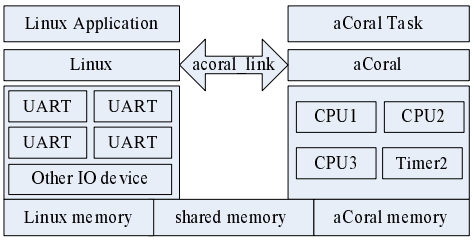

Figure 1: Design of DualOSCCF.

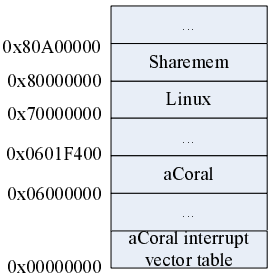

Figure 2: Address allocation.

data, synchronizing, and exclusive accessing for shared resources between Linux and aCoral.

3. IMPLEMENTATION OF DUALOSCCF

3.1 Run Dual-OS

Linux and aCoral run on the unified memory address space. In order to ensure Dual-OS execute correctly, we implement basic mechanisms for distributing address space for Dual-OS images, storing the interrupt vector tables of Dual-OS, distributing external interrupts, booting aCoral from Linux and so on.

3.1.1 Address space allocation

Linux works together with aCoral. The address space needed for the two OS must be independent. The address allocation is shown in Figure 2, address space occupied by Linux, aCoral and shared memory is from different address. As a result of aCoral's no use of MMU in PB11 MPCore version, the interrupt vector table of aCoral must be put in the address of 0x0. Linux enables MMU and its interrupt vector table is mapped to the upper address 0xFFFF0000.

3.1.2 Resource allocation and interrupt binding

In DualOSCCF, Linux is the main OS and manages most hardware components (e.g. clock, network card, serial port, memory controller and so on), while aCoral is the slave OS and only manages the clock. There are four external clock sources on the development board, in which Timer0 is allocated to Linux and its interrupt is bound to CPU0, Timer2 is allocated to aCoral and its interrupt is bound to CPU1. Other external interrupts are all bound to CPU0. Thus the resource conflicts between Dual-OS are avoided.

3.1.3 Boot Dual-OS

Booting Dual-OS is completed in three stages: the first stage is to boot Linux, the second is to load acoral_link module and the last is that the slave CPUs boot aCoral in SMP way.

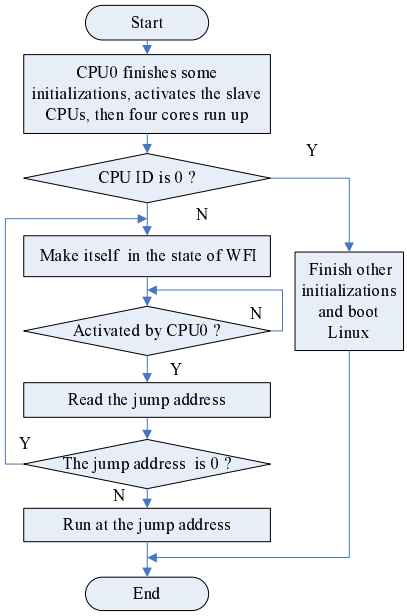

Figure 3: Booting Linux.

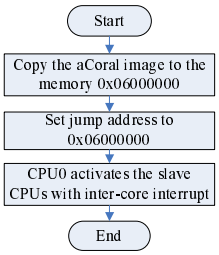

Figure 4: Loading acoral_link.

1) *Booting Linux*: The flow chart of booting Linux is shown in Figure 3. After powered up, only CPU0 is running now. The slave CPUs are in a state of Wait For Interrupt (WFI). They will be activated after CPU0 finishing certain steps of initialization, then four cores can be booted and run correctly. Finally, four cores judge whether their CPU ID is CPU0. If not, the core makes itself in WFI state again and waits for activation from CPU0.

2) *Loading acoral_link*: After Linux's booting and initialization, acoral_link module is loaded. The procedure is shown in Figure 4. Firstly, Linux copies aCoral image from Linux's file system to physical address 0x06000000. Then it sets a jump address to 0x06000000 and activates slave CPUs by inter-core interrupt. Thus, slave CPUs in WFI state (in Figure 3) continue to run, read the jump address, jump to the address to run and then enter the third stage.

3) *Booting aCoral*: The booting procedure of aCoral is shown in Figure 5. The slave CPUs run aCoral image after activation. Firstly they disable their caches and interrupts, CPU2 and CPU3 make themselves in WFI state. After CPU1's initializations, CPU2 and CPU3 are activated respectively. Then aCoral continues the left initializations. Thus the three cores run aCoral in SMP way.

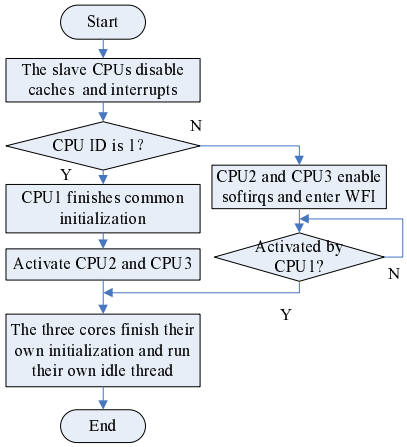

Figure 5: Booting aCoral.

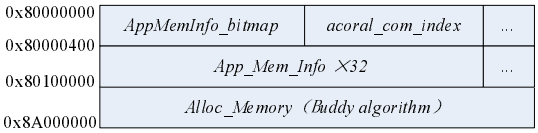

Figure 6: Shared memory allocation.

3.2 Shared memory management

Linux and aCoral are different operating systems, so traditional inter-process communication mechanism cannot be used directly (e.g. pipeline, message queue and semaphore). Therefore, shared memory is a simple and effective way for communication between the two operating systems. As showed in Figure 2, we allocate 160M for shared memory, from physical address 0x80000000 to 0x8A000000. Linux accesses it through virtual address while aCoral access it through physical address, so data transmission requires address translation. In Figure 6, memory from 0x80000000 to 0x80100000 is reserved for some particular data structure with fixed size, which is mainly used for the communication between Linux and aCoral.

App_Mem_Info structure records the information of task which is sent from Linux to aCoral. There are 32 structures stored sequentially, so aCoral supports up to 32 tasks running concurrently. *AppMemInfo_bitmap* is a 32-bit integer and is used as a bitmap for indexing *App_Mem_Info* structure, in which each bit indicates an *App_Mem_Info*. When an *an App_Mem_Info* is applied by a task, the corresponding bit is set to 1. Bitmap can represent the waiting tasks in the queue. *Acoral_com_index* is an integer indicating that which task is finished currently. When a task finishes, aCoral writes the task ID to *Acoral_com_index* and sends inter-core interrupt, Linux will enter interrupt handler and read the value to know which task is finished and then clear the corresponding bit in *AppMemInfo_bitmap*. *Alloc_Memory* describes a dynamic memory region that is allocated and reclaimed by the improved buddy algorithm which can improve the memory utilization. The code, parameters and return value of tasks are stored in this region.

3.3 Inter-core interrupt

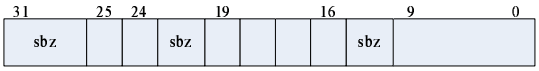

Figure 7: Inter-core interrupt register.

```
for( i1=0; i1<m; i1++)
    for( i2=0; i2<n; i2++)
        for( i3=0; i3<p; i3++)
            C[i1 ][ i2 ] += A[i1][ i3 ]*B[i3 ][ i2 ];
```

Figure 8: Matrix multiplication.

Inter-core interrupt of MPCore is implemented by softirq, which is an effective mechanism in multi-core communication, So MPCore has designed a register supporting inter-core interrupt [2], as Figure 7 showing. The bits in the sbz region are invalid. Bits 0∼9 are the softirq ID. Bits 16∼19, in which every bit represents a CPU ID (CPU0∼CPU3). Set the corresponding bit to 1 if sending a softirq to a core. Bit 24, 25 are control bits.

For Linux only using softirq 1 of CPU0 and aCoral only using softirq 1 and 2 of CPU1, thus we can add softirq 3 in Linux and aCoral respectively for inter-core communication [15]. When Linux allocates a task to aCoral, it triggers softirq 3 of aCoral. The interrupt handler searches the bitmap (*AppMemInfo_bitmap*), acquires an *App_Mem_Info* for the task information and creates a thread to execute. When the task finishes, aCoral triggers softirq 3 of Linux. The interrupt handler reads *acoral_com_index* which indicating the task whose task ID equaling the value is finished and wakes up the blocked process to read the result.

3.4 Task scheduling and load balancing

DualOSCCF provides a set of multi-core mechanisms for synchronization, mutex and task scheduling in order to support users to decompose a large computational task into several small tasks and distributed them on multiple cores to execute in parallel. aCoral runs in SMP, in which each core has its own task ready queue and waiting queue. aCoral, based on certain real-time scheduling policy (RM, EDF), selects the highest-priority task to execute from ready queue. The priority range in aCoral is from 3 to 31, the fewer the value, the larger the priority.

The workload balance policy of DualOSCCF is based on the task number in the ready queue in each core. When aCoral receives tasks distributed from Linux, it will search the idlest core first and create thread on the core.

4. EXAMPLE: MATRIX MULTIPLICATION ON DUALOSCCF

This section presents an example of matrix multiplication ($A \times B = C$) to illustrate the efficiency of DualOSCCF. The size of matrix A is m lines and p columns, matrix B is p lines and n columns, thus the result matrix C is m lines and n columns. The traditional matrix multiplication is achieved by triple loops as Figure 8 shows. This computation is designed as a thread in Linux.

In order to take advantage of aCoral's strength and parallel

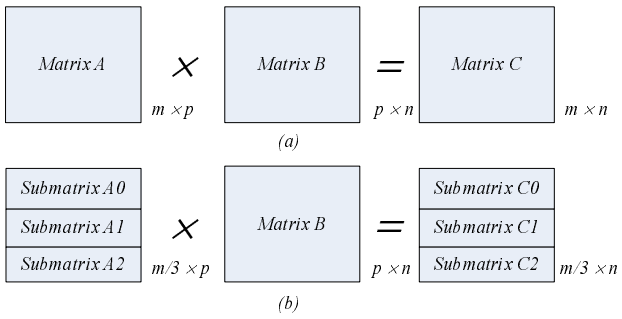

Figure 9: Matrix decomposition. (a) Traditional matrix multiplication. (b) Modified matrix multiplication.

capability of DualOSCCF, matrix A (Figure 9(a))is divided into three submatrixes *A0*, *A1* and *A2* by line equally, as Figure 9(b) showing, then *A0*, *A1* and *A2* multiply matrix B respectively to get three submatrixes *C0*, *C1* and *C2*. Thus the traditional thread is divided into three subtasks: $A0 \times B$, $A1 \times B$, $A2 \times B$, so aCoral creates three threads and allocates them on core 1, core 2 and core 3 to calculate respectively.

The implementation of modified matrix multiplication on DualOSCCF is as Figure 10:
①User writes the source code and gets two executable files *acoral_matrix.o* and *linux_matrix* after compiled. *linux_matrix* is executed on Linux and *acoral_matrix.o* is executed on aCoral.
②Linux creates a process to run *linux_matrix*, which completes the initialization of task information and calls *acoral_link* module, then the process enters kernel mode from user mode.
③The process reads task information and allocates three pieces of memory for *app_addr*, *para_addr* and *ret_addr* in shared memory, which are used for storing *acoral_matrix.o*, matrix A , B and result C respectively. Then the process copies *acoral_matrix.o* to *app_addr* and matrix A and B to *para_addr*, searches *appMemInfo_bitmap* to get an empty *App_Mem_Info* structure and sets the bit in bitmap to 1. aCoral can only access shared memory through physical address, so after *app_addr*, *para_addr* and *ret_addr* are translated into physical address, the process writes them into *App_Mem_Info* to finish the initialization of task information. Sending inter-core interrupt to aCoral, then the process is blocked and waits for aCoral's finishing computation.
④aCoral receives inter-core interrupt and enters interrupt handler, searches the bit which is 1 in bitmap and reads the corresponding *App_Mem_Info* to get task parameter. Then aCoral creates a thread schedules the threads according to certain Real-Time scheduling policy.
⑤The thread on aCoral executes *acoral_matrix.o* from address *app_addr*, reads matrix A and B in shared memory, finishes computation and writes the result to *ret_addr*. The thread sends inter-core interrupt to Linux after finished.
⑥After receiving inter-core interrupt, Linux enters into interrupt handler, clears the bit of the task in bitmap and wakes up the blocked process in ③.
⑦After the process waken up, the computation result is copied to matrix C from *ret_addr*. Then the process reclaims the allocated shared memory and returns to user mode.

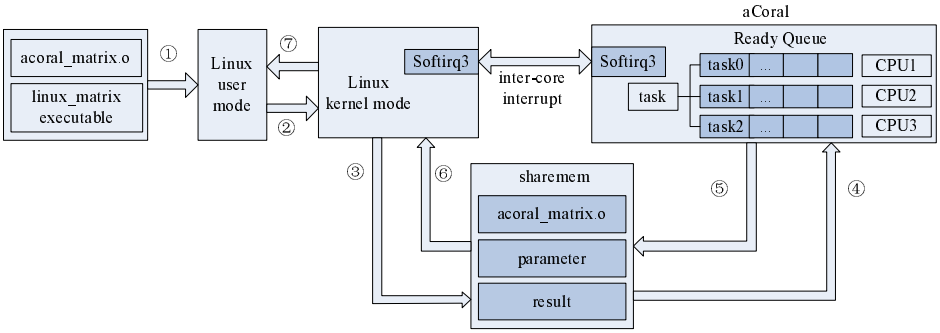

Figure 10: Procedure of matrix multiplication.

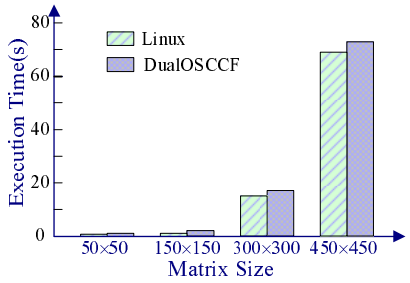

Figure 11: Matrix multiplication with single thread.

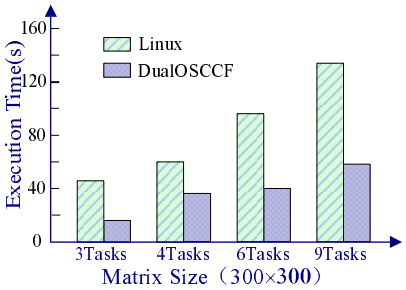

Figure 12: Matrix multiplication with multiple threads.

5. TEST AND ANALYSIS

In order to test and evaluate the efficiency of DualOSCCF platform, we design three experiments about matrix multiplication on single-core Linux platform and multi-core DualOSCCF.

1) Four groups of matrix multiplication are designed to test on both platforms. The sizes of matrixes are respectively 50×50, 150×150, 300×300 and 450×450, as figure 11 shows. Each group is designed as a single thread. The result shows that the execution time on DualOSCCF platform is more than the Linux's, that is, DualOSCCF's computation performance is worse than single-core Linux platform in traditional matrix multiplication. The main reason is that, for a single thread, just one of the three cores in aCoral is used, while DualOSCCF's extra overhead is not negligible, which consists of the time for data copy, inter-core communication as well as synchronization and mutex between Dual-OS. Data copy time refers to the time for copying matrix A and B to shared memory and copying the result from shared memory back to matrix C. Besides this, with the increase of matrix size, the extra overhead becomes larger correspondingly.

2) On the basis of 1), we further use multiple threads on both platforms to calculate the matrix multiplication. Here four groups of multiple threads are designed. In each group, there are 3, 4, 6 and 9 threads as shown in Figure 12. The sizes of matrixes are all 300×300. From testing result, it is shown that Linux platform's computing time rises linearly with the increasing of thread number, while acoral's computing time doubles when every three threads increase. The DualOSCCF's performance improves nearly 2.7 times when 3 threads execute. In a multi-thread environment, threads execute concurrently on Linux platform and only one core is utilized, while DualOSCCF can take the advantage of aCoral's 3-core computing strengths and execute every three threads in parallel. As a result, with the increase of threads number, DualOSCCF has more advantages than Linux platform.

3) For a large-scaled matrix multiplication, in order to make full use of the computation capability of 4 cores, we decompose a computational task into several subtasks and allocates them on different cores to execute in parallel (as shown in Figure 9). Here three groups of matrix multiplications with matrix size of 150×150, 300×300 and 450×450 are designed to be tested on both platforms. Decompose each matrix multiplication into three subtasks according to Figure 9(b), thus DualOSCCF creates threads and distributes them on three cores respectively. Task decomposition makes no sense for single-core Linux platform, so matrix multiplication isn't decomposed when tested on it. Figure 13 shows that DualOSCCF's computing performance improves nearly 2.6 times compared with Linux platform when running the 450×450 matrix multiplication.

6. CONCLUSION

In this paper, based on ARM11MPCore four-core embedded processor, we present a (Linux+aCoral) collaborative computing platform DualOSCCF, which provides the details of booting Dual-OS, and inter-core interrupt, shared memory and so on. Then we introduce the application development procedure and specification on DualOSCCF. Finally a set of matrix computations on single-core and multi-core platform are conducted respectively. The results show that DualOSC-

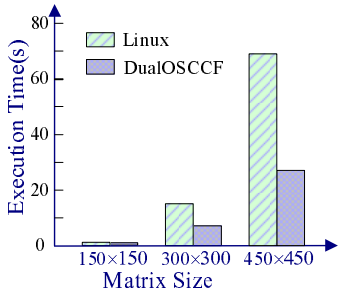

Figure 13: Matrix multiplication decomposition.

CF platform can make full use of multi-core computing resource by decomposing and paralleling the computation on multiple cores.

7. REFERENCES

[1] A., B. D., and Virat, A. Fftc: Fastest fourier transform for the ibm cell broadband engine. *Lecture Notes in Computer Science 4873 LNCS*, 1 (2007), 172–184.

[2] ARM. Arm11 mpcore processor revision: r2p0 technical reference manual. *http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/DDI0360F_arm11_mpcore_r2p0_trm.pdf*.

[3] ARM. Realview platform baseboard for arm11 mpcore. *http://www.general-files.com/download/gs4c29f1e4h32i0/DUI0351E_realview_platform_baseboard_for_arm11_mpcore_ug.pdf.html*.

[4] Bahga, A., and Madisetti, V. K. A dynamic resource management and scheduling environment for embedded multimedia and communications platforms. *IEEE EMBEDDED SYSTEMS LETTERS 3*, 1 (March 2011).

[5] Bansal, S., Kumar, P., and Singh, K. Dealing with heterogeneity through limited duplication for scheduling precedence constrained task graphs. *Journal of Parallel and Distributed Computing 65*, 4 (Apr 2005), 479–491.

[6] Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., and Stiller, S. Improved multiprocessor global schedulability analysis. *Real-Time Systems 46*, 1 (Sep 2010), 3–24.

[7] Becchi, P., and Crowley, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers* (May 2006), ACM:Association for Computing Machinery, pp. 29–40.

[8] Chen, Q. P., and GuiJu, L. Real-time video compressing under dsp/bios. In *Proceedings of SPIE: The International Society for Optical Engineering* (Jan 2009).

[9] Colic, A., Kalva, H., and Furht, B. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems* (May 2009), pp. 3264–3270.

[10] Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., 2011.

[11] GoMez-Luna, J., GonzaLez-Linares, J. M., Benavides, J. I., and Guil, N. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing 72*, 9 (Sep 2012), 1117–1126.

[12] Hagras, T., and Janecek, J. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing 31*, 7 (Jul 2005), 653–670.

[13] Kalinov, P. Y. Scalability of heterogeneous parallel systems. *Programming and Computing Software 32*, 1 (Jan 2006), 1–7.

[14] Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing* (May 2012), pp. 341–352.

[15] Mauerer, W. *Professional Linux kernel Architecture*. Wiley Publishing, America, 2008.

[16] Munshi, A., Gaster, B., Mattson, T. G., Fung, J., and Ginsburg, D. *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.

[17] Naghibzadeh, M., and Kim, K. H. K. A modified version of rate-monotonic scheduling algorithm and its efficiency assessment. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (Jan 2002), ACM:Association for Computing Machinery.

[18] Qin, A. K., Raimondo, F., Forbes, F., and Ong, Y. S. An improved cuda-based implementation of differential evolution on gpu. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (Jul 2012), pp. 991–998.

[19] Reid, and Keith. The ibm cell. *Advanced Imaging 21*, 4 (2006), 12–16.

[20] XiuFeng, Z., KeRen, S., and LiNan, Z. A faster wavelet analysis algorithm based on dsp/bios. *Lecture Notes in Computer Science 270*, 3 (2004), 2018–2023.