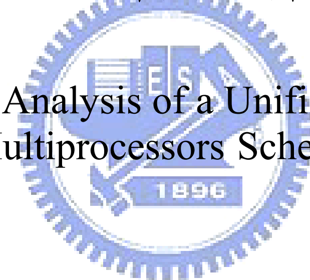
國立交通大學

資訊工程系

碩 士 論 文

高度整合非對稱多處理核心工作排程之
設計與分析



Design and Analysis of a Unified Asymmetric
Multiprocessors Scheduler

研 究 生：王志鵬

指導教授：蔡淳仁 博士

中 華 民 國 九 十 四 年 六 月

高度整合非對稱多處理核心工作排程之設計與分析
Design and Analysis of a Unified Asymmetric Multiprocessors Scheduler

研 究 生：王志鵬 Student：Chih Peng, Wang

指導教授：蔡淳仁 Advisor：Chun Jen, Tsai

國 立 交 通 大 學
資 訊 工 程 系
碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

國　立　交　通　大　學

博碩士論文全文電子檔著作權授權書

（提供授權人裝訂於紙本論文書名頁之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學＿資訊工程＿系所＿＿＿＿＿＿組，＿93＿ 學年度第_2_學期取得碩士學位之論文。

論文題目：高度整合非對稱多處理核心工作排程之設計與分析

指導教授：蔡淳仁 博士

■ 同意　□不同意

本人茲將本著作，以非專屬、無償授權國立交通大學與台灣聯合大學系統圖書館：基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學及台灣聯合大學系統圖書館得不限地域、時間與次數，以紙本、光碟或數位化等各種方法收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行線上檢索、閱覽、下載或列印。

論文全文上載網路公開之範圍及時間：	
本校及台灣聯合大學系統區域網路	■ 中華民國 94 年 8 月 15 日公開
校外網際網路	■ 中華民國 94 年 8 月 15 日公開

授 權 人：王志鵬

親筆簽名：＿＿＿＿＿＿＿＿＿＿＿＿

中華民國　94　年　8　月　3　日

國　立　交　通　大　學

博碩士紙本論文著作權授權書

（提供授權人裝訂於全文電子檔授權書之次頁用）

本授權書所授權之學位論文，為本人於國立交通大學＿資訊工程＿系所＿＿＿＿組，＿93＿ 學年度第＿2＿學期取得碩士學位之論文。

論文題目：高度整合非對稱多處理核心工作排程之設計與分析

指導教授：蔡淳仁 博士

■ 同意

本人茲將本著作，以非專屬、無償授權國立交通大學，基於推動讀者間「資源共享、互惠合作」之理念，與回饋社會與學術研究之目的，國立交通大學圖書館得以紙本收錄、重製與利用；於著作權法合理使用範圍內，讀者得進行閱覽或列印。

本論文為本人向經濟部智慧局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：＿＿＿＿＿＿＿＿＿＿，請將論文延至＿＿年＿＿月＿＿日再公開。

授 權 人：王志鵬

親筆簽名：＿＿＿＿＿＿＿＿＿＿＿＿

中華民國　94　年　8　月　3　日

國家圖書館博碩士論文電子檔案上網授權書

ID:GT009217605

本授權書所授權之論文為授權人在國立交通大學_電機資訊_學院_資訊工程_系所 __________ 組 _93_學年度第_2_學期取得碩士學位之論文。

論文題目：高度整合非對稱多處理核心工作排程之設計與分析

指導教授：蔡淳仁 博士

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文及論文電子檔以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽、下載或列印。

※ 讀者基於非營利性質之線上檢索、閱覽、下載或列印上列論文，應依著作權法相關規定辦理。

授權人：王志鵬

親筆簽名：________________

民國 94 年 8 月 3 日

國立交通大學

論文口試委員會審定書

本校 資訊工程系 碩士班 王志鵬 君

所提論文:

Design and Analysis of a Unified Asymmetric Multiprocessors Scheduler

高度整合非對稱多處理核心工作排程之設計與分析

合於碩士資格水準、業經本委員會評審認可。

口試委員：

指導教授：

系主任：

中 華 民 國 九十四 年 六 月 二十二 日

高度整合非對稱多處理核心工作排程之設計與分析

學生：王志鵬　　　　　　　　　　指導教授：蔡淳仁 博士

國立交通大學資訊工程學系（研究所）碩士班

摘　　要

現今大多數嵌入式多媒體平台使用非對稱多處理核心平台。一個非對稱多處理核心通常包含一個通用型微處理器以及一個或多個數位訊號處理器。針對這樣的系統，目前大多數的工作分配是採用在開發時的靜態分配。然而在新世代多媒體處理系統中的多變性，當運作時的系統狀態與開發時所假設的系統狀態有所差異時，整體系統效能可能會有所降低。本論文提出一個在非對稱多處理核心系統上的動態高度整合工作排程，藉由此工作排程可以獲得較佳的系統效能。新的程式寫作方法類似多執行緒的程式寫作。初步的結果顯示本架構非常適合複雜的嵌入式系統。

Design and Analysis of a Unified Asymmetric Multiprocessors Scheduler

Student: Chih Peng, Wang　　　　　Advisor: Dr. Chun Jen, Tsai

Institute of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

Most embedded multimedia devices today uses asymmetric multiprocessor (AMP) platforms. An AMP is usually composed of a General Purpose Processor (GPP) core and one or more Digital Signal Processor (DSP) cores. For such systems, a common practice is to perform static task partition during development time. However, due to the dynamic nature of new generations of multimedia embedded systems, the performance of the system maybe hindered greatly when the runtime system state is different from the assumed static state at development time. This thesis proposes a dynamic asymmetric multiprocessor scheduling framework that can reach better runtime system performance by using a single unified task scheduler. A new programming practice similar to multi-thread programming is also proposed in order to facilitate this approach. Initial results show that this framework is very suitable for complex embedded systems.

誌　　　謝

在這篇論文的寫作過程中，感謝許多人對我的支持與幫助。首先要感謝的是我的指導教授蔡淳仁博士，感謝教授在研究過程中提供了完善的設備資源以及寶貴的經驗與指導，更在教授的身上學到許多研究的方法以及態度。感謝家人在研究過程中無論是精神上或是經濟上的幫助，讓我可以專心的投入研究工作，努力學習。感謝同學以及朋友的支持，讓我在研究上遭遇挫折或阻礙時，能夠勇於面對挑戰。最後感謝交通大學資訊工程系，這裡提供了一個完善的研究環境，讓我順利完成學業並成長，讓我更有信心面對未來的挑戰。

Table of Content

摘要 ... i
Abstract ... ii
誌謝 ... iii
Table of Content ... iv
List of Figures ... v
List of Tables ... vii
1. Introduction ... 1
2. Previous Work ... 2
3. Design of The Proposed AMP Scheduler ... 5
3.1. The Proposed AMP Scheduler ... 5
3.1.1. Power Consumption ... 7
3.1.2. Execution Time ... 8
3.1.3. Deadline Fulfillness ... 11
3.1.4. Loading Balance ... 12
4. Implementation Details of The AMP Scheduler ... 15
4.1. Introduction to OMAP 5912 ... 15
4.1.1. OMAP 5912 Application Processor ... 15
4.1.2. OMAP 5912 Memory Map ... 17
4.1.3. Memory Traffic Controller ... 18
4.2. Introduction to OMAP 5912 Starter Kit ... 18
4.3. Introduction to DSP Gateway ... 19
4.3.1. DSP Gateway Linux Device Driver ... 20
4.3.2. DSP Gateway DSP/BIOS Kernel ... 21
4.3.3. Inter-Processor Communication ... 21
4.4. Implementation ... 23
5. Experimental Results and Analysis ... 33
5.1. Dynamically Task Dispatching Rate Experiment ... 33
5.2. M4V Interpolation Module Experiment ... 35
5.3. Dynamic Task Dispatching – No OS Environment ... 36
5.4. DSP Gateway Data Transfer Experiment ... 37
6. Conclusion and Future Work ... 42
7. References ... 43

List of Figures

Figure 1. The Proposed AMP Scheduler 6
Figure 2. Task Dispatch Example........ 9
Figure 3. Task Dispatch Example – The Best Ratio........ 9
Figure 4. Rate Transition Diagram........ 13
Figure 5. OMAP 5912 Functional Block Diagram 16
Figure 6. SDRAM Mapping for DSP Space........ 18
Figure 7. OMAP 5912 Starter Kit 19
Figure 8. DSP Gateway Driver Block Chart........ 20
Figure 9. DSP Software Block Chart........ 21
Figure 10. Mailbox........ 22
Figure 11. IPBUF Structure 22
Figure 12. DSP Dynamic Loading Mechanism Block Chart........ 23
Figure 13. The Dsp_dld with Task Registrar........ 24
Figure 14. Pure ARM MPEG 4 Video Codec Interpolation Module 27
Figure 15. Active Sending DSP Task – Read Before Sending........ 28
Figure 16. Active Sending DSP Task – Read After Sending 28
Figure 17. Passive Receiving DSP Task........ 28
Figure 18. Task Information Register........ 28
Figure 19. Dual Mode M4V Interpolate with Single-Thread 29
Figure 20. Dual Mode M4V Interpolate with Multi-Thread........ 29
Figure 21. Multi-Thread M4V Interpolate Module – GPP Thread 30
Figure 22. Multi-Thread M4V Interpolate Module – DSP Thread 30
Figure 23. Multi-Thread M4V Interpolate Module – DSP Data Listener 31
Figure 24. Multi-Thread M4V Interpolate Module – Control Thread 31
Figure 25. Input/Output of Interpolation 33
Figure 26. GPP to DSP Transfer Rate........ 37
Figure 27. GPP to DSP to GPP Transfer Rate........ 38
Figure 28. DSP Gateway Invoke Times........ 38
Figure 29. DSP Gateway Different Size Transfer 39
Figure 30. Linux System Call 40

Figure 31. Read() system call flow 41

List of Tables

Table 1. Unbalance Penalty Example 10

Table 2. Tasks Information 11

Table 3. OMAP 5912 DSP Internal RAM Memories 17

Table 4. AMP Control Interface Command – Task Operation 25

Table 5. Task_info Data Structure 25

Table 6. AMP Control Interface Command – Lambda Operation 26

Table 7. λ Set 34

Table 8. Task Information In Experiment 34

Table 9. Task Dispatching Result 34

Table 10. M4V Interpolation Module Experiment Result 35

Table 11. Time per Computing Unit 36

Table 12. Time per Computing Unit Ratio 36

Table 13. Experiment Group Setting 37

1. Introduction

The complexity of embedded system grows rapidly due to new mobile multimedia applications. Uniprocessor platforms are not suitable for these applications since they require high core frequency in order to handle massive multimedia data processing tasks. However, higher core frequency consumes more power and produces more heat, which is inapt for small form factor embedded systems. Therefore, a common practice for mobile devices is to adopt multiprocessor solutions to increase system performance.

In particular, asymmetric multiprocessor architecture has been widely used for embedded systems development (for example, for cell phones). In this architecture, a general purpose RISC processor (GPP) core and a digital signal processor (DSP) core are integrated into a system-on-chip (SoC), which can handle embedded system tasks efficiently, especially for multimedia applications. However, existing real-time operating systems for such architecture typically adopt a loosely-coupled approach. Task partitions between the two cores are typically done offline and two separate schedulers are employed to perform task scheduling for the two cores independently. This paradigm works properly for traditional mobile applications where the GPP core is typically slow and functionally limited and the application tasks can be put into a simple foreground/background working model.

New generations of multimedia applications and devices make this kind of loosely-coupled system design obsolete. There are at least three reasons that call for a new approach for real-time scheduler designs. First of all, new GPPs today are much more powerful than old ones. Many of them even include special instructions for DSP tasks. Secondly, multimedia applications has become so complicated and dynamic that run-time load balance between the GPP core and the DSP core are crucial for system performance and power consumption reduction. Thirdly, many multimedia applications are more memory-centric than computation-centric. Quite often multimedia data are encapsulated in transport streams, which are parsed out by the GPP. Depending on the inter-processor communication cost at runtime, it may not be possible to determine offline whether the GPP should pass the data over to the DSP for computation.

2. Previous Work

There are many researches on schedulers for multiprocessor architecture in last twenty years. Until now, most multiprocessors scheduling algorithms concentrated on systems of symmetric multiprocessors (SMP) and static tasks partition ([1], [4], [5], [6], [7], [8], [9], [12], [17], [22]). Multiprocessor scheduling techniques in homogeneous multiprocessor platforms can be classified into two general class, partition and global scheduling. Under partition scheduling, each processor has its own task queue, including ready and wait queue, and schedules tasks with local priority space independently from any other processors. Each task is assigned to a particular processor when arriving, ends at the same processor, and will not migrate to other processors during its life cycle. Unlike partition scheduling, global scheduling stores all ready tasks in a single ready queue, and uses a single system-wide priority space. Whenever the scheduler using global scheduling is invoked, the highest-priority task is selected from global ready queue and executed regardless of which processor is being scheduled. These have worked well for existing homogeneous multiprocessors platforms.

A symmetric multiprocessor system can provide better overall system performance than a uniprocessor system [18]. With the gaining popularity of multimedia devices in recent years, the focus has been shifted to asymmetric multiprocessor (AMP) systems. The main reason why AMP systems are used for embedded devices is because that they provide the best performance/clock ratio for the execution of a wide variety of tasks.

Wendorf et al. [15] proposed a number of scheduling policies, ranging from asymmetric master/slave scheduling to symmetric scheduling, for multiprocessor platforms. According to their experiments, "OS Preempt" policy provides the best performance in almost all situations for AMP systems. Moreover, an AMP system using the OS Preempt scheduling policy can perform as good as a fully symmetric system. Their results also indicate that the overhead of context switching and shared resource contention in asymmetric systems are relatively minor factors in overall system performance.

A simple model of master/slave architecture is presented by Greenberg and Wright in [2] along with two scheduling algorithms. In this proposal, a subset of the system calls, which are referred to as the kernel calls, can only be executed on the master. The remaining system

calls are referred to as the user calls. When a slave process makes a kernel call, the slave processor returns the process to the master, rather than services the call by itself. The kernel calls are serialized and may not be independent since these calls may update data that influence the whole system. In the proposed design, jobs not running on any processors are waiting in one of the two queues, the master queue or the slave queue. Jobs in the master queue are all in kernel mode and jobs in the slave queue are all in user mode. A slave processor can take jobs from the slave queue only and the master processor can take jobs from either queue. Two scheduling algorithms are proposed to balance between queue-switching overhead reduction and scheduling flexibility. They also proposed a way to find P^*, the optimal number of slave processors in a single-master processor environment.

In [3], Avritzer et al. developed an analytical performance modeling approach for load sharing policies in highly asymmetric systems that schedule jobs based on global system state. In the system described in [3], hosts have many different speeds which are subject to heterogeneous workloads. They also introduced a threshold type load-sharing algorithm for distributed asymmetric systems, the algorithm varies the thresholds dynamically, adjusting them to the load in order to keep an optimal number of tasks in each hosts. In this paper, they modeled the job routing algorithms by building a global state Markov chain and computing upper and lower bounds on the total system average delay. They concluded that carefully tuned algorithms for load sharing in the asymmetric environment provide a significant improvement in performance over simpler algorithms.

For resource sharing, Saewong and Rajkumar [25] proposed the use of a Cooperative Scheduling Server (CSS), which is a dedicated server that manages one specific controlled resource while using a controlling resource, to control multiple resources access from a single CSS. A CSS is created on a controlling resource (such as a CPU) to handle all local requests for a controlled resource (such as disk access). The CSS reserves a sufficient amount of capacity for controlling resources as needed to fulfill the obligations it has for accessing controlled resources. Because there are scheduling policies for both controlling and controlled resources, co-scheduling design must be employed. Some important considerations of the co-scheduling design in [25] are as follows: 1) scheduling mismatch due to heterogeneity of resource scheduling policies, 2) conjunctive admission control, 3) resource synchronization, and 4) efficient resource utilization.

For embedded multimedia applications, such as 3G mobile phones, both control operations and massive data processing operations are very important. There are some architecture proposals ([16], [23]) efficiently integrate these two different types of computing

units into one AMP SoC. However, most of these systems are designed in a loosely-coupled manner. For example, Gai et al. [24] discussed the problem of multiprocessors scheduling for asymmetric architectures composed by a general purpose processor (GPP) and a digital signal processor (DSP). Two task queues are used in their design, one for regular tasks (for GPP) and the other for DSP tasks. When the DSP is idle, the scheduler always selects the task with higher priority between the tasks at the head of the two queues. When the DSP is active, the scheduler only selects the highest priority task from the regular queue.

In the next section, we will propose a tightly-coupled working model and the associated scheduler design.

3. Design of The Proposed AMP Scheduler

3.1. The Proposed AMP Scheduler

The key concept of the proposed AMP scheduler is to facilitate a tightly-coupled working model. Without loss of generality, assume that there is one GPP core and one DSP core in the target system. In the tightly-coupled model, a task can be assigned to either the GPP or DSP at runtime. When a new task arrives, the unified scheduler will oversee the runtime status of both processor cores and decide which core is more suitable for executing the new task. In our design, the scheduler computes a cost function based on power consumption, computation complexity, deadline fulfillness, and loading balance in order to make a decision for task dispatching.

Since different processor cores execute different binaries, to enable the proposed tightly-coupled model, a new programming practice must be adopted. The new programming model is somewhat similar to single thread vs. multi-thread programming. In the OS, new system service calls are provided for the application to register dual-core versions of executable images into the kernel at runtime. Note that registration of a dual-core executable image does not create a task and enter it into the task queues. Another API must be called explicitly to start a (dual-core) task, which will enter the single universal task queue. This is similar to explicitly calling a system service to start a new thread of a process. The unified scheduler will then dispatch the task either to the GPP or the DSP based on a cost function.

Figure 1 shows the proposed AMP scheduler. It is composed of a cost function evaluator, an AMP scheduler, a version registrar, a resource monitor, and the task interface.

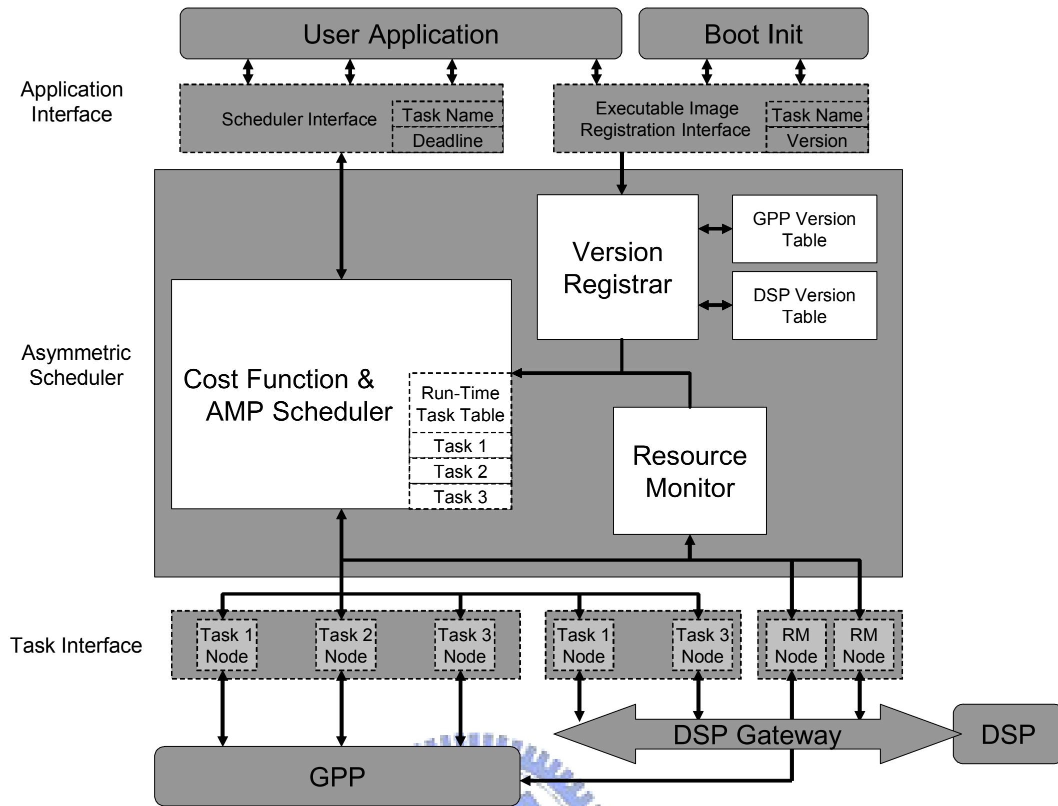


Figure 1. The Proposed AMP Scheduler

The AMP scheduler dispatches a task based on the cost function value and manages running tasks, the version registrar records available executable images (refer to as services in this paper) in the GPP version table and the DSP version table, and the resource monitor watches GPP-side and DSP-side status and provides information for the cost function evaluator. The run-time task table records information and status of tasks running on GPP and DSP-side.

The task interface is an interface between the proposed AMP scheduler and the processing cores. In the proposed design, there are three types of task nodes, the GPP task node, the DSP task node, and the system task node. The GPP task node provides APIs for managing tasks running on the GPP, the DSP task node provides APIs for managing tasks running on the DSP, and the system task node provides APIs for retrieving and monitoring system status.

Equation (1) shows the cost function used by the proposed scheduler to choose the target processor for a task.

$$C=\lambda_1 \cdot C_{power}+\lambda_2 \cdot C_{execution}+\lambda_3 \cdot C_{deadlines}+\lambda_4 \cdot C_{ld_bal} \quad (1)$$

In equation (1), C_{power} is the power consumption cost of computation and data accessing on the GPP or the DSP, $C_{execution}$ is the task execution time on the GPP or the DSP, $C_{deadline}$ is deadline fulfillness based on the task execution time and the deadline, and C_{ld_bal} is the load balance factor based on the task queue lengths on the GPP-side and the DSP-side.

By selecting different values of λ's in equation (1), the cost function can adapt to different system requirements. For example, if the remaining power capacity is low, we can increase λ in order to save more power at the cost of slower response time and poor deadline fulfillness. In the next subsections, we will describe the design of the cost function in detail.

3.1.1. Power Consumption

Power consumption is a major factor in embedded system design because power is the most critical resource for mobile applications. Multiprocessor platforms often have the advantage of being more energy efficient than uniprocessor platforms [31] at same performance.

Effective power usage is not only an important issue in hardware design but also in software design. In the proposed design, power consumption of a task can be further divided into that for setup, computation, and data access. Equation (2) tries to capture these factors:

$$C_{power} = P_{Core_INIT} + P_{Core} + P_{Core_DA} \quad (2)$$

In equation (2), P_{Core_INIT} is the power consumption required to setup the task on a processing core (either the GPP or the DSP), P_{Core} is the power consumption for computation on the core, and P_{Core_DA} is the power consumption for data access from the core. It is important to note the necessity of P_{Core_INIT} in the design. Before executing a task, the system must establish the communication channel between the processor cores, register the task and allocate necessary run-time resources. The power consumption for all these operations may not be negligible and are summarized by P_{Core_INIT}.

3.1.2. Execution Time

Similar to the cost of power consumption, the cost for computation can be divided into that for initialization, computation, and data access. In addition, on some systems, if a task is dispatched to DSP, then there is additional time to deal with channel setup and data communication between GPP and DSP. However, this cost can be rolled into the setup time for DSP. The general cost function of execution time on a core for a task is summarized in equation (3).

$$C_{execution} = T_{Core_INIT} + T_{Core} + T_{Core_DA} \tag{3}$$

T_{Core_INIT} is the initial setup time of a task on either the GPP or the DSP core. Besides allocating runtime resources, including memory and task node, for the task, T_{Core_INIT} also includes communication channel setup and configuration time. T_{Core} is the computing time of a task on a core. This is task-specific. Finally, T_{Core_DA} is the data access time. It includes data transfer time between DSP and GPP if necessary, and memory access time during processing.

In order to take advantage of multiprocessor platform, *"Unbalance Penalty"* is introduced for balancing task dispatching ratio dynamically.

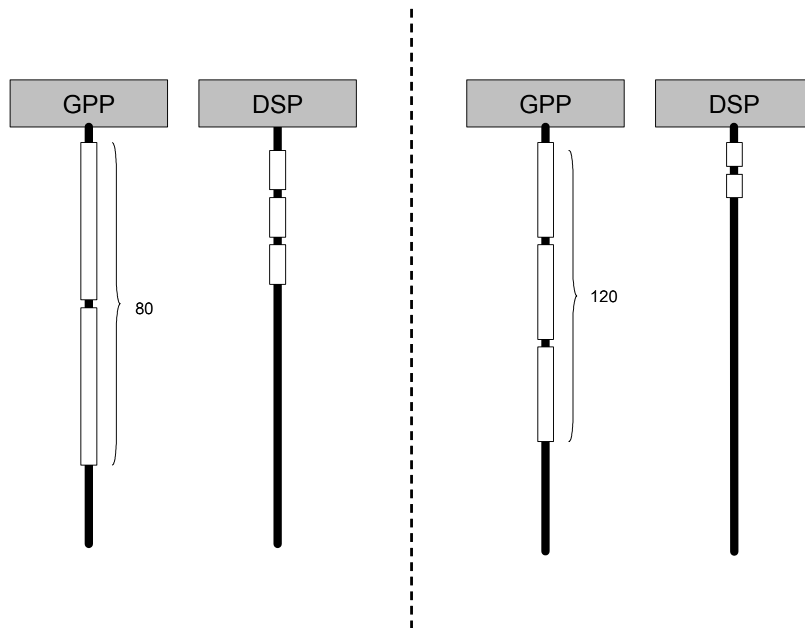


Figure 2. Task Dispatch Example

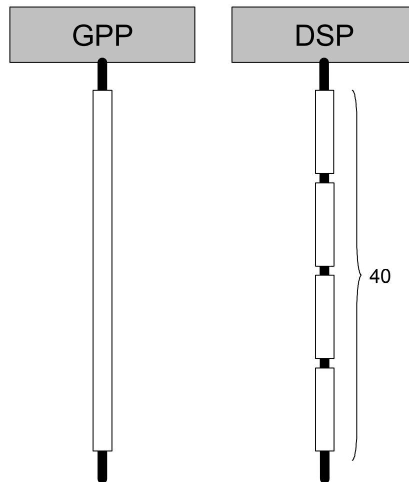


Figure 3. Task Dispatch Example – The Best Ratio

Assuming that there are five units of tasks, the DSP version execution time of the task is 10 per unit and the ARM version is 40 per unit. Figure 2 shows non-optimal dispatching ratios. Although these two examples can take advantage of multiprocessors and reach total computation time, they are not the optimal schedules. Figure 3 shows optimal task dispatching ratio. The task can achieve the lowest response time when the task ratio is inversely proportional to the ratio of the DSP and the ARM execution time of the task. The optimal dispatching ratio is 4 in this example. Most problems related to multitasking scheduling on multiprocessor systems have been proven to be NP-complete ([13], [14], [18], [19]). Under multitasking environment, the *"Unbalance Penalty"* method can achieve optimal dispatching ratio for each task, but overall scheduling decision may not optimal. The *"Unbalance Penalty"* method is sub-optimal solution under multitasking environment.

Stage	DSP Count	GPP Count	Dynamic Ratio	Unbalance Penalty	DSP Cost	GPP Cost
Init	4	1	4 (Optimal Ratio)			
1-1				0	10	40
1-2	5	1	5			
2-1				10	20	40
2-2	6	1	6			
3-1				20	30	40
3-2	7	1	7			
4-1				30	40	40
4-2	8	1	8			
5-1				40	50	40
5-2	8	2	4			
6-1				0	10	40

Table 1. Unbalance Penalty Example

Table 1 shows an example of unbalance penalty works. Note that this example assumes the cost function only concerns about the task execution time. The task can obtain the minimum response time if the DSP to ARM dispatching ratio is 4. Using unbalance penalty mechanism, the cost function can maintain a better dispatching ratio. The second row in Table 1 is the initial stage. The cost function will record the optimal dispatching ratio in the task table. The stage 1-1 shows there is a task unit calls the cost function. There is no unbalance penalty for this task unit since the dynamic dispatching is equals to the optimal dispatching ratio in this stage. The stage 1-2 to 4-2 show the difference between the optimal dispatching ratio and the dynamic dispatching ratio is bigger and bigger since the unbalance penalty value is not big enough to influence the result of the cost function. The stage 5-1 shows the unbalance penalty value is big enough to change the result and the stage 5-2 shows the dynamic dispatching ratio returns to the optimal dispatching ratio. Equation (4) shows how to calculate the unbalance penalty value.

$$V_{ubp} = (R_{Dynamic}-R_{Optimal}) \cdot C_{DSP} \tag{4}$$

V_{ubp} is the unbalance penalty value, $R_{Dynamic}$ is the dynamic dispatching ratio, $R_{Optimal}$ is the optimal dispatching ratio, and C_{DSP} is the DSP execution cost.

3.1.3. Deadline Fulfillness

In a real-time system, we want to process a task before the deadline. In the proposed cost function, deadline fulfillness function is described by equation (5) if the task is executed on the DSP.

$$C_{deadline} = \beta \cdot |T_g - T_c| \quad (5)$$

In equation (5), β is the ratio of the task execution time of running on the DSP core versus on the GPP core. For example, if a motion estimate task processed on the DSP core is twice as fast than on the GPP core, then we can β is calculated by equation (6).

$$\beta = \frac{T_{DSP_ME}}{T_{GPP_ME}} = 2 \quad (6)$$

T_g is the deadline given by the user application. In our design, we assume that the system is a soft real time system.

T_c is the estimated task completion time, which is estimated by summation of task computing time in the task queue that has priority higher than (or equal to) the target task. An example is given as follows.

Task	Priority	Computing Time (ms)
1	1	10
2	3	20
3	2	30

Table 2. Tasks Information

In Table 1, assume that there are three tasks already in the task queue and task 4 is a new arriving task. The priority of task 4 equals to 2 and computing time equals to 40. Because the priorities of task 1 and 3 are higher than or equal to task 4, these two tasks may block

execution of task 4. T_c for task 4 is estimated as in equation (7), and $T_{arrival}$ is the task arrival time (t in this case).

$$T_c = T_{arrival}+T_{task1}+T_{task3}+T_{task4} = t+80 \tag{7}$$

Assuming that T_g equals to t', then the deadline fulfillness cost is calculated in (8).

$$C_{deadline} = \beta \cdot | t' - (t+80)| \tag{8}$$

With this formulation, the cost becomes smaller as the completion time approaches the deadline (note that we assume a soft real-time system).

3.1.4. Loading Balance

In multiprocessor platforms, we want to balance the processing core's utilization since one can get better overall system performance this way. In equation (9), M is the queue length on the DSP side and N is the queue length on the GPP side.

$$C_{ld_bal} = \begin{cases} \left| \Pr_M^{DSP} - \Pr_{N+1}^{GPP} \right| \\ \left| \Pr_{M+1}^{DSP} - \Pr_N^{GPP} \right| \end{cases} \tag{9}$$

$\Pr_M^{DSP}$ and $\Pr_N^{GPP}$ are probabilities when the DSP queue length is M and the GPP queue length is N. We want to make the difference in probability between two queues as small as possible. In this equation, we calculate two possible ways when dispatch a new arrival task by adopting the method for solving birth-death problem presented in [3].

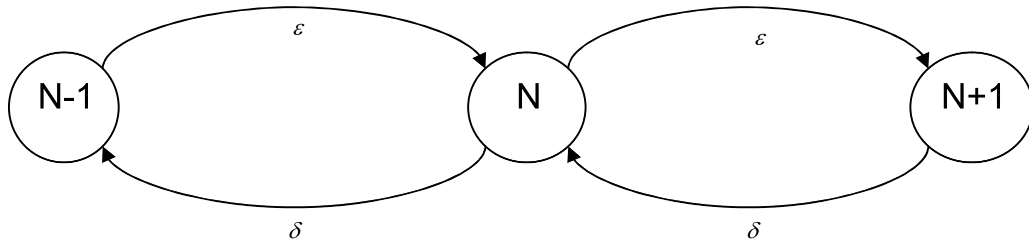


Figure 4. Rate Transition Diagram

In Figure 2, ε is the task arrival rate and δ is the task completion rate. If the system is steady in state N, the total out flow equals to the total in flow and the equation shows is shown in (9). PrN is the probability of the system in state N.

$$\begin{aligned} 0 &= -(\varepsilon+\delta)\,\mathrm{Pr}_N + \delta\,\mathrm{Pr}_{N+1} + \varepsilon\,\mathrm{Pr}_{N-1} \quad (N \ge 1), \\ 0 &= -\varepsilon\,\mathrm{Pr}_0 + \delta\,\mathrm{Pr}_1 \end{aligned} \tag{10}$$

or

$$\begin{aligned} \mathrm{Pr}_{N+1} &= \frac{\varepsilon+\delta}{\delta}\mathrm{Pr}_N - \frac{\varepsilon}{\delta}\mathrm{Pr}_{N-1} \quad (N \ge 1), \\ \mathrm{Pr}_1 &= \frac{\varepsilon}{\delta}\mathrm{Pr}_0 \end{aligned} \tag{11}$$

We use the equations show in (10) iteratively and obtain a sequence of state probabilities, Pr1, Pr2, Pr3, …, each in terms of Pr0,

$$\mathrm{Pr}_N = \mathrm{Pr}_0 \prod_{i=1}^{N}\left(\frac{\varepsilon}{\delta}\right) = \mathrm{Pr}_0\left(\frac{\varepsilon}{\delta}\right)^N \quad (N \ge 1) \tag{12}$$

To get Pr0, probabilities must sum to 1 and it follows that

$$1 = \sum_{N=0}^{\infty}\left(\frac{\varepsilon}{\delta}\right)^N \mathrm{Pr}_0 \quad (N \ge 1) \tag{13}$$

And we define γ as the ratio ε/δ, and obtain

$$\mathrm{Pr}_0 = \frac{1}{\sum_{N=0}^{\infty}\gamma^N} \quad (N \ge 1) \tag{14}$$

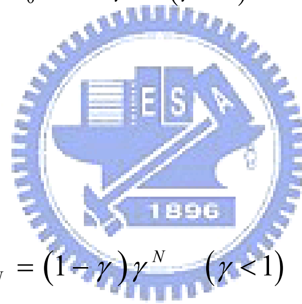
$\sum_{N=0}^{\infty} \gamma^N$ is the geometric series $1 + \gamma + \gamma^2 + \gamma^3 + \ldots$ and converges if and only if $\gamma < 1$. If $\varepsilon > \delta$, the mean arrival rate is greater than the mean completion rate, and the system will grow up unlimited. And if $\varepsilon = \delta$, it means that the system never services all incoming task and turn off with some tasks still wait for processing. So we assume that $\varepsilon < \delta$ or $\gamma < 1$ in our system and use well-known expression for the sum of the terms of a geometric progression,

$$\sum_{N=0}^{\infty} \gamma^N = \frac{1}{1-\gamma} \quad (\gamma < 1) \tag{15}$$

By combining equations (14) and (15), we obtain

$$\Pr_0 = 1 - \gamma \quad (\gamma < 1) \tag{16}$$

and



$$\Pr_N = (1-\gamma)\gamma^N \quad (\gamma < 1) \tag{17}$$

4. Implementation Details of The AMP Scheduler

The proposed scheduler is implemented on an OMAP 5912 platform running Linux. In this chapter, we will first give an introduction to the OMAP 5912 application processor and the OMAP 5912 Starter Kit development board used for the implementation, followed by an introduction to the DSP gateway package used for communication between the GPP core and the DSP core. Note that the overhead of the DSP gateway package is quite high and is not suitable for a tightly-coupled system for practical applications. The reason it is used in the implementation is merely for fast prototyping of the proposed system. Finally, some details about the implementation will be given in section 4.3.

4.1. Introduction to OMAP 5912

4.1.1. OMAP 5912 Application Processor

OMAP 5912 Starter Kit (OSK5912) uses an OMAP 5912 processor. OMAP 5912 integrates a TMS320C55x DSP core and an ARM926EJ-S RISC core. The C55x DSP core provides high performance and low power consumption for digital signal processing tasks. The ARM9 RISC core is very popular for embedded systems. OMAP 5912 is suitable for multimedia embedded devices and can achieve better performance through dividing an application into tasks and dispatch these tasks to two cores appropriately.

Figure 5 shows OMAP 5912 functional block diagram.

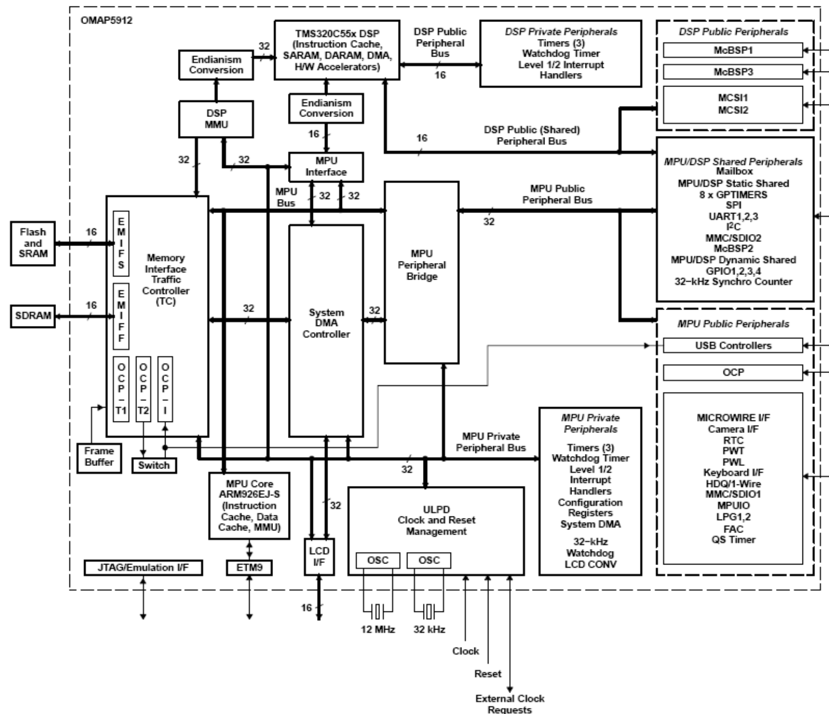


Figure 5. OMAP 5912 Functional Block Diagram

The technical features of ARM926EJ-S RISC core and TMS320C55x DSP core are listed as follows.

- ARM926EJ-S
  - Support for 32-Bit and 16-Bit (Thumb® Mode) Instruction Sets
  - 16K-Byte Instruction Cache
  - 8K-Byte Data Cache
  - Data and Program Memory Management Unit (MMU)
  - 17-Word Write Buffer
  - Two 64-Entry Translation Look-Aside Buffers (TLBs) for MMUs

- TMS320C55x
  - One/Two Instructions Executed per Cycle
  - Dual Multipliers (Two Multiply-Accumulates per Cycle)
  - Two Arithmetic/Logic Units
  - Five Internal Data/Operand Buses (3 Read Buses and 2 Write Buses)

- 32K x 16-Bit On-Chip Dual-Access RAM (DARAM) (64K Bytes)
- 48K x 16-Bit On-Chip Single-Access RAM (SARAM) (96K Bytes)
- Instruction Cache (24K Bytes)
- Video Hardware Accelerators for DCT, iDCT, Pixel Interpolation, and Motion Estimation for Video Compression

4.1.2. OMAP 5912 Memory Map

Table 3 shows memory mapping of the DSP internal DARAM and SARAM. Note that PDROM can not be seen in MPU physical address.

	DSP Byte Address	MPU Physical Address	Linux Virtual Address	size
DARAM	0x000000 | 0x00ffff	0xe0000000 | 0xe000ffff	0xe0000000 | 0xe000ffff	64kB
SARAM	0x010000 | 0x027fff	0xe0010000 | 0xe0027fff	0xe0010000 | 0xe0027fff	96kB
PDROM	0xff8000 | 0xffffff	(not seen in MPU space)		32kB

Table 3. OMAP 5912 DSP Internal RAM Memories

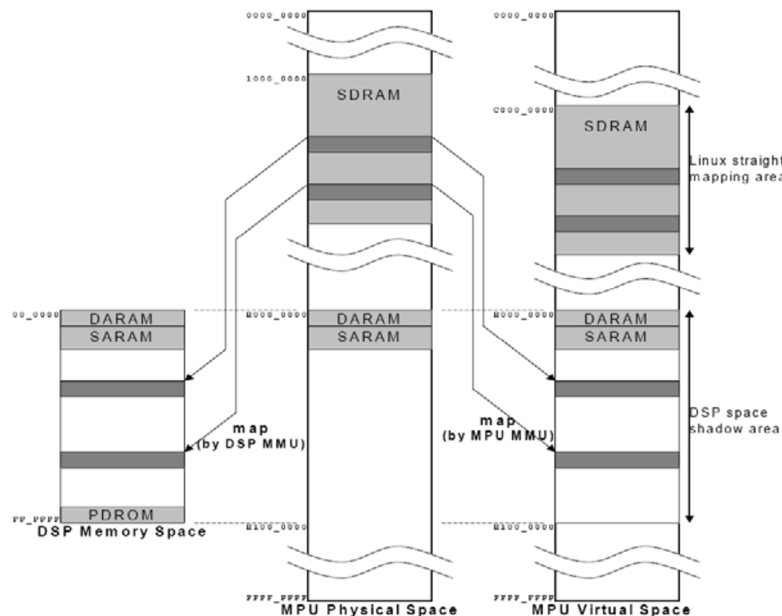


Figure 6. SDRAM Mapping for DSP Space

Besides on-chip DARAM and SARAM, DSP also can use external SDRAM by mapping it to the DSP memory space through the DSP MMU. Figure 6 shows how MPU maps SDRAM to DSP space. When the MPU maps a memory block to the DSP space, it also maps to the DSP space shadow area in the MPU virtual space so the offsets in the DSP space and the shadow area become same. With this mapping, address exchanges between DSP space and MPU virtual space can be done with very simple operations.

4.1.3. Memory Traffic Controller

Memory Traffic Controller (TC) is an important component in OMAP 5912 processor. It provides and controls two high speed memory interfaces, EMIFS and EMIFF, for DSP and MPU to access external memory.

- External memory interface slow (EMIFS) connects external device memories, such as common flash memory. This interface enables 16-bit data accesses and provides four chip-selects – each chip-select is able to support up to 64M bytes address space through a 25-bit address bus.
- External memory interface fast (EMIFF) is a memory interface that enables 16-bit data SDRAM memory access. It supports connection a 64 Mbytes SDRAM at maximum. It also provides two bank selection bits and 16-bit width address. The OMAP 5912 provides interfacing with a maximum of four banks of 64M x 16-bit SDRAM memory with DDR capability.

4.2. Introduction to OMAP 5912 Starter Kit

OSK5912 is a highly integrated hardware and software platform targeted at application processing devices, mobile communications, and video and image processing. OSK is designed to run general embedded operating systems on the ARM side and TI DSP/BIOS real-time kernel on the DSP-side.

Figure 7 shows OSK5912 front view.

Figure 7. OMAP 5912 Starter Kit

The following paragraph lists some OSK5912 product features.

- Hardware Features
  - Texas Instruments TMS320C55xx core operating at 192 MHz.
  - ARM9 core operating at 192 MHz.
  - TLV320AIC23 codec
  - 32 Mega Bytes DDR RAM
  - 32 Mega Bytes on board Flash ROM
  - 10 MBPS Ethernet port
  - On board IEEE 1149.1 JTAG connector for optional emulation

- Software Features
  - Compatible with MontaVista's Linux for OSK5912
  - Compatible with OMAP Code Composer Studio from Texas Instruments

4.3. Introduction to DSP Gateway

DSP Gateway ([14]) is an open source project developed by Toshihiro Kobayashi for inter-processor communication mechanism on Linux for OMAP family. The DSP Gateway consists of a Linux device driver on the ARM side and a DSP-side kernel library. The Linux

device driver provides a convenient interface so that an application on GPP-side can communicate with DSP through normal device system calls. The DSP-side kernel library provides multi-task environment and APIs for user tasks.

4.3.1. DSP Gateway Linux Device Driver

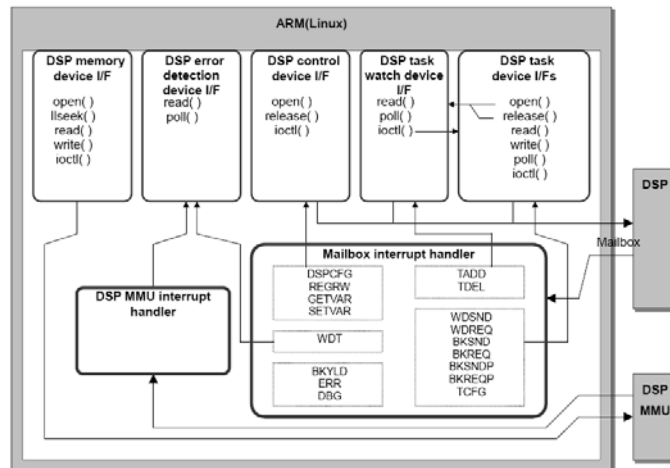


Figure 8. DSP Gateway Driver Block Chart

Figure 8 shows GPP-side functionalities provided by DSP Gateway Linux device driver. The Linux device driver communicates with DSP through two Mailboxes, one for GPP to DSP and another for DSP to GPP. It also provides five task device interfaces.

- DSP Task Devices

  The DSP task devices provide interfaces to the DSP tasks for Linux applications. Programs communicating with the DSP tasks can achieve sending or receiving data to/from DSP by reading from or writing to those devices.

- DSP Control Device

  The DSP control device provides DSP control API for Linux. Through this device, Linux applications can control DSP reset, set specified DSP reset vector address, and performs other DSP control commands.

- DSP Memory Device

  The DSP memory device provides the access to the DSP memory space for the DSP program loader in Linux side. Programs also can extend the usable DSP memory range by mapping external SDRAM block to the DSP memory space through this interface.

- DSP Task Watch Device

  The DSP task watch device provides functionalities needed for the DSP Dynamic Loader Daemon.

- DSP Error Detection Device

  The DSP error detection device provides the ability of detecting error from DSP for Linux applications, such as watchdog timer expiration and DSP MMU error interrupt.

4.3.2. DSP Gateway DSP/BIOS Kernel

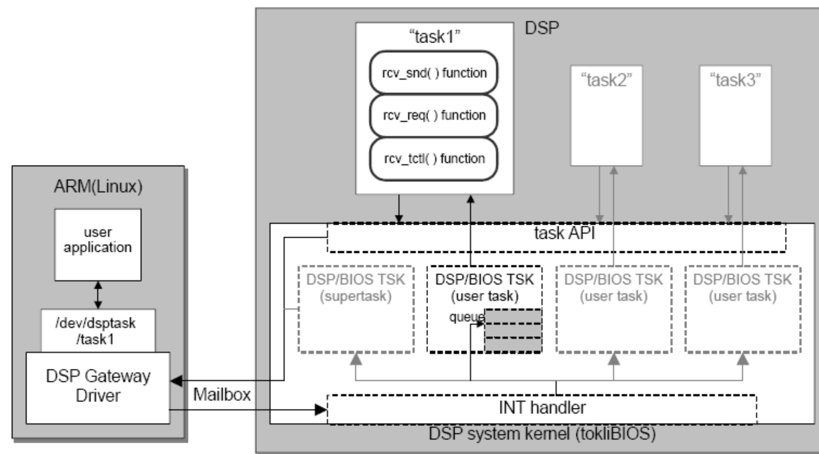


Figure 9. DSP Software Block Chart

Figure 9 shows the DSP Software Block Chart. When a Linux user application accesses the DSP task device, the Linux device driver generates a Mailbox command to DSP. In DSP side, the system kernel receives the Mailbox command and registers it into the queue of the corresponding DSP/BIOS TSK.

4.3.3. Inter-Processor Communication

The ARM-DSP inter-processor communication is implemented using the mailbox mechanism.

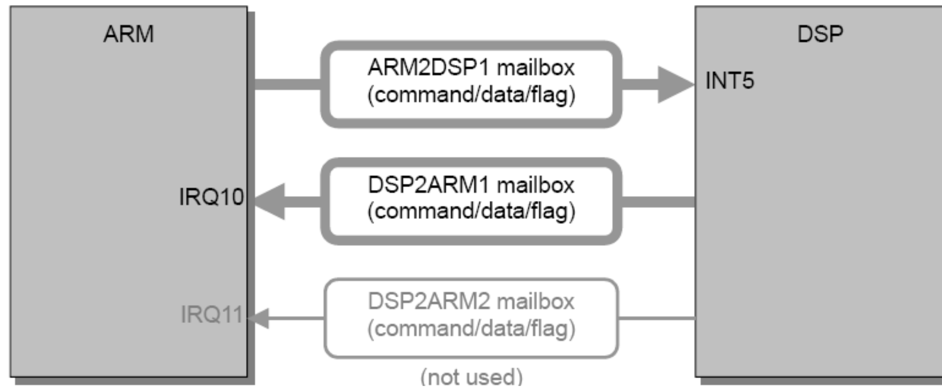


Figure 10. Mailbox

Figure 10 shows the mailbox mechanism between the ARM and the DSP. There are three mailbox register sets. One is for the ARM to send messages and issue an interrupt to the DSP. The other two are for the DSP to send messages and issue an interrupt to the ARM. Each mailbox register set consists of two 16-bit registers and a 1-bit flag register. The DSP Gateway only uses two of mailbox register sets, one for ARM to DSP and another for DSP to ARM, for inter-processor communication.

Transferring a large amount of data with only mailbox registers between the ARM and the DSP is not efficient. DSP Gateway introduces Inter-Processor Buffer (IPBUF) for large block data transfer. The IPBUF can be placed at the DSP internal SARAM and DARAM, or the external SDRAM block which is mapped to the DSP memory space. Figure 11 shows IPBUF structure in detail.

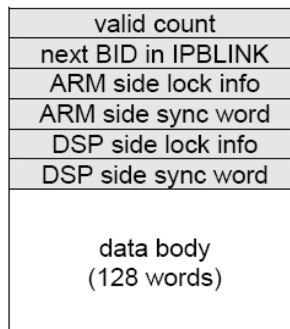


Figure 11. IPBUF Structure

DSP Gateway also provides shared memory mechanism for transferring or sharing data between the DSP and the ARM. This mechanism relies on the DSP MMU and the ARM MMU. By setting the DSP MMU and the ARM MMU, the DSP and the ARM can access the same memory space. This mechanism only supports mapping SDRAM as shared memory. In order to ensure data consistence between the DSP and the ARM, D-cache will be disabled to mapped memory space and this would have an impact on the performance on the ARM-side.

4.4. Implementation

The implementation of the proposed system is based on Linux 2.6.11 kernel patched by [11] and [13]. We use the dsp_dld [12] in DSP Gateway package to load DSP applications to the DSP core from an application (or system service) running under Linux. The dsp_dld is a utility program that manages DSP tasks and resources for the DSP Gateway.

Figure 12 shows how DSP dynamic loader daemon (dsp_dld) works.

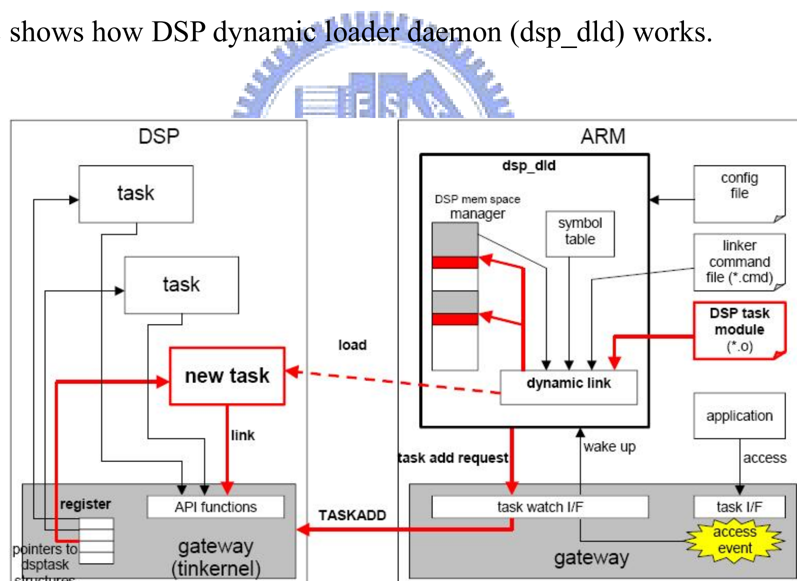


Figure 12. DSP Dynamic Loading Mechanism Block Chart

At startup time, the dsp_dld create DSP task device file in ARM-side /dev directory as the DSP task interface to the dynamic tasks, boot the DSP up using the DSP Gateway kernel, initializes the memory space manager, and creates the symbol table.

When an open access event is passed to the dsp_dld through /dev/dspctl/twch, the

dsp_dld loads the corresponding task module to DSP memory and the task is added to the system. When a close access event occurs, the dsp_dld removes the corresponding task from the system. Rely on this mechanism, DSP Gateway can link and load the DSP task dynamically. The memory space manager in the dsp_dld manages the memory space on DSP-side. External RAM block for a DSP task is mapped dynamically at loading time and lasts to the end of a DSP task.

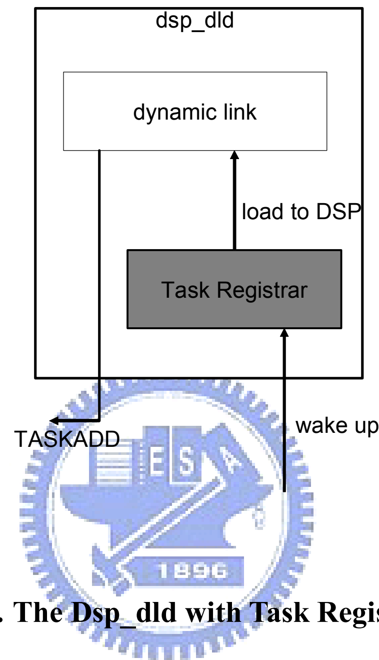


Figure 13. The Dsp_dld with Task Registrar

There are two ways to register task information into the proposed design. Figure 13 shows the dsp_dld with the task register interface. When an access event wakes the dsp_dld, it not only loads/unload a DSP task to/from DSP but also register/unregister task information into the proposed design. The second way is to register task information through AMP control interface. AMP control interface is a control interface implemented as a Linux driver and provides a device file interface at /dev/amp/ctl. Programs can use the ioctl system call to register task information or control the AMP scheduler parameters. Table 4 and Table 6 show AMP control interface commands.

Command Name	CMD Value	Parameter	Return Value	Short Description
OMAP_AMP_IOCTL_CAL_COST	0x15	Char *	Int	Calculate Task Cost
OMAP_AMP_IOCTL_TSK_REG	0x16	Tsk_info *	Int	Register Task Information
OMAP_AMP_IOCTL_TSK_GET	0x17	Tsk_info *	Tsk_info *	Get Task Information
OMAP_AMP_IOCTL_TSK_DEL	0x18	Tsk_info *	Int	Delete Task Information
OMAP_AMP_IOCTL_TSK_INIT	0x19	None	Int	Initiate Task Table

Table 4. AMP Control Interface Command – Task Operation

- OMAP_AMP_IOCTL_CAL_COST

  This command is used for calculating a task cost specified by input char parameter. When the task registrar receives the command, it calls the cost calculator and returns a lower cost value between two cores. Note that this command doesn't change values in the run-time task table and will not influence the runtime cost calculator value.

- OMAP_AMP_IOCTL_TSK_REG

  This command is used for registering a task into GPP or DSP version table. Table 5 shows Task_info data structure and all field is need for registering a task.

Variable	Type	Description
Version	Int	Task version – GPP or DSP
Name	Char *	Task Name – Maximum length is 16
Power_cost	Int	Task power consumption information
Computing_cost	Int	Task computing time information

Table 5. Task_info Data Structure

- OMAP_AMP_IOCTL_TSK_GET

  This command is used for getting task information. Version and name fields in Task_info data structure are need for this command.

- OMAP_AMP_IOCTL_TSK_DEL

This command is used for deleting task from GPP or DSP version table. Version and name fields in Task_info data structure are need for this command.

- OMAP_AMP_IOCTL_TSK_INIT
  Initiate GPP and DSP version table. All information in version tables will be cleaned away.

Command Name	CMD Value	Parameter	Return Value	Description
OMAP_AMP_IOCTL_ SET_LMD	0x1	Int value[4]	Int	Set four lambda values in Cost Function.
OMAP_AMP_IOCTL_ GET_LMD	0x2	Int value[4]	Int value[4]	Get four lambda values in Cost Function.
OMAP_AMP_IOCTL_ POWER_LMD_ADD	0x3	None	Int	Increase power lambda values in Cost Function by 1.
OMAP_AMP_IOCTL_ POWER_LMD_SUB	0x4	None	Int	Decrease power lambda values in Cost Function by 1.
OMAP_AMP_IOCTL_ COMPU_LMD_ADD	0x5	None	Int	Increase execution time lambda value in Cost Function by 1.
OMAP_AMP_IOCTL_ COMPU_LMD_SUB	0x6	None	Int	Decrease execution time lambda value in Cost Function by 1.
OMAP_AMP_IOCTL_ DEADL_LMD_ADD	0x7	None	Int	Increase deadline fulfillness lambda value in Cost Function by 1.
OMAP_AMP_IOCTL_ DEADL_LMD_SUB	0x8	None	Int	Decrease deadline fulfillness lambda value in Cost Function by 1.
OMAP_AMP_IOCTL_ LDBAL_LMD_ADD	0x9	None	Int	Increase loading balance lambda value in Cost Function by 1.
OMAP_AMP_IOCTL_ LDBAL_LMD_SUB	0xA	None	Int	Decrease loading balance lambda value in Cost Function by 1.

Table 6. AMP Control Interface Command – Lambda Operation

All commands in Table 6 are used for adjusting lambda values in the cost function. The first two commands can set or get four lambda values at once. The remaining commands are used for increasing or decreasing the specified lambda value by 1.

In order to use the proposed AMP scheduler, programmers only need to port the program to DSP and pack data into one transfer unit, write it to DSP task device, and read the returned data from DSP task device on the GPP side. The proposed design tries to reduce the impact of porting existing programs as much as possible. The following paragraph shows an example of how to modify the existing program.

```
while (Interpolate_Dis_y < (EDGE_SIZE+144))
{
  while (Interpolate_Dis_x < (EDGE_SIZE+176))
  {
    idx = Interpolate_Dix_x+Interpolate_ARM_y*edged_width;
    Interpolate_Dix_x += 16;
    halfpel16x16_h16(&h_ptr[idx], &n_ptr[idx], edged_width, rounding);
    halfpel16x16_v16(&v_ptr[idx], &n_ptr[idx], edged_width, rounding);
    halfpel16x16_hv16(&hv_ptr[idx], &n_ptr[idx], edged_width, rounding);
  }
  Interpolate_Dis_y += 16;
  Interpolate_Dis_x = EDGE_SIZE;
}
```

Figure 14. Pure ARM MPEG 4 Video Codec Interpolation Module

Figure 14 shows GPP MPEG 4 video codec (m4v) interpolation module. Its input data is one macro block (MB) and output data are three MBs in every iteration of the while-loop.

In the DSP Gateway, there are two DSP task data receiving type and sending type, active and passive. Only passive receiving and active sending type task suits the proposed design. Figure 15 and Figure 16 show how an active sending DSP task works. Figure 15 shows GPP calls read system call before DSP issues WDSND/BKSND interrupt. Read system call will be blocked until DSP issues sending interrupt and return as soon as receiving sending interrupt. Figure 16 shows GPP class read system call after DSP issues WDSND/BKSND interrupt. Read system call will return immediately. Figure 17 shows how a passive receiving DSP task works. Write system call will return immediately and no need to wait for DSP issues any interrupt.

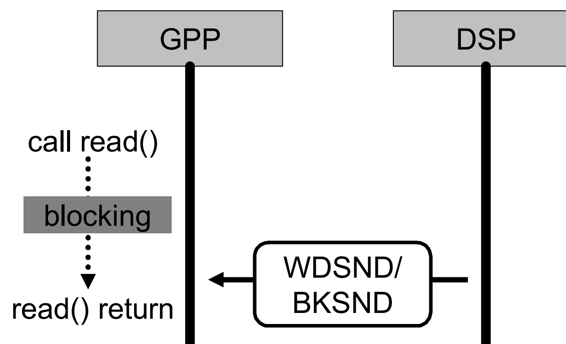


Figure 15. Active Sending DSP Task – Read Before Sending

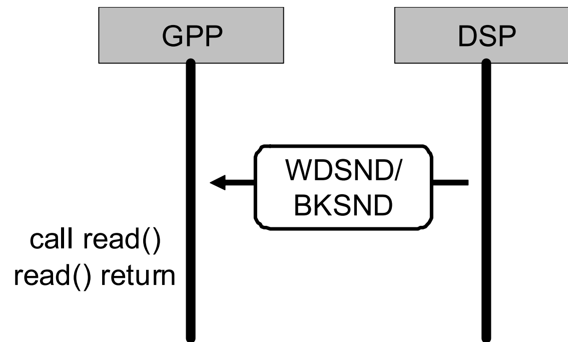


Figure 16. Active Sending DSP Task – Read After Sending

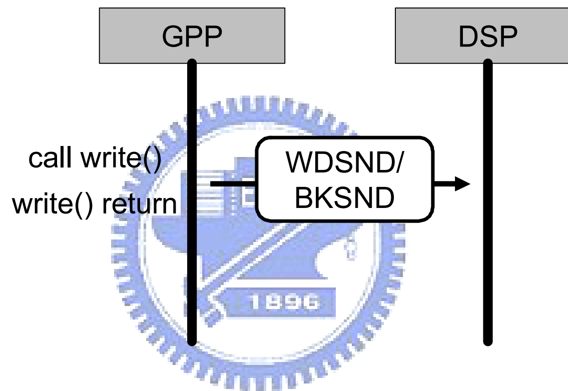


Figure 17. Passive Receiving DSP Task

```
struct tsk_info *tsk;
tsk->version = AMP_DSP;
tsk->name = “m4v_interpolate”;
tsk->power_cost = 10;
tsk->execution_time = 10;
ioctl(amp_ctl_fd, OMAP_AMP_IOCTL_TSK_REG, tsk);
tsk->version = AMP_GPP;
tsk->power_cost = 5;
tsk->exection_time = 20;
ioctl(amp_ctl_fd, OMAP_AMP_IOCTL_TSK_REG, tsk);
```

Figure 18. Task Information Register

Figure 18 shows how to register tasks into the proposed design at runtime. It should be done before using the proposed design. Besides registering at runtime, the task also can be registered at system boot time.

If the program only uses one thread to computing data on GPP-side and communicate with DSP. Due to GPP may not receive data sent by DSP immediately, DSP will idle before GPP receives data and transfers next data.

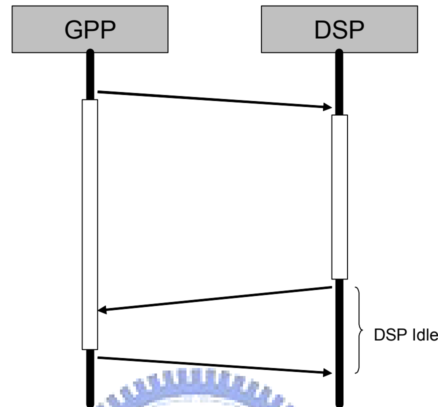


Figure 19. Dual Mode M4V Interpolate with Single-Thread

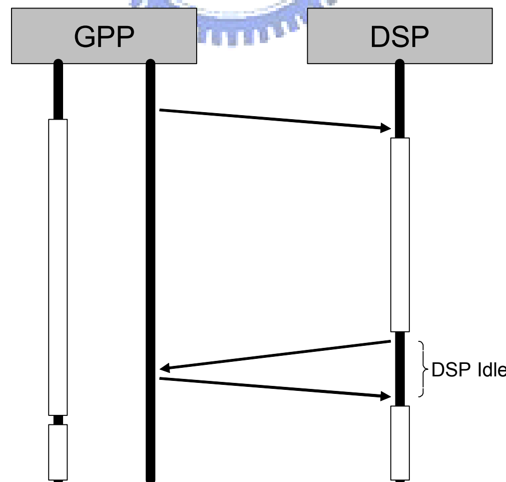


Figure 20. Dual Mode M4V Interpolate with Multi-Thread

Figure 19 and Figure 20 show how a single-thread and a multi-thread program run. A multi-thread program uses two threads on GPP-side, one for process data on GPP-side and

another for transferring or receiving data from DSP. By using multi-thread, the DSP idle time can minimize to data communicating time.

Thread GPP:

```
Void *cal(void *)
{
  Interpolate_ARM_x = Interpolate_Dis_x;
  Interpolate_ARM_y = Interpolate_Dis_y;
  idx = Interpolate_ARM_x+Interpolate_ARM_y*edged_width;
  halfpel16x16_h16(&h_ptr[idx], &n_ptr[idx], edged_width, rounding);
  halfpel16x16_v16(&v_ptr[idx], &n_ptr[idx], edged_width, rounding);
  halfpel16x16_hv16(&hv_ptr[idx], &n_ptr[idx], edged_width, rounding);
  pthread_exit(NULL);
}
```

Figure 21. Multi-Thread M4V Interpolate Module – GPP Thread

Thread DSP:

```
Static Uns cal(struct dsptask *task, Uns bid, Uns cnt)
{
  src_data = udata;
  out_h = out_data;
  out_v = out_data+256;
  out_hv = out_data+512;

  halfpel16x16_h16(out_h, src_data, 0, rounding);
  halfpel16x16_v16(out_v, src_data, 0, rounding);
  halfpel16x16_hv16(out_hv, src_data, 0, rounding);

  bksnd(task, bid, 768);
}
```

Figure 22. Multi-Thread M4V Interpolate Module – DSP Thread

```
Thread Listener:

Void *dsp_listener(void *)
{
  Interpolate_DSP_x = Interpolate_Dis_x;
  Interpolate_DSP_y = Interpolate_Dis_y;
  read(dsp_int_fd, read_buf, 768);
  idx = Interpolate_DSP_x+Interpolate_DSP_y*edged_width;
  memcpy(&h_ptr[idx], read_buf, 256);
  memcpy(&v_ptr[idx], &read_buf[256], 256);
  memcpy(&hv_ptr[idx], &read_buf[256], 256);
  pthread_exit(NULL);
}
```

Figure 23. Multi-Thread M4V Interpolate Module – DSP Data Listener

```
Thread Control:

Pthread_t dsp_pt, arm_pt;
data_size = 17*17;  // One 16x16 macro block with one extra row on bottom
                    //  and one extra column on right
while (Interpolate_Dis_y < (EDGE_SIZE+144))
{
  while (Interpolate_Dis_x < (EDGE_SIZE+176))
  {
    idx = Interpolate_Dis_x+Interpolate_Dis_y*edged_width;
    ret = ioctl(amp_ctl_fd, OMAP_AMP_IOCTL_DSC, f_ptr);
    if (ret) pthread_create(&arm_pt, NULL, f_ptr, NULL);
    else pthread_create(&dsp_pt, NULL, f_ptr, NULL);
    Interpolate_Dis_x += 16;
  }
  Interpolate_Dis_y += 16;
  Interpolate_Dis_x = EDGE_SIZE;
}
```

Figure 24. Multi-Thread M4V Interpolate Module – Control Thread

Figure 21 and Figure 22 show an example to implement a multi-thread program. The DSP and GPP version modules can be implemented with minimal differences. Besides indexes operations before function calls (halfpel16x16_h16, halfpel16x16_v16, and halfpel16x16_hv16), the main difference between these two modules is termination

function call. In GPP version module, the program exits by invoking the pthread_exit function call and the DSP version module exits by invoking the bksnd function call. In the future, we will implement a wrapped routine to eliminate this difference. Figure 23 shows a GPP-side thread to communication with DSP-side thread. The thread will be blocked on read system call until DSP-side thread completed the task process and returns data to GPP. After receiving data from DSP, the thread writes data to corresponding index and call pthread_exit to terminate itself. Figure 24 shows a main program flow controller. The program uses ioctl system call to invoke the proposed design and creates corresponding thread using pthread depends on the returned function pointer and value of the ioctl system call.

5. Experimental Results and Analysis

In this chapter, the computation components from an MPEG-4 Simple Profile encoder are used to test the proposed tightly-coupled AMP system. The interpolation module in MPEG-4 video codec is used in the following experiment. In MPEG-4 Simple Profile, motion compensation (MC) and motion estimation (ME) are done with half-pixel accuracy. Therefore, sub-pixel interpolation of the original reference frame pixels for both MC and ME is necessary. Figure 25 shows the input and output to the interpolation module.

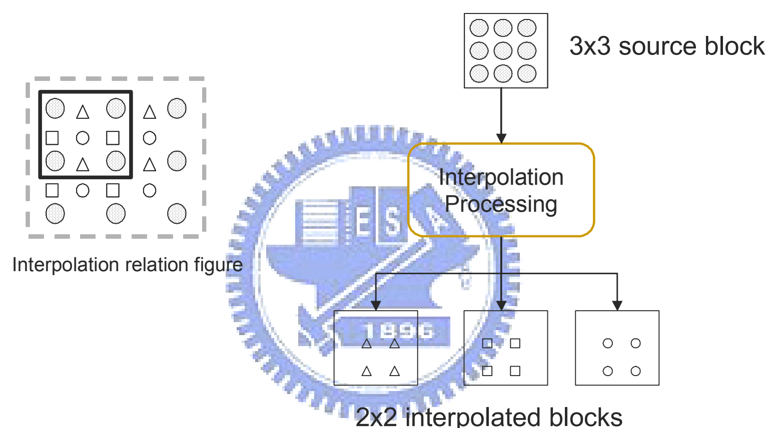


Figure 25. Input/Output of Interpolation

5.1. Dynamically Task Dispatching Rate Experiment

In this experiment, the goal of the experiment is to test the task dispatching result under different task information settings and λ values. The experiment settings are showed in following two tables and explained in following paragraphs.

	Power	Execution	Deadline	Load Balance
λ set 1	1	1	1	1
λ set 2	0	1	0	0

Table 7. λ Set

Table 7 shows the two λ set used in the following experiment. These λ values are introduced in the chapter 3. By setting different λ values, factors in the cost function can be calculated with different weight. The λ set 1 in the second row of Table 7 means the cost function concerns about all factors, power consumption, execution time, deadline fulfillness and load balance, as equal. The λ set 2 in third row means the cost function only concerns about execution time.

Test 1	Power	Execution
GPP	50	40
DSP	30	10

Test 2	Power	Execution
GPP	50	20
DSP	30	10

Table 8. Task Information In Experiment

Table 8 shows the task information in the experiment. In the proposed design, the program is requested to provide the power consumption and the execution time information of the task. In the test 1, the power consumption value of the GPP task is 50 and the execution time value is 40, and the power consumption value of the DSP task is 30 and the execution time value is 10. The test 2 is similar to the test 1. Note that all values in Table 8 are presumed values for experiment only and not measured data.

Test 1	GPP	DSP
λ set 1	25	241
λ set 2	30	236

Test 2	GPP	DSP
λ set 1	38	226
λ set 2	53	211

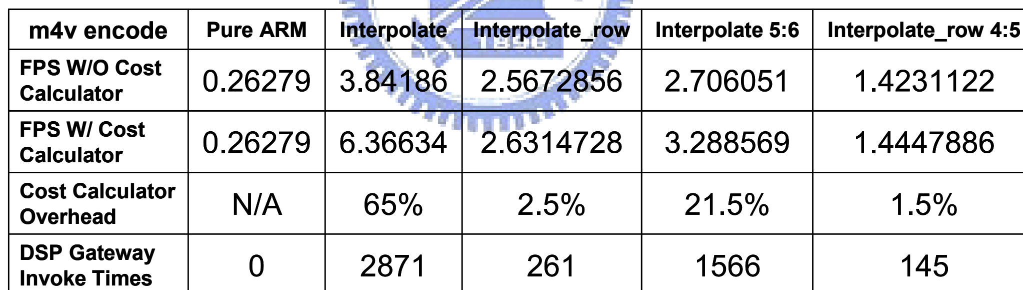
Table 9. Task Dispatching Result

Table 9 shows the results of task dispatching based on settings showed in Table 7 and

Table 8. In the second row of the test 1 and the test 2, although the power consumption and execution time value of DSP task are smaller than GPP task, there are still some task will be assigned to GPP because the value of the other two factors are weighted as large as the power consumption and the execution time. Besides that, the intervention of the unbalance penalty mechanism also influences the result of the cost function. In the third row of the test 1 and the test 2, the DSP to GPP dispatching ratio is smaller than λ set 1 because λ set 2 only concerns about execution time factor and the influence of the unbalance penalty mechanism is larger than λ set 1 relatively.

5.2. M4V Interpolation Module Experiment

Table 10 shows the computing time of the interpolator (for motion compensation and motion estimation) under different operating modes. The test sequence used is the QCIF version of the FOREMAN sequence and its length is 30 frames. The encoding configuration in the experiment is the first frame is encoded as I-frame and all following are encoded as P-frame. The interpolation module in the experiment is invoked 2871 times in the entire encoding flow.



m4v encode	Pure ARM	Interpolate	Interpolate_row	Interpolate 5:6	Interpolate_row 4:5
FPS W/O Cost Calculator	0.26279	3.84186	2.5672856	2.706051	1.4231122
FPS W/ Cost Calculator	0.26279	6.36634	2.6314728	3.288569	1.4447886
Cost Calculator Overhead	N/A	65%	2.5%	21.5%	1.5%
DSP Gateway Invoke Times	0	2871	261	1566	145

Table 10. M4V Interpolation Module Experiment Result

The interpolate_row in fourth row stands for the program transfer one row data to DSP at once. The interpolate 5:6 in fifth row stands for the task distribution ratio between GPP and DSP is 5:6. The experiment result shows that less DSP Gateway invoke times results in lower interpolation time. In Table 10, it is obvious that the results using DSP interpolation module is much slower than pure ARM module. In the following sections, we will explain how this happens.

5.3. Dynamic Task Dispatching – No OS Environment

The experiment is based on the work of [10]. Table 2 shows computing time for different computing units, including motion estimation (ME), interpolation, and discrete cosine transform (DCT), under ARM/DSP/Dual mode. The test sequence used is the QCIF version of the STEFAN sequence and its length is 150 frames. These tests were conducted on a TI OMAP 1510 platform (the PSI Innovator).

Under ARM mode, the computing units in Table 2 are processed on ARM core and didn't use any DSP hardware extension. Under DSP mode, the computing units are processed on DSP core using C55x hardware extension. Under dual mode, the computing units are processed on both cores. Table 3 shows time per computing unit ratio under all modes.

Table 12 shows time per computing unit ratio under all modes.

Unit: us

	ARM	DSP	Dual
ME	3883	357	349
Interpolation	441	282	227
DCT	764	342	276

Table 11. Time per Computing Unit

	ARM	DSP	Dual
ME	11.12	1.02	1
Interpolation	1.94	1.24	1
DCT	2.77	1.24	1

Table 12. Time per Computing Unit Ratio

This experiment shows that the dual version can reach faster execution time than pure ARM and pure DSP version. In the next section we will try to find out where is overhead come from in our implementation system.

5.4. DSP Gateway Data Transfer Experiment

The next experiment tests the data transfer time between two cores. Table 4 shows the data size and iterations of transfer of our experiment. In group 1, for example, a Linux application transfers 12672 bytes of data three times to the DSP core through SDRAM/SARAM/DARAM. All groups in our experiment transferred same amount of data (38016 bytes) at different number of iterations. In this experiment, ARM and DSP run at 192 MHz, and Traffic Controller (TC) runs at 96 MHz.

Group	1	2	3	4	5	6	7	8
Transfer Size (bytes)	12672	6336	3168	1584	792	396	198	176
Transfer Time	3	6	12	24	48	96	192	216

Table 13. Experiment Group Setting

Due to the limitation of SARAM and DARAM capacity, there are no group 1 result for SARAM and group 1, 2, and 3 results for DARAM experiments.

Figure 26 and Figure 27 show transfer rates from GPP to DSP and from GPP to DSP to GPP. All results show transfer time influences transfer rate.

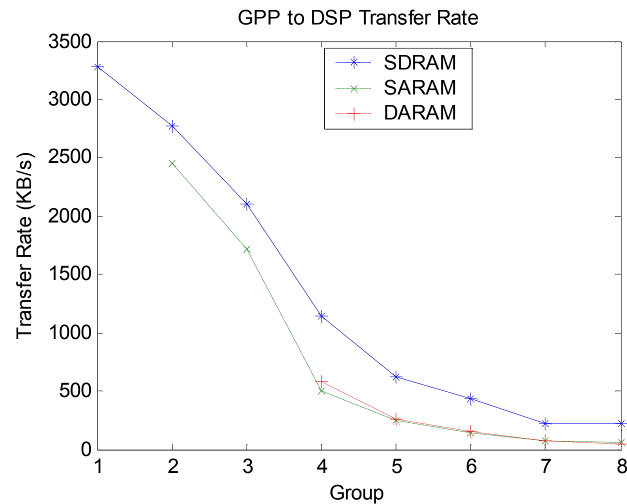


Figure 26. GPP to DSP Transfer Rate

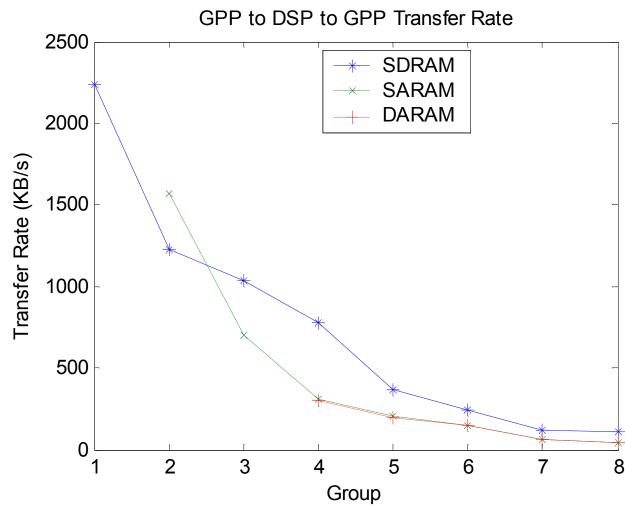


Figure 27. GPP to DSP to GPP Transfer Rate

GPP writes data to SDRAM through EMIFF Interface in Memory Interface Traffic Controller, and to SARAM and DARAM through MPU Interface (MPUI). It is obvious that transfers data through SDRAM is faster than through SARAM and DARAM.

1896

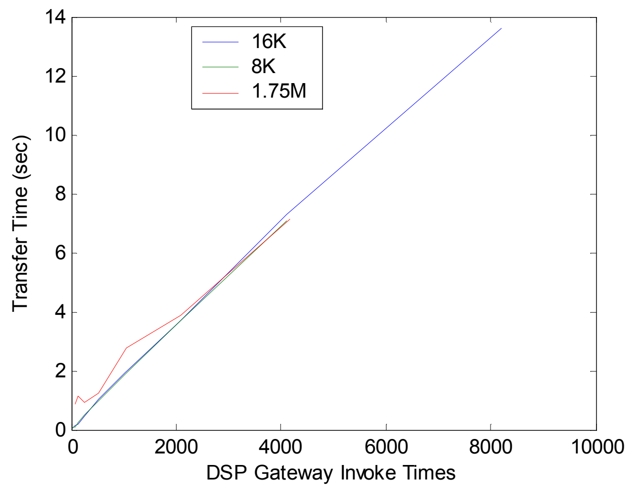


Figure 28. DSP Gateway Invoke Times

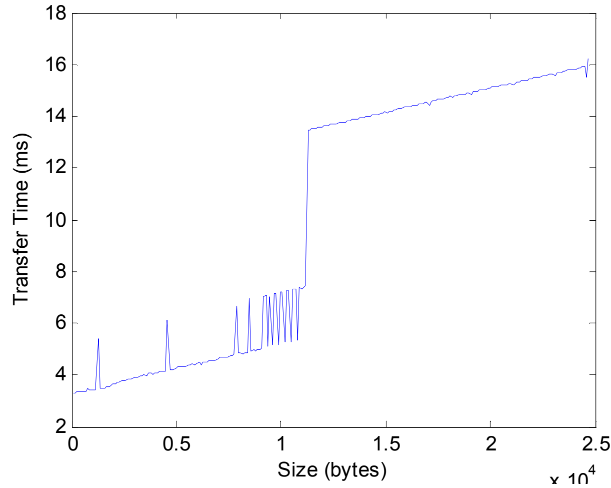


Figure 29. DSP Gateway Different Size Transfer

Figure 28 shows the relationship between the DSP Gateway invoke time and the total transferred time under different total transferred data size. Three different data size, 8KB, 16KB and 1.75MB, are tested in the experiment and the invoke time range of DSP Gateway is from 1 to 8192. The result of the experiment shows that the total transfer time most depended on DSP Gateway invoke time and the different transferred size is a less important factor. Figure 29 shows the relationship between the total transferred size and transfer time. The experiment transfers different data sizes with the same DSP Gateway invoke time. Compared with Figure 28, the slope of Figure 29 is smaller than Figure 28. It is obvious that the DSP Gateway invoke time is a more important influence factor to transfer time than the transferred data size.

In the experiment, more transferring times using DSP Gateway result in lower transfer rate. Linux kernel system calls and device driver subsystem using by DSP Gateway are significant overhead in this experiment. Figure 30 shows a system call flow.

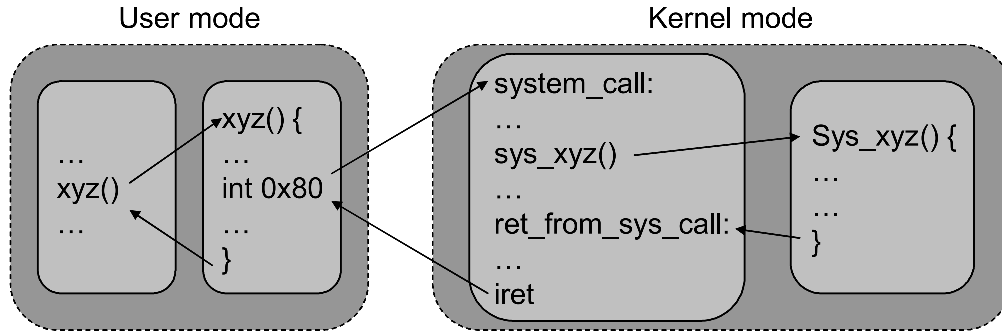


Figure 30. Linux System Call

When a user application invokes a system call xyz(), the wrapper routine in libc standard library is called and issues a 0x80 interrupt. After interrupt, the program enters kernel mode, jumps to system_call table, finds the corresponding system call routine, and the system call routine services the call then returns to user mode.

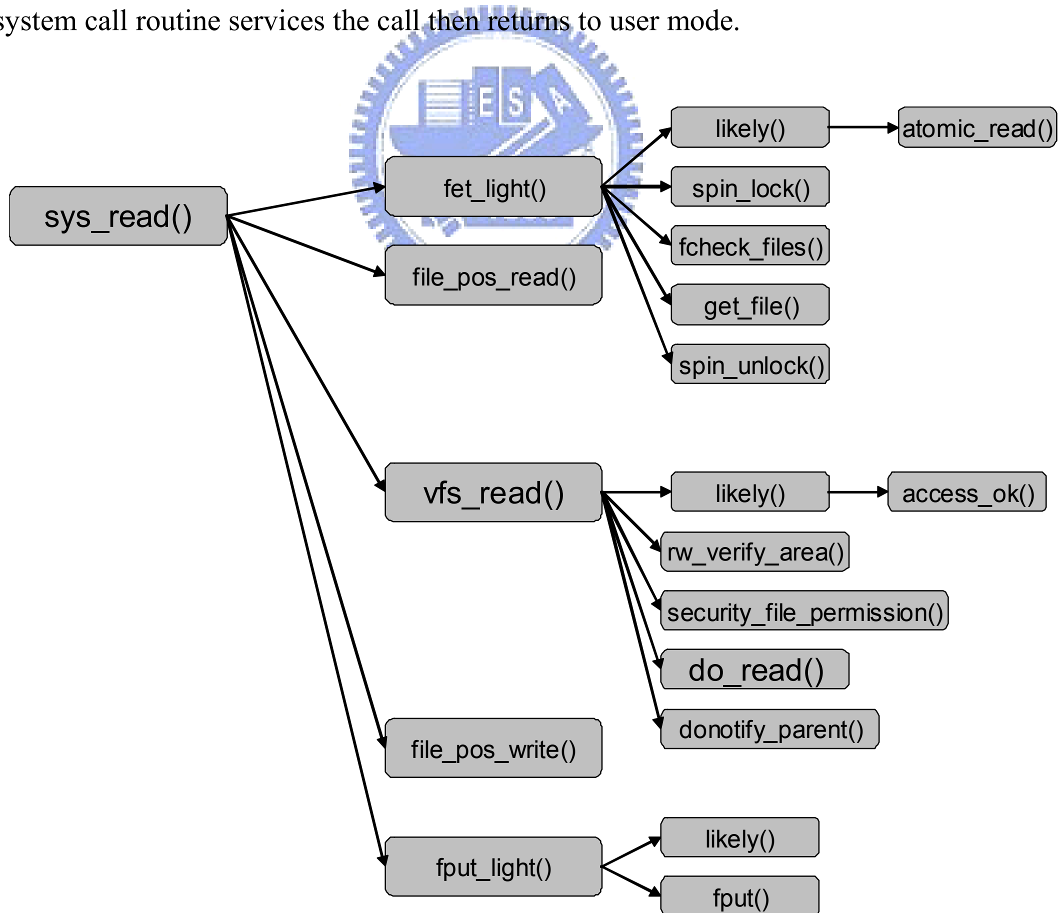


Figure 31. Read() system call flow

Figure 31 shows how a read() system call works. There are more than 10 functions need by a single read() system call. Most of them are for checking or protecting the kernel. The Linux system call flow showed above can be simplified in the embedded system because the embedded system environment is much simply than existing microcomputer environment.

Besides Linux kernel system calls overhead, every GPP to DSP and GPP to DSP to GPP transfer bring 2 and 5 mailbox interrupts. This mechanism results in considerable impact on transfer rate and overall system performance.

6. Conclusion and Future Work

In this paper, we propose a unified scheduler for tightly-coupled AMP systems. The scheduler performs dynamic task partitioning between the GPP core and the DSP core at runtime based on the system state. A sophisticated cost function is proposed to take into account runtime power consumption, execution time, deadline fulfillness, and load balance simultaneously. Even though the system requires a new programming practice for efficient implementation, the impact on the programmers is minimal. This approach is more promising for next generation's multimedia embedded systems than the common loosely-coupled dual-core systems used today. Current scheduler is implemented as an add-on component to the Linux kernel for proof of concept. In the future, a native OS kernel will be implemented base on this design for more efficient communication between two cores.

7. References

[1] Abdelzaher T., Andersson B., Jonsson J., Sharma V., and Nguyen M. The Aperiodic Multiprocessor Utilization Bound for Liquid Tasks. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, September 2002.

[2] Albert G. Greenberg and Paul E. Wright. Design and Analysis of Master/Slave Multiprocessors. *IEEE Transactions on Computers*, VOL.40, NO.8, August 1991.

[3] Alberto Avritzer, Mario Gerla, Berthier A. N. Ribeiro, Jack W. Carlyle and Walter J. Karplus. The Advantage of Dynamic Tuning in Distributed Asymmetric Systems. In *Proceedings of INFOCOM*, 1990.

[4] Anderson J., and Srinivasan A. Early Release Fair Scheduling. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, June 2000.

[5] Anderson J., and Srinivasan A. Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, June 2001.

[6] Andersson B., Baruah S., and Jansson J. Static-Priority Scheduling on Multiprocessors. In *Proceedings of the IEEE Read-Time Systems Symposium*, December 2001.

[7] Andersson B., and Jonsson J. Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or Not to Partition. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, December 2000.

[8] Baruah S., Cohen N., Plaxton G., and Varvel D. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica 15*, 6, 600-625. June 1996.

[9] Burchard A., Liebeherr J., Oh Y., and Son S. H. Assigning Real-Time Tasks to Homogeneous Multiprocessor Systems. *IEEE Transactions on Computers*, 44, 12, 1429-1442. December 1995.

[10] Chien-Tang Tseng, Chih-Peng Wang, and Chun-Jen Tsai. Dynamic MPEG-4 Video Encoder Partitioning on Asymmetric Dual-Core Platforms. In *Proceedings of the 15th VLSI Design/CAD Symposium*, 2004.

[11] Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory 3rd Edition*. February 6, 1998.

[12] Ha R., and Liu J. W. S. Validating Timing Constraints in Multiprocessors and Distributed Real-Time Systems. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems,* June 1994.

[13] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Butazzo. Implications of Classical

Scheduling Results for Real-Time Systems. Technical Report UM-CS-94-089, Computer Science Department, University of Massachusetts, 1994.

[14] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384-393, June 1975.

[15] James W. Wendorf, Roli G. Wendorf and Hideyuki Tokuda. Scheduling Operating System Processing on Small-Scale Multiprocessors. In *Proceedings of the Twenty-Second Annual Hawaii International Conference*, 1989.

[16] K. K. P. Research. Increasing functionality in set-top boxes. In *Proceedings of IIC-Korea, Seoul*, 2001.

[17] Lopez J. M., Garcia M., Diaz J. L., and Garcia D. F. Worst-Case Utilization Bound for EDF Scheduling in Real-Time Multiprocessor Systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, June 2000.

[18] M. R. Garey and D. S. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal on Computing*, 4(4):397-411, 1975.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[20] Maurice J. Bach and S. J. Buroff. Multiprocessors UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733-1749, October 1984.

[21] Momtchil Momtchev and Philippe Marquet. An Asymmetric Real-Time Scheduling for Linux. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[22] Oh D. I., and Baker T. P. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems: The International Journal of Time-Critical Computing*, 15, 183-192. 1998.

[23] *OMAP5912 Applications Processor Data Manual*. Texas Instruments. Dallas, Texas. [Online]. Available: http//www.ti.com.

[24] Paolo Gai, Luca Abeni and Giorgio Buttazzo. Multiprocessor DSP Scheduling in System-on-a-chip Architectures. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[25] Saowanee Saewong and Ragunathan Rajkumar. Cooperative Scheduling of Multiple Resources. In *Proceedings of 20th IEEE Real-Time Systems Symposium*, 1999.

[26] S. Sriram, and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000.

[27] The OMAP Linux Kernel Team. *Linux 2.6.11 omap1 patch file*. [Online]. Available: http://www.muru.com/linux/omap/.

[28] Toshihiro Kobayashi. *DSP Gateway Dynamic Loader Daemon (dsp_dld) Specification*. May 7, 2005.

[29] Toshihiro Kobayashi. *DSP Gateway Linux 2.6.11 omap1 patch file*. [Online]. Available: http://dspgateway.sourceforge.net.

[30] Toshihiro Kobayashi. *Linux DSP Gateway Specification Rev 3.2*. May 4, 2005.

[31] Wayne Wolf. *Computers as Components: Principles of Embedded Computing Systems Design*. Morgan Kaufmann Publishers, 2000.