國 立 交 通 大 學

電信工程學系

碩 士 論 文

嵌入式雙核心微處理器之耗電分析與評估器

Energy Estimator/Analyzer for Embedded Dual Core Processor

研究生：許君豪

指導教授：曹孝櫟 、李育民 教授

中 華 民 國 九 十 五 年 十 月

嵌入式雙核心微處理器之耗電分析與評估器
Energy Estimator/Analyzer for Embedded Dual Core Processor

研　究　生：許君豪	Student：Chun-Hao Hsu
指導教授：曹孝櫟	Advisor：Shiao-Li Tsao
李育民	Yu-Min Lee

國　立　交　通　大　學
電　信　工　程　研　究　所
碩　士　論　文

A Thesis

Submitted to Institute of Communication Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Communication Engineering

October 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年十月

嵌入式雙核心微處理器之耗電分析與評估器

學生：許君豪　　　　指導教授：曹孝櫟、李育民

國立交通大學電信工程學系碩士班

摘要

近年來，嵌入式系統常採用異質雙核心或多核心微處理器來增進效能並減少成本及耗電，例如行動電話、個人數位助理等。如何在異質雙核心平台上設計出符合效能且節省耗電的軟體是非常重要的議題。目前已經有相當多軟體耗電模擬、分析之技術和工具來幫助評估之軟體耗電。然而，這些技術與工具軟體大多針對單核心微處理器。目前尚無適合評估異質雙核心微處理器耗電的方法和工具被提出。本論文提出評估與分析異質雙核心軟體耗電之方法並且實做成工具軟體。此工具軟體不僅評估雙核心應用程式之總耗電，並且分析其耗電與時間之分佈。此工具軟體可幫助程式設計者更有效率、更簡便地觀察並診斷雙核心應用程式之耗電問題。我們混合多種高層次方法建立耗電模型，並藉由硬體計數器追蹤軟體動態執行時之耗電與時間分佈。實驗結果顯示評估之耗電與實際量測最大誤差在百分之五以內。

Energy Estimator/Analyzer for Embedded Dual Core Processor

Student: Chun-Hao Hsu Advisor: Shiao-Li Tsao, Yu-Min Lee

Department of Communication Engineering
National Chiao Tung University

Abstract

Recently, embedded systems such as mobile phones adopt heterogeneous dual-core or multi-core processors to improve the performance and reduce the cost and power. One of the critical issues for such an embedded system using heterogeneous dual-core processor is the software and system designs while maximize the computation power of a heterogeneous dual-core and minimize the power consumption of a system. Hence, the software design for heterogeneous dual-core is very important for a low power embedded system using dual-core processors. In order to evaluate the power consumption of the embedded software, a number of power modeling and analysis techniques and tools were proposed. Unfortunately, these models and tools are designed for single core processor, and there is no suitable mythology for modeling and analyzing power consumption for a heterogeneous embedded dual-core processor. In this thesis, an instrument-based tool for energy consumption estimation and analysis of the embedded software running on a dual-core processor is proposed and developed. This tool not only estimates the total consumed energy/time but also analyzes the energy/time distribution of dual-core applications. With the aid of this tool, dual-core application designers could observe and diagnose the power consumption of the applications efficiently and easily. It also provides an opportunity for further software energy optimization. In this work, we apply hybrid high-level modeling techniques and profile the energy and time through hardware timers. Experimental results demonstrate that the max error is less than 5%.

致謝

本篇論文得以順利完成，首先要感謝我的指導老師，曹孝櫟教授。感謝曹老師這些年來的指導與教誨，讓我在做研究的過程中，能解決遭遇到的困難並且養成獨立思考的能力。除了專業上的協助，老師工作及待人處世的態度，更是讓人佩服。很幸運在求學的過程中，能遇到如此傑出的老師，相信這兩年的學習，對我未來的人生將有莫大的幫助。謝謝老師!

我還要感謝實驗室的好朋友們。感謝一正學長、建明學長、海倫學姊的照顧與指導；感謝金璋、凱翔、邦翔、宥霖、雅聯、誌謙、中暉、政龍、建臻等同學及學弟，在課業學習上和生活中的支持、關心及鼓勵，讓我生活中隨時充滿活力與樂趣，謝謝你們。

最後我要感謝家人和女朋友的鼓勵、支持和體諒，你們是我生活的原動力。謹以此論文，獻給你們，謝謝。

目錄

圖目錄

表目錄

I. Introduction

The performance requirement of embedded system grows persistently in order to support modern mobile multimedia applications. More and more embedded system designers are turning toward dual-core processors rather than higher-frequency processors to achieve better performance without incurring significant power consumption penalty. Therefore, embedded application programmers are expected to have increasing opportunities to develop applications on such dual-core embedded platforms that typically contain a general-purpose processor (GPP) running control-oriented task and a digital signal processor (DSP) processing high computation tasks. Considering asymmetric hardware architecture, inter-processor communication overhead between heterogeneous cores, and even the presence of operating system, software energy optimization for dual-core applications becomes a tough work. Without any support, currently, dual-core application programmers could only design applications depending on their own experiences and knowledge, and verify if the power consumption is beyond what can be tolerated through physical current measurement. However, the results from physical measurement could neither explain the power/energy consumption behavior nor indicate the power/energy consumption problem of the dual-core program. It is difficult and inefficient for further power optimization due to lack of information. Thus it can be seen that there is a high demand for dual-core energy estimation and analysis tool that helps to save much time wasted on optimizing the insignificant part of the program.

Previous works on software power estimation typically integrated processor power models into available performance simulators and can be roughly classified into two categories: the cycle-level and instruction-level methods. In cycle-level methods, the power models for each sub-component in the processor are constructed. Power is estimated by calculating the consumption of every component in the target processor cycle by cycle through execution-driven cycle-accurate simulators. Tools implemented with these methods require detailed information about the micro-architecture [1, 2, 3]. However, such low-level information for off-the-shelf processors is generally unable to be obtained by user.

Measurement based instruction level power analysis approach was first presented in [4, 5] to model the power of a given instruction sequence. The basic idea is that the total energy of a program can be modeled as the sum of base energy cost for each instruction. Furthermore, inter-instruction effect, cache misses and pipeline stalls are also taken into consideration for more accurate result. The base cost for an instruction is the average current obtained by executing this instruction repeatedly in an infinite loop. The inter-instruction effect is incurred from extra switches of circuit when two different instructions are executed consecutively. Instruction level power estimation tools are often based on instruction set simulators or tracers that output complete trace of the program during execution. The main drawback is that it takes a lot of measurement for processor with complex instruction set. To solve this problem, a technique to group the instructions into classes is proposed in [6].

Power simulators implemented with above two approaches are quite useful for software energy analysis and optimization in the early design phase. However, they are often too slow for large programs and do not support operating systems. Sinha [7] demonstrated that the variation in the current consumption across different instructions is quite low in some RISC processors such as ARM processors and the energy consumption varies only about 8% among the different programs. It is unnecessary to perform detail instruction trace and inter-instruction analysis when estimating software energy consumption on such platforms. A flat power model for all instructions in the program is applied. And the energy prediction for applications or operating system running on such general purpose processors is simplified to timing estimation or profiling [8].

Software macromodeling [9] and function level [10] power estimation are also strategies proposed to speed up power simulation by means of pre-characterization of programs. The precharacterized macro-operations such as application functions [10], operating system calls [11,12,13], and even for whole tasks obtained form measurement or low level power simulation are stored as energy library in the forms of a constant, a table or equations. Software profilers or simulators are responsible to collect the parameters for these models during execution.

Software profiling is also an old research topic to gather the statistics of interest during execution for performance or energy optimization. Energy profiling is typically used to associate energy consumption to the source code and identify the energy hotspots of the program. Existing energy profiling techniques can be roughly classified into two categories: monitoring-based and modeling-based profiling. Monitoring-based energy profiling [14, 15] frequently interrupts the target platform at runtime to access the program execution context such as program counter and process identifier. At the same time, the current measured from digital multimeter is mapped to software procedures or functions. Modeling-based energy profiling models the energy cost of specific events and then counts the occurrence of the events during execution. The common ways to track the activation events can be done by executing instrumented program, sampling system's state periodically, or modified simulator.

Although a lot techniques and tools on software energy estimation and profiling have been proposed before, none of them targeted towards heterogeneous dual-core processor. In this paper, we propose an energy estimation and analysis tool for dual-core programs and implement it on the popular dual-core embedded processor: OMAP5912. Our work focuses on providing programmers with insight into high-level energy consumption and grouping the information according to the characteristics of dual-core programs. The output of this tool shows both energy and timing profiling. Our aim is to help designer perform high level power-performance tradeoff and optimization rather than functions or algorithms on local processor.

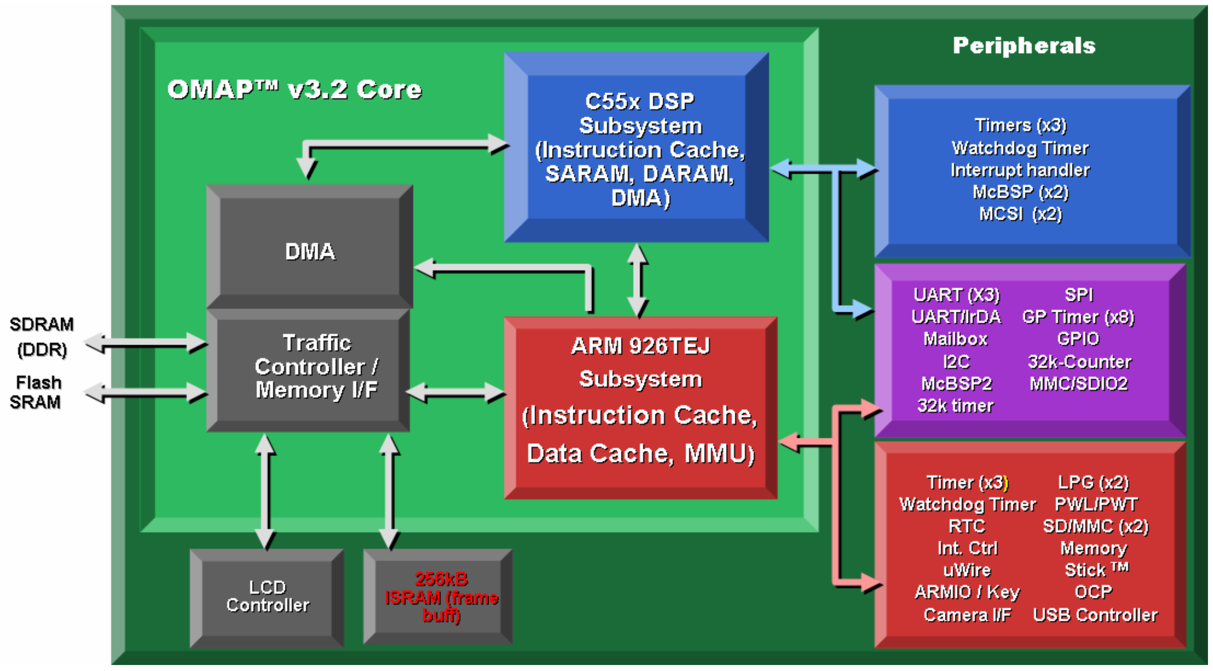
Without available tools which can simulate and trace the behavior of whole dual-core program, we applied hybrid high-level approaches such as reduced instruction level modeling, function level modeling and even physical measurement to model different parts of the dual-core program according to their power consumption behavior. The parts of energy which are hard to be obtained from traditional simulation or measurement are estimated by the power models built in our tool. For example, Linux-side operating system, DSP driver functions, and DSP-side library functions are built in our tool. On the other hand, we leave power model of the task-dependent DSP algorithm which can be pre-characterized easily by traditional approaches as input of the tool. All these high-level power models are represented in the form of average current/power. Finally, we profile the time spent on them through code instrumented in application, kernel and DSP driver. Implementation details are described in Section III.

The rest of the paper is organized as follows. In Section II, we introduce the background technologies of the hardware architecture of dual-core processors, the inter-processor communication mechanism and procedures, and the schedule mechanism in DSP kernel. Section III presents the design and implementation details of this tool. Section IV discusses the experiment results. Finally, conclusions are made in Section V.

II. Background

A. Introduction to OMAP5912

OMAP 5912 Start Kit (OSK5912) includes a dual-core processor OMAP 5912 [16] .OMAP 5912 processor integrates an ARM926EJ-S RISC core and a TMS320C55x DSP core. The C55x DSP core features high performance and low power consumption and is usually in charge of high-computation and real-time jobs. The ARM9 core, which OS typically runs on it, generally performs user interfacing and other house keeping functions. Taking the advantages of both ARM and DSP cores makes OMAP5912 become a powerful multimedia embedded processor.



Peripherals
OMAP™ v3.2 Core
C55x DSP Subsystem (Instruction Cache, SARAM, DARAM, DMA)
DMA
Traffic Controller / Memory I/F
ARM 926TEJ Subsystem (Instruction Cache, Data Cache, MMU)
SDRAM (DDR)
Flash SRAM
LCD Controller
256kB ISRAM (frame buff)
Timers (x3) Watchdog Timer Interrupt handler McBSP (x2) MCSI (x2)
UART (X3) UART/IrDA Mailbox I2C McBSP2 32k timer
SPI GP Timer (x8) GPIO 32k-Counter MMC/SDIO2
Timer (x3) Watchdog Timer RTC Int. Ctrl uWire ARMIO / Key Camera I/F
LPG (x2) PWL/PWT SD/MMC (x2) Memory Stick ™ OCP USB Controller

Figure 1. The OMAP5912 hardware architecture

Figure 1 shows the architecture of the OMAP5912. On-chip caches inside both ARM and DSP reduce the fetch times to external memory. And the memory management units (MMU) support virtual-to-physical memory translation and memory protection. There are two external memory interfaces which separately connect to synchronous DRAM and standard asynchronous memories devices such as SRAM or FLASH. There is an internal memory interface that connects to on chip SRAM to save time/power for frequently used data, such as LCD frame buffer. The OMAP platform contains rich peripherals and peripheral interfaces to support media application. Some of the peripherals are shared and the others are own by either ARM or DSP privately. A LCD controller is also included to support a direct connection to the LCD panel.

B. Inter-processor communication mechanism of OMAP 5912

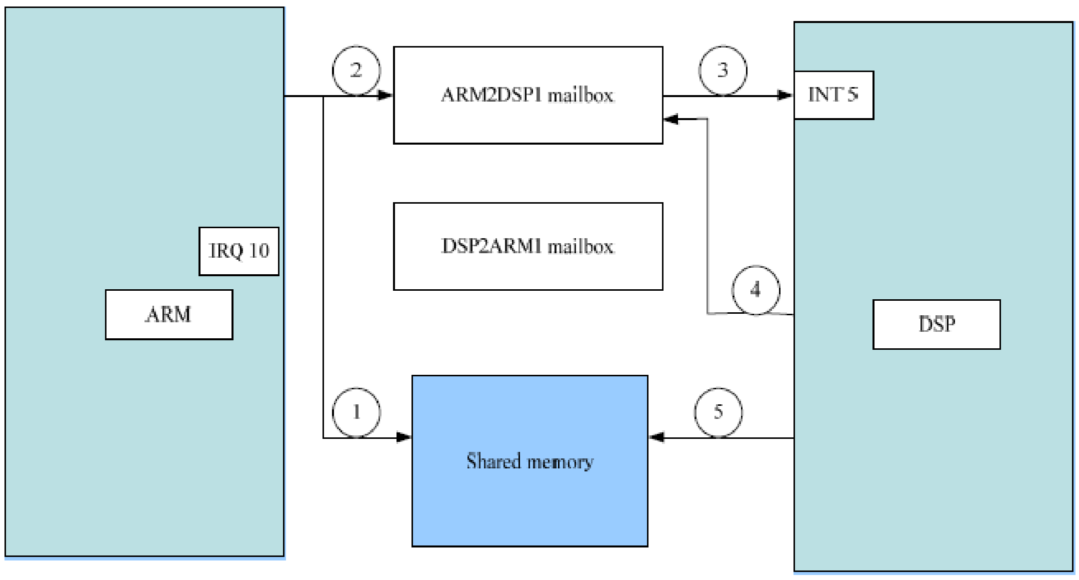


Figure 2. Inter-processor communication mechanism and procedures

For embedded multi-processors such as OMAP, the most frequently used communication mechanism is to implement message passing on shared memory architecture and synchronization via inter-processor interrupt. Figure 2 is an example to illustrate the IPC mechanism and procedures on TI OAMP. In OMAP, there is a global memory that MPU see and control. Both MPU and DSP have local memories. MPU can read and write DSP local memories. Mailbox is the hardware that serves as the synchronization mechanism. It is composed of a set of registers and can generate interrupt to the other processor. When ARM transfers a block of data to DSP, the steps are as follow: 1.ARM writes data to shared memory. 2. ARM writes some information about the size or address of the shared data into registers in mailbox. 3. The mailbox automatically generates an interrupt to DSP and correspondent Interrupt Service Routine (ISR) runs. 4. ISR reads data from registers in mailbox 5. ISR reads data from shared memory and executes the request task.

Many high-level communication primitives provide APIs for inter-processor communication and synchronization. Programmers can directly use the communication primitives to implement parallel programs without handling underlying hardware directly. For OMAP, there is an IPC software, DSPGateway [17], which supports popular Linux operating system.

C. Introduction to DSP Gateway

DSP Gateway [18] is a software and mechanism which provides high level programming model to help programmers easily use ARM and DSP at the same time. DSP Gateway includes Linux driver and DSP-side libraries. They are responsible for hardware settings, interrupt handlings, and communications in between. With the help from DSP Gateway, dual-core applications can be developed without low-level hardware knowledge. From ARM side, Linux applications can communicate with DSP tasks through device files. On DSP side, tasks can easily be synchronized with ARM side by using the API provided by DSP Gateway.

C.1. Introduction to DSP/BIOS and DSP System Kernel (tokliBIOS)

DSP/BIOS [19] is a scalable real-time multi-tasking kernel which manages scheduling and hardware resources on DSP. It is designed for applications that require real-time scheduling and synchronization. DSP/BIOS provides several types of program threads with different priorities. Each thread type has different characteristics. The thread types are: hardware interrupts (HWI), software interrupts (SWI), tasks (TSK), and background thread (IDL), as shown in Figure 3.

HWI functions are the threads with the highest priority in a DSP/BIOS application. An HWI function, also called an interrupt service routine, is triggered after a hardware interrupt occurs. Software interrupts which have priorities lower than hardware interrupts are triggered by calling SWI functions from the program. Both HWI and SWI threads always run to completion. Tasks own lower priority than software interrupts and higher priority than the background thread. Unlike HWI and SWI, Tasks can be suspended during execution when waiting specific events and necessary resources. Background thread which executes the idle loop (IDL) has the lowest priority in a DSP/BIOS application. The background thread runs continuously until it is preempted by higher-priority threads.

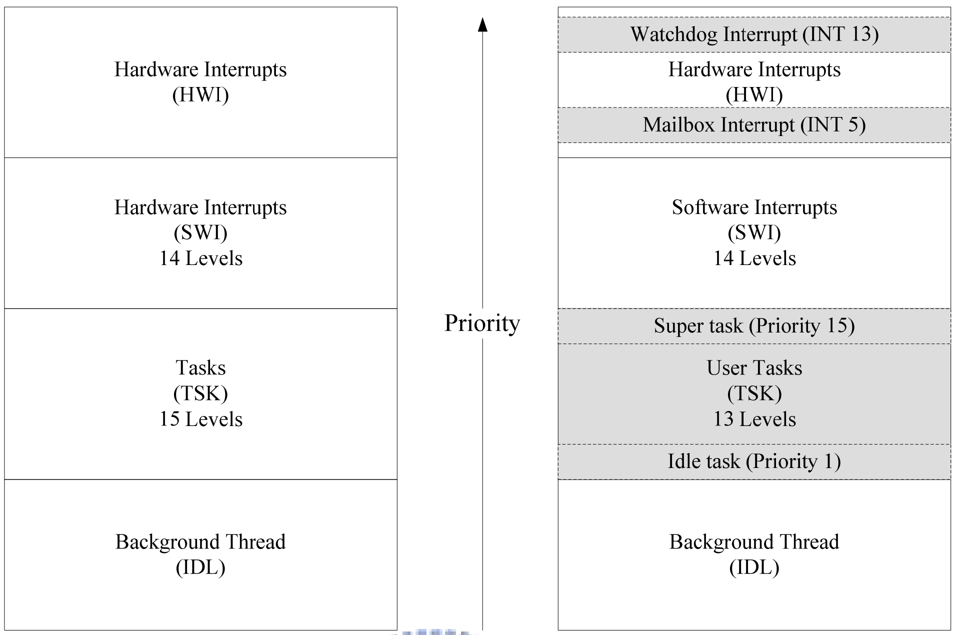


Figure 3. Priority of DSP/BIOS threads and TokliBIOS threads

TokliBIOS is a library developed on the basis of DSP/BIOS. It is provided by DSPGateway to enable inter-processor communication. All user tasks are created as DSP/BIOS task threads. Their priority can be set in the range of 2 to 14. By default, there are two system tasks named supertask and idle task are created. The supertask performs housekeeping jobs such as managing shared buffers for system. The priority of the supertask is 15, higher than any user tasks. The Sleep/Idle task is the task which is executed when all other tasks have nothing to do. The priority of the idle task is 1, lower than any user tasks.

C.2. Thread Scheduling

Each TSK object is always in one of four possible states, as shown in Figure 4: 1. Running state, which means the task is the one actually executing on the processor; 2. Ready state, which means the task is scheduled for execution in case processor is available; 3. Blocked state, which means the task cannot execute until a particular event occurs within the system; or 4. Terminated state, which means the task is terminated and does not execute again.

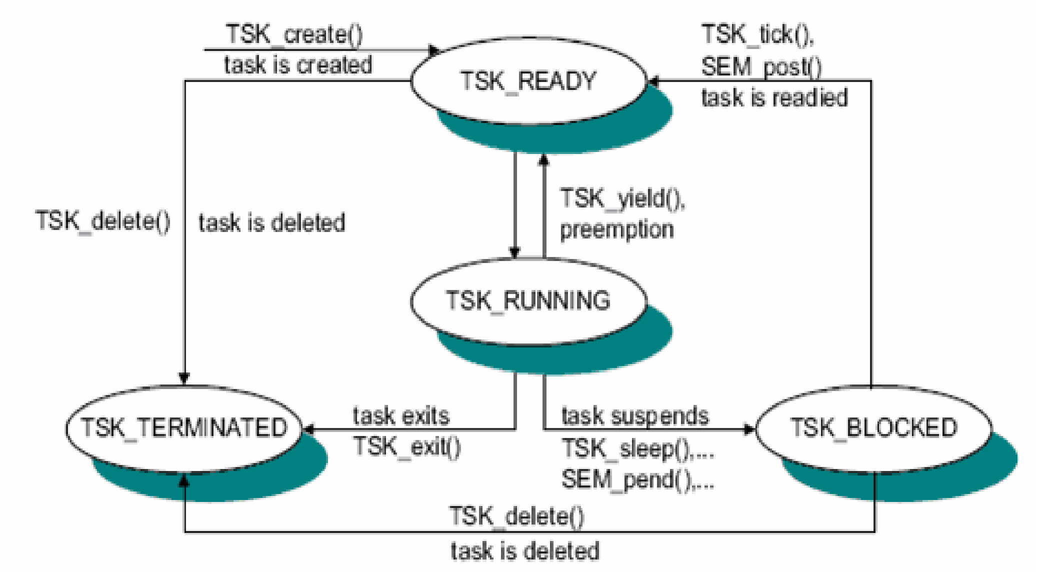


Figure 4. Execution states of DSP/BIOS tasks

Tasks are scheduled for execution according to a priority level assigned to the application. Unlike many time-sharing operating systems, DSP/BIOS immediately preempts the current task whenever a task of higher priority becomes ready to run. There can be no more than one running task. Hence, no ready task has a priority level greater than that of the currently running task. When a task is preempted by a software or hardware interrupt, the task is still TSK_RUNNING because the task will run when the preemption ends. The running task becomes TSK_BLOCKED when it calls a function such as SEM_pend or TSK_sleep that causes the current task to suspend its execution. Tasks can move into this state when they are performing certain I/O operations, awaiting availability of some shared resource, or idling. The running task becomes TSK_TERMINATED by calling TSK_exit, which is automatically called if and when a task returns from its top-level function.

A task that is currently TSK_BLOCKED transitions to the ready state in response to a particular event. After becoming TSK_READY, this task is scheduled for execution according to its priority.

C.3. DSPGateway IPC procedures

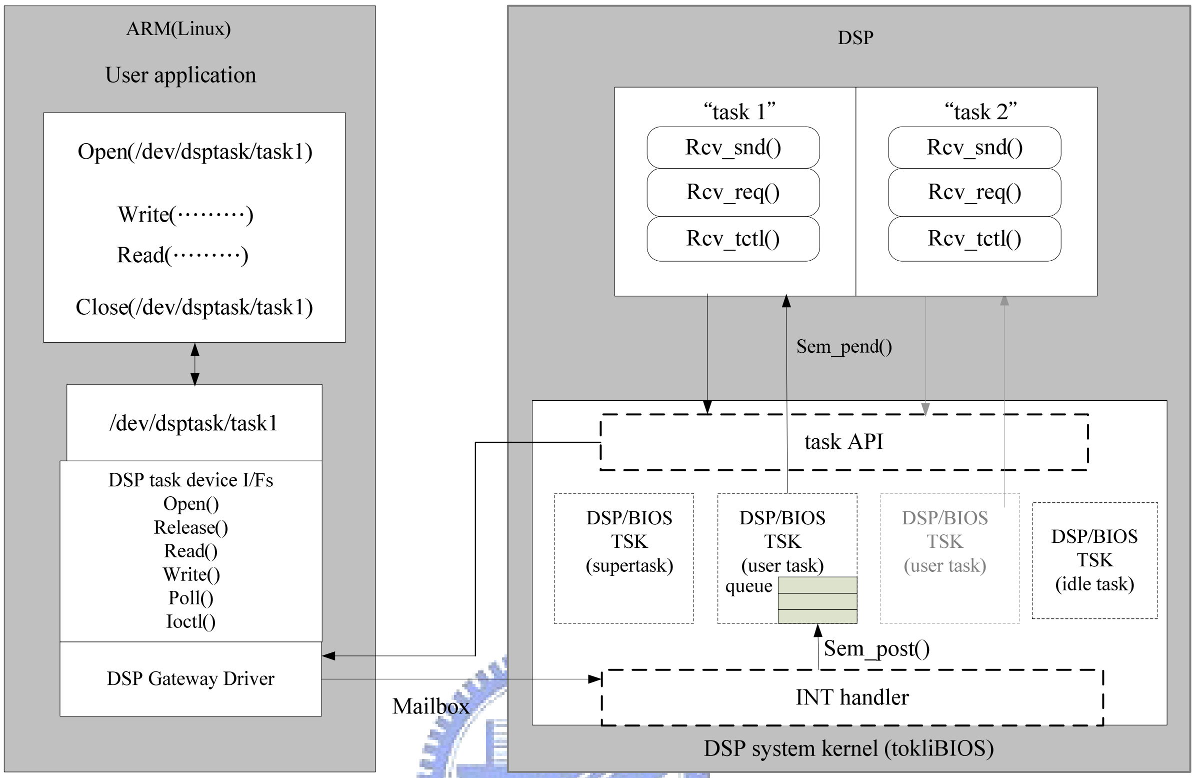


Figure 5. DSPGateway IPC procedures and software block chart

Figure 5 shows the inter-processor communication procedures. There is one user application in Linux and there are two dsp tasks in DSP. In Linux side, the IPC system calls are called by user application to communicate with DSP. In dsp tasks, Rcv_snd(), Rcv_req() and Rcv_tctl() are corresponding functions implemented to respond write(), read(), and ioctl() system calls from Linux-side application. When a Linux user application calls an system call such and accesses to the DSP task device, /dev/dsptask/task1 for example, the 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 and call Sem_post() to increase semaphore count. If task1 is in TSK_BLOCKED state originally, it would switch to TSK_READY state in respond of the semaphore event. After all threads or tasks that have higher priorities than task1 finish, task1 is scheduled and runs. After getting a command from mailbox queue, it would decrease the semaphore counts and then process the commands by calling corresponding task function. The task functions can send back Mailbox commands to ARM by calling task API functions in the tokliBIOS. Until mailbox queue is empty, task1 would switch to TSK_BLOCKED state again and the next lower priority task is scheduled and run. If there are no other tasks to run, the defaulted idle task runs.

III. Design and Implementation

In this section, we describe the design and implementation of the proposed tool.

A. *Design overview*

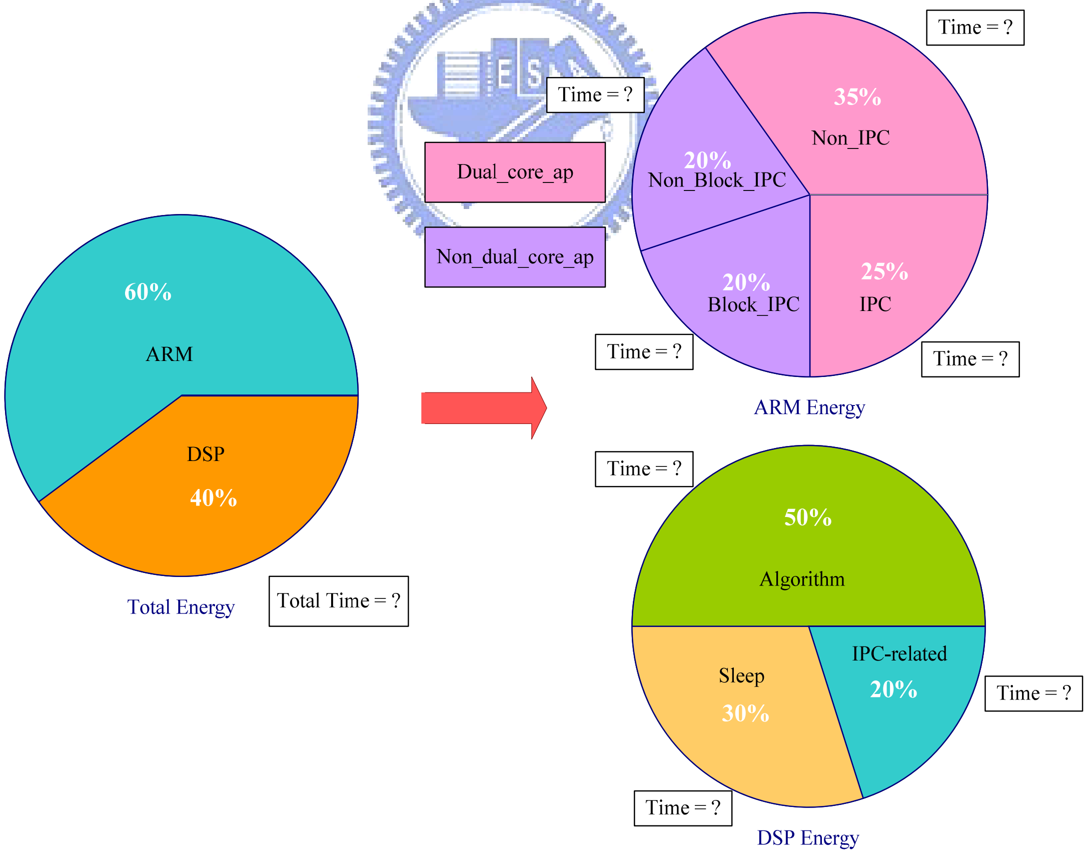


Figure 6. The output information of the tool

Before going into implementation details, we first describe about what information are provided by this tool. Figure 6 shows the output information of this tool. First of all, the tool shows the total energy and execution time consumed by entire dual-core processor during the execution period of the dual-core program. And it also shows the percentages of total energy that spends on ARM and DSP. After understanding the energy/power consumed by ARM and DSP, the energy/time information is further mapped to the each portion of the software of interest.

On ARM side, the dual-core application runs on multitask environment. The energy/time is necessary to be divided into the dual-core application part, denoted as Dual_core_ap, and the other part, denoted as Non_dual_core_ap. The information about how much IPC overhead it takes to communicate with DSP is also important. It can be used to design a more energy/time efficient way for inter-processor communication. The energy/time overhead when performing blocking read/write with IPC system call, named Block_IPC, is also isolated from Non_daul_core_ap part to find out the synchronization problem. It indicates the time/energy consumed by waiting for the response form DSP.

On DSP side, high-computation algorithms such as audio/video codec are implemented inside dsp task in the form of function calls. These algorithms typically consume high energy, so it is isolated as an independent part, named algorithm part. When there is no command or data sent from Linux-side application, DSP executes sleeptask and consumes lower energy. The energy/time consumed by sleep task is aggregated into the Sleep part. This is important information to understand the workload of DSP. There are still some operations to interact with Linux-side application such as IPC functions, IPC-related interrupt service routine, and functions to maintain IPC-related resources. So we group all of them into IPC-related part.

B. Implementation overview

B.1. Measurement Environment

OSK5912 supports individual power measurement capabilities. The current consumed by various components such as entire OMAP5912 core, DSP core, SDRAM, and IOs can be measured separately. The power sources are split into groups and fed through individual jumpers. The current consumption of whole OMAP5912 core and DSP core can be measured by inserting a digital oscilloscope into these jumpers. So we can characterize the power consumption of DSP and MCU domain individually and construct model for them.

B.2. Hardware Timers

Most embedded platforms today provide hardware timers that are user programmable for profiling use. For instance, OSK5912 comes with eight general purpose timers and six operating system timers. General timers are on the shared bus, and both the ARM and DSP can use them. Operating system timers are private peripherals own by either ARM or DSP. Three of the operating system timers are controlled by the MPU, and the other three are controlled by the DSP. All these timer counters can be read and written while counting. They also can be started and stopped at anytime. The clock source fed into these counters can be selected and configured as needed. Profiling with these timers incurs low overhead, high resolution and high accuracy.

B.3. Develop Environment and Software architecture

The implementation is based on Linux 2.6.16 kernel and DSPGateway is configured as default DSP driver. For simplification, we assume that the preemptible kernel option in kernel configuration is not selected. Both ARM and DSP run at 96M Hz.

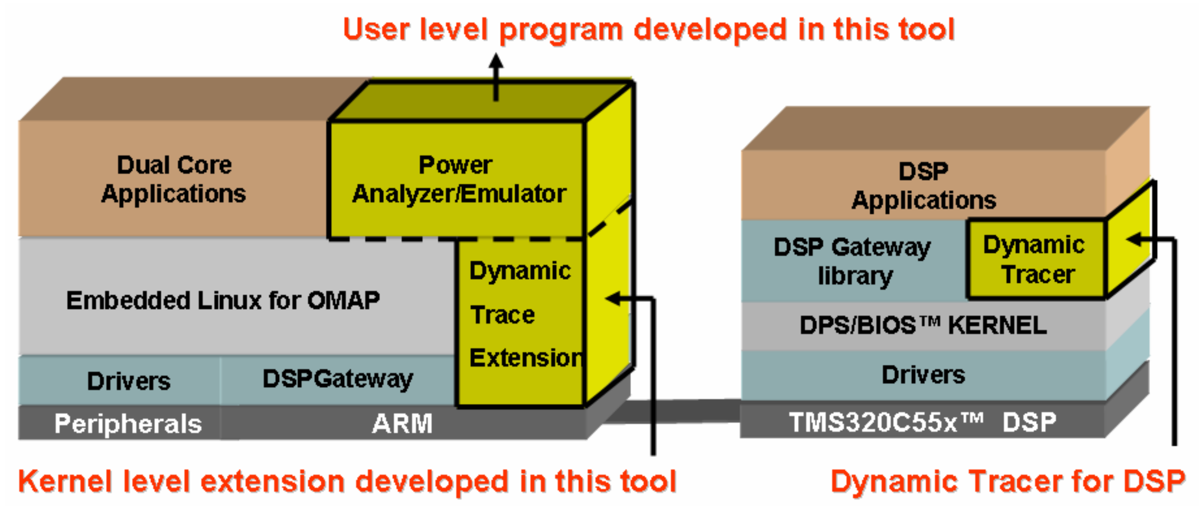


Figure 7. Software architecture of the develop environment and the tool

Figure 7 shows the software architecture of the tool. Embedded operating system runs on ARM processor. Applications on the top level access underlying hardware through operating system and drivers in the middle layers. In the same way, dual-core applications communicate with DSP through OS and DSP driver, DSPGateway. On the other hand, a micro-kernel, named DSP/BIOS which manages the hardware and schedules the tasks also runs on the DSP. Applications running on DSP communicate with application running on ARM by calling DSPGateway library API. The energy estimation and analysis tool we develop is composed of the three components, the three yellow blocks in Figure 7. The Energy Analyzer/Estimator running at user level is the primary program that main functions, such as power consumption models and user interface, are implemented in it. The Dynamic Trace Extension is the patches in both Linux kernel and DSPGateway, while the Dynamic Tracer in DSP is the instrumented version DSPGateway library. They profile the events which occur in runtime and provide the information to the Energy Analyzer/Estimator.

C. Energy Estimation and Analysis for dual-core application on ARM processor

C.1. Power modeling for ARM processor

The power consumption model for ARM core is based on Sinha's work [7]. After testing the free embedded benchmarks, MiBench, it shows the similar result that the current/power variations among different programs running in ARM9 core are low. Therefore, the power consumption of the ARM core is simply divided into active state and idle state. When a task in active queue is scheduled, the processor enters active state. Otherwise, the processor executes idle task and enters idle state. The power in idle state is lower.

	Current (mA)	Power (mW)
Active state	135	216
Idle state	95	152

Table 1. The current and power consumption of active and idle states of ARM processor

C.2. Energy estimation for dual-core application running on ARM processor

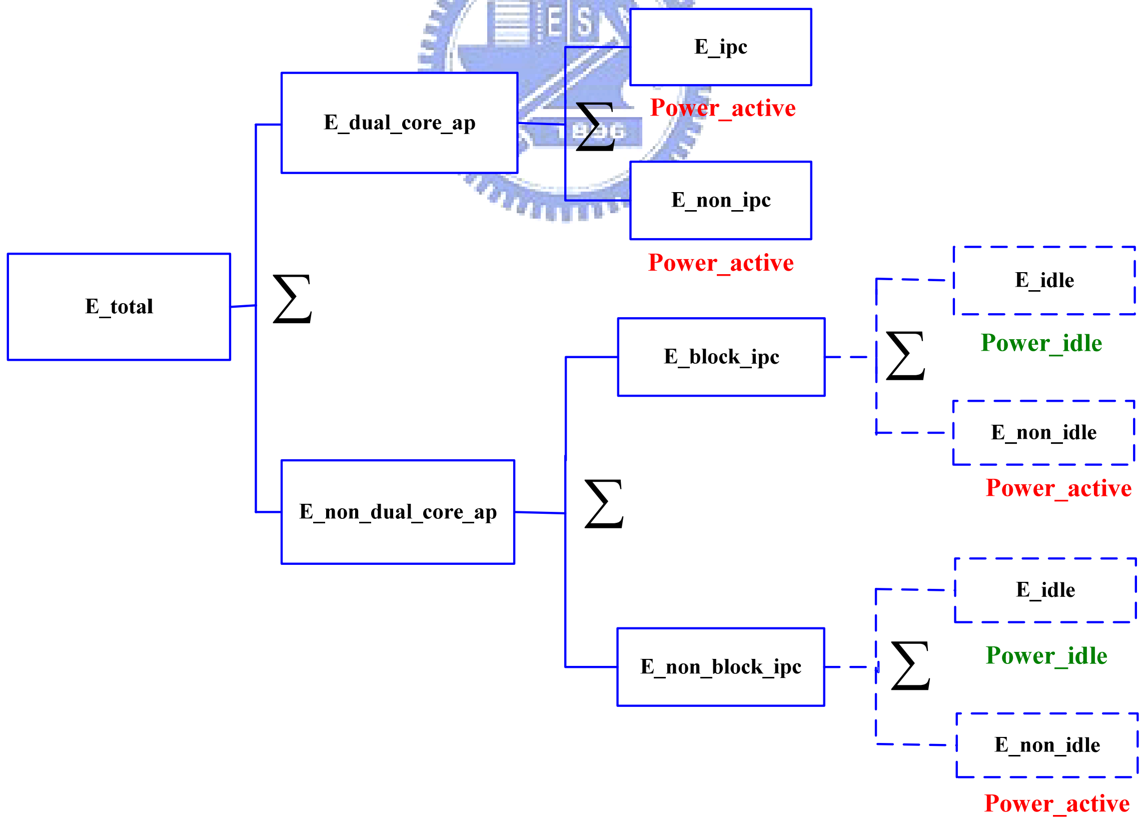


Figure 8. The partition tree for energy profiling

The energy of the each component shown in Figure 8 needs to be collected in the runtime. What we need to do is to profile the time spend on each component. We can get the energy of the most underlying blocks by multiplying the time and the power. Finally, energy of the upper layer components can be obtained by summing the energy of the branching blocks.

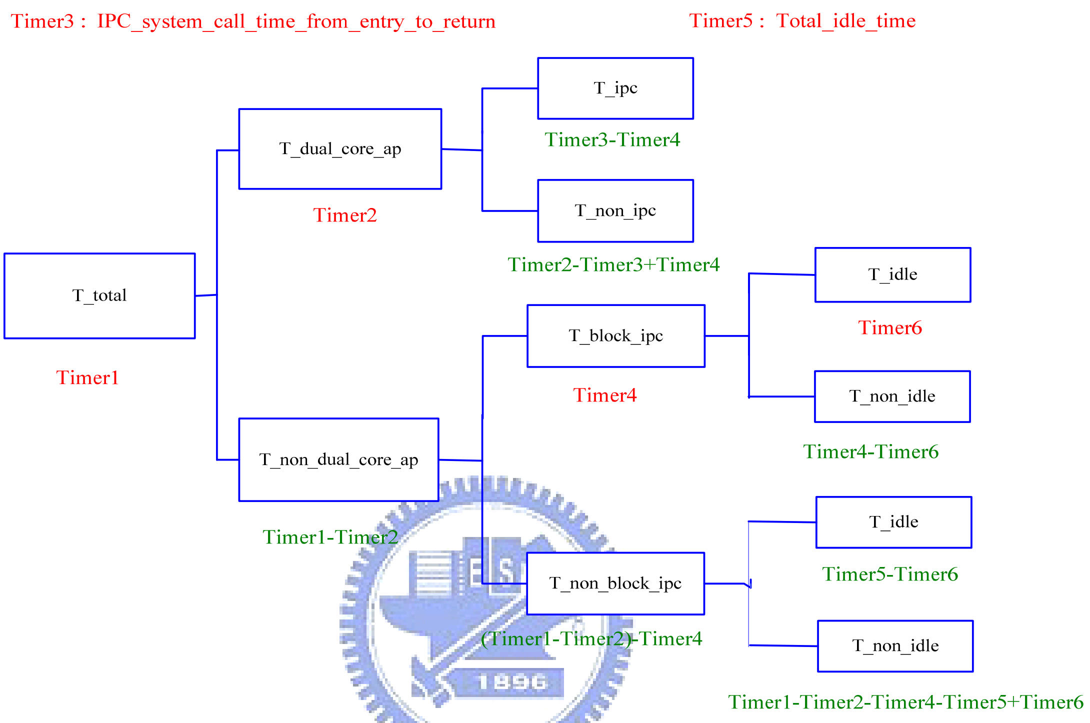


Figure 9. The configuration of the timers for time profiling

As shown in Figure 9, we configure six timers to perform time profiling. Each timer delicately tracks one execution path. The functions which start and stop the timers are injected into proper positions in kernel and DSP drivers. And all necessary time information can be derived by these six timers.

Following we are going to explain the implementation details about where the timers are injected in kernel and DSP driver and how them work.

Total execution time (T_total)

Function	Kernel file	Description
Fork()		Our tool calls fork() function to create a new process
execve()		libc function is called

sys_execve()		libc calls kernel system call
sys_execve()	arch/arm/kernel/process.c	Arrive to kernel side
do_execve()	fs/exec.c	open file and do some preparation
search_binary_handler()	fs/exec.c	find out type of executable
load_elf_binary()	fs/binfmt_elf.c	load executable and create user segment
start_thread()	include/asm-arm/processor.h	pass control to program code

Table 2. Startup process of an ELF binary

The dual-core application is executed by our tool. First of all, the tool forks a child process, and the child process is added to run queue [20]. When the child process is scheduled by scheduler and starts to run, it calls execve() function to execute the dual-core application. As shown in table 2, a series of functions are triggered in the kernel to initialize the execution environment of the dual-core application. At last, start_thread() function is called to start executing the dual-core application. So we starts the timer1 just after calling the start_thread() function. After the dual-core application finishes and returns, it calls do_exit() function to enter zombie state. We inject timer1_stop() function in do_exit() function in exit.c to stop timer1. After reading the number of the counter, total time period during the execution of the dual-core program is obtained.

Dual-core application time (T_dual_core_ap), IPC system call time (T_ipc_system_call_time_from_entry_to_return) and Blocking IPC time (T_block_ipc)

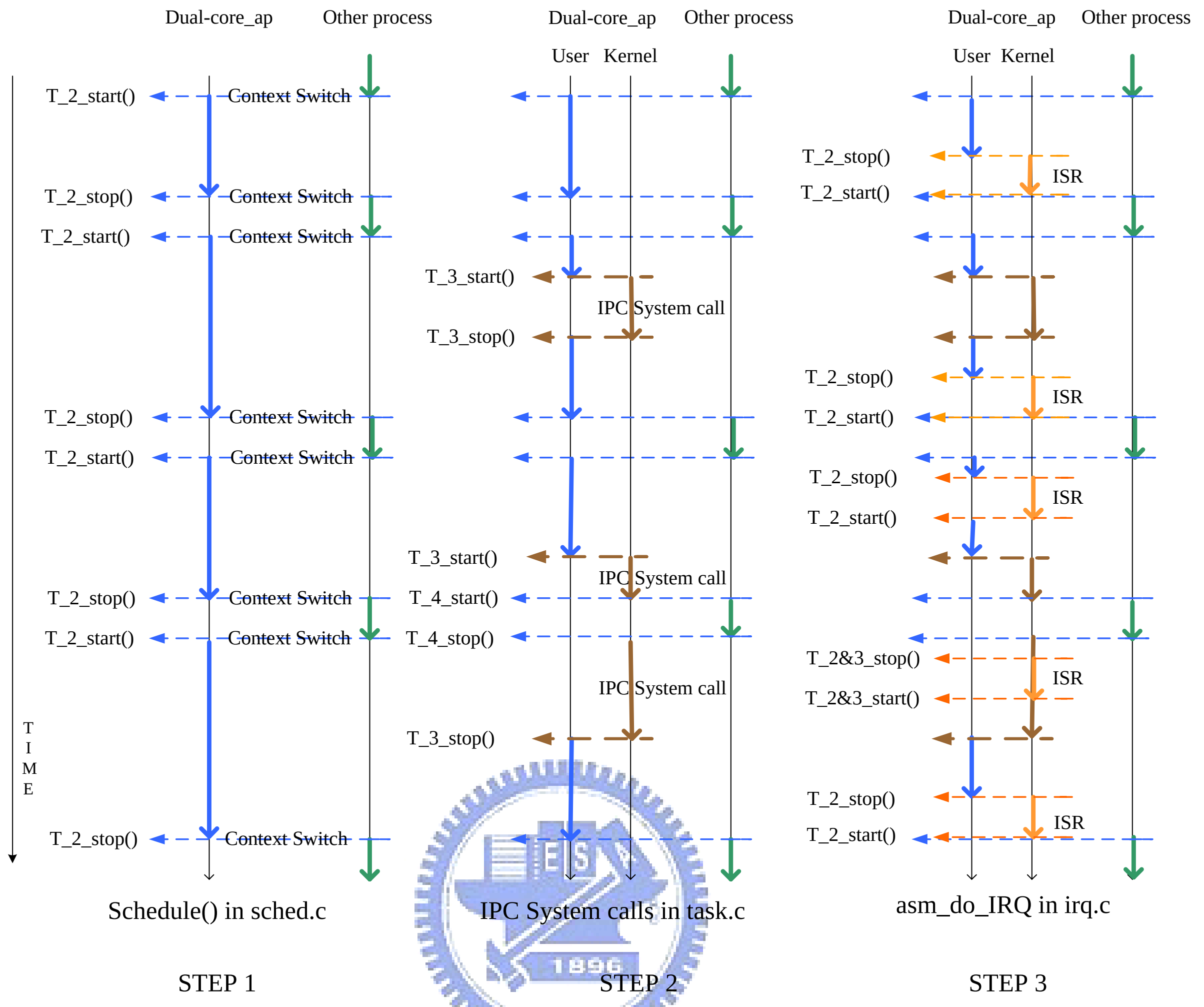


Figure 10. The steps to inject timers in kernel and DSP drivers for time profiling

Figure 10 shows how we get these three time information in following three steps:

Step1: The execution time of dual-core application is recorded by Timer2. Each time when dual-core application is scheduled to run, Timer2 starts counting. And in the same way, Timer2 stops when dual-core application is preempted by other processes. The T_2_start() and T_2_stop() functions are injected in schedule() function in kernel source file sched.c.

Step2: Timer3 is responsible to accumulate the time spent on IPC system calls such as read(), write(), open() and close(). All we need is to start/stop Timer3 at the entry/exit of every function that implements the IPC system calls in DSP drivers. However, it is possible to block and call schedule() function when waiting the response from DSP. After receiving the response signal from DSP, dual-core application will be waken up and scheduled to finish the remaining job in IPC system call. So we need to start/stop Timer4 before/after calling schedule() function.

Step3: Various interrupts could occur and run corresponding ISRs in the period of dual-core application and IPC system calls. However the time/energy should be classified into non-dual-core application part. In Linux kernel, all interrupts are handled by asm_do_IRQ() function. So codes are injected to stop/start timer2 and timer3 at the entry/exit of the asm_do_IRQ() function.

Idle time in dual-core application and idle time in blocking IPC

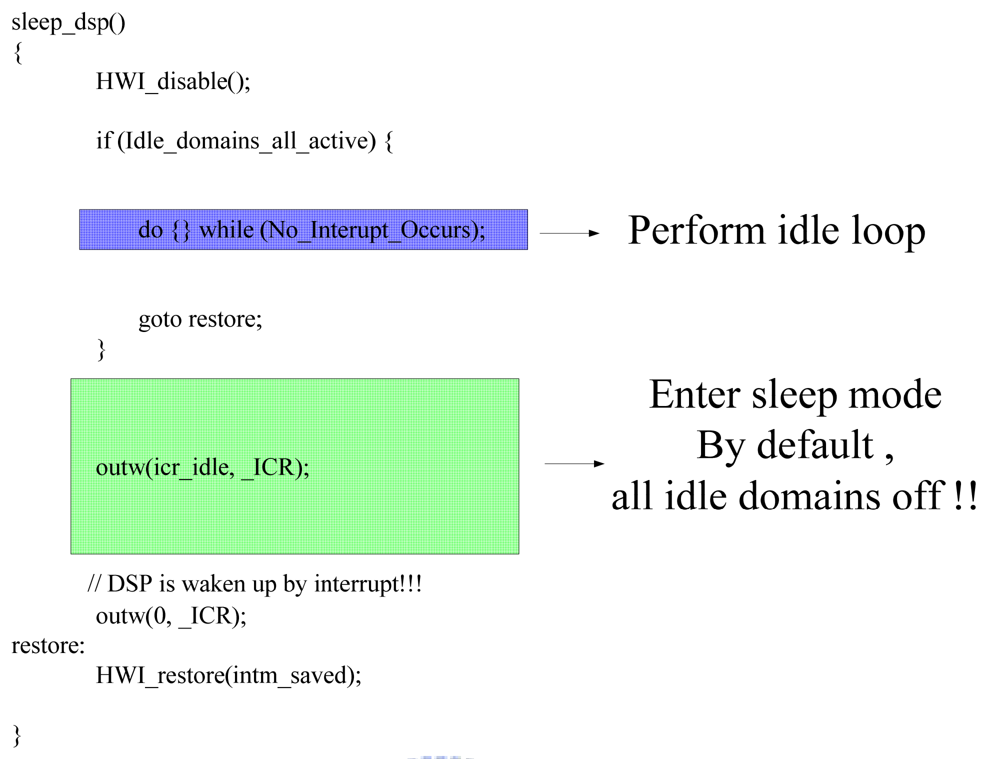
Like the way to record the execution time of dual-core application, we need to inject code to schedule() function to start/stop these two timer when idle task is scheduled in/out. Of course, asm_do_IRQ() function is also modified to stop/start the timers when interrupts occurs during execution period of idle task.

D. Energy estimation and analysis for dual-core application on DSP processor

Because DSP/BIOS is not an open source kernel, we could not fully control the runtime behavior in DSP through instrumentation. Some assumptions and simplifications are necessary. DSPGateway supports static task and on-demand task. We assume that only single user task which is linked with the tokliBIOS kernel statically runs on DSP. Along with the defaulted supertask and idle task, there are three tasks on DSP. The way these tasks are partitioned is according to the implementation and design of DSPGateway. However, we analyze the jobs they performed and regroup them according to the information we are interested in. Following we are going to introduce the jobs performed by these three tasks.

Sleeptask

Sleeptask is the task which is executed when there is no other task to run. The behavior of sleeptask is close related to power management mechanism of DSP. Therefore, we introduce the power management of DSP first. The DSP CPU and peripherals contain several clock domains that can be turned off individually to conserve power. The active/idle status of the various domains is controlled by the idle control register. When the DSP software executes the IDLE instruction, the clock domains are configured according to the settings of the idle control register. DSPGateway also implements mailbox command about power management for ARM to control these domains through DSP. By default, DSPGateway driver turns off all idle domains to save power when sleeptask is executed. However, when ARM needs to access to DSP internal memories, DSP is not allowed to turn all domains off. Before ARM accesses to DSP internal memories, it sends a mailbox command with PMCMD=ENABLE to DSP. DSP then sets all domains enable and perform idle loop when sleeptask is executed next time. Another mailbox command with PMCMD=ENABLE is sent to DSP if the access to DSP internal memories is no need anymore. Figure 11 shows the pseudo code of the sleep_dsp() function which is executed when sleeptask is scheduled. When entering sleep_dsp() function, all hardware interrupts are disabled. If all idle domains are enabled, it performs idle loop until interrupt occurs. Otherwise, it turns off all idle domains and enter sleep mode. When interrupts occur, DSP is wakening up and turns on all idle domains. At last, it enables and handles the interrupts.

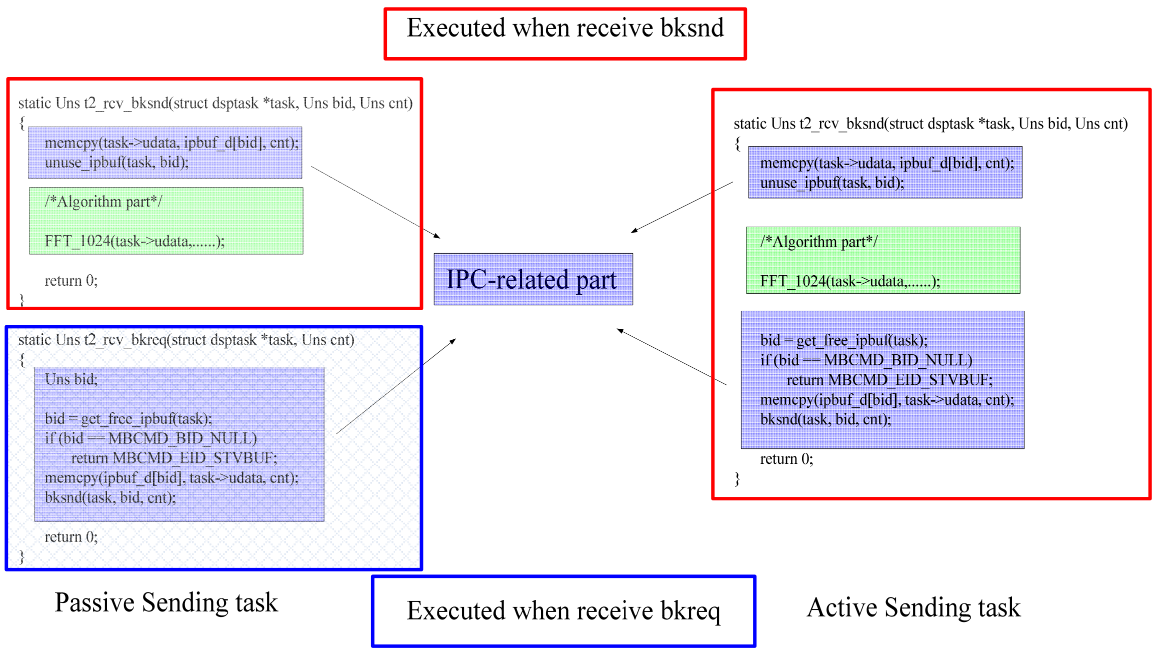


```
sleep_dsp()
{
        HWI_disable();

        if (Idle_domains_all_active) {

            do {} while (No_Interupt_Occurs);

            goto restore;
        }

        outw(icr_idle, _ICR);

       // DSP is waken up by interrupt!!!
        outw(0, _ICR);
restore:
        HWI_restore(intm_saved);

}
```

Figure 11. Pseudo code of sleep_dsp() function

Dsptask

Dsptask is the task that user should implement. DSP typically executes computation-intensive algorithms or jobs which partitioned form Linux-side application to enable real-time services. Figure 12 shows that dsptasks implemented with different communication types can be roughly divided into algorithm part and IPC-related part. The behavior in IPC-related part typically prepares the data for algorithm by calling memcpy() function provided by standard C library and IPC functions implemented by DSPGateway. After executing the algorithm, the code inside IPC-related part sends the data back actively or passively by calling memcpy() function provided by C library and IPC functions.



```
static Uns t2_rcv_bksnd(struct dsptask *task, Uns bid, Uns cnt)
{
    memcpy(task->udata, ipbuf_d[bid], cnt);
    unuse_ipbuf(task, bid);

    /*Algorithm part*/

    FFT_1024(task->udata,......);

    return 0;
}
```

```
static Uns t2_rcv_bkreq(struct dsptask *task, Uns cnt)
{
    Uns bid;

    bid = get_free_ipbuf(task);
    if (bid == MBCMD_BID_NULL)
        return MBCMD_EID_STVBUF;
    memcpy(ipbuf_d[bid], task->udata, cnt);
    bksnd(task, bid, cnt);

    return 0;
}
```

```
static Uns t2_rcv_bksnd(struct dsptask *task, Uns bid, Uns cnt)
{
    memcpy(task->udata, ipbuf_d[bid], cnt);
    unuse_ipbuf(task, bid);

    /*Algorithm part*/

    FFT_1024(task->udata,......);

    bid = get_free_ipbuf(task);
    if (bid == MBCMD_BID_NULL)
        return MBCMD_EID_STVBUF;
    memcpy(ipbuf_d[bid], task->udata, cnt);
    bksnd(task, bid, cnt);

    return 0;
}
```

Figure 12. Overview of the behavior of Dsptask

Supertask

Supertask is the task which performs system services. Functions implemented inside are for different purposes such as system initialization and debug. Only two of them are commonly executed in the runtime. One of them is power management function which set idle control register when receiving MBCMD_PM command. The other one is release_ipbuf() function which yield the ownership of a Global IPBUF line to the other processor in order to keep the numbers of Global IPBUF balance between ARM and DSP.

Reclassification according to software behavior of DSP

Form above observation, we find that DSP almost works for algorithm and IPC-related job when awake. So we reclassified them into sleep, IPC-related, and algorithm parts and profile the energy/time for them.

D.1. Power characterization for DSP processor

In order to understand the power consumption behavior of DSP and pre-analyze the power consumption of DSP algorithm, we take instruction level power analysis approach to characterize the power consumption of DSP. We group the instructions into common classes on the basis of the previous work [22] which studied on power consumption characterization of the same TMS320C55x DSP. The experiment result shows that the current consumption varies high among different instructions. This is due to different CPU utilization by different instructions. Some DSP instructions designed for specific algorithms usually trigger several arithmetic units and perform several memory accesses within a cycle. These computation-based instructions highly utilize CPU and consume high current, while control-related instructions generally consume far less current. The algorithms run on DSP are generally highly optimized. The energy behavior of these algorithms is dominated by several instructions executing in a loop. Hence, the current consumption between different algorithms varies high. On the other hand, the instructions designed for specific algorithms are not shown in control-based functions. So we expect that the variation between control-based functions should be low.

Power modeling for IPC-related functions

We take software function level power modeling approaches to establish energy library for these IPC-related functions. The results show that the current consumption among them varies low. It is not necessary to distinguish from them. So we applied a flat power model between them.

IPC Functions	Average Current(mA)
get_free_ipbuf()	50.04
unuse_ipbuf()	50.37
Wdsnd()	49.63
Bksnd()	49.03
Bksndp()	50.14

Wdreq()	48.73
Bkreq()	49.47
Bkreqp()	49.91

Table 3. The current and power consumption of IPC-related functions

Power modeling for Sleeptask

The power consumption of the sleeptask falls into two states. One is the idle state when executing idle loop. The other is sleep state when all idle domains are turned off. We found that the current consumption of DSP is almost zero when all idle domains are off.

	Current (mA)	*Power (mW)*
Idle	*39*	*62.4*
Sleep	*0~1*	*0~1.6*

Table 4. The current and power consumption of idle and sleep state of DSP

D.2. Energy estimation and analysis for dual-core application on DSP

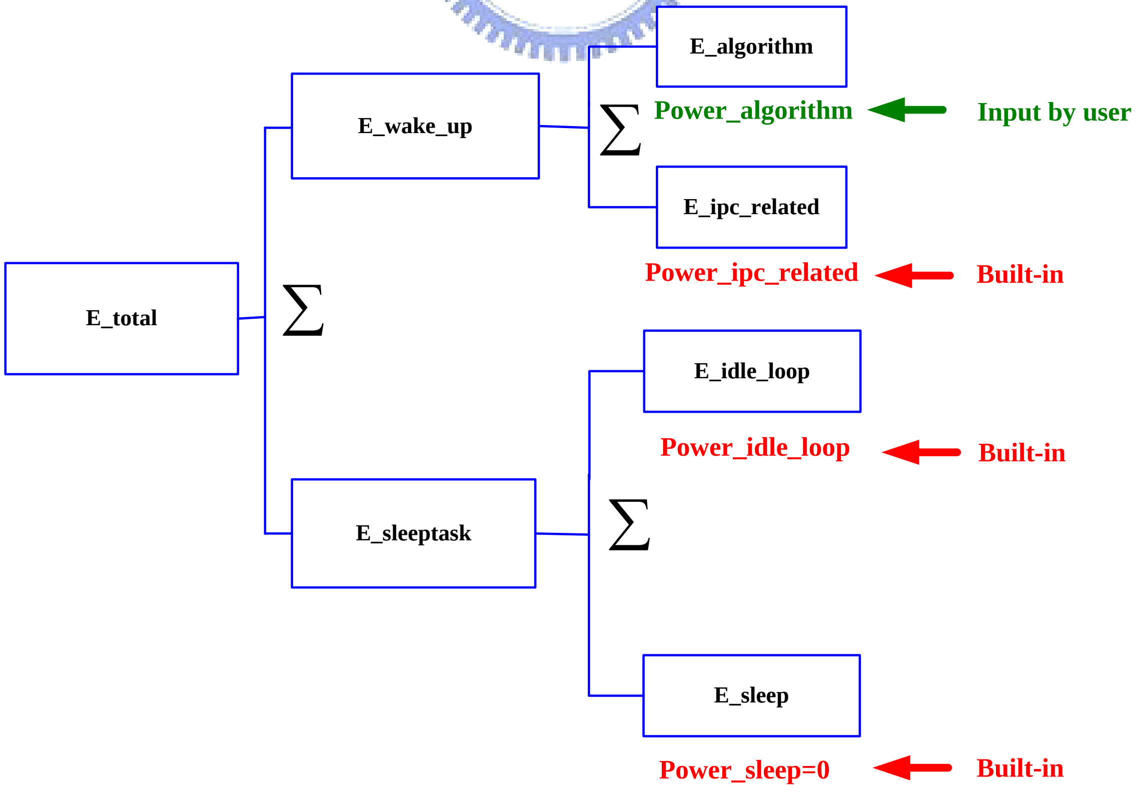


Figure 13. The partition tree for energy profiling on DSP

The figure 13 shows the energy information we need to know. With the three build-in power models and the one inputted by user, energy profiling can be obtained through time profiling.

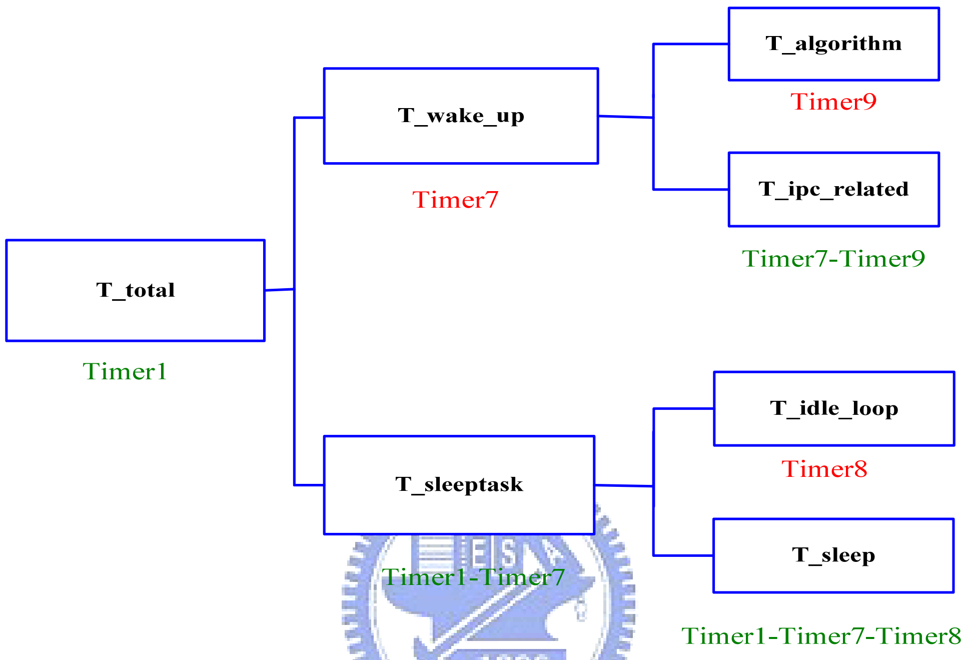
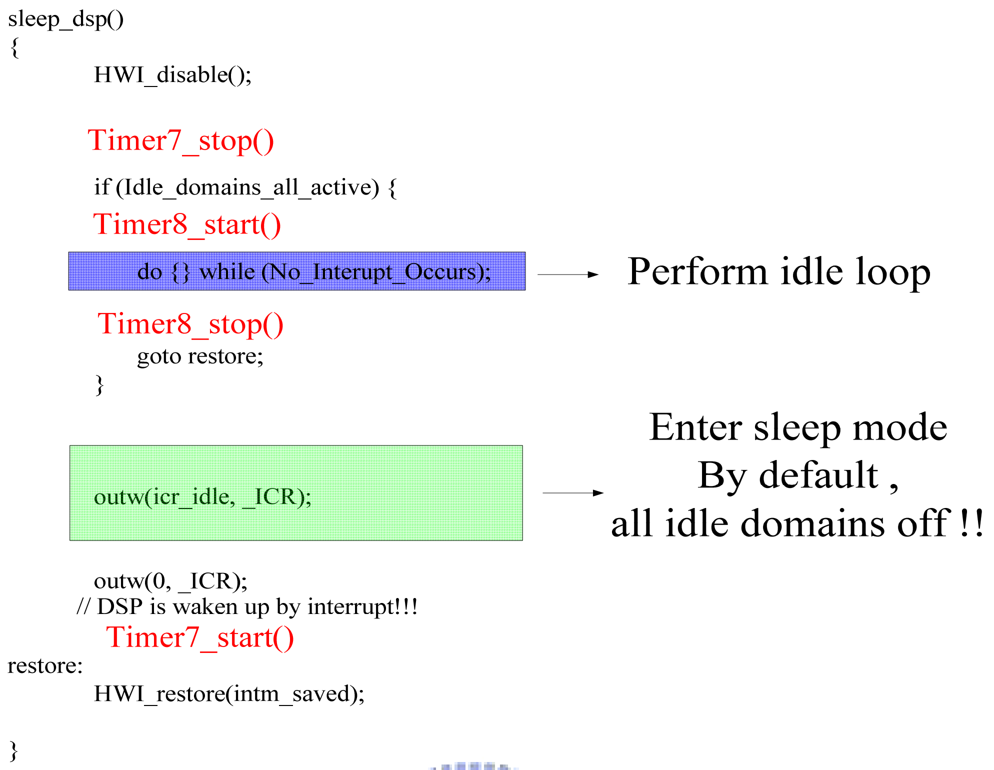


Figure 14. The configuration of timers for time profiling on DSP

As shown in Fig 14, we configure three timers to get all necessary information. Following we explain where to inject the timers.

DSP wake up time(T_wake_up) and DSP idle loop time(T_idle_loop)

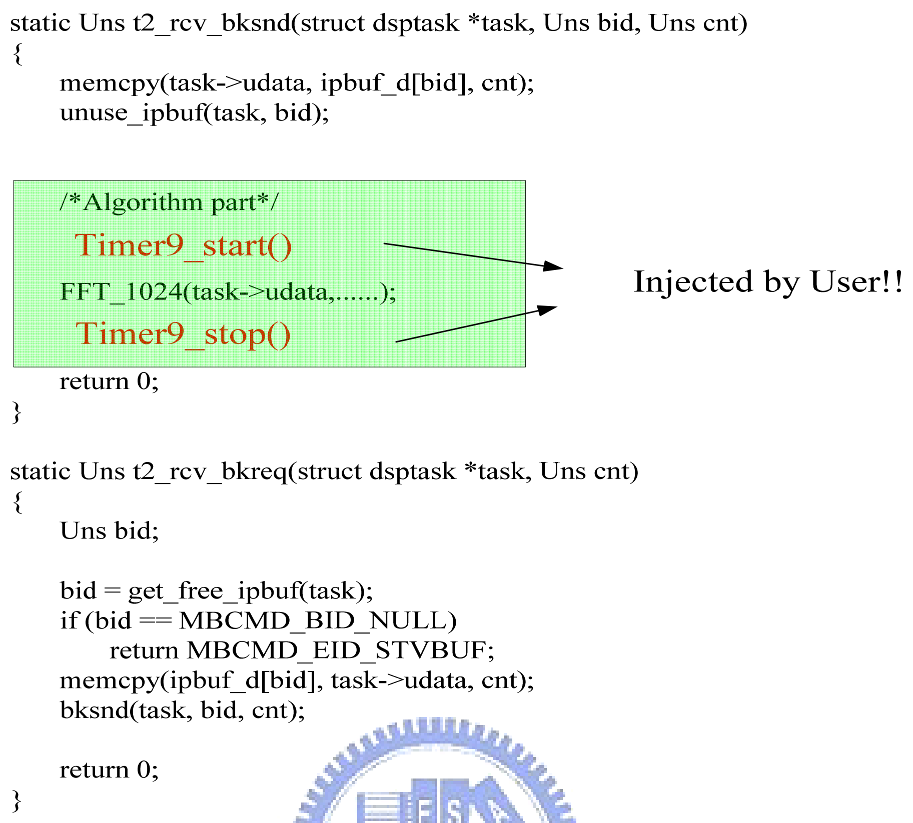


```
sleep_dsp()
{
        HWI_disable();

        Timer7_stop()

        if (Idle_domains_all_active) {
        Timer8_start()
          do {} while (No_Interupt_Occurs);
          Timer8_stop()
            goto restore;
        }

        outw(icr_idle, _ICR);

        outw(0, _ICR);
      // DSP is waken up by interrupt!!!
          Timer7_start()
restore:
        HWI_restore(intm_saved);

}
```

Figure 15. The configuration of timers for time profiling on DSP

As shown in Figure15, we start Timer7 each time it wakes up and exits the sleep_dsp() function. And stop it when it enter sleep_dsp() function. Also we start timer8 whenever it starts to execute idle loop, and stop it when it exits.

DSP algorithm total time(Time_algorithm_total)



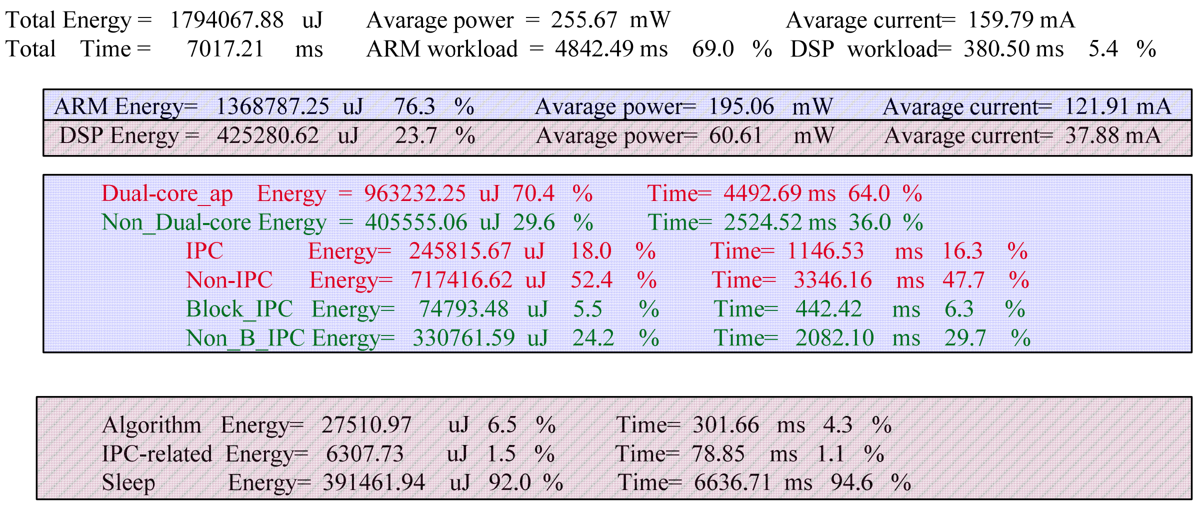
```
static Uns t2_rcv_bksnd(struct dsptask *task, Uns bid, Uns cnt)
{
    memcpy(task->udata, ipbuf_d[bid], cnt);
    unuse_ipbuf(task, bid);

    /*Algorithm part*/
     Timer9_start()
    FFT_1024(task->udata,......);
     Timer9_stop()
    return 0;
}

static Uns t2_rcv_bkreq(struct dsptask *task, Uns cnt)
{
    Uns bid;

    bid = get_free_ipbuf(task);
    if (bid == MBCMD_BID_NULL)
        return MBCMD_EID_STVBUF;
    memcpy(ipbuf_d[bid], task->udata, cnt);
    bksnd(task, bid, cnt);

    return 0;
}
```

Figure 16. The configuration of timers for time profiling on DSP

As shown in Figure 16, the users insert the timer9_start()/timer9_stop() function to start/stop profiling the algorithm by them.

IV. Experiments Results

A. An Example of energy/time profile



```
Total Energy =  1794067.88   uJ     Avarage power  = 255.67  mW              Avarage current=  159.79 mA
Total    Time =      7017.21     ms     ARM workload  = 4842.49 ms    69.0   %   DSP  workload=  380.50 ms     5.4   %

 ARM Energy=   1368787.25  uJ     76.3   %          Avarage power=  195.06   mW       Avarage current=  121.91 mA
 DSP Energy =   425280.62   uJ      23.7   %          Avarage power=  60.61      mW       Avarage current=  37.88 mA

        Dual-core_ap    Energy   =  963232.25  uJ  70.4   %         Time=  4492.69 ms  64.0  %
        Non_Dual-core Energy   =  405555.06  uJ  29.6   %         Time=  2524.52 ms  36.0  %
                   IPC              Energy=   245815.67  uJ    18.0    %         Time=  1146.53     ms   16.3     %
                   Non-IPC       Energy=   717416.62  uJ    52.4    %         Time=   3346.16    ms   47.7     %
                   Block_IPC    Energy=    74793.48   uJ    5.5      %         Time=   442.42     ms    6.3      %
                   Non_B_IPC Energy=   330761.59  uJ    24.2    %         Time=   2082.10    ms    29.7     %

        Algorithm   Energy=   27510.97      uJ   6.5   %         Time=  301.66   ms   4.3   %
        IPC-related  Energy=   6307.73      uJ   1.5   %         Time=  78.85    ms   1.1   %
        Sleep            Energy=  391461.94    uJ   92.0  %        Time=  6636.71  ms   94.6   %
```

Figure 17. Example energy/time profile

Figure 17 shows an example of the energy/time profile. The results are presented in text-mode.

B. *Error analysis*

Following we are going to analyze the error between physical current measurement and the current estimated by this tool. First of all, we choose various often used algorithms and assign them to ARM and DSP. The average current of the DSP algorithm are obtained form measurement in advance and feed it into the tool. To make the current measurable, the test programs are also executed in a loop.   In table 5, it shows that the max error is less than 5%.

	Description		Average current (mA)		
	ARM	DSP	E	M	%E
Program1	Bubble Sort	Radix-2 complex forward FFT	159.80	154.3	3.44
Program2	FFT	Forward and Inverse DCT	165.57	164.48	0.6
Program3	Convolution	Sum of absolute	168.85	163.1	3.2

		differences on a single 16x16 block			
Program4	Matrix Transform	JPEG variable length coding following ITU-T (CCITT) T.81 standard	159.41	157.4	1.2
Program5	Find minimum value in an array	Hw_dct_idct	167.26	164.1	1.88
Program6	Autocorrelation	FIR direct form	168.55 mA	164.5	2.4
Program7	Square root	Matrix multiplication	160.44	156	2.77
Program8	Matrix multiplication	Double-precision IIR filter	168.31	165.3	1.79

Table 5. Error analysis of the tool

V. Conclusions

In this work, a tool for energy consumption estimation and analysis of the embedded software running on a dual-core processor is proposed and developed. This tool can provide energy/time information of dual-core program with low overhead and high accuracy. With the help of this tool, dual-core program designers could easily understand the energy/time distribution and then optimize the most critical part of software more efficiently.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations,” Proceedings of the 27th International Symposium on Computer Architecture (ISCA), June 2000.

[2] W. Ye, N. Vijaykrishan, M. Kandemir, and M. J. Irwin, “The Design and Use of Simple-Power: A Cycle-Accurate Energy Estimation Tool,” Proceedings of the Design

Automation Conference, June 2000.

[3] The SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/~tnm/power/

[4] V. Tiwari, S. Malik, and A. Wolfe, “Power analysis of embedded software: A first step toward software power minimization,” IEEE Trans.VLSI Syst., vol. 2, pp. 437–445, Dec. 1994.

[5] V. Tiwari, S. Malik, A. Wolfe, and M.T.C. Lee, “Instruction level power analysis and optimization of software,” J. VLSI Signal Processing, vol. 13, no. 2, pp. 1-18, 1996.

[6] M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita, “Power analysis and minimization techniques for embedded DSP software,” IEEE Transactions on VLSI Systems, pages 1-14, March 1997.

[7] A. Sinha and A.Chandrakasan, “JouleTrack – A Web Based Tool for Software Energy Profiling,” Proc. 38th Design Automation Conference, June 2001.

[8] A. Sinha, N. Ickes, and A. Chandrakasan, “Instruction level and operating system profiling for energy exposed software,” IEEE Trans. on VLSI, 11(6), December 2003.

[9] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, “Highlevel Software Energy Macro-modelling,” in Proc. ACM/IEEE Design Automation Conference, Las Vegas, Nevada, USA, June 2001.

[10] G. Qu, N. Kawabe, K. Usami, and M. Potkonjak, “Function-level power estimation methodology for microprocessors,” in Proc. Design Automation Conf., June 2000, pp. 810–813

[11] T. Tan, A. Raghunathan, and N. Jha, "Embedded Operating System Energy Analysis and Macro-Modeling," International Conference on Computer Design, pp. 515-222, 2002.

[12] A. Acquaviva, L. Benini, and A. Ricco', "Energy Characterization of Embedded Real-Time Operating Systems," in L. Benini, M. Kandemir, J. Ramanujam, Compilers and Operating Systems for Low Power, Kluwer Academic Publishers 2003.

[13] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. Jha, "Analysis of Power Dissipation in Embedded Systems using Real-Time Operating Systems," IEEE

Transactions on CAD, Vol. 22, no. 5, pp. 615-627, May 2003.

[14] J. Flinn and M. Satyanarayanan, “Powerscope: A Tool for Profiling the Energy Usage of Mobile Applications,” Proc. IEEE Workshop Mobile Computing Systems and Applications (WMCSA 1999), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 2-10.

[15] D. Shin et al., “Energy-Monitoring Tool for Low-Power Embedded Programs,” IEEE Design and Test of Computers, 19(4), 2002.

[16] OMAP5912 Applications Processor Data Manual. Texas Instruments. Dallas, Texas. [Online]. Avallable:http//www.ti.com.

[17] DSP Gateway for Linux, http://dspgateway.sourceforge.net/.

[18] Toshihiro Kobayashi, Kiyotaka Takahashi, Linux DSP Gateway Specification Rev3.3, Nokia Corporation, December 7 2005.

[19] TMS320 DSP/BIOS User's Guide Rev B(SPRU423B), Texas Instruments.

[20] Robert Love. Linux Kernel Development. Sams, 2004.

[21] The OMAP Linux Kernel Team Linux 2.6.16 omap1 patch file [Online] http://www.muru.com/linux/omap/.

[22] V. Paliouras, J. Vounckx, and D. Verkest, “Power Consumption Characterisation of the Texas Instruments TMS320VC5510 DSP,” (Eds.): PATMOS 2005, LNCS 3728, pp. 561–570, 2005.