Novel method to reduce Inter Core Communication in a Multi Core System

Jayanth H.
Robert Bosch Engineering and Business Solutions Ltd
Bangalore. India

Umadevi V., PhD
Department of Computer Science and Engineering,
BMS College of Engineering
Bangalore, India

Gurudath A. S.
Robert Bosch Engineering and Business Solutions Ltd
Bangalore. India

ABSTRACT
The architectural advancements in desktop computing have made embedded devices in real time applications to adopt multi core architectures. The main challenge in multi core programming is the process of communication between the different executing cores. Effectiveness of parallel programming in multi core architectures lies in method used for communication. Communication using shared cache is one of the popular approaches. This paper discusses in detail one of the novel methods of inter core communication. Correctness of the algorithm has been based on results obtained on a hard real time system.

General Terms
Memory Locking, Embedded Hypervisor

Keywords
Multi core, data bank, memory lock, functional split

1. INTRODUCTION
Embedded systems with multi core architectures have been dominating the electronic applications in this decade. Features that are implemented in these embedded systems are increasing exponentially [1]. Features which are inter dependent and executed on different cores face a road block in their execution if inter core communication is not managed in a robust manner. Failure of a proper method for inter core communication [2] has hindered the parallel programming capacity of the multi core system. Communication between different cores takes place through memory interactions. Several ideas on inter core communication has been discussed in [3]. A novel method of inter core communication has been described in this paper. This method of communication is applicable to those applications in which, the parameters involved, are not changing rapidly. Results were quantified on a real time automotive application.

1.1 Terminologies
- Task: It is a set of program instructions loaded to memory.
- Data Bank: A section of memory where data is stored.
- Producer Task: The task which produces or updates the information. It stores the information in transmission data bank.
- Consumer task: The task which uses the produced information for further processing. It utilizes the information stored in receiving data bank. Same task can produce and consume the information.

2. EXISTING COMMUNICATION PATTERN
2.1 Single Core System
Let two tasks A and B execute in a single core processor. Initially, task A is executed. Execution of task B depends on the results produced by task A. After task A finishes its execution, it writes the data to memory. Task B then fetches the data and starts its execution. Task A which executes initially is called as a producer task and task B which utilizes the data produced by task A is called as a consumer task. Problem of data synchronization and memory access does not arise in single core communication. Authors of [4] have provided comprehensive details of single core system. Figure 1 illustrates single core system.

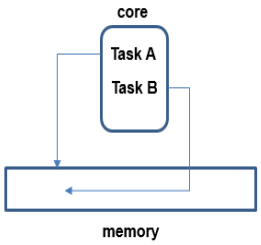

Fig 1: Single Core System

2.2 Multi Core System
Consider the same tasks A and B executing in a dual core system. Both the tasks are executing in different cores. Advantage of a multi core system lies in the possibility of tasks executing in parallel. With parallel execution, problem of data consistency arises. Several methods for data consistency have been defined in [5]. If the system under test is a hard real time system, consistency methods have to be performed in a very short time interval. An automobile system with multi core architecture has been considered for our analysis. In this system, synchronization is brought by activating task B only after task A completes its execution. After task A finishes, Task A_End is executed, context switching occurs and Task B_Start begins executing. After Task B_Start finishes execution, Task B starts its execution.

If task B is the only task in core 1 and if it is dependent on task A, core 1 would be unused for a very long time. Figure 2 illustrates multi core system.

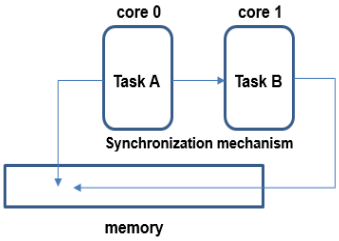

Fig 2: Multi Core System

In any embedded system, certain quantities may not vary in very fast intervals of time. Ex: temperature, altitude in an automobile system. The functions containing these quantities are executed in all cycles even though the values of the quantities are same. Processor would be busy and its executing capacity would be wasted as the results produced by these functions remain the same. Runtime improvements can be brought about by executing these quantities only when its value changes.

3. PROPOSED MODEL: INTELLIGENT COMMUNICATION

Transmission Data bank is the part of memory updated by the producer task and receiving data bank is the part of memory that provides the data to consumer task. Transmission data bank would be initially locked. Producer task updates the transmission data bank whenever it produces the data. Once the data updating is over, the transmission data bank would be released and the receiver data bank would be updated. Problems of data synchronization between the different cores would be solved in this approach. The core on which the consumer task is supposed to run can execute other tasks when the consumer task is busy waiting for data. Figure 3 illustrates the mechanism of intelligent communication.

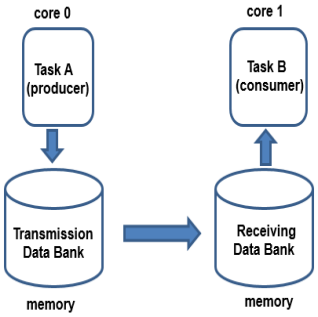

Fig 3: Intelligent Communication

3.1 State Diagrams

When the producer task arrives, it would be in ready state. The transmitter data bank is free and hence the producer task would move to running state. After its execution, the producer task would stay in suspended state until further updating. When the consumer task arrives and if the receiver data bank is updated, consumer task would move to running state. After its execution, it would stay in suspended state. If the receiver data bank is not updated, the consumer task would move to suspend state and waits there until further updating. Figure 4 and 5 illustrates the state transitions for consumer and producer tasks.

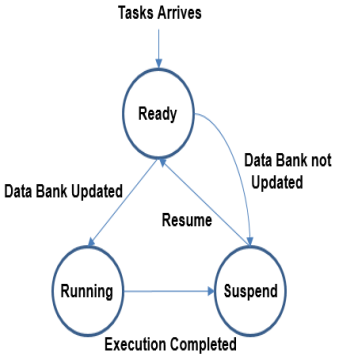

Fig 4: Consumer Task

Fig 5: Producer Task

3.2 Memory Locking

One of the major contributors for success of any real time system is its ability to handle the problem of data consistency [6]. Data inconsistency due to memory related operations could occur when a particular portion of memory which is written by a particular core is interrupted by other core for reading or writing to/from the same location of memory. An operating system should always have the ability to restrict access to a particular section of memory. Excessive paging and swapping of memory can also cause consistency related issues. To solve the problem of data consistency, memory locking is implemented. Specific regions of physical memory are identified. Virtual pages are created and each page has a locking variable. Locking and unlocking the pages depend upon the value of the locking variable [7]. Each page can be locked by a task and can be unlocked only by the locking task.

Each producer task has an exclusive privilege to lock a specific portion of the virtual address space into the physical memory. Pages locked in such manner are exempted from further paging until the producer task finishes its execution and unlocks the portion of memory. The consumer task checks for the value of the locking variable. If the value of the locking variable is not reset, the consumer tasks keeps polling at regular intervals of time. If the value is reset, then the data required for its execution is ready. It copies the values of transmission data bank to the receiver data bank.

Databanks are also allocated priorities. If the same task is the producer task for more than one information, data bank corresponding to the most important information would have the highest priority. Then the producer task would have to write data to the data bank of the highest priority. To unlock a particular page, the producer task resets the unlocking variable. Receiver data bank is generally unlocked. If the receiver data bank is the transmission data bank for some other task, the concept of locking and unlocking would be applied to it. Both the data banks would be locked during data

transfer. The common locking variable would be stored in a location know to both the tasks.

3.3 Data transfer between memory banks

The address of the transmission data bank and receiving data bank are known in prior. Copying data from the transmission data bank to the receiver data bank is carried out with the help of linked lists. Authors of [8] have described the concept of linked list and the various methods of memory transfer. The transmission data bank pushes the data into the list and the receiver data bank pops the data from the list. Multiple pop operations are carried out simultaneously across the cores.

3.4 Functional Split

Functionalities across the system are identified. These functionalities contain several tasks executing inside them. Dependencies which could exist between the functions are removed and they are scheduled across the cores. This process of identifying and splitting it across the cores is called as functional split.

The concept of functional split has been described by the authors of [9]. Advantages have also been described there. Figure 6 illustrates the concept of functional split. An automotive project is considered for analysis. Some of the common functions present in any automotive project are torque system, temperature system, fuel system etc. Currently all of them are combined and scheduled based on their time locality. In the method described, every function is isolated and executed separately. A data bank is created for each function. This section is called a data bank. If this function is a producer, it updates the transmission bank and if it is a consumer, it copies the data to the receiver data bank. Data transfer between a transmission data bank on a particular core and a receiving data bank on other core is handled by the arbitration of the communication busses that are present between the cores.

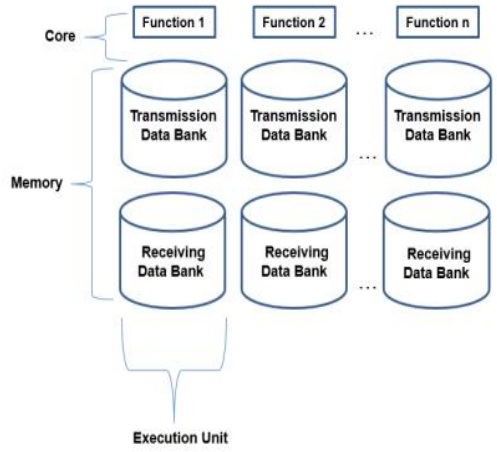

Fig 6: Functional Split

3.5 Embedded Hypervisor

Embedded hypervisors allow different components to share a common hardware host [10]. They grant the control of memory to different cores in our intelligent approach [11]. Consider the scenario where function 1 is a producer task and is executing on core 1. It updates the transmission data bank. Function 2 and 3 are consumer tasks on core 2 and 3 respectively. Both require data from the same producer task. Priority based round robin scheduling is introduced. Since functional split is carried out and their priority is already known, embedded hypervisor allows only the core with highest priority to arbitrate for the buss. General arbitration technique is considered when priority of all cores is same. Each newly introduced core corresponds to a new functionality added. The embedded hypervisor previously used would still be the same but with more arbitration for memory.

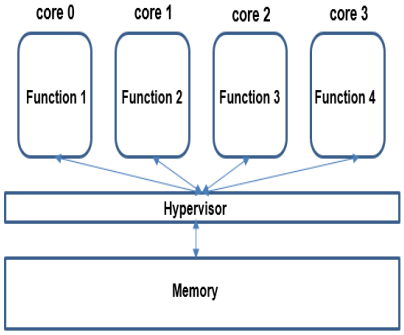

Fig 7: Embedded hypervisor

Fig 8: Data Format for Transmission

Concept of embedded hypervisor and information propagated across the different cores by the embedded hypervisor is depicted in figure 7 and 8. The field 'core', gives the details of the core in which the current function is being executed. 'Priority' field gives the details of the function and its associated priority. The address where the transmission data bank is stored is provided by 'address' field. The approximate time by which the transmission data bank could change is given by 'time to live'. Data corruption during transmission is recognized by cyclic redundancy check ('crc') field.

4. ALGORITHM

The below mentioned algorithm has been used for implementation.

INPUT

'm' cores

'n' tasks

PROCESS

Step 1: Initialize the embedded hypervisor. Each core acts as a virtual processor.

Step 2: Propagate function information across all cores. Each function is considered a task.

Step 3: Identify all the slow varying quantities in all task.

Step 4: Based on data utilization and updating, categorize the tasks as producer and consumer.

Step 5: Create transmission and receiver data banks and assign priorities. Begin the execution.

Step 6: Producer task locks the transmission data bank.

Step 7: Producer task performs the required operations and stores data in the transmission data bank.

Step 8: Data in transmission data bank is copied to receiver

data bank using linked lists.

Step 9: Transmission data bank is cleared.

OUTPUT

Consumer task utilizes the copied data for its execution. If the Receiver data bank is not the transmission data bank of some other task, clear the receiver data bank.

5. IMPLEMENTATION AND RESULTS

Use case from an automotive application was considered to demonstrate the correctness of the communication method described in this paper. The algorithm described in section 4 was implemented and the results obtained are discussed in this section. The details of the automotive application are not discussed because of legal bindings. In the automotive application under test, tasks are classified based on the time at which they are called by the processor for execution. The runtime of each task is shown in table 1.

Table 1. Runtime of Individual Tasks

Task	Runtime
20ms	6236.44µs
50ms	1122.96µs
100ms	6314.12µs
200ms	68.76µs
1000ms	269.72µs

These tasks have several parameters whose value change, after execution of a particular condition. This condition occurs only after a specific period of time. These parameters are classified as slow varying quantities. Such quantities were identified and runtime improvements due to their non-execution and intelligent communication, at various intervals were recorded.

The total runtime improvement for Interval 1 is 2.584%, for interval 2 it is 5.172% and for interval 3 it is 10.338%. A significant improvement in runtime is observed by application of the method described in this paper. Figure 9 and 10 illustrates the runtime improvements for individual tasks and the entire system.

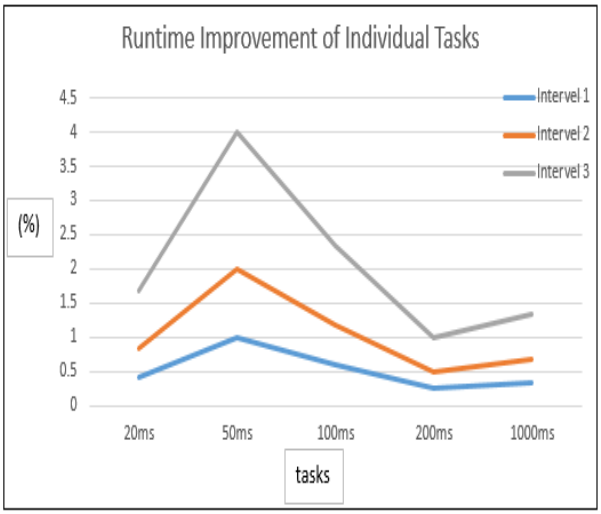

Fig 9: Runtime Improvement of individual tasks

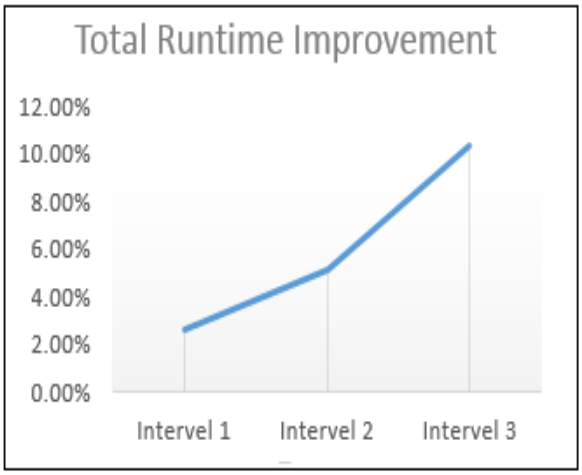

Fig 10: Total Runtime Improvement

Following inferences were drawn from the obtained results:

Let‘t’ be the total cycle time of the system under consideration. Let ‘R’ be the set of all possible runtime values. The runtime of all the tasks is given by ‘f’, which is a function of total cycle time. Let dt be the change in runtime due to errors in the system. The total runtime is given by

$$\text{Runtime total} = \int f(t)\,dt,\ dt! = 0 \text{ and } f(t) \in R\ . \quad (1)$$

Let ‘a’ and ‘b’ be the time interval where runtime has to be found out. [a, b] are mapped into [x(i-1),x(i)] for runtime measurements.

$$a = x0 <= t1 <= x1 <= \ldots\ldots <= xn = b. \quad (2)$$

Therefore the final runtime between the interval [a, b] is given by

$$r = \sum f(T(i))\Delta I\ . \quad (3)$$

where T(i) is a recognizable point between [x(i-1),x(i)] and $\Delta I = x(i) - x(i-1)$

This system comprised of several tasks. Let task ‘a’ be one of them. Let, the total runtime of the system be = ‘p’. The runtime of the task ‘a’ before optimization be = ‘m’. The runtime of the task ‘a’ after optimization be = ‘n’. The total runtime of the system after optimization of task ‘a’ be ‘q’. The difference in runtime of task ‘a’ before and after optimization = ‘o’ = ‘m’ – ‘n’. Improved Runtime ‘R’ is given by

$$\text{‘R’} = (\text{‘p’} - \text{‘q’})* \text{‘o}\ . \quad (4)$$

6. FUTURE

The data stored in receiver banks could be transferred to the private cache of the core using direct memory access (DMA) [12].DMA allows memory transfers to take place without the usage of central processing unit (CPU) or the core. The address of the memory banks and the cache are known in prior. The processor initializes the direct message access controller. Cycle stealing mode of direct memory access would be chosen.

DMA controller carries on the activities of memory transfer and the CPU can be used for other purposes. This would minimize the arbitration for the main system bus. Shadow tables would be used to maintain consistency between the receiver data bank and the private cache. Since slow varying data is being considered, private cache would be cleared after fixed interval of time.

7. ACKNOWLEDGEMENTS

We would like to thank Dr. S.R.Krishnamurthy, former principal and HOD of Computer Science and Engineering Department, BMS College of Engineering for his support and guidance. We also express our indebted gratitude towards Robert Bosch Engineering and Business Solutions Ltd for providing all the necessary support.

8. REFERENCES

[1]. Future of Embedded Systems, http://www.techonline.com/ electrical-engineers/education-training/webinars/4429820/The-Future-of-Embedded-Systems,Wind River Systems, May 1, 2014

[2]. Shin'ichi Miura, Toshihiro Hanawa, TaisukeBoku, Mitsuhisa Sato: XMCAPI: Inter-Core Communication Interface on Multi-chip Embedded Systems, Ninth IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, pp. 397-402, 2011.

[3]. HengQuan, Ruijing Xiao, Kaidi You, Bei Huang, Xiaoyang Zeng, Zhiyi Yu.: A Simple High-Efficient Inter-Core Communication Mechanism for Multi-Core Systems, State Key Laboratory of ASIC and System, Fudan University, Shanghai.

[4]. VilmaTomço, AnetaDeliu, IgliTafa: A Trade-off between Complexity and Performance over Multi-core Systems,International Conference on Advances in Computing,Communications and Informatics (ICACCI), pp. 1509-1514, 2014.

[5]. Haibo Zeng, Marco Di Natale: Mechanisms for Guaranteeing Data Consistency and Flow Preservation in AUTOSAR Software on Multi-core Platforms, pp. 140-149, 2011.

[6]. Embedded Control Systems Design and Learning, https://en.wikibooks.org/wiki/Embedded_Control_Systems_Design/Learning_from_failure.

[7]. Hongwei Zhou, Rangyu Deng, Zefu Dai, Xiaobo Yan, Ying Zhang and Caixia Sun: The virtual open page buffer for multi-core andmulti-thread processors, 2014 IEEE International Conference on High Performance Computing and Communications (HPCC), 2014 IEEE 6th InternationalSymposium on Cyberspace Safety and Security (CSS) and 2014 IEEE 11th International Conference on Embedded Softwareand Systems (ICESS), pp. 290-297,2014.

[8]. Longfei Tan , Zhao Han,Chunguang Chen,Yinghua He ; Kunlong Zhang : A Non-blocking Self-Organizing Linked List Algorithm, Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference, pp. 71-76, 2012

[9]. Jayanth.H, Umadevi.V, Gurudath A.S : Intelligent task allocation in multi core environment, International Journal of Computer applications, pp. 34-39, 2015.

[10]. Crespo A. , Ripoll, I , MasmanoM: Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach, Dependable Computing Conference (EDCC), 67 – 72, 2010.

[11]. Embedded Hypervisors, https://en.wikipedia.org/ wiki/ Embedded_hypervisor.

[12]. DMAhttps://www.techopedia.com/definition/2770/dynamic-random-access-memory-dram.