International Journal of Computer Applications (0975 – 8887)
Volume 115 – No. 13, April 2015

Intelligent Task Allocation in Multi Core Environment

Jayanth H
Department of Computer Science and Engineering, BMS College of Engineering, Bangalore, India

Umadevi V
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. Constrained power availability but ever increasing performance requirements are the main reason for this migration. Failure to allocate tasks to specific cores would result in some tasks running while other tasks in other cores remaining idle. The efficiency of the entire system would decrease and the tasks with higher priority could cause bottlenecks. In this work, we propose a model which could analyze, split and allocate the tasks to cores. The results of the proposed model for a real time automobile application were observed to be effective on multi core architecture.

General Terms
Task Scheduling, Resource Allocation.

Keywords
Cores, Symbolic Model Verifier, Scheduler.

1. INTRODUCTION
The architectural advancements in desktop computing have made embedded devices in real time applications to adopt multi core architectures. The main reason is the ever increasing performance requirements but constrained power dissipation. Automotive embedded multi core real time systems have specific characteristics and differ strongly to other embedded multi core systems. Car manufactures have introduced multi core Electronic Control Units (ECU). These ECU's offer greater levels of performance improvements in automobile applications. Multi core architectures in the ECU reduce the complexity of previous system which had several ECU's and a lot of interconnection between them. Multi core ECU's should provide predictability and must ensure that all responses will be met [1].

At present, task allocation is done manually which may result in core overloading and performance bottlenecks. Some of the tasks might not get an opportunity to execute. Automatic allocation of tasks based on functionality will increase the utilization and performance of the multi core system. Hence in this work we propose a model to analyze, split and allocate tasks to cores automatically.

2. BACKGROUND
Automobiles have become the machines of the world. They are becoming software oriented and more user friendly software features are being introduced in automobiles. To incorporate all these features, multi core architecture is being introduced. Analysis of multi core architectures in automobiles has been described by the authors of [2].

These automotive embedded multi core real time systems have specific characteristics and differ strongly from other embedded multi core systems. Their main requirement is to complete the execution of a task before its deadline. A lot of research in static scheduling and parallelization of applications are being carried out for a long time [3]. Not all of those are intended for real-time systems such as those used in the automotive sector. As the possibility of multi core architectures being used in all future automobile systems are very high, evaluation of the current loading pattern and possible improvements has to be checked. Some of the load balancing algorithms is described in [4], [5]. But, most studies investigate on small problems and do not use realistic data. In contrast, an entire real time project has been considered in this paper and analysis has been carried out.

3. MODEL PROPOSED FOR AUTOMATIC TASK ALLOCATION
In this section we describe our proposed model which automatically allocates the tasks to the cores based on functionality. Following are the common terms used in this research paper

Core: Processor is a component which reads instructions and executes them. They could be reading or writing data. The processor consists of several execution units, cache, busses etc in a single chip. This execution unit is generically defined as core. Multi core processors have several execution units. The number of cores generally depends upon the application concerned.

Task: Smallest part of any program that can be managed by the scheduler of the operating system is called task. Scheduler is responsible for allocating these tasks to different cores.

Functionality of a task: Each task has its own set of properties. These properties could be dependent or independent of each other. These properties are collectively called functionality of a task.

We have considered tri core architecture for our analysis.

Example: Task1 has three functionalities F1, F2 & F3. Similarly the functionalities of task 2 and task 3 are shown in figure 1.

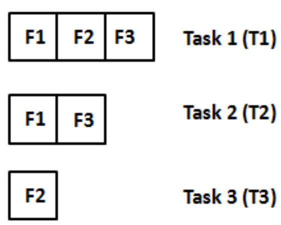

Fig 1: Relation between tasks and its functions

In the manual method, tasks are allocated across several cores without considering the functionalities. The task to core assignment in manual method is shown in figure 2.

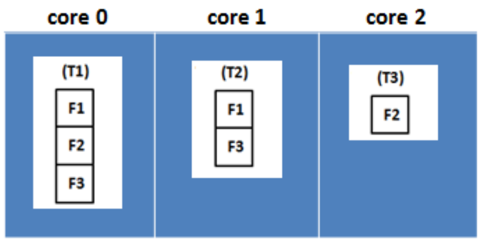

Fig 2: Task to core assignment in manual method

In our proposed model, the functionalities of the tasks are analyzed and then task assignment to core is done based on the functionalities. Figure 3 illustrates the task assignment to core based on the functionality.

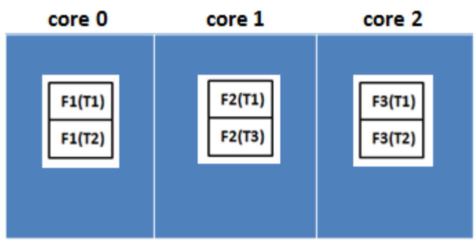

Fig 3: Task assignment in automated method

Following in this section we discuss in detail the sub modules of our proposed model which allocates tasks to cores based on the functionalities. In the sub modules labeled as (a) and (b) in figure 4, all information related to the hardware and the software descriptions are statically analyzed and the extracted information is provided as an input to the model. The software component description contains the information of the operating system used in the vehicle ECU. The resource requirements of the ECU and constraints of the system are identified. Parts of the software description which affect the scheduling scheme are identified and the information is extracted. Sub modules labeled as (c) in figure 4 shows our proposed model, which is discussed in detail in the further sub sections.

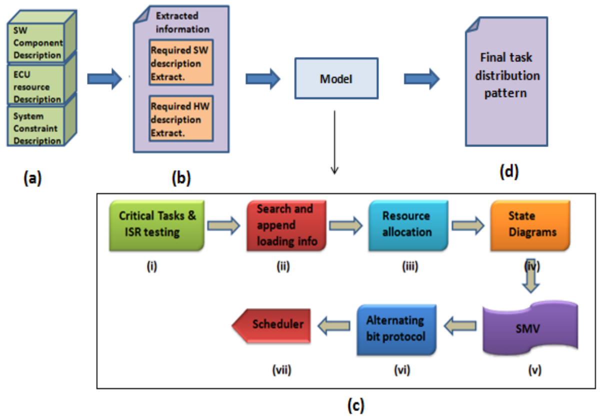

Fig 4: Automated method of task allocation to cores

3.1 Critical Tasks and Interrupt Service routine testing

Initially we identify all critical tasks of the system. The criticality of the tasks can be based on their deadline or priority. These tasks will be isolated and executed on a separate core. For example the engine synchronous tasks which control the functionalities of the engine and the injection system in an automobile are considered critical in automotive applications. These tasks begin execution after engine reaches a particular rpm. Many critical applications depend upon these tasks for execution and hence changing their scheduling sequence during runtime should be avoided. The operating system produces several interrupts which trigger several tasks to execute based on a particular condition. All interrupts could be isolated and executed along with the engine synchronous tasks.

3.2 Task cluster creation and appending load information

A set of tasks with similar priority is called as a task cluster. Following procedure has been employed to create a task cluster. Priority information of the tasks is extracted and stored in a queue along with the task name. A threshold priority is decided which separates the preemptive and the cooperative task sets. The first element of the queue is considered and is compared with all other elements. Sorting techniques like Selection or Insertion sort as defined by the author of [6] is used. Finally queues for both preemptive and cooperative task cluster where the first element would be the task with the highest priority would be present. A look up table containing the load contributed by each task is created. This information is appended to the respective tasks in the queue. By creating specific task clusters, core allocation will be done in a seamless manner and task clusters with higher priority will be allocated to the core

3.3 Resource Allocation

After isolation of the critical tasks and appending the load information, the resources associated with the tasks are identified. All resources associated with these tasks are stored in the global memory. When a task is executed on a particular core, considerable amount of operating system overhead is involved in accessing the resources in the global memory. This overhead affects the performance of the core. To reduce this overhead, the resources used by the cores are identified and stored in the local memory of the core i.e. cache. Figure 5 gives a pictorial representation of the resource allocation concept.

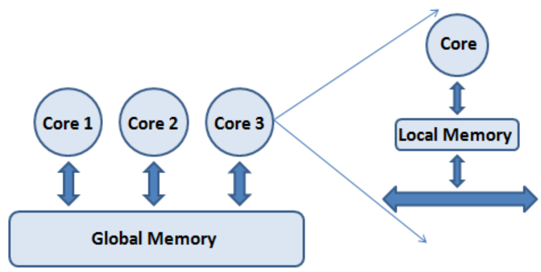

Fig 5: Resource allocation for the given multi core system

Consider a task T1. T1 utilizes global resources r1. If T1 is executed on core 1, r1can be explicitly stored in the local memory of core 1. If the concerned resources are dependent on other resources for their usage or the value of the resources gets updated after a particular task executes, a copy of the resource could be created and this copy could be stored in the cache. Updating the resources will be after a fixed number of cycles. The access time of several processors to global memory and the local memory comparison has been done in [7].

3.4 State Diagram Construction

State Diagrams are used to define and describe the system under concern. The major criteria for state diagram construction are the assumptions that the system concerned has finite number of states. State transitions are done based on the occurrence of events or any conditions. Several approaches can be used to build a state diagram. One of them has been described by the authors of [8]. Construction of state diagram is beyond the scope of this paper.

The tasks which are resource ready are considered for the construction of the state diagram. The construction of state diagram plays a vital role in automatic task allocation.

The system consists of three cores. Let n1, n2 and n3 be the initial states. Let t1 (trying for core 1), t2 (trying for core 2) and t3 (trying for core 3) represent the trying states. Let c1 (core 1), c2 (core 2) and c3 (core 3) represent the core allocated states. During startup, all cores are in their initial states. Each task set may try to check the core for execution but only one task set at a time. After a particular task set has tried and the core is allocated to it, a new task set may also try for the core. Core allocation at this instant depends upon the priority of the task set. The properties considered while building the state diagram are

1. Safety: Only one task will be allocated to any core and only one core can be executed at any instant.
2. Liveliness: Whenever a task set requests a core for execution, it will eventually be permitted.
3. Non Blocking: A task set can always request to execute on a particular core.
4. No strict sequencing: A task need not execute on a core in a strict path.

The term t1n2n3 specifies that a particular task set has tried for core1. Core2 and core3 are idle and can be allocated to any other task set. The term c1n2n3 specifies that core1 has been allocated and other cores are free. Various transitions from S0 (State 0) to S18 (State 18) are shown in figure 6. Each task set may follow any path for its core allocation. The core returns back to the original state after task execution. The transitions from one state to another will be after fixed cycle of time and it is assumed that there are no transitions to the same state.

3.5 Symbolic Model Verifier (SMV)

SMV provides a language for describing the state diagrams and its specifications. The inputs to this model are the state diagram and all the pre conditions that a program should follow while executing. One such condition will be the user specifying the priority of the tasks and the intended core on which a task should execute. The state diagram constructed in the previous section is inputted to SMV. SMV provides readability to our application. SMV analyses the state diagram and verifies the correctness of the given input. If all preconditions are satisfied, a 'True' output will be produced. Failure in satisfying the pre conditions results in a 'False' output and its trace will also be produced. To verify that our system satisfies our properties, we

1. Model the system using description language and arrive at a model μ.
2. Code the property using specific language of the model resulting in φ.
3. Run the model with μ and φ.

SMV is executed in batch mode in Linux or in the command prompt in windows. All models defined here work with satisfaction notation i.e. the satisfactory relationship between the model and the formula.

$\mu \models \varphi$, where $\models$ represents semantic entailment.

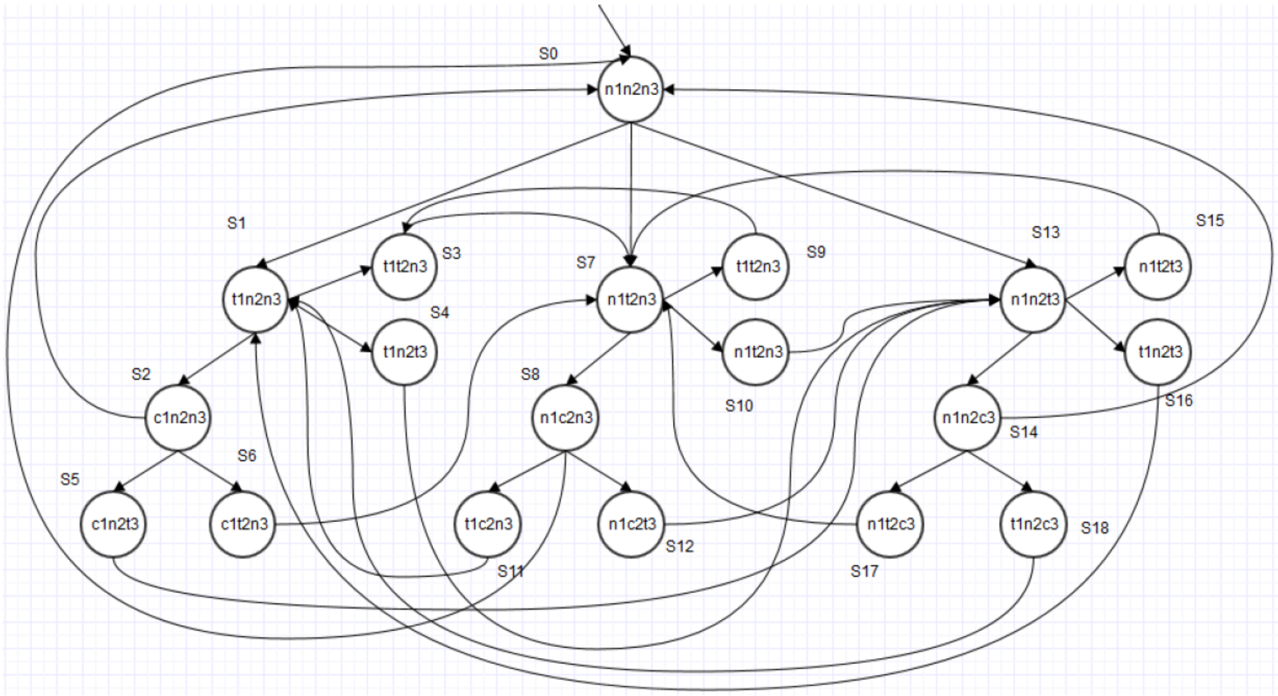

Fig 6: State diagram of the multi core system

State diagram verification

1. Safety: This property is generally expressed in linear time temporal logic. G¬ (c1∧c2∧c3) is satisfied in the initial state.
2. Liveliness: This property is generally expressed in branching time temporal logic. This is expressible as G (t1->Fc1) ∨ G (t2->Fc2) ∨ G (t3->Fc3).
3. Non Blocking: For every state satisfying n1, there is always a successor satisfying t1. The same concept can be applied for states n2 and n3.
4. No strict sequencing: This property is satisfied by the pre conditions assigned to the model.

'G' signifies all future states and 'F' signifies some future state in logic. The state diagram is common to all tri core architecture but the path for each core allocation can vary. Modules in SMV can be constructed synchronously i.e. modules are executed with each global clock tick and asynchronously i.e. modules can be chosen non-deterministically and executed. Usage of SMV prevents the rigorous manual testing involved in determining the task allocation to each core. SMV automatically finds the path which satisfies all our constraints and provides this information to the scheduler.

3.6 Alternating bit protocol

The connection between the scheduler and SMV is assumed to be 'lossy' i.e. messages might be lost during connection but the messages will not be corrupted. Alternating bit protocol is used to transmit the information from SMV to the scheduler. ABP guarantee's that infinite losses will not occur between the sender and receiver and the sender sends the message until it receives the acknowledgement from the receiver. Four agents namely sender, receiver, message channel, acknowledgement channels are present. The sender transmits the message with a control bit. The authors of [9] have described several ways of implementing this protocol.

If the expected control bit is received, acknowledgement is sent and message transmission begins. If the sender/receiver does not receive the control bits, the previous message will be resent. If the connection between the scheduler and SMV are not lossy, the core allocation information can be directly passed on to the scheduler. ABP is similar to Stop and Wait ARQ in computer networks [10] except that it keeps sending the messages. The sender i.e. SMV is partially independent of the receiver. The scheduler allocates the tasks to cores during runtime. When SMV sends the task allocation information to the scheduler, the scheduler might be busy in executing the tasks on some cores. This might lead to loss of task allocation information. Usage of ABP prevents this loss of this information as it continuously transmits this information to the scheduler.

3.7 Scheduler

The scheduler performs the task to core assignment as suggested by the Symbolic Model Verifier. Sub module (d) contains the final task distribution pattern which can be finalized as per the project requirements. The generalized algorithm of our proposed model for task allocation to cores is as shown in figure 7.

INPUT
'm' cores
'n' tasks
(1 to k) functionalities in each task
PROCESS
Step1: Identify all critical and interrupt service routine tasks
Step2: Isolate and execute them in a separate core
Step3: Sort the tasks based on priority
Step4: Append load information and create a queue for pre emptive and cooperative task set.
Step5: Identify all the resources associated with the task.
Step6: Allocate the resource statically to local memory
Step7: The state diagram is built using the tool.
Step8: The model inputs the state diagram to SMV.
Step 9: ABP send the task to core assignment to the scheduler.
OUTPUT
Scheduler allocates the tasks to the cores.

Fig 7: Generalized algorithm for automatic task allocation

4. RESULTS

In this section we discuss the results obtained for our proposed method of task allocation. An automobile system with tri core architecture is considered. Let T1, T2 and T3 be the tasks executing in the system. Details of the tasks and its functionality are shown in figure 8.

Tasks	Functionality
T1	Injection system, Engine system, Power train system, Interrupt
T2	Safety system, Engine system, Interrupt
T3	Entertainment system, Interrupt

Fig 8: Sample task details of an automobile system

In the manual method, tasks are directly allocated to the cores without considering the functionality. In our proposed model, all engine system and interrupt related tasks are combined together on core 1 as they cause high loading. Functionalities related to entertainment system and injection system is combined on core 2. Safety related functionalities and power train system is executed on core 3. Now T1 executes for 40ms and T2 for 30ms and T3 for 35ms.

The result for the task distribution in manual method is shown in figure 9. The utilization of each core is also shown in the figure. The results for the task distribution in automated method are shown in figure 10. Task allocations to respective cores are performed by the scheduler after verification by SMV. The loading of core 1 and core 2 has decreased and that of core 3 has increased by a large amount. Clearly load balancing has been achieved in the automated method. The overall utilization of the multi core processor has also improved.

Core	Task	Functionality	Execution Time	Utilization (%)
1	T1	Injection system, Engine system, Power train system, Interrupt	50ms	50
2	T2	Safety system, Engine system, Interrupt	40ms	40
3	T3	Entertainment system, Interrupt	10ms	10

Fig 9: Performance utilization of cores in manual method

Core	Functionality	Execution Time	Utilization (%)
1	Engine system, Interrupt	40ms	40
2	Injection system, Entertainment system	30ms	30
3	Power train system, Safety system	35ms	35

Fig 10: Performance utilization of cores in automatic method

Figure 11 illustrates the comparison of utilization in manual and automated methods. In the automated method, the utilization of all the cores is almost same.

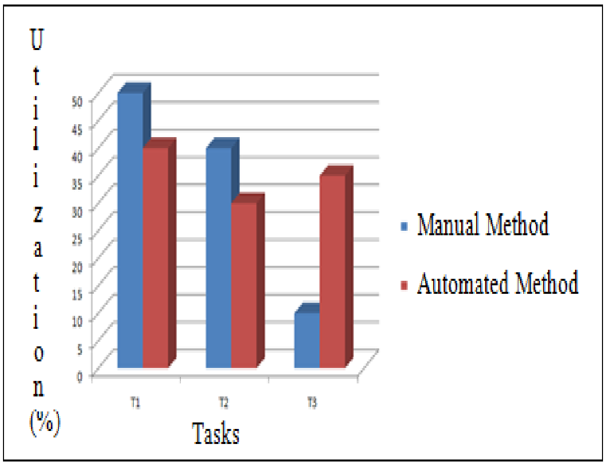

Fig 11: Utilization comparison between manual and automatic method

A desktop computer with installed memory of 3GB, a tri core processor, 32 bit operating system was chosen for our second example. Fig 12 illustrates the functionalities chosen for our testing and the core utilization in the manual method.

Core	Tasks	Functionality	Utilization (%)
1	T1	DOS Game, copying a large word document	30
2	T2	Searching file contents in EXCEL	10
3	T3	Playing audio file, Running a PERL Script	60

Fig 12: Sample task details in a desktop computer

The results of the automated method are shown in figure 13.

Core	Functionality	Utilization (%)
1	DOS Game	20
2	Searching file contents in EXCEL, Running a PERL Script, copying a large word document	40
3	Playing audio file	40

Fig 13: Core Utilization in automated method

5. CONCLUSION

In automotive sector, an automated tool for task allocation has never been proposed and implemented. For the real time applications like automobile, designing a model to automatically allocate the tasks to multi core systems is required for better utilization of cores and to avoid bottlenecks in task allocation to cores. In this work, we have proposed a model which allocates the tasks to the cores in the runtime. This model will solve the problem of bottlenecks during task execution as resource requirements and criticality of the tasks are considered while allocating these tasks to the cores. The model was subjected to verification for a real time automobile application. The results of verification showed approximately equal utilization of all the cores.

Currently AUTOSAR has imposed limitations on the use of dynamic scheduling in automobiles. It is widely predicted that this limitation will be removed in a very short time. Our model can be realized in automobiles after that. The idea of constructing an adaptive state diagram can be one of the future scopes of the project.

The entire process from extracting the resource information, task clustering to final task distribution can be developed as a tool. This tool would require the user to only input the required files at the beginning and it would give the final task distribution.

6. ACKNOWLEDGEMENTS

The work reported in this paper is supported by the college through the Technical Education Quality Improvement Program [TEQIP-II] of the MHRD, Government of India. We would like to thank Dr. S.R.Krishnamurthy, former principal and HOD of Computer Science and Engineering, BMS College of Engineering for his support and guidance. We also express our gratitude to entire faculty of BMS College of Engineering for their support in completing this paper. We also express our indebted gratitude towards Robert Bosch Engineering and Business Solutions Ltd for providing all the necessary support.

7. REFERENCES

[1] Santosh Kumar Jena and Prof. M. B Srinivas , "On The Suitability of Multi-Core Processing for Embedded Automotive Systems", International Conference on Cyber-Enabled Distributed Computing and Knowledge Discover, pp. 315-322, 2012.

[2] Abhinesh S, Kathiresh M, Neelaveni R,"Analysis of Multi Core Architecture for Automotive Applications", International Conference on Embedded Systems – (ICES 2014),pp.76-79,2014.

[3] Andersson B, “Static-priority scheduling on multiprocessors”, Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE, pp.193-202, 2001

[4] Kun-Ming Yu, Shu-Hao Wu, “An Efficient Load Balancing Multi-core Frequent Patterns Mining Algorithm”, 2011 International Joint Conference of IEEE TrustCom, pp. 1408-1412, 2011.

[5] La´ercioL.Pilla, Philippe O.A.Navaux, Grande do Sul,Porto Alegre, Christiane P Ribeiro, PierreCoucheney, FrancoisBroquedis, BrunoGaujal, Jean-Franc¸ oisM´ehaut, ”Asymptotically Optimal Load Balancing for Hierarchical Multi-Core Systems”, 2012 IEEE 18th International Conference on Parallel and Distributed Systems, pp. 236-243, 2012.

[6] Yashavant P. Kanetkar, Data Structures Through C, BPB Publications,pp. 351-394, 2011.

[7] Memory Comparison specifications , URL: http://www.sisoftware.net/?d=qa&f=gpu_mem_latency&l=en&a= [Online accessed in June 2014].

[8] Jung Ho Bae, Heung Seok Chae, “An automatic Approach to generating State Diagram from Contract-Based Class”, Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the, pp. 323-331,2009.

[9] Michael Huth and Mark Ryan, Logic in Computer Science Modelling and Reasoning about Systems,pp.172-203 2011.

[10] Andrew S Tanenbaum, Wetherall, Computer Networks, Prentice Hall,pp. 211-228, 4th edition.