Trends in Task Allocation Techniques for Multicore Systems

Arun Kumar Sundar Rajan[1], Shriram K Vasudevan[2], Nirmala Devi M[3]
[1,3]Department of Electronics and Communication Engineering, Amrita School of Engineering Coimbatore, Amrita Vishwa Vidyapeetham, Amrita University, India
[2]Department of Computer Science Engineering, Amrita School of Engineering Coimbatore, Amrita Vishwa Vidyapeetham, Amrita University, India

Article Info

Article history:

Received Feb 11, 2016
Revised Sep 7, 2016
Accepted Sep 21, 2016

Keyword:

Decentralized scheme
Global scheme
Hardware speculation
Partitioned scheme
Real time operating system
Scheduling algorithm
Thread level parallelism

ABSTRACT

As the functionality in real-time embedded systems becoming complex, there has been a demand for higher computation capability, exploitation of parallelism and effective usage of the resources. Further, technological limitations in uniprocessor in terms of power consumption, saturation in instruction level parallelism, delay in access of memory blocks, etc. led to the emergence of multicore. Multicore design has its challenges as well. Increase in number of cores has raised the demand for proper load distribution, parallelizing existing sequential codes, enabling effective communication and synchronization between cores, memory and I / O devices. This paper had identified and compared the distribution schemes for task distribution in a multicore environment and also the algorithms suitable for decentralized task scheduling scheme. In addition, this paper had addressed the techniques of formulating parallel task blocks from sequential code.

Corresponding Author:

Arun Kumar Sundar Rajan,
Department of Electronics and Communication Engineering,
Amrita School of Engineering Coimbatore,
Amrita Vishwa Vidyapeetham, Amrita University, India.
Email: s_arunkumar@cb.amrita.edu

1. INTRODUCTION

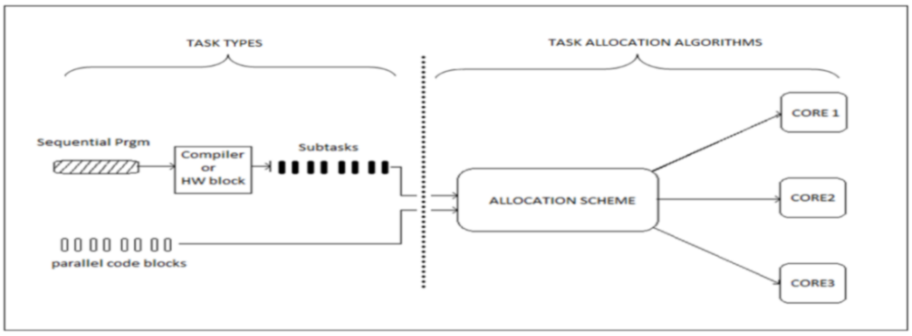
With continuous improvement in semiconductor manufacturing technology, embedding many processing units on a single chip has become a feasible technological trend. Single embedded chip with many individual processing units is termed as multicore and its usage can be seen in many real time applications like nuclear power plants, automobile and aerospace industry. Embedded systems with multicore are developed with expectations to achieve higher performance and faster parallel execution; but the replacement of single core to a multicore hardware alone does not always guarantee the expected higher performance [1-3].

Although each core is an individual processing unit with capability to work independently, higher performance cannot be achieved unless there are steps to ensure proper load distribution and synchronization between cores, memory and Input-Output devices are taken into consideration. Load distribution is handled by a task distributor, which will be capable of avoiding priority inversion between tasks in different cores, longer waiting time and unneccesary missing of task deadlines.

Roles of a task distributor can be divided into 2 levels. First on deciding which core should execute the task. Second is on deciding the time and order of the task execution. Former is termed as allocation activity; while the latter is termed as scheduling activity.

After identifying the need for efficient task distributor and its role, next is to distinguish the type of tasks. Tasks can be broadly classified into two types namely *sequential* and *parallel*. Sequential tasks have to be executed undividedly and as per the program order. Violating this rule can more often lead to data inconsistencies. Parallel tasks are independent code segments that can be executed in parallel among different cores. Handling of task types is elaborated in the Section 4.

With a brief introduction to the basic terminologies (as depicted in Figure 1), this paper begins with an historical overview of the research activities in Section 2. Section 3 describes in detail the task allocation schemes suitable for multicore system.Parallel execution of task blocks using compiler and speculative hardware technique is discussed in Section 4. In addition, control flow and data dependencies that will arise with speculative hardware is also explained with suitable examples. Later, mathematical modeling of task allocation algorithms for decentralized task scheduling scheme is brought out. Finally, a consolidated review is presented in Section 5.



HW – Hardware;

Figure 1. Building Blocks Describing Steps that are Involved from Task Block Formation to Core Allocation

2. RELATED WORK

This Section brings out an overview of the research activities that were carried out with respect to the task scheduling and allocation, starting from single core and time lined to multicore scenario.

Scheduling algorithms were of two types, namely timeline driven approach (also termed as timeline scheduling) and priority driven approach. In timeline scheduling [4], the time period was divided into slots of fixed length and tasks were statically allocated to a slot based on their frequency and execution time. Although timeline scheduling was straightforward to implement, it was fragile under overload conditions-where a task could exceed its predicted execution time and generate a domino kind of effect on the subsequent tasks to surpass their timeline. A solution to the difficulties of timeline driven approach was formulated in priority driven approach. In this approach, tasks were assigned a-prior and scheduler worked based on the priority value.

Out of the many priority based approaches [5-7], the predominant were Rate Monotonic (RM) – where the priorities were statically assigned to the tasks and Earliest Deadline First (EDF) – where the priorities were dynamically assigned to the tasks, in achieving real time behavior for single core environment. In parallel, experiments on finding suitable and efficient algorithms for multicore environment had gained importance. Experiments combining priority based algorithms (RM and EDF) that were successful in single core was tried over multicore environment. Major concerns in mapping such algorithms were complexity, lack of reusability and scheduling anomalies with respect to load distribution and synchronization between the cores [8-10].

In order to achieve effective load distribution, researchers had come up with various heuristic task scheduling techniques based on execution time, completion time of the task and also combinations of algorithms such as MIN–MAX duration algorithms, switching algorithm, sufferage algorithms etc., [11]. Another approach was to reduce the makespan by utilization of parallel task models. Combination of sequential and parallel task scheduling algorithms like MIN / MAX – Round Robin, MIN / MAX – Load Balance, MIN / MAX – Min Number of Task and MIN / MAX – NTWP was experimented [12]. Details of these algorithms are explained in Section 5 of this paper.

Task scheduling algorithms, which were more or less a combination of single core algorithms, were assumed to bring in the required performance uplift to a multicore system. However the results were not fruitful. A strong supplement from effective task allocation scheme was need of the hour. Typically, the allocation schemes can be classified under global, partitioned or decentralized schemes. Section 4 of this paper focuses in detail about the task allocation schemes, their methods and their pros & cons.

Brandenburg [13] performed an experimental study between partitioned EDF and global EDF using LITMUS^{RT} - a derivative of Linux Kernel on 24-core Intel Xeon platform. His experiment showed that partitioned EDF would be more preferable for hard real-time system while global EDF would be more effective in a soft real-time system. Another extensive work in the priority based algorithm for multicore was done by Zapata & Alvarez [14], where RM and EDF algorithms were subjected to best known heuristic functions: worst fit (WF), first fit (FF), best fit (BF) and next fit (NF) for both partitioned and global scheme. Further static scheduling on partitioned scheme for single core and multicore procedure were demonstrated [15-16].

Meanwhile Lauzac et al., [17] had performed a comparative study of global RM and partitioned RM scheme. Their observation also showed that for soft real-time systems-global scheduling can perform better because overloaded processors can distribute tasks to under loaded processors dynamically, whereas in a partitioned scheme a task allocation was fixed. They had also pointed out the scheduling effects of RM with respect to delay seen in low priority tasks under global and partitioning scheme of a multicore environment.

Lee et al., [18] had come up with three algorithms that operate on decentralized scheme namely Local scheduling, Internodes scheduling and Queue balancing. Each of their technique was evaluated in a clustered environment and their experimental results showed of throughput improvement using decentralized scheme.

With growing importance for parallelism, which can bring in higher performance, real-time applications started to focus on parallel task models. Traditionally compiler and hardware speculation based approaches were used to divide sequential tasks; however it has become complicated for compiler to statically determine all the dependencies especially for programs with dynamic data structures. Control flow and data dependencies were also common to occur while partitioning a task based on speculation. In Section 2 of this paper, partitioning techniques and dependencies that would arise are explained in detail with examples.

Diversification of task into parallel models and applying the traditional deadline algorithms were described in works of Li et al., [19]. Another approach in combining DAG direct acyclic graphs was described in works of Saifullah et al., [20]. These approaches could not handle dependencies that were dynamic in nature.

In order to handle dynamic dependencies, hardware based approach using cache like structures-address resolution buffer (ARB) for centralized shared memory multiprocessor [21] and speculative versioning cache (SVC) for distributed shared memory multiprocessor [22] have been explored. In these techniques, each core buffered the value and value was written to main memory only when the operation was committed, else discarded by buffer flushing techniques. Extensive report on speculative multithread architecture can be seen in the works of Sohi and Vijaykumar [23]. Speculation for data to avoid dependency was also tried and the same can be seen in the works of Hammond et al., [24].

Of late, sequential program can be converted to parallel program using OpenMP Fork-join structure. Best and worst case scenarios of using this fork-join model were illustrated in the works of Lakshmanan et al., [25]. For better understanding of scheduling terminology, book by Labrosse [26] on µc-OS II and for topics on parallelism, book by Hennessy & Patterson [27] on Computer Architecture is recommended as they provide good insight.

3. PARALLEL EXECUTION OF TASKS

In order to improve the performance and to utilize the capability of multicore system, tasks are split into smaller blocks and made to execute simultaneously in more than one core. This Section addresses on techniques needed for parallel code execution.

3.1. Task Types

Tasks are broadly classified into two types namely, serial or sequential tasks and parallel or high performance computing tasks. Sequential tasks have to be executed undividedly and as per the program order. Violating this rule would lead to data inconsistencies. On the other hand, parallel tasks are code segments that are independent and as their name suggests, can be operated in parallel. Advantage of parallel task is reduction in overall execution time. The category under which a task falls purely depends on the system design.

Based on dependency in coding, some Sections of serial task are divided into subtasks and each subtask is made to run in parallel among the cores. But who performs the breaking of tasks into subtasks? Subtasks, which are contiguous parts of instruction stream and are executed in parallel, is identified either by programmer during the design and coding phase or by compiler or by using a speculative hardware (Sohi&Vijaykumar (2009)). Details of each technique are elaborated in the following subSection.

Once the subtasks are identified, they can be allocated to one of multiple cores, either statically or dynamically, using the allocation schemes. Section 4 elaborates on the allocation scheme. Parallelism achieved by breaking a program into subtasks is often referred to as Thread level parallelism TLP and this technique is different from Instruction-level parallelism ILP, as each subtask in TLP can contain hundreds to millions of instructions that are executed in parallel with other subtasks.

Architecture that uses threads or subtasks to execute a program in parallel is referred to Multithread Architecture. Eventually, this type of architecture can facilitate simultaneous execution of multiple tasks, as well as multiple subtasks / threads. Along with the benefits of faster execution due to parallelism in multithreaded architecture, certain overheads, namely resource contention-shared memory handling, synchronization and load imbalance follow.

In the upcoming part, techniques using compiler and hardware to breakdown a serial task into parallel are discussed.

3.2. Compiler Based Parallelism

Formulation of subtasks using complier is static in nature. During compilation of a program, the complier identifies code sequences where same set of operations are applied to multiple data items and facilitates parallel operation to that code sequence . For example, consider iteration for adding 'n' objects of an array.

Let 'A' be the Array of size ' n ' , 'e' be the execution time for 1 addition operation and the number of cores be 'm' There will also be additional execution time for branching operation and this branching time is currently ignored for easy understanding of the underlying concept.

In a sequential uniprocessor, the loop performing addition will iterate *n* times and the result will be available only after the duration of ***n * e*** and the same is demonstrated in Algorithm 1.

```
Initialize
m← number of Cores(C)
A← array of 'n' elements
Sum ← result of addition operation

m   = 1; uniprocessor/single cores
Sum = 0; initial value
i = 0;

for every 'i' do until 'i' is equal to 'n'
      Sum = Sum + A[i];
      Increment i;
end for
```

Algorithm 1. Addition operation in single core sequential execution
*Complexity: n * e ; n- number of entries to add, e- execution time for 1 addition operation*

In a multithreaded architecture with the support of compiler, the addition operation can be partitioned among the cores to complete in parallel. This technique would then consume $(n/m + m) * e$ execution time. Additional effort of *m* is included to indicate that results from *m* cores have to be merged/summed up together. In Algorithm 2, function for split, merge and add is shown for reference.

```
Initialize
m← number of Cores(C)
A← array of 'n' elements
Sum ← result of addition operation

Split the 'n' addition operations among 'm' cores,
      1              to n/m  for core $P_0$
      [n/m + 1]      to 2n/m for core $P_1$
      ….
```

```
     ….
     [n(m-1)/m + 1] to n for core $P_m$

add(start,end):
i = start;
count = count + 1; to identify the subtask number
for every 'i' do until 'i' is equal to 'end' cores
     Sum [count] = Sum [count] + A[i];
     Increment i;
end for

merge:
for every 'i' do until 'i' is equal to 'm' cores
     Result = Sum (Values calculated in each core);
     Increment i;
end for
```

*Algorithm 2. Addition operation in Multithreaded Architecture with complier optimization Complexity: (n/m + m) * e; n- number of entries to add, e- execution time for 1 addition operation, m-number of cores.*

3.3. Speculatively Multithreaded Architecture

In Speculatively multithreaded architecture, hardware plays an important role in partitioning sequential program into subtasks. The hardware does the partitioning by speculating that the subtasks are independent. Each subtask is spawned from another subtask and made to proceed in parallel. The key idea in speculation is to allow out of order execution but forces the instructions to commit in order. Hierarchy of two level instruction commitment is required in this architecture: Instructions within each core (subtasks) must locally commit and the overall task among all cores must globally commit in given program order.

Dependency hazards that would arise in a speculative approach and possible solutions for handling the dependency are discussed as follows.

3.3.1. Dependency

Though this architecture speculates that the subtasks are independent, in real time scenarios the subtasks can have control or/and data dependencies. Control dependencies will occur when a branch decision inside a subtask determines which subtask to be executed next. So similar to branch predictors seen in ILP, subtask level predictors are needed to predict the flow. Likewise data dependencies can also arise when a memory or register value calculated in one subtask is consumed in other subtasks. If the hardware detects control or data dependencies across the subtasks, then the hardware enforces the correct order of execution of dependent instructions among the subtasks. Hardware would be able to detect the dependency only during execution or early if ReOrder Buffers (ROB) – common to all cores is used in speculation.

If there had been a violation of control-flow dependency due to misprediction or a data dependency where consumer had executed before its producer, then the hardware rolls back the offending subtasks and all subtasks in later program order [23]. As a general safety procedure all the subtasks following the offending subtasks are rolled back. Mechanisms to identify subtasks that are not affected, helps to reduce number of unwanted rollbacks. For example, consider the below code snippet which highlights the control flow dependency.

```
Subtask A
{
while(1)
      {
       .....
       if (condition > 0)
         .....
         spawn subtask B
         .....
       else
         spawn subtask C
        }
}
```

Control flow of the program is partitioned into three subtasks namely A, B and C. Subtask A is a loop body where each iteration is an instance of the subtask. One instance of Subtask A is predicted correctly and spawns subtask B. Hardware predicts that next iteration will also choose subtask B and spawns another instance of subtask B. The second prediction is incorrect!! Now the roll back mechanism flushes the instance of subtask B and loads with subtask C. With better task level predictors, roll back cost can be averted.

Similarly, data dependencies can also bring down the performance of multithreaded architecture. A key idea for detecting dependencies is to keep track of the program order among the subtasks. When one subtask execution determines the next, task prediction can be done correctly only if the next sequence in program order is known. Similarly, this ordering is critical in detecting and enforcing true register and memory dependencies.

3.3.2. Hazards

For example, consider the below code snippet which highlights on the data dependency

```
mult   a,b,c;      here, a = b * c
sub    x,y,z;      here, x = y - z
mult   d,v,x;      here, d = v * x
store  d,10(r1);   here, d is stored in memory [10+R1]
add    y,a,x;      here, y = a + x
div    d,u,t;      here, d = u /t
store  d,8(r2);    here, d is stored in memory [8+R2]
```

**R1 and R2 are registers of the controller.*

This sequential code block is divided into two subtasks A and B to facilitate parallel execution. Note: Subtasks must be contiguous code blocks.

Subtask A

```
mult  a,b,c;
sub   x,y,z;
mult  d,v,x;
store d,10(r1);
```

Subtask B

```
add   y,a,x;
div   d,u,t;
store d,8(r2);
```

There are 3 possible data inconsistencies that may rise from parallel execution of subtask A and subtask B namely, RAW (Read After Write) violation if **`add`** instruction of subtask B reads the source '**a**' *before* **`mult`** instruction of subtask A writes on to it. Subtask B will have old value of '**a**', which is different from the actual program order. WAR (Write After Read) violation if **`sub`** instruction of subtask A reads '**y**' from register *after* **`add`** instruction of subtask B writes to the same source '**y**'. This results in '**x**' to have incorrect value for subtask A. WAW (Write After Write) violation if **`store`** instruction of subtask A and subtask B interchange in execution order. Both instructions may store *same value at both the memory locations* which is incorrect. **`Div`** or **`mult`** result in subtasks will be totally lost and only the last executed result will be replicated in both the memories.

3.3.3. Solution

The key idea to achieve performance and at the same time to avoid hazards is to allow out of order execution for parallelism and to force the instructions to commit in program order. Hierarchy of two level instruction commit is required: Instructions within each core (subtasks) must locally commit and the overall task among all cores must globally commit in given program order.

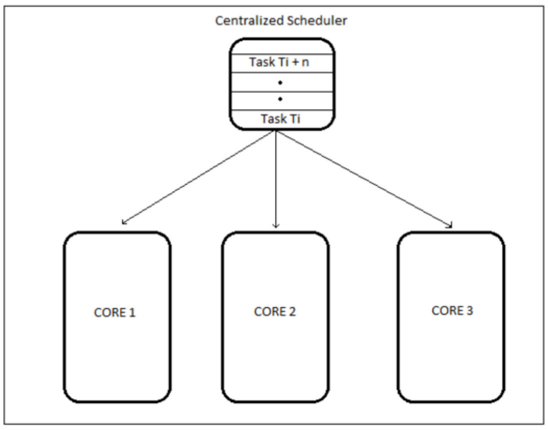
4. TASK ALLOCATION SCHEMES

Having information on the possibilities of task formation next is to distribute tasks among the cores. The techniques for task allocation are classified as global scheme, partitioned scheme and decentralized scheme. This Section elaborates in detail on the benefits and drawbacks of each scheme.

4.1. Global Scheme

In the Global scheme (Figure 2), a centralized scheduler manages task allocation to individual cores and controls the scheduling sequence globally. Being a centralized scheduler, it knows the status of tasks already allocated to the core, so whenever a high priority task is ready to execute, it can be immediately assigned the processing unit for execution. By this method, the occurrence of priority inversion is avoided.

"Priority inversion is said to have occurred when a higher priority task is waiting in the ready queue of one core while a lower priority task is selected to execute on another core". [12]



n – number of tasks and T_i - ith Task in the central queue

Figure 2. Global Scheme with 3 cores and a centralized scheduler

In this scheme, tasks are not always fixed to a particular core. Migration is a feature where tasks can be moved from one core to another and this feature is supported in this global scheme. Even during the task execution, tasks can be moved from one core to another. Scheduling static priority tasks under global scheme by extending uniprocessor RM algorithm experimented by Andersson et al., [11]. Consider a scenario wherein the task *T1* running in Core1 is preempted by high priority task *T2;* now the centralized scheduler allocates *T2* to Core1 and puts task *T1* on hold until any core becomes available. Once a core becomes available, say Core2 and if task *T1* is currently the highest priority task in ready queue, then it is scheduled to resume its execution from Core2.

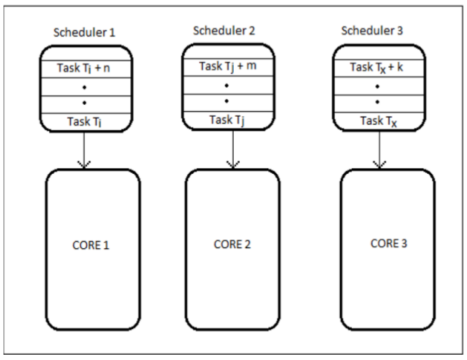
This scheme may also be referred to as centralized scheme-as it maintains a centralized ready queue and it ensures that no same task is being executed at the same time, redundantly in multiple cores. Constraint of this scheme is scalability in sense of amount of tasks and cores-a single scheduler has to manage.

4.2. Partitioned Scheme

In the partitioned scheme (Figure 3), the tasks are allocated to individual cores statically by the system designer and henceforth the tasks are executed only in their respective designated cores. Scheduling is fixed in nature, where the partitioning has to be decided during the design or implementation phase.

The partitioning scheme is preferred to global scheme, as scheduling for multicore can be seen as an algorithm for scheduling on single core, to which a great variety of scheduling algorithms already exist and tasks are executed only on the same designated core. Static scheduling under partitioned scheme was experimented in literature [12], [15-16].

Efficiency of allocation depends on the design of how the tasks are allocated across the cores. Accurately identifying which task runs at a given moment is not possible and this makes the scheme vulnerable to priority inversion. Hence it is the designer's responsibility to ensure that no two tasks are redundantly executed in different cores.



n,m,k – number of tasks allocated to each core and T_i - i^{th} Task of that core

Figure 3. Partitioned Scheme with 3 cores and each core having an individual scheduler

4.3. Distributed or Decentralized Scheme

Decentralized scheme combines the advantages seen in global and partitioned scheme, thereby reducing priority inversion and at the same time improving the system performance. In the decentralized scheme (Figure 4), there are two levels of scheduler. Normally, the levels of scheduler are referred as *L1* and *L2*. Level 1 scheduler (*L1*) is similar to the one in centralized scheme and its duty is to allocate the dynamically queued up tasks to one of the cores based on the availability. Level 2 scheduler (*L2*) is specific to each core and it handles the tasks that are allocated to it by *L1* [12], [18].

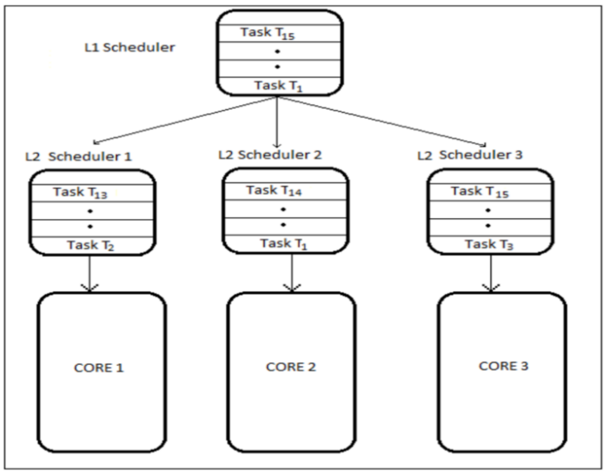


Figure 4. Decentralized Scheme with 3 L2 schedulers and 1 L1 scheduler

To summarize, *L1* performs the task allocation, like a master of all cores and *L2* performs the management of assigned core. So, for a system with 3 cores, there will be 1-*L1* and 3-*L2* schedulers, as depicted in Figure 4. Scheduler L2 also maintains information about tasks pertaining to that core, like state of the tasks in that core, number of tasks in ready and wait queue and finally priority of the tasks. Since *L1* is the master of all cores, it can access the status information via *L2* before allocating a task to the core. In this decentralized scheme, efficient design of the *L1* scheduler provides an effective solution for the task scheduling problem (Kim & Lee (2015)). The *L1* scheduler should be able to efficiently distribute tasks among cores and with minimal overhead.

5. TASK ALLOCATION ALGORITHMS

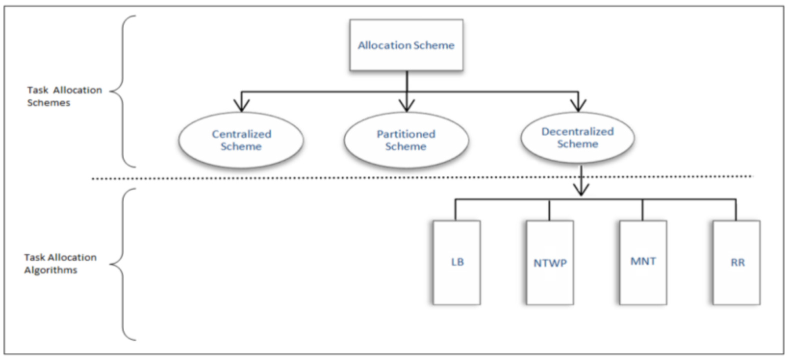
With a clear view on the allocation schemes, techniques to handle sequential code in multithread architecture and implicit dependencies that would arise; next step is to understand the algorithms. This

Section begins with introduction to the terminologies used in mathematical representation of an algorithm, followed by the core allocation algorithms that can be used to distribute tasks between the cores.

Consider a multicore system *S* with *n* cores and each core is denoted as C_i, for all $i = 1$ *to n*. Let's assume that this multicore system *S* organizes the program into *m* tasks and each task is denoted as T_j, for all $j = 1$ *to m* with corresponding execution time E_j and task priority P_j. Based on the execution time E_j, the expected remaining execution time [ERT] of the task on a particular core C_i and the expected start time [EST] of the new task that will be allocated by *L1* to the core C_i can be calculated.

5.1. Core allocation techniques

In this part, algorithms for decentralized scheme with dynamic task inflow are presented (Figure 5). Role of these algorithms is to dynamically select a core and allocate tasks to the selected core. Various algorithms pertaining to the study are discussed as follows.



LB - Load Balance; **NTWP** - Number of Tasks, Waiting time and Priority
MNT - Minimum Number of Tasks; **RR** - Round Robin

Figure 5. Allocation Scheme and Associated Algorithm

5.1.1. Round Robin (RR) algorithm

In Round Robin algorithm, every task is treated with equal priority and job of *L1* scheduler is to distribute the tasks, as and when they arrive, to different cores one by one. Pseudo code for the RR algorithm is presented in Algorithm 3. For example,

If task 'T_j' arrives first, scheduler allocates this task to Core 'C_i' followed by task 'T_{j+1}'to core 'C_{i+1}' and so on.

```
Initialize
n← number of Cores(C)
m← number of Tasks to be allocated(T)
i← core number
for each task 'j' do until 'j' is equal to 'm'
if ('i' is less than or equal to 'n') do
      assign Task '$T_j$' to Core '$C_i$'
      increment 'i' to select next Core
elsedo
      reinitialize 'i' to select first core
end for
```

Algorithm 3. The Round Robin algorithm for 'm' tasks across 'n'cores.

In this round robin fashion, allocation depends only on the arrival order of the tasks. Priority is ignored as all tasks are treated equal.

5.1.2. Minimum Number of Tasks (MNT) algorithm

In MNT algorithm, job of the *L1* scheduler is to distribute tasks to the core containing minimum number of tasks already allocated [12]. In this algorithm, allocation is based on the arrival order of tasks and priority is ignored. Pseudo code for the MNT algorithm is presented in Algorithm 4. For example,

If Core 'C_i' *has 5 tasks,* 'C_{i+1}' *has 7 tasks and* 'C_{i+2}' *has 3 tasks. When task 'T_x' arrives next, scheduler allocates this task to Core* 'C_{i+2}', *which has minimum number of tasks.*

MNT algorithm does not consider about the variation in execution time of each task, which is common in real time systems.

```
Initialize
n← number of Cores(C)
m← number of Tasks already allocated to each core
i← core number
Tx← new task that has arrived

for each core 'i' do until 'i' is equal to 'n'
index←Core number corresponding to Min(mi)
end for

assign Task 'Tx' to Core 'Cindex'
```

Algorithm 4. MNT algorithm which works based on minimum number of allocated tasks on each core.

5.1.3. Number of Tasks, Waiting time and Priority (NTWP) algorithm

NTWP algorithm is an extension of MNT algorithm, with task priority being considered before the allotment is made. Since real time system, used in most applications, cannot treat all tasks to have equal priority – tasks with higher priority have to be given preference in execution [12]. Pseudo code for the NTWP algorithm is presented in Algorithm 5.

In NTWP algorithm, job of the *L1* scheduler is to check for minimum number of tasks assigned to each core, and then checks for cores with minimum ERT and finally chooses the core with minimum sum of task priorities. Core with minimum sum of priority also implies that this core contains many low priority tasks compared to other cores and there is a high possibility of reduced waiting time for high priority tasks in this core. NTWP algorithm was designed in order to reduce priority inversion.

```
Initialize
n← number of Cores(C)
mi← number of Tasks already allocated to each core
i← core number
Tx← new task that has arrived
Ei← total execution time of mi tasks in each core
Pi← sum of priorities of mi tasks in each core
for each core 'i' do until 'i' is equal to 'n'
      index←Core number corresponding to Min(mi)
      flag← 0
      if there are multiple values for 'index' do
            Ei = Sum (execution times of tasks mi)in core
            index_1 = Compute Min(Ei)
            flag  ← 1
            if there are multiple values for 'index_1' do
                  Pi = Sum(priorities of task mi)in core
                  index_2 = Compute Min(Pi)
                  flag  ← 2
end for
```

```
if flag == 0 do
      Assign task $T_x$ to the $C_{index}$
else if flag == 1 do
      Assign task $T_x$ to the $C_{index\_1}$
else do
      Assign task $T_x$ to the $C_{index\_2}$
```

Algorithm 5. NTWP algorithmis based on number of tasks, waiting time and task priority.

5.1.4. Load Balance (LB) algorithm

In LB algorithm, first job of *L1* scheduler is to calculate the total execution time of tasks already assigned on each core. Then the algorithm has to select a core which has the minimum total execution time for allocation of new task. Pseudo code for the LB algorithm is presented in Algorithm 6. In continuation with example for MNT algorithm,

If Core ‘C_i’ has 5 tasks and overall execution time ‘E_i’ = 5ms, ‘C_{i+1}’ has 7 tasks, with ‘E_{i+1}’ = 4ms and ‘C_{i+2}’ has 3 tasks, with ‘E_{i+2}’ = 8ms. When task ‘Tx’ arrives next, scheduler allocates this task to Core ‘C_{i+1}’ which has minimum execution time; on contrary to core ‘C_{i+2}’ from MNT algorithm.

```
Initialize
n← number of Cores(C)
m← number of Tasks already allocated to each core
i← core number
$T_x$← new task that has arrived
for each core ‘i’ do until ‘i’ is equal to ‘n’
      $E_i$ = Sum (execution times of tasks $m_i$) in core
end for
index = min(Ei)
assign Task ‘$T_x$’ to Core ‘$C_{index}$’
```

Algorithm 6. LB algorithmis based on minimum execution time and not on number of tasks

Despite the number of tasks present, total execution time is considered before the new task is allocated. In this algorithm, allocation is based on the arrival order of tasks and priority is ignored.

5.2. Comparison of Task Allocation Schemes

Behavioral attributes of the 3 allocation schemes is summarized in the Table 1.

Table 1. Summary of the Allocation Schemes

	Global Scheme	Partitioned Scheme	Decentralized Scheme
No of schedulers	Only 1 scheduler is required	"n" schedulers ; here 'n' corresponds to number of cores	"(n+1)" schedulers ; here 'n' corresponds to number of cores
Technique	Dynamic allocation	Static allocation	Dynamic allocation
Advantages	No priority inversion, Migration is supported.	Scalable to more cores, Handles concurrent task execution.	Scalability, Reduced makespan, Reduced Priority inversion, Handles concurrent task execution.
Drawbacks	Scalability	Priority inversion, Burden on the designer for appropriate partitioning.	Complexity in handling more schedulers.
Algorithms	EDF and RM variants	EDF and RM for Sequential	EDF/RM with NTWP, MNT, RR and LB for sequential deadline oriented tasks.
		MIN–MIN, OLB, MAX-MIN and Sufferage for concurrent tasks.	MIN/MAX ordering with NTWP, MNT, RR and LB for concurrent tasks.

6. CONCLUSION

In this paper, three task allocation schemes that can be used in multicore system namely: global, partitioned and decentralized scheme are presented. Global scheme benefits in preventing priority inversion, duplication/redundant execution of tasks, and also facilitating migration feature. However, from our references it is evident that when the number of cores increases, performance from global scheme comes down – "scalability" issue. On the other hand, partitioned scheme – where individual cores are statically assigned a set of tasks, facilitates good performance with scalability. Drawback of the partitioned scheme is priority inversion, where a high priority task is waiting when low priority task is being executed in another core. Taking into consideration of the drawbacks from global and partitioned scheme, a new decentralized scheme is introduced. Appropriate task allocation algorithm combined with decentralized scheme reduces priority inversion and overall makespan.

This paper has also presented four allocation algorithms used in decentralized scheme namely RR, MNT, NTWP and LB. Evaluation of the allocation algorithms in term of selection criteria, shows NTWP to have better performance in reducing priority inversion and makespan. This paper has also presented techniques for parallelism by partitioning a sequential task into subtasks that can be executed in parallel. Control flow and data dependencies that would arise from sequential task that are also addressed. Examples using traditional technique of exploiting compiler for ILP is extendable to achieve TLP; however to facilitate dynamism, hardware techniques using ARB and SVC are preferred.

Major idea of this paper is to summarize the terminologies, major events and researches that had occurred with respect to multicore allocation and scheduling activity. Future work lies in evaluating the allocation and scheduling algorithms on project which has combination of serial and parallel code blocks on a multicore RTOS.

REFERENCES

[1] Rho S., *et al.*, "Guest Editorial: Challenges of Embedded Systems as They Evolve into M2M, Internet of Things,"

[2] *ACM Transactions on Embedded Computing Systems* (TECS), vol/issue: 15(2), pp. 34, 2016.

[3] Biondi A., *et al.*, "Moving From Single-Core to Multicore: Initial Findings on a Fuel Injection Case Study," *SAE Technical Paper*, 2016.

[4] Awadalla M. and Konsowa H., "Performance Enhancement of Multicore Architecture," *International Journal of Electrical and Computer Engineering*, vol/issue: 5(4), pp. 669, 2015.

[5] Jensen E. D., *et al.*, "A Time-Driven Scheduling Model for Real-Time Operating Systems," In *RTSS*, vol. 85, pp. 112-122, 1985.

[6] Liu C. L. and Layland J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM* (JACM), vol/issue: 20(1), pp. 46-61, 1973.

[7] Lehoczky J., *et al.*, "The rate monotonic scheduling algorithm: Exact characterization and average case behaviour," *In Real Time Systems Symposium*, IEEE, pp. 166-171, 1989.

[8] Buttazzo G. C., "Rate monotonic vs. EDF: judgment day," *Real-Time Systems*, vol/issue: 29(1), pp. 5-26, 2005.

[9] Baker T. P., "Multiprocessor EDF and deadline monotonic schedulability analysis," *Innull*, IEEE, pp. 120, 2003.

[10] Goossens J., *et al.*, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-time systems*, vol/issue: 25(2-3), pp. 187-205, 2003.

[11] Pillai A. S. and Isha T. B., "Ec-A: A task allocation algorithm for energy minimization in multiprocessor systems," *Middle-East Journal of Scientific Research*, vol/issue: 18(6), pp. 779-87, 2013.

[12] Chaturvedi A. K. and Sahu R., "New heuristic for scheduling of independent tasks in computational grid," *International Journal of Grid and Distributed Computing*, vol/issue: 4(3), pp. 25-36, 2011.

[13] Kim S. C. and Lee S., "Decentralized task scheduling for a fixed priority multicore embedded RTOS," *Computing*, vol/issue: 97(6), pp. 543-55, 2015.

[14] Brandenburg B. B., "Scheduling and locking in multiprocessor real-time operating systems," Doctoral dissertation, University of North Carolina at Chapel Hill.

[15] Zapata O. U. and Alvarez P. M., "Edf and rm multiprocessor scheduling algorithms: Survey and performance evaluation," *Seccion de Computacion Av. IPN*, pp. 2508, 2005.

[16] Andersson B., *et al.*, "Static-priority scheduling on multiprocessors," In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pp. 193-202, 2001.

[17] Crespo A., *et al.*, "Static Scheduling Generation for Multicore Partitioned Systems," In *Information Science and Applications (ICISA)*, pp. 511-522, 2016.

[18] Lauzac S., *et al.*, "Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor," In *Real-Time Systems, 1998.Proceedings. 10th Euromicro Workshop*, IEEE, pp. 188-195, 1998.

[19] Lee J., *et al.*, "Decentralized dynamic scheduling across heterogeneous multi-core desktop grids," *In Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), IEEE International Symposium*, pp. 1-9, 2010.

[20] Li J., *et al.*, "Analysis of federated and global scheduling for parallel real-time tasks," In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference*, IEEE, pp. 85-96, 2014.

[21] Saifullah A., *et al.*, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Systems*, vol/issue: 49(4), pp. 404-35, 2013.

[22] Franklin M. and Sohi G. S., "ARB: A hardware mechanism for dynamic reordering of memory references," *Computers, IEEE Transactions on*, vol/issue: 45(5), pp. 552-71, 1996.
[23] Gopal S., *et al.*, "Speculative versioning cache. InHigh-Performance Computer Architecture," *Proceedings, Fourth International Symposium*, IEEE, pp. 195-205, 1998.
[24] Sohi G. S. and Vijaykumar T. N., "Speculatively Multithreaded Architectures," In *Multicore Processors and Systems*, Springer US, pp. 111-143, 2009.
[25] Hammond L., *et al.*, "Data speculation support for a chip multiprocessor," *ACM,* 1998.
[26] Lakshmanan K., *et al.*, "Scheduling parallel real-time tasks on multi-core processors," In *Real-Time Systems Symposium (RTSS)*, IEEE 31st, pp. 259-268, 2010.
[27] Labrosse J. J., "MicroC/OS-II," *R & D Books,* pp. 9, 1998.
[28] Hennessy J. L. and Patterson D. A., "Computer architecture: a quantitative approach," *Elsevier,* 2011.