POWER AWARE TASK SCHEDULING

ON HOMOGENEOUS MULTI-CORE

SYSTEMS

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Shwetha Shankar, B.E.

San Marcos, Texas
May 2013

POWER AWARE TASK SCHEDULING

ON HOMOGENEOUS MULTI-CORE

SYSTEMS

Committee Members Approved:

_____
Dan Tamir, Chair

_____
Apan Qasem

_____
Mina Guirguis

Approved:

_____
J. Michael Willoughby
Dean of the Graduate College

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

I take this opportunity to thank my principal advisor Dr. Dan Tamir for his invaluable advice, suggestions, and support during the course of this work. It is inspiring to see his painstaking attention to detail and commitment to students. I have been fortunate to have his insightful guidance.

My special thanks to Dr. Apan Qasem for his timely ideas, comments, and suggestions which were instrumental in completing my thesis work.

I thank Dr. Mina Guirguis for participating as a member in my thesis committee, reading my dissertation, and providing useful suggestions.

I want to express my immense gratitude to my family for their constant support and encouragement in my pursuit of Master's Degree. My husband Kumar has been my driving force and my daughter Krithi has been my inspiration. I acknowledge their patience and understanding which has been very crucial in my education path.

This manuscript was submitted on December 10th, 2012.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

POWER AWARE TASK SCHEDULING

ON HOMOGENEOUS MULTI-CORE

SYSTEMS

by

Shwetha Shankar

Texas State University-San Marcos

May 2013

SUPERVISING PROFESSOR: DR. DAN TAMIR

Excessive power consumption affects the reliability of processors, requires expensive cooling mechanisms, reduces battery lifetime, and causes extensive damage to the device. Hence, managing the power consumption and performance of processors is an important aspect of chip design.

This research aims to achieve efficient multi-core power monitoring and control via operating system based power-aware task scheduling. There is a significant amount of research on efficient OS task scheduling algorithms involving performance criteria like execution time. However, there is considerable scope for developing *power and performance* efficient scheduling policies.

The main objectives of power aware scheduling are: 1) lowering processor's power consumption level, 2) maintaining the system within an allowable power envelope, 3) supporting hot-spot elimination, and 4) balancing the power consumption across processors. These objectives are achieved by incorporating power characteristics into the scheduling policies. It is desired, however, to achieve these goals without drastically affecting performance.

Generally, intra-core task scheduling policies engage in selecting a task to execute from a queue of ready tasks. On the other hand, inter-core task migration policies refer to the process of moving ready tasks from one processor's queue to another processor's queue. A special case of task migration is known as task stealing. Task stealing policies involve the concept of a *starving* thief processor stealing a task from a *loaded* victim processor. Therefore, *Task Scheduling* policies in general refer to the broad area of intra-core task scheduling and inter-core task stealing policies.

This study concentrates on the two steps that are part of the OS task scheduling in a multi-core system, namely, intra-core task scheduling and inter-core task stealing. In an

attempt to achieve maximum power efficiency, both the intra-core task scheduling and inter-core task stealing policies have been manipulated to consider the power aspects of processors and tasks.

Moreover, this thesis explores classical single-core task scheduling policies such as Round Robin (RR), Shortest Remaining Time First (SRTF), and Highest Response Ratio Next (HRRN) by employing power features into the task selection policy. A power-based intra-core scheduling policy called Highest Energy-delay-product based Cost function Next (HECN) that integrates HRRN and Energy-Delay-Product into the selection criteria is determined to be the most promising power efficient policy.

In addition, power aware techniques for task migration in a multi-core system are investigated. Ten variants of the work stealing policy have been devised. Under these policies, a thief processor considers both the power and the performance attributes of the system in the process of selecting a victim processor. In addition, the thief's task selection criterion includes power aspects of tasks that reside on potential victims.

A simulator has been developed to enable efficient evaluation of the formulated single and multi-core scheduling policies. The simulator features the ability to perform power aware and / or power agnostic intra-core task scheduling and inter-core task stealing while operating at a relatively high level of abstraction. Simulations have been performed for different task generation scenarios to thoroughly exploit all scheduling policies. The simulator has the capability to provide performance measures of important metrics such

as energy consumption level, turnaround time, and completion time so that the effect on power and performance can be analyzed.

The experiments conducted show that the intra-core HECN scheduling policy coupled with power aware inter-core stealing policies have good potential for power efficient task scheduling with tolerable effect on performance.

I. INTRODUCTION

Power is a dominant obstacle for performance improvements in the VLSI technology. Excessive power consumption affects the reliability of processors. The higher the power dissipation, the higher the heat generated. This in turn requires costly cooling mechanisms, affects battery lifetime, and causes damage to semi-conductor devices. Hence monitoring the power consumption is of high importance in the semi-conductor industry.

This study aims to address this significant power management issue by concentrating on scheduling techniques available at the Operating System (OS) level. *Intra-core task scheduling* policies concentrate on selecting a ready task for a processor while *inter-core task migration* policies focus on moving ready tasks between processors. *Task stealing*, a specific type of task migration, is a multi-core scheduling algorithm that achieves efficient dynamic load-balancing. *Task scheduling* encompasses the broad area of intra-core task scheduling and inter-core task stealing. In the classical work-stealing environment, processors that are executing tasks are referred to as *workers* while *idle* processors are potential *thieves* (or *stealers*). Depending on the state, working or idle, processors make choices with regard to available tasks. Each worker must choose the

next task to be executed. If the idle processor becomes a thief, it must choose the victim processor and the task to steal. The performance of the task scheduling algorithms depends heavily on the task choice. From the classical OS scheduling policies like *First Come First Serve* and *Round Robin* to the more sophisticated OS scheduling policies like *Multi-level Feedback Queue* scheduler and *Completely Fair Scheduler*, these algorithms do not consider the issues of power consumption but instead mainly take into consideration performance criteria like execution time and/or priority of tasks while selecting the next process to run on an idle processor.

There is significant amount of research on algorithms involving execution time as the task selection criteria, focusing on real-time applications, and interacting with hardware. However, research on power aware task scheduling strategies that focus on power consumption issues and integrate power and performance metrics in the selection criteria has considerable opportunities for extension. This study incorporates both execution time and power considerations into the OS based task scheduling on homogeneous multi-core systems.

Problem Definition

Maintaining a homogeneous multi-core system within an allowable power envelope and/or balancing the power consumption across processors without drastically affecting performance are the main problems addressed in this paper. The main objective is to devise an efficient power aware multi-core OS task scheduler for single core and multi-

core systems so that both execution and power consumption of the task are taken into consideration. In addition, this study aims to find mechanisms to lower processor's power consumption level and support hot-spot elimination. These objectives are achieved by integrating power characteristics into the intra-core task scheduling and inter-core stealing policies.

Assumptions

This study assumes that a system has a set of homogeneous processors and the service time of tasks to be executed is known a priori. In addition, it is assumed that estimates of the power consumption rates of individual executable tasks are known.

Hypothesis

It is possible to devise power aware OS based single core and multi-core scheduling strategies by extending classical intra-core task scheduling policies and formulating variants of the inter-core work stealing algorithm to include the power characteristic of processor/tasks while stealing a task and/or selecting a victim processor achieving a higher level of power efficiency without significant effect on the performance or execution time of the processes.

Contribution

This research has been successful in identifying techniques to improve power and performance for both single and multi-core systems. The main contributions of this study are listed below:

1. A power aware intra-core task scheduling policy, referred to as HECN, that considerably reduces the energy consumption level and improves the turnaround time of a processor has been developed.
2. Power efficient inter-core task stealing policies that significantly reduce the energy consumption variance across processors and produces a noticeable improvement in the completion time, for different workload scenarios, have been devised.

Overview

The thesis report is organized in the following way. Chapter 2 gives a brief description of the Operating System concepts pertaining to CPU task scheduling. It discusses classical task scheduling policies that form the basis for power aware scheduling algorithms. With the aim of achieving power efficiency at the OS level via power aware task scheduling, a technique to integrate power characteristics into the selection criteria of task scheduling is introduced. Chapter 3 describes relevant research conducted with regard to OS level power management techniques. The literature survey shows that significant research is yet to be done and provokes studies seeking cost-effective power efficient OS task scheduling policies for single and multi-core systems. This research explores this aspect further. Chapter 4 outlines the power aware task scheduling policies that have been

explored. It provides details on the experimental setup used to evaluate the devised power efficient policies. The emphasis is on the details of the steps involved in simulating an OS based task scheduling environment. The in-depth simulation steps enable the developed simulator to thoroughly exercise the scheduling policies and analyze the potential in these methods. Chapter 5 presents the details of all the simulation experiments conducted with varied task generation scenarios. The results of each experiment are shown with figures that compare the different power aware and power agnostic policies. The outcome of the experiments is analyzed and the behavior and effect of introducing power features into OS task scheduling is studied. Chapter 6 provides an overall analysis of the simulations conducted and draws conclusions from the combined results of all the experiments. Finally, chapter 7 provides conclusion in the form of a report on the main contribution of this study. It throws light on the fact that the research aimed to achieve power aware scheduling policies with minimal impact on performance and has been successful in suggesting power aware techniques with high potential. In addition, proposals for future research work have been recommended.

II. BACKGROUND

This section provides background on the Operating System concepts with respect to a multiprogramming environment.

Task Scheduling Concepts and Terminology

A *Process* is a program ready for execution. A process includes the program code as well as additional components. The process includes the text (code), current activity, stack, heap, and data section. A process can be in one of five states, namely:

New: The process has entered the system.

Running: The Process is executing on a processor.

Blocked: The process is waiting for an event, such as an I/O, to complete.

Ready: The process is waiting in the *ready queue* and is ready to be assigned to a processor.

Terminated: The process has completed execution.

Figure 1 illustrates the different states of a process.

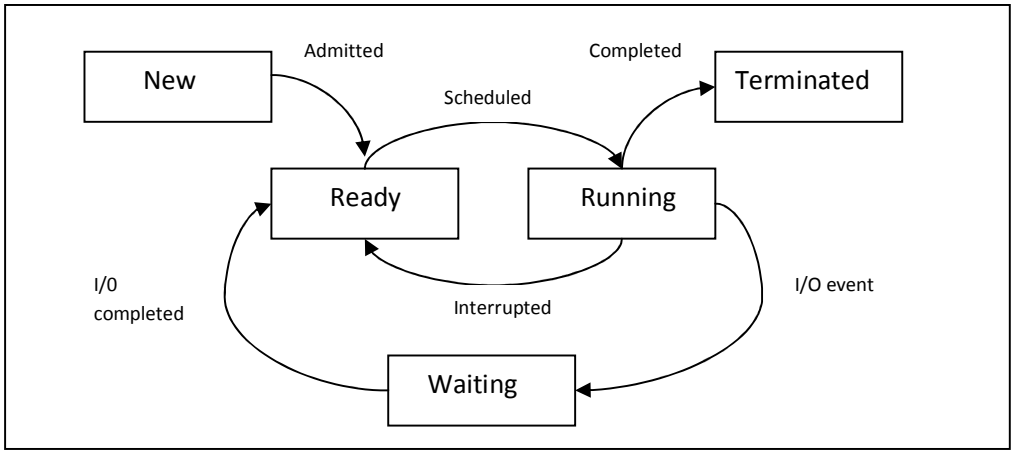


Figure 1: Process State Diagram

A *Process Control Block (PCB)* is used by the operating system to represent a process. The PCB contains several pieces of information associated with a specific process, including process id, process state, and scheduling information. Another term for PCB is *Task Control Block.* For the purpose of this research report, the PCB has been referred to as simply a *task.*

Each task is placed in a different *Task Queue* based on the state of the task.
A *Ready Queue* has tasks ready to run, a *Blocked Queue* contains tasks waiting for I/O operation to complete, and a *Completed Queue* stores tasks that have been completed.

Early computers ran one process at a time. While the process is waiting for an I/O event to complete, the CPU is idle. In multiprogramming, several processes that are in the ready state are kept in memory. If one process has to wait, the operating system *takes the CPU* away from that process and gives the CPU to another process. The objective of

multiprogramming is to reduce CPU idle time and maximize the CPU utilization. The activity of selecting a process to execute on an idle processor is known as *CPU Intra-core Task Scheduling*. Basically, *intra-core* t*ask scheduling* moves a selected process from the ready state to the running state.

A task *Service Time or Execution Time* is the estimate of the total time a task requires to complete execution on a processor. *Waiting Time* is any time that a task spends, in different queues, in the system waiting to be allocated to a processor. The term *Time-Slice* refers to a pre-determined time that a processor is allocated to execute a task before it is released and re-allocated to the next waiting task. *Task Eviction is* the process of removing a task from a processor and moving it from the running state to the ready state. *Task Switching* includes task eviction and task replacement via intra-core task scheduling.

Non-preemptive intra-core task scheduling implies that a task is removed from a processor only upon completion or if a task is waiting for an I/O event to complete. *Preemptive intra-core task scheduling* implies that a currently running task is evicted due to time-slice constraint or because another high priority task just switched from the waiting state to the ready state.

Classical Intra-core Task Scheduling Policies

This study uses several traditional intra-core task scheduling policies as the basis for deriving power aware scheduling policies. The two important types of intra-core task scheduling policies are preemptive and non-preemptive policies. A few examples of both these types of policies are provided next:

(I) The main non-preemptive intra-core task scheduling policies are:

1. First-Come First-Served (FCFS); is the simplest intra-core task scheduling policy. The task that arrives in the ready queue first is allocated to a processor first.

2. Shortest Job First (SJF); as the name suggests, the policy picks a task with the least service time first from the ready queue.

3. Highest Response Ratio Next (HRRN); the task with the highest response ratio in the ready queue is picked next.

$$HRRN = \frac{(w+s)}{s}$$, where w = waiting time of a task, s = service time of a task.

This ratio gives priority to a task with shorter service time. In addition it gives consideration to a task that has been waiting for a long time.

All the above policies are non-preemptive since the next task in the queue is picked for execution only after the current task is completed. The next section discusses preemptive policies.

(II) The main preemptive intra-core scheduling policies are:

1. Shortest Remaining Time First (SRTF); this policy is similar to the SJF scheduling policy described above; but, since it is preemptive, the selection of the shortest task is made every time a new task arrives to the ready queue.

2. Round Robin (RR); this policy is similar to the FCFS scheduling policy mentioned earlier, but preemption is added to reallocate the processor to the next task in the ready queue after a preset time slice.

3. Round Robin with priority: In addition to selecting a new task after a preset time quantum, this policy selects the next task based on the priority of the task instead of directly picking the first task in the queue as done in FCFS policy.

Power Aware Task Scheduling

Most scheduling policies, take into account execution or service time of a task. In addition to service time, a *Power Aware Task Scheduling* policy takes into consideration the power consumption rate of a task.

Power is the rate of energy used. To effectively combine power and service time of a task, a metric called *Energy Delay Product* is considered and derived below.

$$\text{TaskPower (P)} = \frac{E}{s}; \text{ where } E = \text{Energy of a task}, s = \text{service time of a task.}$$

$$\text{Energy Delay Product (EDP)} = s \times E; \text{ where } s = \text{delay time or service time of a task.}$$

$$\Rightarrow \text{Task Energy(E)} = s \times P; \text{ where } P = \text{power of a task.}$$

$$\Rightarrow \text{EDP} = s \times s \times P = s^2 \times P.$$

In this work, power aware task scheduling policies consider EDP rather than the service time of the task. Experiments are conducted to determine an effective power aware task

scheduling policy. These experiments are described in detail later and the results show the potential in power aware task scheduling.

Multi-core Task Scheduling

The scheduling policies discussed so far focus on single core processors. This section talks about extending the scheduling policy to consider multi-core processors.

Task Matching refers to the process of allocating newly arrived tasks to processors/cores by matching parameters of a given task to parameters of a given processor/core (other terms for this are task distribution).

Task Migration literally means moving tasks from the ready queue of one core to the ready queue of another core (e.g., task stealing or work stealing). Several performance metrics described in the previous section, such as the length of the ready queues of each core, the total energy consumed by every core, the anticipated wait time of tasks in the ready queue of cores, and the anticipated completion time of these tasks can be used to characterize the state of a multi-core system. In general, especially for a homogeneous multi-core system, it is desirable to maintain a balance with respect to these parameters among cores. A system (or a state of the system) is referred to as *balanced* if the variance of important parameters among the cores is low. This balance can be achieved through *Task Migratio*n. This is discussed in depth later. In general, the OS is responsible for task matching, intra-core scheduling, and inter-core migration.

Performance Metrics

The performance of the single and multi-core task scheduling policies can be evaluated based on several important metrics. The following are a list of performance measures that can be utilized to study the behavior of scheduling policies.

1. Completion Time - the total time taken to complete executing an entire workload (predetermined set of tasks).
2. CPU Utilization Percentage - the percentage of the completion time that the processor is busy executing tasks.
3. Idle Time Percentage- the percentage of the completion time that the processor is un-utilized and idle.
4. Throughput - the number of tasks completed per time unit.
5. Turnaround Time - the total time a task spends in the system from the time it enters the system until it is completed.
6. Energy Consumed - the energy consumed by a processor in a time unit.
7. Ready Queue Length - the length of the ready queue.
8. Wait Time - the time a task spends in any queue waiting to be executed.
9. Remaining Energy - the estimate of the energy of the all tasks remaining in the ready queue.
10. Remaining Service Time - the estimate of the execution time of all the tasks remaining in the ready queue.

III. LITERATURE SURVEY

This section discusses the relevant research available on single and multi-core task scheduling policies that consider the *energy consumption* of processors.

Task Scheduling Policies

Kashif et al. propose a Priority-based Multilevel Feedback Queue Scheduler (PMLFQS) for mobile devices [9]. PMLFQS is a work-conserving algorithm that uses different CPU speeds for different queues to minimize the overall energy consumed by the CPU for each task. Another policy called Dynamic Voltage and Frequency Scaling (DVFS), shares a similar approach to this policy where the frequency of the processor is adjusted to conserve power [10]. The paper, however, focuses on changes to CPU speed to reflect energy efficiency on single core processors. On the other hand, this research study suggests changes at the software level, enabling a multi-core operating system (OS) to incorporate energy efficiency considerations into the scheduling algorithm.

Wu et al. propose LTEDF (Low Thermal Early Deadline First), a temperature-aware task scheduling algorithm for real-time multi-core systems [11]. In LTEDF, a History Coolest Neighborhood First (HCNF) task allocation algorithm is employed to balance the

temperature loads. If cores are thermally saturated, task migration is performed to alleviate thermal saturation. Therefore, tasks are queued based on deadline priority but selected based on the power and temperature contribution of each task. The paper is focused on real-time systems and on lowering the peak power and temperature consumptions. This study, however, concentrates on non-real-time applications. Moreover, rather than limiting the considerations to peak power, this research considers balancing the power consumption across processors in the system.

Zhou et al. propose an algorithm referred to as THRESHHOT that is based on the observation that, given two tasks, one that is hot (i.e., a high power consuming task) and one that is cool (i.e., a low power consuming task), executing the hot task before the cool one results in a lower final temperature than the reversed order as long as executing the hot task itself does not violate the thermal threshold [12]. Consequently, at each step THRESHHOT selects the hottest task that does not exceed the thermal threshold using an online temperature estimator, leveraging the performance counter-based power estimation. The paper however, focuses on batch processes on a single core and is intended to lower final core temperature. This study aims to consider varying type of processes (beyond batch processes) on a multi-core system with a focus on lowering the variance in energy consumption across processors that in turn balances the temperature of processors as well.

Work Stealing Algorithms

This section looks at several work stealing algorithms to understand the variations in the work stealing process.

Quintin et al. detail the Classic Work Stealing Algorithm. A starving processor, with the number of tasks in the ready queue less than a fixed threshold, is referred to as a thief. The thief identifies a processor, known as the victim, at random and steals the oldest task from the victim [13]. In addition, they propose the idea of grouping processors as *Leader* or *Slave*. The risk of congestion between huge groups of processors arises with the amount of transferred data. To limit this risk, they chose to restrict in each group, the number of processors that can steal from another group. In each group, only one processor sends remote steal requests. This processor is called a Leader. The leader oversees a group of slave processors. Therefore, each leader gives work to the cluster if there is not enough work, and keeps the large tasks to efficiently balance the load between leaders. Leaders execute only global tasks and balance the load between slave groups. Slaves perform the classical work-stealing algorithm within their group. The policy described in this paper performs stealing at two levels, leader and slave level, that may lead to redundancy. Instead in this research, stealing policies are being devised for a homogeneous system such that all processors (that have load imbalance) can initiate stealing with the help of one efficient central unit.

Sarkar et al. propose two policies [14]. In the first *Work-first* policy, the processor executes the spawned task eagerly and leaves the continuation to be stolen. In the second

Help-first policy, the processor makes the spawned task available for stealing and continues execution on the parent task. This paper discusses policies mainly for parallel workload with several spawned tasks that requires prior knowledge of the level of parallelism and task dependency. On the other hand, this work aims at developing power aware policies for all types of workload but still allowing the victim processor to decide on the task to volunteer.

Agarwal et al. propose a Central Task Scheduler that can maintain information of all the processors in the system [15]. The thief computer sends a request to the Task Scheduler and is routed to heavily loaded computer for stealing tasks. The thief computer cannot have more than half the load of the victim computer after work stealing.
The paper discusses a central scheduler that monitors the loaded processors .On the other hand, this research study goes a step further by having a global scheduler that tries to balance the load and energy consumption across processors.

Sudarshan et al. discuss a similar policy that mainly consists of a Dispatcher and nodes [16]. The main server forms a Minimal Spanning Tree (MST) of the idle nodes. If any node is in an idle state or busy state, it has to transmit message to the dispatcher. As soon as an IDLE node is given work it detaches itself from the MST. After this, the detached node begins independent processing of the workload it is assigned. The dispatcher's role is the management of tasks, including maintenance of load balancing, monitoring the status of each node, selection for nodes for task execution, and assignment and adjustment of tasks for each node. Whenever a node joins or exits the system, the

table of candidate nodes is updated. The paper proposes a dedicated monitor for idle processors and focuses on CPU utilization while this study goes beyond considering the idle processors by monitoring the power consumption and load of running processors as well.

Robison et al. propose that if a processor t_1 spawns a task that has affinity for another processor t_2, processor t_1 puts a pointer to the task in t_2's Mailbox. If a processor is idle, before it resorts to stealing, it checks the mailbox and first processes those tasks in FIFO order. Since this mailed task is a part of the general pool, there could be more than one thief attempting to steal it. So there is an *idle flag* associated with each processor. The flag indicates whether a processor is trying to steal work. Thieves are not allowed to steal a task that has been mailed to a processor whose idle flag indicates it is idle [17]. The paper involves scheduling tasks with predetermined affinity to processors but this research involves tasks that can be executed on any processor in the system with the same level of efficiency and therefore focuses on the power characteristics of the task and processor during scheduling.

Faxén et al. suggest two policies [18]. The first is *Sampling Victim selection*. In this policy, a thief does not steal the first task it finds. Instead, it samples several potential victims and selects the one with the task that is closest to the root of the computation. The second policy is the *Set Based Victim selection.* If there are a significant number of active thieves in the system, each thief only attempts to steal from a subset of the other workers. The Sampling Victim policy determines the best task to steal based on the time of arrival

in the system while the Set Based policy limits stealing to a subset of processors. However, in this research, the policies consider all the available victim tasks but choose the best task based on the power considerations of the task.

IV. EXPERIMENT SETUP

This section describes the simulation environment and details the simulation steps. The simulator emulates a multi-core processor system having a central unit that enables CPU task scheduling similar to an Operating System.

Simulator Environment

The four major components required to simulate an OS based task scheduling environment are: a Central Unit, a group of Processors, a set of Tasks, and a few Task Queues. Within the simulator, each of these components has been developed as individual modules or simulation units. The simulator is implemented as a finite-state machine and the functionality of the system is driven primarily by the state of two of the components mentioned earlier; namely, a Processor and a Task. The following are the main states of these two components.

Task States:

1. Executing on a Processor
2. Ready (in ready queue)
3. Blocked [for I/O] (in blocked queue)
4. Completed

Basically, a Task that is ready to be executed can be waiting in the ready queue, a task that is selected from the ready queue can be executing on the processor, a task can be locked due to I/O interrupts, and a task can be complete and terminated. A task can fork or spawn a new task that is added to the ready queue; but, this has not been addressed in this research.

Processor States:

a. Running or Executing a task
b. Idle
c. Working at particular frequency using DVFS (This is a potential state for future work; but, it is out of the scope of this research)
d. Turned off (by the firmware or by the OS; but, it is also out of the scope of this research)

Hence, a processor can be executing a task and consume power based on the task's power consumption rate. In fact, the processor can be idle and consume power based on a pre-determined idle power consumption rate.

Figure 2 shows the main properties and functions of the simulator components along with the relationship between components.

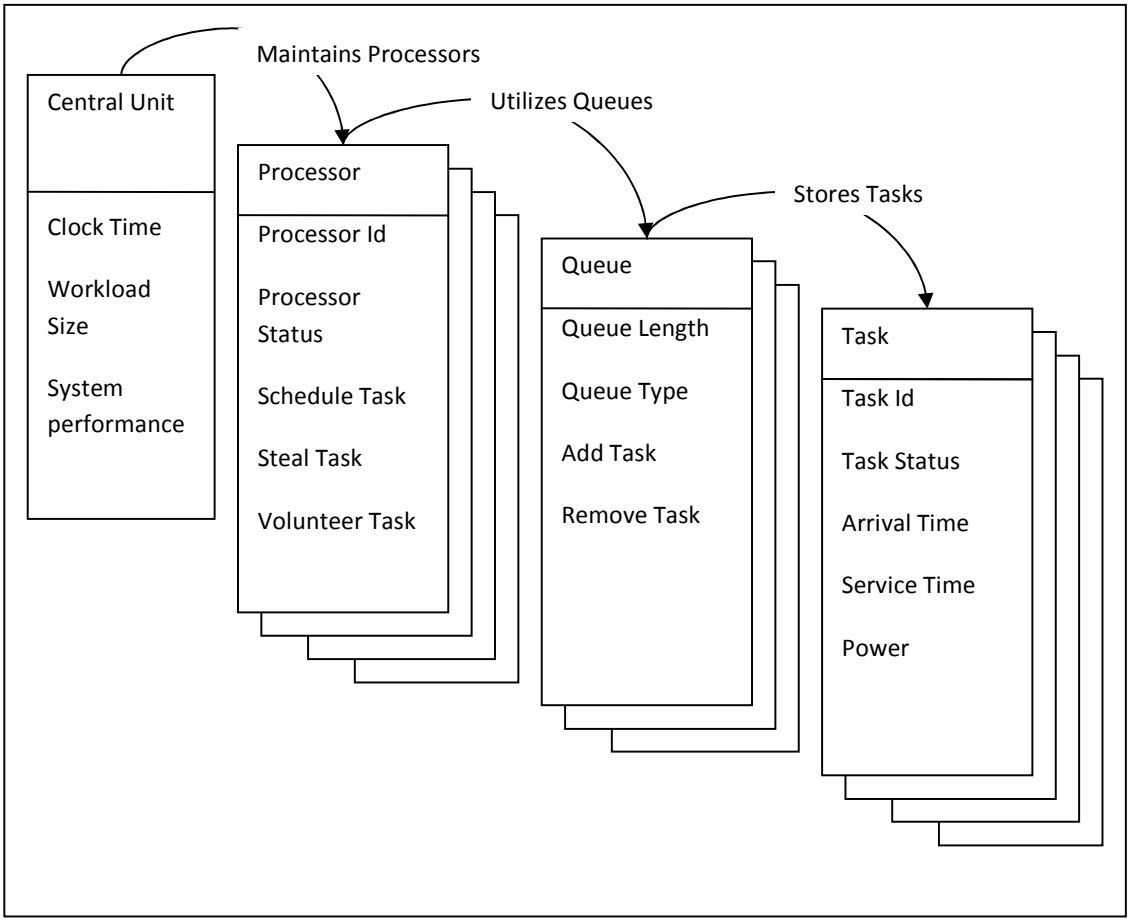


Figure 2: Simulator Components

Every *Task* is identified in the system using a unique task id. Each task has an attribute of power and execution time associated with it. The processor schedules tasks for execution based on these task properties.

The *Central Unit* has been simulated to maintain global knowledge of the entire system much like an operating system. This unit generates tasks, sets task properties, and allocates these tasks to processors for execution. In addition, it enables intra-core task scheduling on each processor, monitors the task load of every processor, and enables

inter-core task stealing between processors. Furthermore, this unit monitors the performance of all processors, reports system status, and updates simulation clock time.

All the *Processors* in the system are identical making the system homogeneous. Each processor simulated in the system has a scheduler module that determines the task to be executed next. In addition, there is a processing module that simulates execution of the task assigned to it. Three *Queues*, namely, the ready, the blocked, and the completed task queues are used by the processing module to store tasks. The tasks allocated to the processor by the central unit are initially placed in the ready queue. As the simulation progresses, the scheduler module moves tasks between the three queues depending on the state of a task. Based upon the length of the ready queue, the processor can be considered as *Starved* when the number of tasks in the ready queue is below a fixed threshold T_s or *Loaded* when the number of tasks in the ready queue is above a given threshold T_l. A starved processor, known as a thief, picks a victim (loaded) processor identified by the starving processor and / or by the central unit. The selected victim then volunteers a task to the thief. This concept is discussed in detail later in the *Task Migration* section.

Figure 3 depicts the high-level interaction between the processing module and central unit of the simulator discussed above.

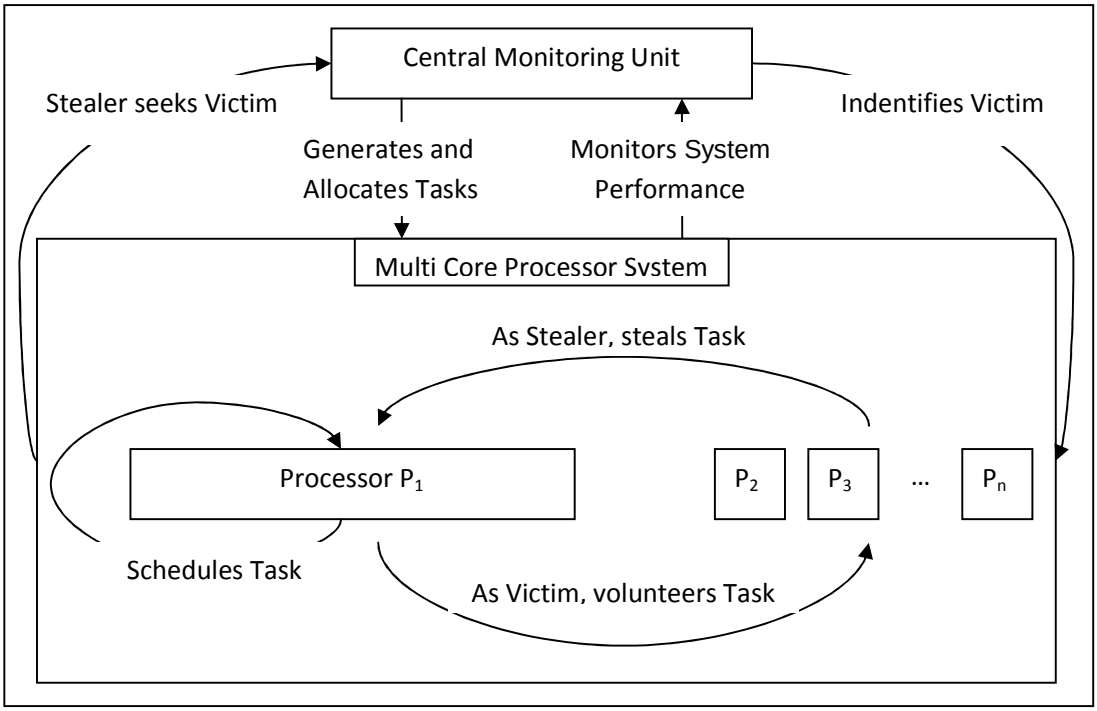


Figure 3: High Level Simulator Framework

Simulation Time Units

This section discusses the basic time units used in a simulation. The following are the two main time units:

- A processor atomic time unit is referred to as a *tick*. A *tick* is assumed to represent n-cycles of execution by a processor.

- The operating system atomic unit is called a *slice*. A *slice* is derived from ticks and is represented as k-ticks.

The simulator is time based as opposed to event based; in this time based simulation paradigm, ticks and slices are the two main time units. A tick is the time set for the OS to perform basic operations such as task switching. A simulation slice has the same meaning as a time slice in the context of OS. Most of the OS operations (e.g., intra-core scheduling, inter-core stealing, etc.) occur at slice boundaries. The processor status is updated on each tick. On the other hand, system status is updated on each slice.

Simulation Steps

This section describes all the steps involved during the simulation. Figure 4 shows the main steps involved in the simulation.

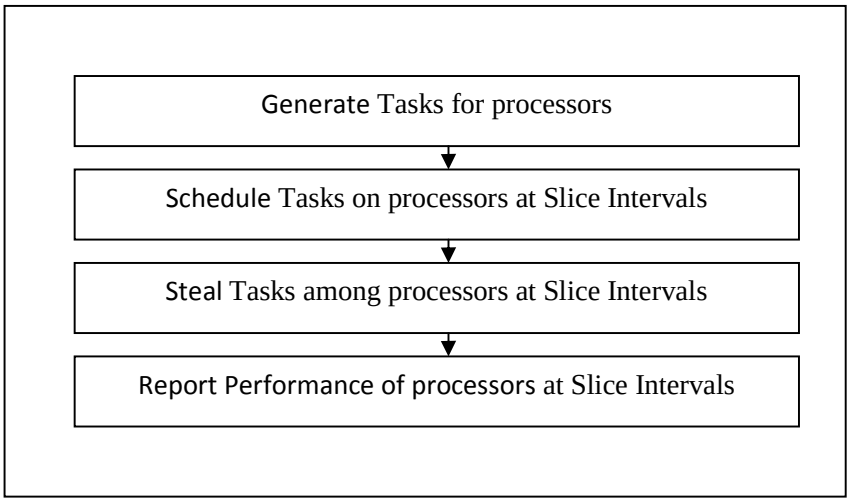


Figure 4: Simulation Steps

First tasks are generated. These tasks are then scheduled on different processors. Next, stealing is performed between processors. Finally, performance of all the processors during the entire simulation is reported. These simulation steps are elaborated next.

Simulation Step 1: Task Generation

The first step in the simulation is to establish a task load also known as *workload*. This simulator synthetically creates tasks. In this thesis, the arrival (generation) of tasks in the system follows a Poisson distribution. This provides a setting that is close to the certain realistic scenario. Each task has attributes of power and execution time associated with it. Every task is given a random attribute of power and execution time that follows an exponential distribution as it efficiently represents a system with varying tasks. The unit for the assigned task power is *Joules/tick*. The assignment is based on the realistic power estimates of a set of tasks that are executed, in a four core system, on a core running at 1 GHz. The average arrival rate, average task power, and average service time of tasks are parameters that can be altered in the simulator.

The simulation can be run for a fixed period of time or until a fixed workload is completed. In the former case, the tasks continually arrive to the system and the simulation runs for the given time period. In the latter case, the tasks are generated until a fixed workload size has been reached and the simulation runs until this workload is completed.

This research study is focused on simulations for a fixed workload. In order to conduct interesting experiments with varying conditions, the average arrival rate of tasks can be manipulated to create the following two scenarios.

- Parallel Workload Scenario - has a fast task arrival rate so all the tasks in the workload arrive to the system early in the simulation period. This causes a sudden

increase in the processors' queue size in the initial period of the simulation. This mode fits a scenario of fine granularity parallelism that has a few tasks, each of which represents a single parallel program, forking numerous tasks.

- Steady State Workload Scenario - involves a slow task arrival rate that spreads the arrival of all tasks in the workload across a long time period in the simulation. In contrast to the parallel workload scenario, here the processors' queue size remains steady through most of the simulation period. This explores a system, such as a communication system, in steady state that handles continuous arrival of tasks.

Simulation Step 2: Intra-core Task Scheduling

Task scheduling policies used for single core scheduling can be utilized for the intra-core scheduling in a multi-core system. There are two main approaches to intra-core scheduling: preemptive and non-preemptive. Nevertheless, in the context of multi-core power aware scheduling, non-preemptive intra-core policies are more restrictive and less interesting, since the constraint of non-preemptive intra-core scheduling limits the OS capability to affect power / performance. Moreover, the research into preemptive intra-core scheduling can be used for evaluating the cost effectiveness of non-preemptive intra-core policies. For example, a slice based preemptive intra-core scheduling with long slices can be used to approximate non-preemptive intra-core scheduling. The opposite is not true. That is, results of research on non-preemptive intra-core scheduling cannot be easily used for evaluating preemptive procedures. Hence, this research concentrates on preemptive intra-core scheduling. The preemption can be synchronous or asynchronous. Nevertheless, in synchronous preemptive intra-core scheduling, the preemption can occur

only on the boundary of an OS atomic unit referred to as slice, which is the most commonly used preemption method. This research explores sliced based synchronous preemptive intra-core scheduling.

The following intra-core scheduling policies have been implemented in the simulator:

1. Round Robin (RR)
2. Shortest Remaining Time First (SRTF)
3. Highest Response Ratio Next (HRRN)

Power Aware Intra-core Task Scheduling Policy

The goal of power aware intra-core scheduling is to significantly improve energy consumption with minimal effect to task completion time. The three intra-core scheduling policies mentioned earlier have been modified in the simulator to consider power as detailed below:

1. Power Aware Round Robin (pRR); in the power agnostic round robin, tasks are evicted at every time slice. The power aware round robin goes a step further and evicts tasks upon reaching a power consumption threshold as well.
2. Power Aware Shortest Remaining Time First (pSRTF); the power agnostic SRTF policy picks a task with shortest time. In contrast, the power aware SRTF policy selects a task with shortest Energy-Delay-Product (EDP).
3. Highest EDP Cost function Next (HECN); the p*ower agnostic* intra-core scheduling mechanism uses the HRRN ratio to derive priority of a task. Since the *Energy-Delay-Product (EDP)* metric is considered as a meaningful combination of *power and time,* a

new *power aware* intra-core scheduling policy called HECN is devised using the EDP metric. The HECN policy determines the *priority* of a task by using a heuristic which is an EDP *based Cost function* that is similar to the HRRN ratio.

The cost function integrates *wait time* and *EDP*, which is mix of two different units. This is not a concern since it is solely used as a heuristic for deciding on the task to execute next. The *EDP* metric allows power characteristics of a task to be included in the task selection process.

Hence, the cost function gives high priority to a task with low power consumption rate. On the other hand, the *wait time* metric ensures that low priority tasks are not kept waiting for an unreasonable amount of time. By combining the two units, a good compromise is achieved between power consumption and performance degradation. The derivation of the cost function is provided next.

$$EDP = s^2 \times p \;;$$

$$HECN = \frac{[w + (EDP)]}{(EDP)}$$, where w = waiting time of task,

s = service time of task,

p = power consumption rate of task

$$\Rightarrow \quad HECN = \frac{[w + (s^2 \times p)]}{(s^2 \times p)}$$

Simulation Step 3: Inter-core Task Migration

The central unit of the simulator enables task migration in the form of work or task stealing. The central unit monitors the following processor level properties to facilitate task stealing.

The Processor Properties are:

1) Starved; a processor with the number of ready tasks below a fixed threshold (T_s).
2) Loaded; a processor with the number of ready tasks below a fixed threshold (T_l).
3) Energy Consumed by the processor so far.
4) Energy Consumed by the processor in last k slices.
5) Remaining Energy; the energy of the tasks in the ready queue shows the potential amount of energy the processor may consume.
6) Remaining Service Time; the service time of the tasks in the ready queue shows the potential amount of time the processor may execute.
7) Ready Queue Length.

Task Migration Process

Task migration occurs if the system is in extreme imbalance and certain cores experience an extremely high peak in a given parameter while other cores experience an extremely low peak in that parameter. In this case, the cores that experience extreme (high or low) values of the given parameter might initiate a task migration transaction. In this study, the *ready queue size* is considered as the parameter that indicates imbalance since it aptly measures the varying workload size of processors. Task or work stealing (or task volunteering) might be an essential remedy to fix the imbalance in the ready queues. A core is considered as *Starved* if the number of tasks in the ready queue falls below a threshold T_s. On the other hand, a core is considered as *Loaded* if the number of tasks in

the ready queue is above a threshold T_s. A core is considered as *Normal* if it is neither starving nor loaded. This type of core does not participate in work stealing.

A *starving* processor is a potential stealer and a *loaded* processor is a potential victim of stealing. A stealer initiates the stealing process by seeking a victim. The stealer identifies a victim. The victim volunteers a task to be stolen. The stealer steals this task by migrating it to its own ready queue. This process is referred to as *Task Stealing or Task Migration* and is being performed at every slice during simulation.

Two stealing models are utilized:

1) The Local Knowledge model; each processor is only aware of its own current status.

2) The Global Knowledge model; each processor is aware of the state of every other processor. This is enabled via the central unit that maintains the status of all processors.

Power Aware Inter-core Task Stealing Policies

The simulated central unit enables the following stealing policies.

(1) Local Knowledge

a. *Random_MinHECN_Task*; the stealer chooses a random processor as a potential victim without knowledge of the processor's load. This victim processor volunteers a task with the lowest HECN. If that randomly chosen processor is not *loaded*, then no stealing occurs.

(2) Global Knowledge

a. *MaxLoaded_MinHECN_Task*; the stealer identifies a processor with the largest ready queue as a victim. This victim processor volunteers a task with the lowest HECN

(presumably is the most power consuming task).

b. *MaxMin_ HECN_Task*; each loaded processor (a potential victim) volunteers a task with lowest HECN (presumably is the most power consuming task). The stealer considers the tasks volunteered by all potential victims and finds a task with the highest HECN (presumably is the least power consuming task) among all volunteered tasks. Hence the name MaxMin, which implies that the MaxHECN task is selected from the available MinHECN tasks.

c. *MaxRemainingService_MinHECN_Task*; the service time of tasks remaining in the ready queue can be used to estimate the time the processor might execute and the power that might be consumed. A queue with highest service time has the maximum potential to increase the power consumption of the processor. Therefore, the stealer picks the processor with a ready queue that has the maximum remaining task service or execution time. The victim processor volunteers a task with the lowest HECN.

d. *MaxRemainingEnergy*; the power of tasks remaining in the ready queue indicates the power that the processor might consume. A queue with high task power has the maximum potential to increase the power consumption of the processor. Hence, the stealer selects the processor with a ready queue that has the maximum remaining task energy. In addition, two variants of this policy are used:

(i) The victim processor volunteers a task with the lowest HECN, *MinHECN_Task.*

(ii) The victim processor volunteers a task with maximum energy, *MaxEnergyTask.*

e. *MaxEnergyInLastKSlices*; the stealer chooses a processor that has consumed the maximum amount of energy in the last *k* slices of the simulation. Again, the two different options for the victim processor are:

(i) The victim processor volunteers a task with the lowest HECN, *MinHECN_Task.*

(ii) The victim processor volunteers a task with maximum energy, *MaxEnergyTask.*

f. *MaxEnergyConsumed*; the stealer opts for a processor that has consumed the maximum energy so far in the simulation. Two further options the victim has are:

(i) The victim processor volunteers a task with the lowest HECN, *MinHECN_Task.*

(ii) The victim processor volunteers a task with maximum energy, *MaxEnergyTask.*

The power agnostic version of the above inter-core task stealing policies uses the HRRN ratio in place of the HECN cost function to determine the task to volunteer.

Each of the three simulation steps described so far can be executed with different variations, depending on the functionality required. The simulator allows these variations to be tested by providing several parameters. Experiments with different scenarios can be performed by altering the values of the following parameters:

Simulation Parameters

1) Multi-core system parameters
    a. Number of cores
    b. Processor power consumption per tick (at idle state)
    c. Slice time (in ticks)
    d. Threshold for starvation/loaded status
    e. Workload size
    f. Stealing policy
2) Task parameters
    a. Task arrival rate
    b. Task power consumption per tick
    c. Task service time
    d. Blocking probability
    e. Unblocking probability

Simulator Modes:

1) The simulation can emulate power aware or power agnostic task scheduling policies.

2) The simulation can emulate task scheduling with and without task stealing.

Simulation Step 4: Performance Reporting

The central unit of the simulator monitors the entire system by capturing and reporting the progress of the simulation and the status of the processors. In addition, the simulator provides all the performance measures on a slice basis. These simulation reports can be used to analyze results, generate graphs, and derive conclusions.

Simulation Flow

This section provides the details of the simulation flow. Figure 5 represents the simulation steps, described in the previous section, using a flowchart.

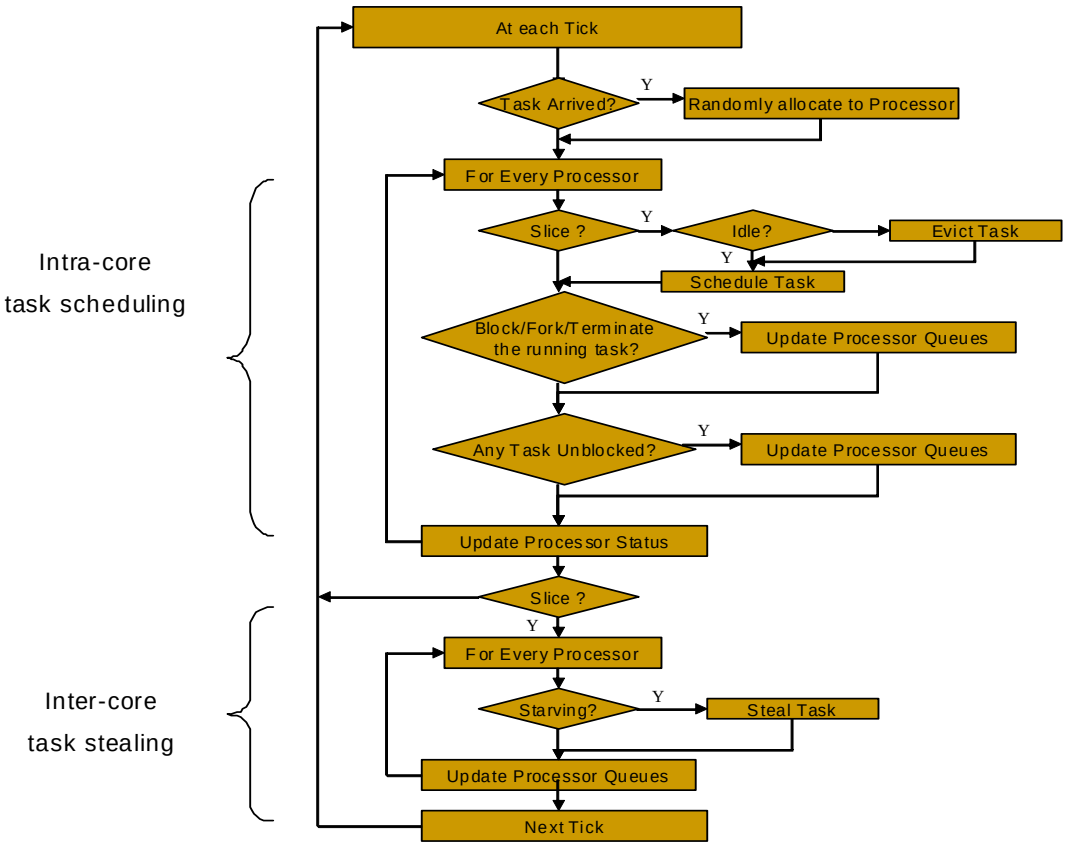


Figure 5: Simulation Flowchart

The simulation progresses on a tick basis. At every tick, the simulator determines if a new task has to be generated. If so, a task is created and randomly allocated to a processor. Next, the intra-core scheduling is performed on every processor and the processor status is updated. Followed by intra-core stealing which is conducted on a slice basis. The simulation proceeds until either the workload is completed or the fixed time period is reached. The next section details the experiments conducted and provides the corresponding performance reports.

V. EXPERIMENTS AND RESULTS

This section reports the experiments conducted as part of this study and provides the results of these experiments. All the devised task scheduling policies have been exercised thoroughly by performing experiments with varying scenarios. Moreover, the experiments comprised of single core and multi-core simulations with both *fixed time-period* and *fixed workload* situations. First the detail of each experiment type is listed. Second, the format used to tabulate the simulation data for both single and multi-core experiments is described. Next, the actual configuration of the parameters used in each experiment is provided. Finally, the result of every experiment is reported and the performance of all the policies is detailed.

Figure 6 contains a flow chart of the different types of experiments performed. Details on each of the experiments are provided later in this section.

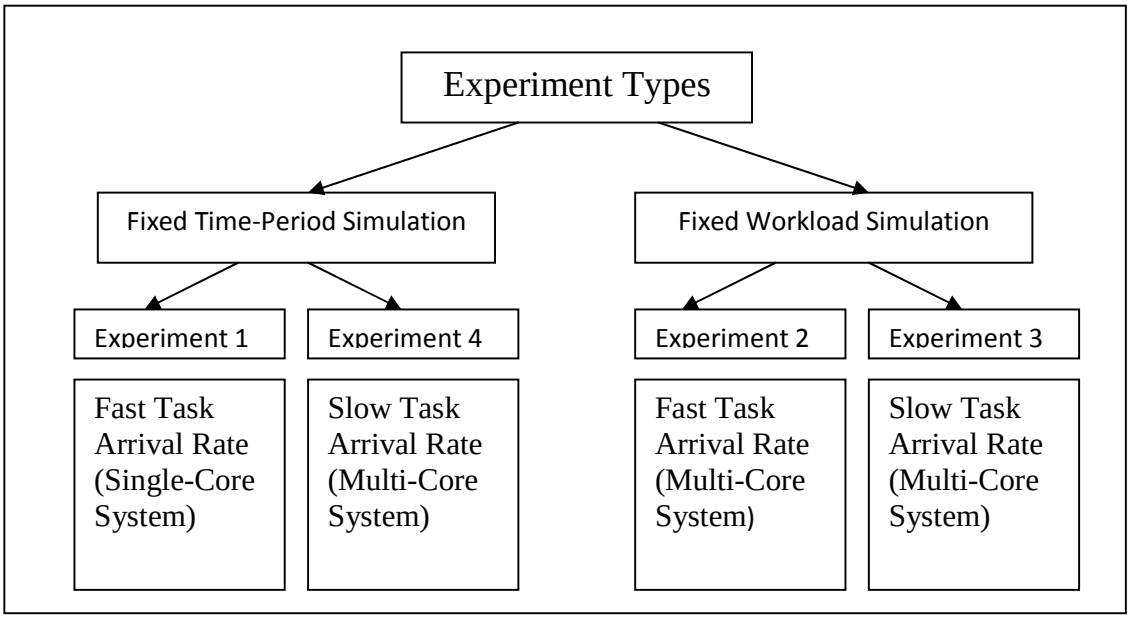


Figure 6: Experiment Overview Chart

The following performance metrics are considered in each simulation to analyze the data and tabulate the results.

1) Energy Consumption Variance; the variance in the energy consumed by processors during the simulation.

2) Average Turnaround Time; the average turnaround time of all the tasks in the workload.

3) Peak Ready-Queue Length; the maximum ready queue length recorded for any processor in any slice of the simulation period.

4) Completion Time; the time required to complete a simulation.

Experiment Data Tabulation Format

This section provides information on the format used to collect data from the experiments in order to analyze and generate results. The tabulation format used for single core and multi-core simulations is explained below.

Single Core Simulation Data Format

The simulations are performed for single core task scheduling policies. Each simulation is repeated several times with different seeds for random number generation. The performance of the processor over the entire simulation period is monitored. The simulation data is recorded for each intra-core task scheduling policy as explained next. First, the performance metrics measured in every simulation are tabulated (c.f., Table 1). Next, the average of each performance metric across all the simulations, performed using different seeds, is calculated. Finally, similar data is gathered for all the intra-core scheduling policies and the simulation results are compared to determine the experiment outcome.

Table 1 provides the format used to tabulate simulation data for each intra-core task scheduling policy.

Table 1: Single Core Simulation Data Table

Simulation data for scheduling policy 1:

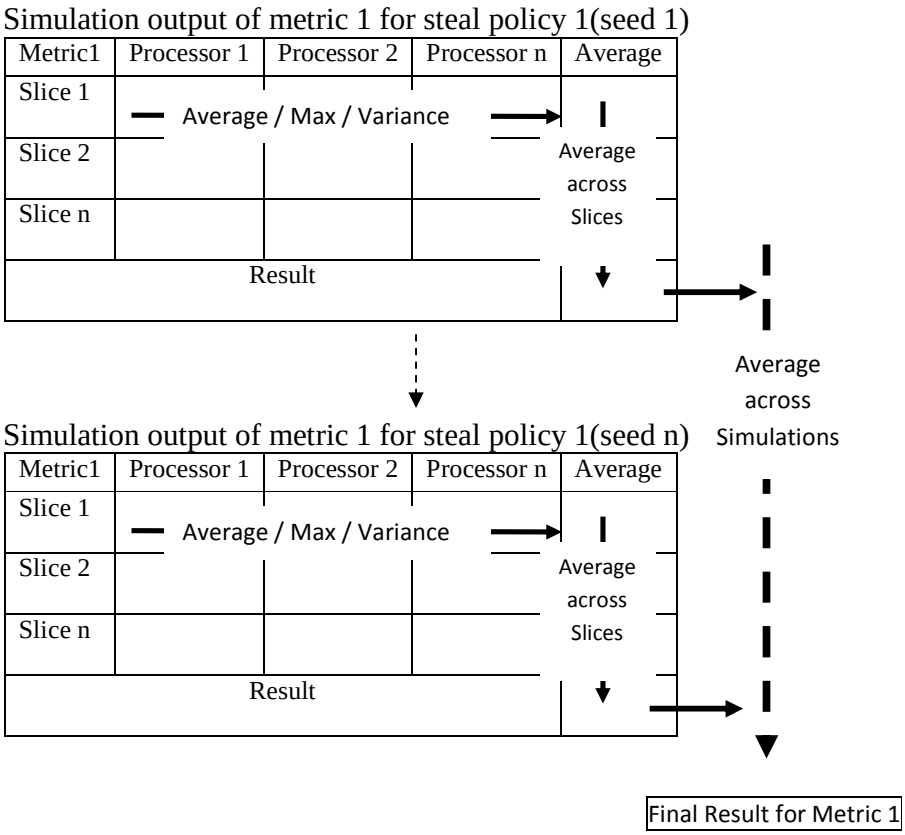
	Metric1	Metric 2	Metric n .....
Seed 1	Average across Simulations	Average across Simulations	Average across Simulations
Seed 2			
Seed n			
Result (Average)	↓	↓	↓

Multi-Core Simulation Data Format

The simulations are performed for all the formulated stealing policies. Each simulation is repeated several times with different random number generation seeds. Every simulation provides performance figures on a slice time basis for all the processors (c.f., Table 2). Data is gathered for each stealing policy as mentioned next. First, for every slice, the output of each performance metric in the simulation is tabulated as an average across all processors. The metric can be tabulated as the variance or maximum across processors as well. Next, the average across slices is determined. Finally, the result of each simulation policy is reported as the average of all the simulations run using different seeds. Similarly, the data is gathered for each stealing policy. The final outcome of an experiment is established by comparing the simulation results of all steal policies.

Table 2 shows the format used to tabulate the data recorded for each performance metric during the simulation.

Table 2: Multi-core Simulation Data Table



Simulator Parameter Configuration

The simulator parameters have been tested with several values to determine a good basic configuration. After careful analysis, a configuration that thoroughly exercises the system functionalities has been selected and shown next.

Table 3 shows the parameter values used for the three main experiments of this study.

Table 3 : Simulator Parameters of Experiments

Experiment Number	1	2	3
No. Of Processors	1	16	16
Fixed Workload Size (No. of tasks)	-	500	1000
Fixed Simulation Time (No. of ticks)	1000	-	-
Average Task Service Time (in ticks)	1	500	400
Average Arrival Rate (per tick)	5	0.5	0.02
Average Task Power (Joules/tick)	5	4	4
Slice Time (No. of ticks)	0.1	100	100
Starved Queue Length (No. of tasks)	-	2	2
Loaded Queue Length (No. of tasks)	-	4	5
Idle Power Consumption Rate (Joules/tick)	-	2	2
Task blocking probability (at each tick)	-	0.01	0.01
Task unblocking probability (at each tick)	-	0.005	0.005

Experiment 4 does not have separate parameters since it analyses a specific phase of the simulation time period in Experiment 3. The details of the experiments are provided in the next section.

Experiments

This section provides details on the four experiments conducted as a part of this study along with information on the simulation results.

Experiment 1: Single Core Task Scheduling

For reasons discussed in the *Intra-core Task Scheduling* step of Chapter IV, the experiments concentrate on preemptive rather than non-preemptive intra-core task scheduling. In this case, the simulations are performed in a single core system for a fixed time-period to compare the intra-core preemptive scheduling policies such as Shortest

Remaining Time First (SRTF), Round Robin (RR), and Round Robin with HECN priority. This experiment is intended to determine the best intra-core task scheduling policy and the most promising policy is to be used as the intra-core scheduling policy in the multi-core experiments. The results from this experiment are provided in the next section. For each intra-core scheduling policy, the total energy consumed, turnaround time, and EDP is presented.

Figure 7 shows the power aware/power agnostic ratio for energy consumed, turnaround time, and EDP metrics for the intra-core task scheduling policies considered. Each metric in Figure 7 is a ratio of the value obtained from the power aware policy over the value obtained from the power agnostic version of the same policy. Thus, a value of less than one indicates that the power aware version is able to improve performance or improve energy consumption.

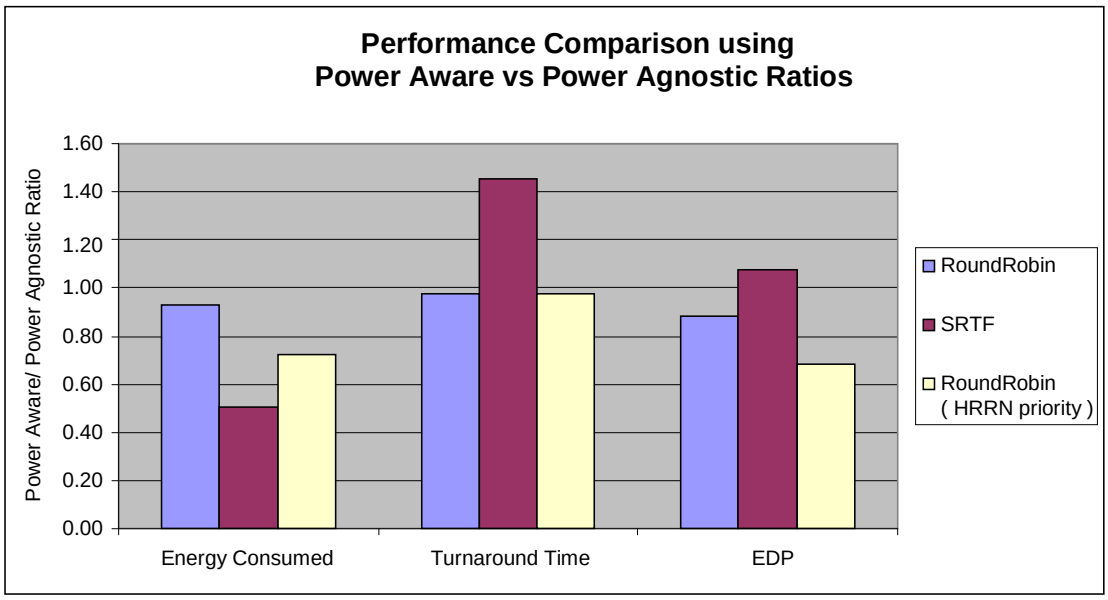


Figure 7: Performance Comparison of single core task scheduling policies

It can be seen that for Round Robin, the ratio of the power consumed under the power aware policy to the power consumed under the power agnostic policy is 0.93. The ratio of the *agnostic* turnaround time to the *aware* turnaround time is 0.97.This implies that there is a minor improvement in power consumption with barely any effect on turnaround time. For SRTF, the energy consumed ratio is 0.51 and turnaround time ratio is 1.46. This demonstrates marked improvement in power consumption; but, with noticeable degradation in the turnaround time. On the other hand, the energy consumed ratio under Round Robin with HECN priority is 0.72 and the turnaround time ratio is 0.97. This shows reasonably good improvement in power consumption with virtually no degradation in the turnaround time. This is further validated with the EDP ratio that, again, shows that Round Robin with HECN has the best improvement in the energy and time metric. The HECN policy considers both the energy demands and the remaining service time in prioritizing tasks for execution. Because of this, it outperforms both the *Round Robin* and *Shortest Remaining Time First* policies. Based on this result, in the simulation experiments discussed next, all the power aware inter-core task stealing policies use HECN policy for intra-core task scheduling and all the power agnostic inter-core task stealing policies use HRRN policy during intra-core task scheduling.

The following three experiments of this section focus on a multi-core system. The next section provides details about figure formats and legend.

Multi-core Experiment Figure Nomenclature

In all the figures of this section, *PAG* denotes power agnostic and *PAW* denotes power aware version of the inter-core task stealing policies. Each figure represents data gathered for a particular performance metric. The *energy consumption variance* is the main performance metric shown in the figures. It is measured in *Joules* since the power unit of a task is Joules/tick as discussed in the *Task Generation Step* in Chapter IV. Every power aware inter-core task stealing policy, shown in the experiment figures, performs the *HECN intra-core scheduling policy* whereas every power agnostic inter-core task stealing policy performs the *HRRN intra-core scheduling policy.* Each policy is denoted by a unique stealing policy and a task scheduling type (PAW or PAG) as shown next with examples.

The high level representation format for a stealing policy is shown below:

< Task Scheduling Type >_<Victim Processor Selection Property >_

<Task Selection Property>

Examples:

(1) PAW_NoSteal => No stealing, only intra-core power aware task scheduling using HECN.

(2) PAG_NoSteal => No stealing, only intra-core power agnostic task scheduling using HRRN.

(3) PAG_MaxEnergyConsumed_MinHRRN_Task => The power agnostic intra-core task scheduling policy uses HRRN. The victim processor is the processor with the *Maximum Energy Consumption* so far. The task with minimum HRRN is stolen.

(4) PAW_MaxEnergyConsumed_MaxEnergyTask => The power aware intra-core task

scheduling policy uses HECN. The victim processor is the processor with the *Maximum Energy Consumption* so far. The task with maximum energy consumption rate is stolen.

(5) PAW_MaxMin_HECN_Task => As the MaxMin steal policy is slightly different, this format implies that the MaxHECN_Task is selected from all the potential MinHECN_Tasks for the power aware version.

(6) PAG_MaxMin_ HRRN_Task => this format implies that the Max HRRN_Task is selected from all the potential Min HRRN_Tasks for the power agnostic version.

The following are the main inferences that can be derived from the experiment figures:

- Comparison between stealing policies identifying the best power aware and best power agnostic policy for a specific metric.
- Comparison between the power aware version (displayed in the first half of each figure) and the power agnostic version (displayed in the second half of each figure) of each policy intended to derive the effect of considering power in each policy.

Experiment 2: Multi-core Task Scheduling for a Parallel Workload Scenario

In this experiment, a fixed workload simulation is performed in a system having a fast task arrival rate. This simulates a *parallel workload* scenario described in Chapter IV. This experiment is intended to study the behavior of the formulated policies and identify the policy that performs the best under this specific scenario. The four main performance figures provided from this experiment are the energy consumption variance, the average

turnaround time, the peak ready-queue length, and the completion time of all the policies. The parallel workload scenario is explained with an example in Figure 8.

Figure 8 shows the processors' average ready queue length in one instance of the simulation.

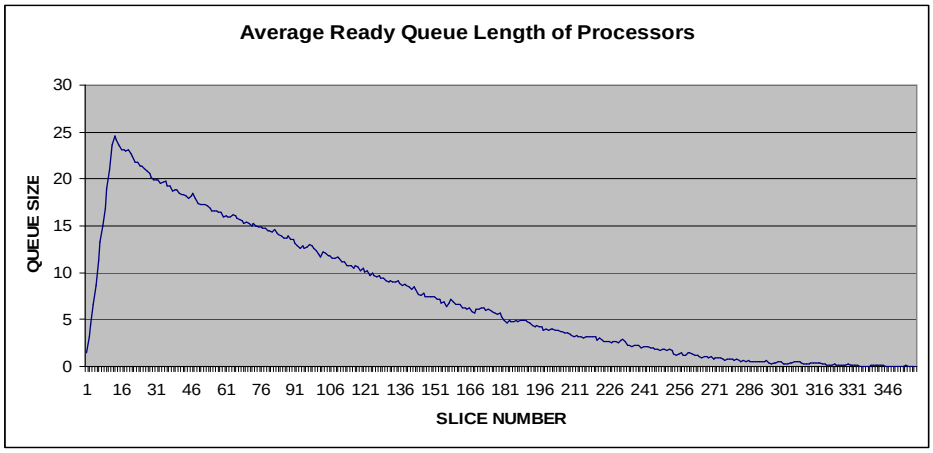


Figure 8: Average Ready Queue Length of Experiment 2

According to the figure, the ready queue length is rapidly increasing in the first few time slices of the simulation and then gradually decreasing as the simulation progresses thereby emulating a *parallel workload scenario*.

Figure 9 shows the processors' energy consumption variance. This is used as an indicator of load balancing.

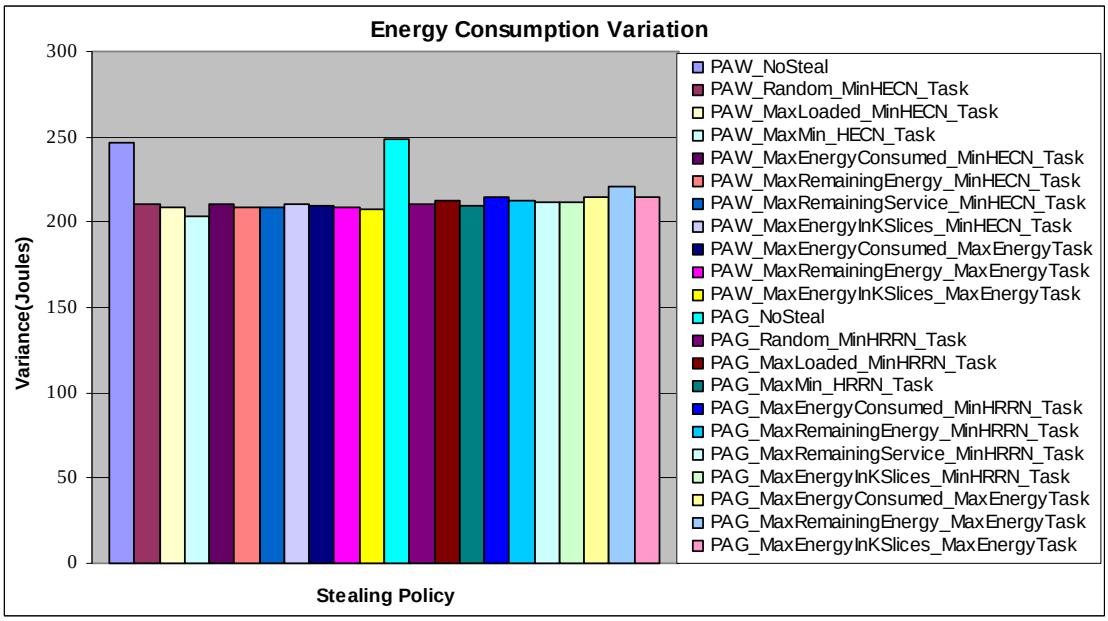


Figure 9: Energy Consumption Variance of Experiment 2

Here, work stealing provides a reduction of about 18% in variance compared to PAG_No Steal policy. The PAW_MaxMin_HECN_Task is the best stealing policy. The power aware policies provide a marginally better power performance than the power agnostic method.

Figure 10 displays the turnaround time.

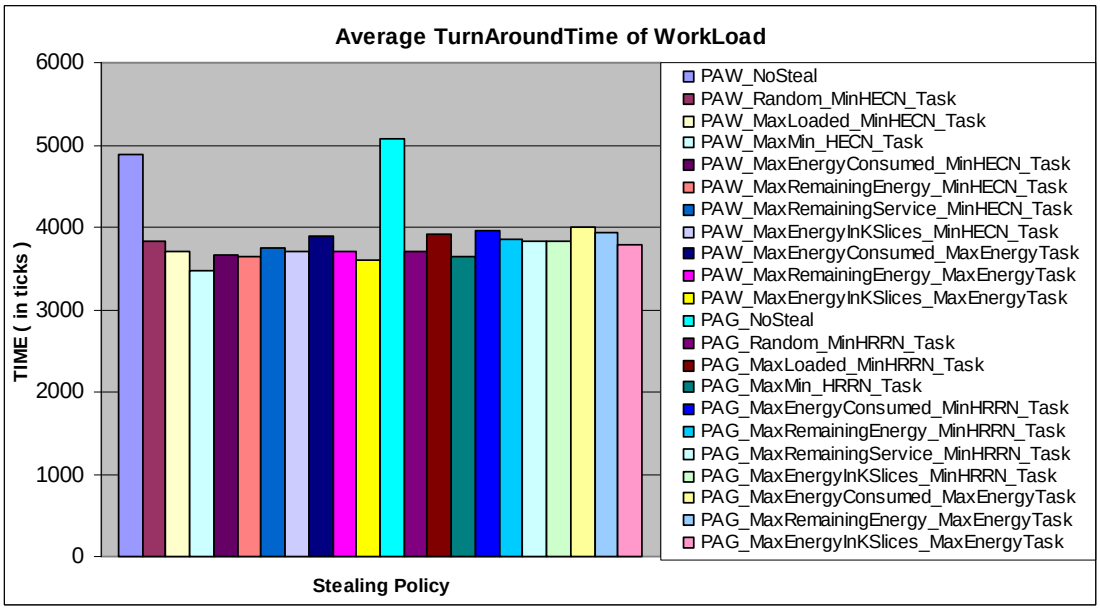


Figure 10: Average Turnaround Time of Experiment 2

In this case, the PAW_NoSteal policy has a lower turnaround time than PAG_NoSteal policy. This implies that power aware intra-core task scheduling, without any stealing, lowers turnaround time by about 4%. By including stealing, the PAW_MaxMin_ HECN_Task is the best stealing policy and it improves (reduces) turnaround time further by approx 31% compared to PAG_NoSteal policy. This shows that in the process of trying to gain power efficiency, time factor is improved as well. This can be due to the fact that the EDP metric used in the selection criteria considers time along with power attributes. Again, power aware is slightly better than power agnostic.

Figure 11 presents the peak ready-queue length.

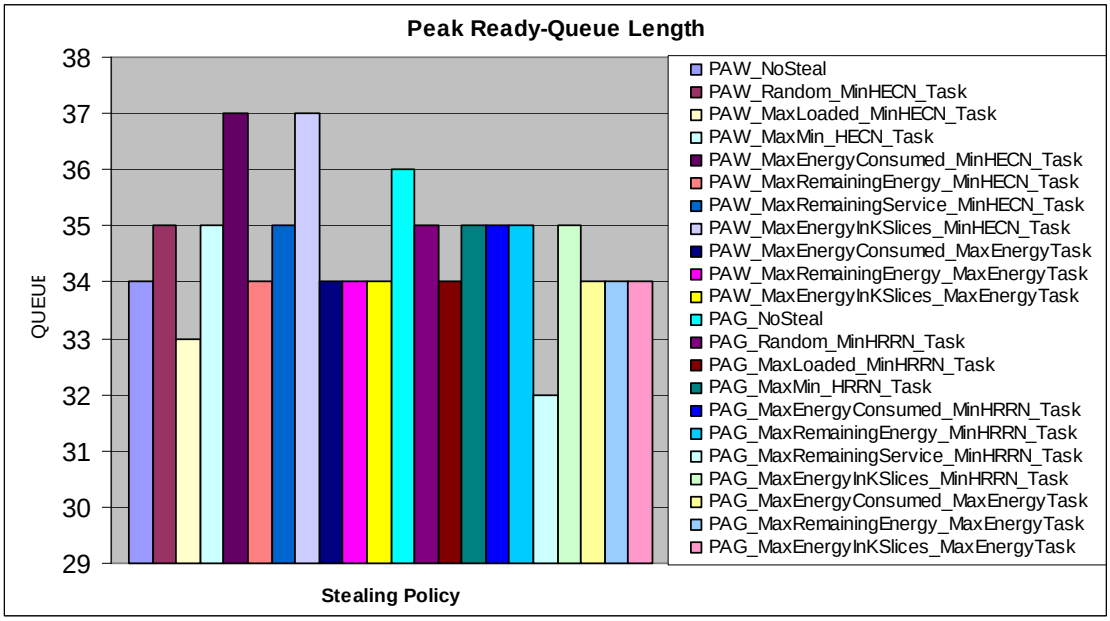


Figure 11: Peak ready-queue length of Experiment 2

Here, again, the PAW_NoSteal policy performs better than PAG_NoSteal policy by lowering the peak ready-queue length by almost 6%. With stealing introduced, the PAW_MaxLoaded_MinHECN_Task policy is the best as it targets stealing from processors with large queues. This policy further reduces the peak queue length by about 8% compared to PAG_NoSteal policy. The PAG_MaxRemainingService_Min HRRN_Task policy is just marginally better than the PAW_MaxLoaded_Min HECN_Task policy.

Figure 12 shows the completion time of the entire workload.

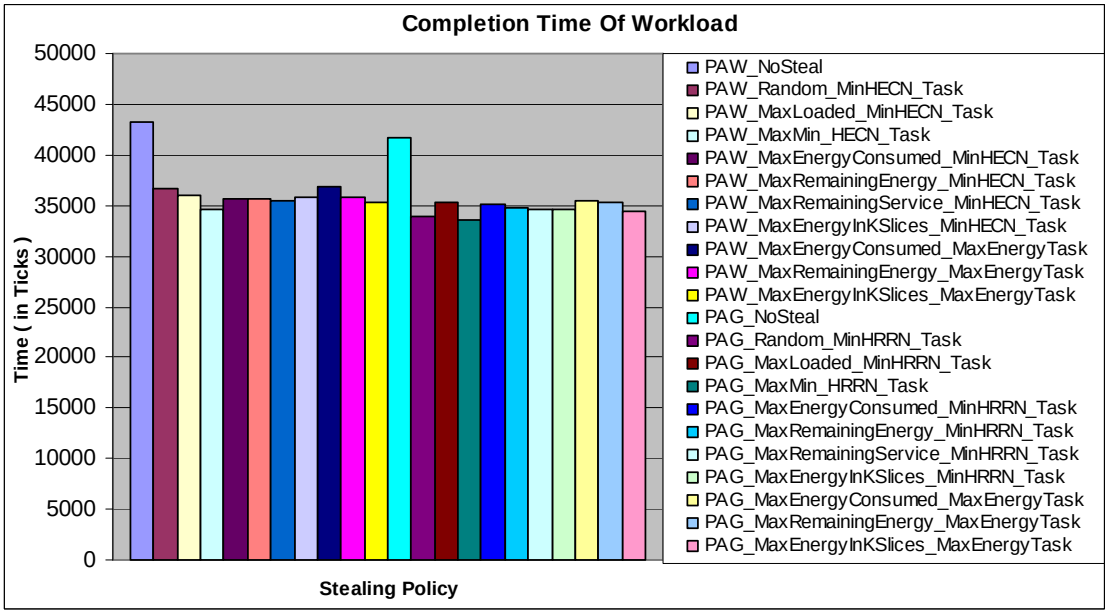


Figure 12: Completion Time of Experiment 2

In this case, PAW_NoSteal policy increases the total completion by about 3.5%. This can be attributed to the fact that power aware scheduling may increase task wait time and there is no stealing to help reduce wait time. On the other hand, stealing significantly reduces the completion time with PAW_MaxMin_ HECN_Task policy being the best stealer as it reduces the completion time by about 17% compared to PAG_NoSteal Policy. The PAG_MaxMin_ HRRN_Task policy is very slightly better than PAW_MaxMin_ HECN_Task policy with just around 3% more reduction. But since PAG_MaxMin_ HRRN_Task policy does not perform as well with regard to the energy consumption variance and turnaround time metrics seen earlier, it is not regarded highly.

From all of the results of this experiment, it can be seen that the PAW_MaxMin_ HECN_Task is the best stealing policy for a fast task arrival rate scenario. It significantly improves three important metrics, namely, energy consumption variance, turnaround time, and completion time.

Experiment 3: Multi-core Task Scheduling for a Steady State Workload Scenario

For this test, a fixed workload simulation is performed in a system having a slow task arrival rate. This emulates a *steady state workload* scenario as described in Chapter IV. This experiment, in similarity to the previous experiment, aims to study the performance of policies until the entire workload is completed but this time, a system with a steady stream of incoming tasks is considered. Figures representing the performance of the policies with regard to the processor's energy consumption variance, the average turnaround time, the peak ready-queue length, and the completion time are provided. The steady state workload scenario is illustrated with a sample case in Figure 13.

Figure 13 shows the processors' average ready queue length in one instance of the simulation.

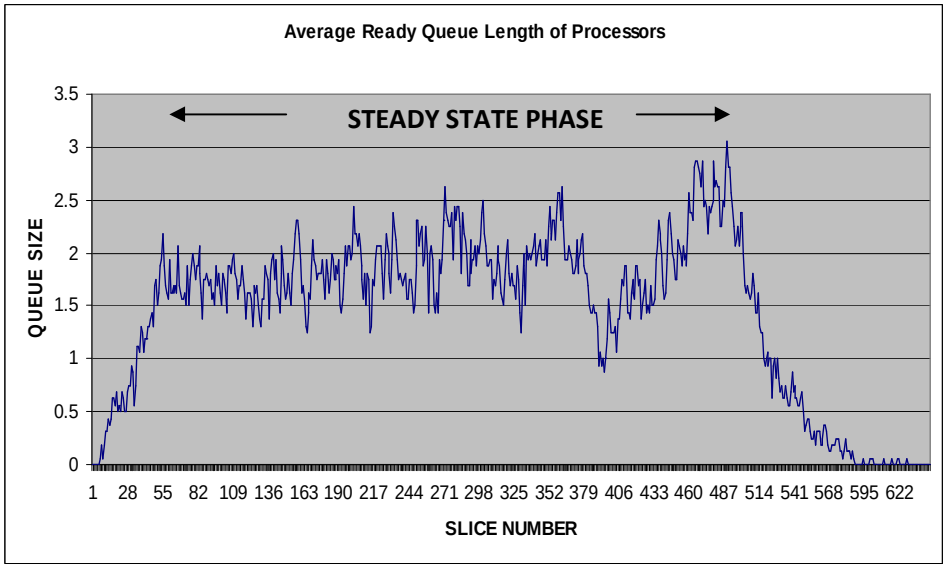


Figure 13: Average Ready Queue Length of Experiment 3

In the first few time slices of the simulation, the ready queue length gradually increases. Then as the simulation progresses, the queue length remains steady for several slices thereby simulating a s*teady state workload scenario*.

Figure 14 shows the processors' energy consumption variance.

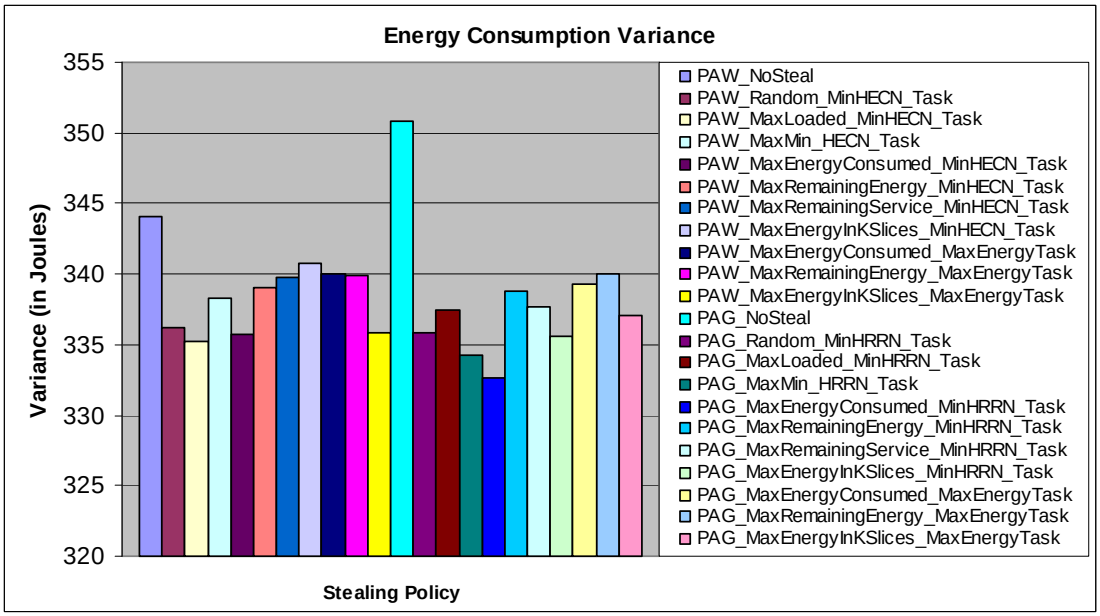


Figure 14: Energy Consumption Variance of Experiment 3

It is noticed that PAW_NoSteal policy performs slightly better than PAG_NoSteal policy by lowering the energy consumption variance by about 2%. By including stealing, the PAW_MaxEnergyInKSlices_MaxEnergyTask is seen as the best power aware stealing policy. This policy further reduces the variance by 5% compared to PAG_NoSteal policy. The PAG_MaxEnergyConsumed_MinHRRN_Task stealing policy provides a marginally better power performance than the PAW_MaxEnergyInKSlices_MaxEnergyTask method but it is not considered significant since it does not perform as well for the turnaround time metric seen next.

Figure 15 displays the turnaround time.

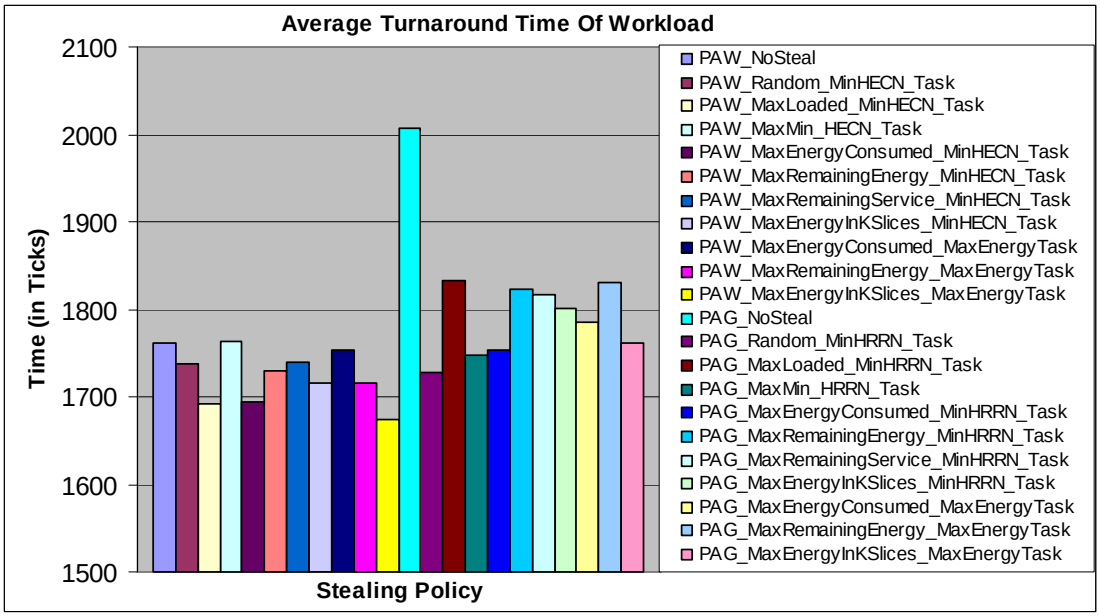


Figure 15: Average Turnaround Time of Experiment 3

Again, PAW_NoSteal policy is better than PAG_NoSteal policy by almost 13%. The power aware intra-core task scheduling coupled with inter-core task stealing further improves turnaround time. PAW_MaxEnergyInKSlices_MaxEnergyTask is again the best stealing policy with approx 17% lower turnaround time compared to PAG_NoSteal policy. The power aware policies are noticeably better than the power agnostic policies.

Figure 16 presents the peak ready-queue length.

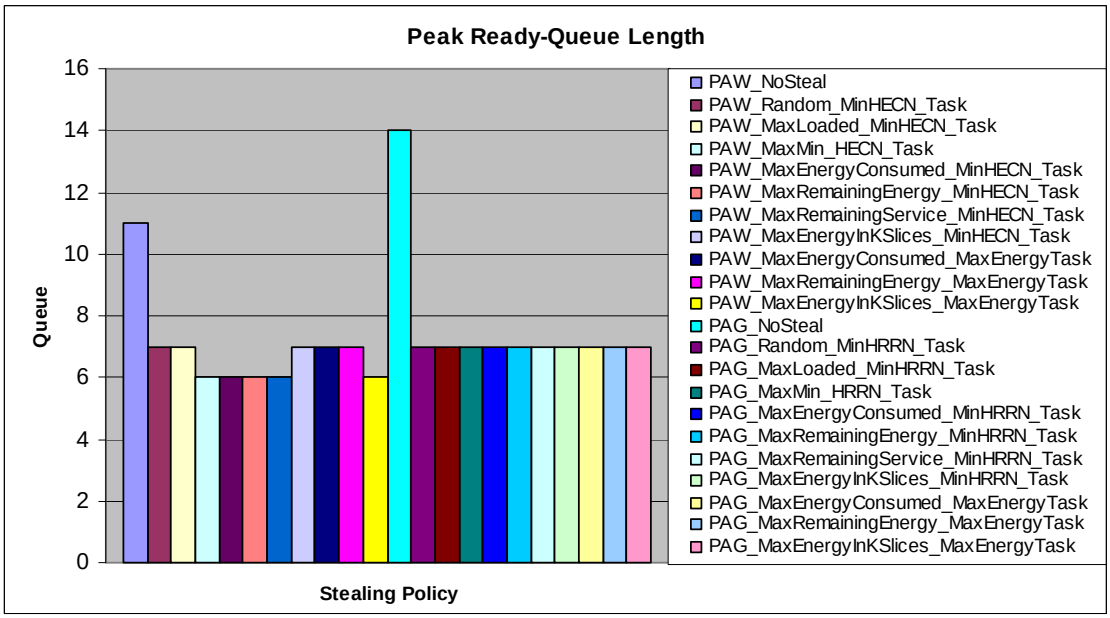


Figure 16: Peak ready-queue length of Experiment 3

Continuing the trend of this experiment, the PAW_NoSteal policy lowers the peak ready-queue length by 21% compared to PAG_NoSteal policy. With stealing included, PAW_MaxEnergyInKSlices_MaxEnergyTask remains one of the best stealing policies as it significantly reduces the peak queue length further by approx 57% compared to PAG_NoSteal policy. Here power aware is marginally better than power agnostic.

Figure 17 shows the completion time of the entire workload.

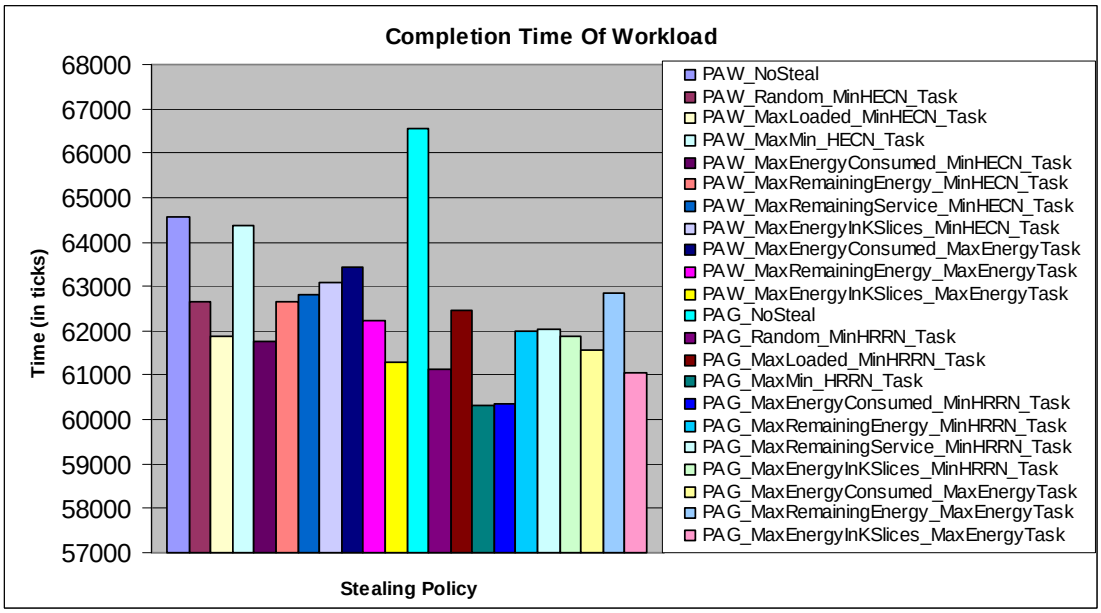


Figure 17: Completion Time of Experiment 3

In this case, an important difference can be noted compared to the previous experiment 2. The PAW_NoSteal policy is better than PAG_NoSteal policy with a 3% lower completion time but the opposite is true in experiment 2 which involves larger ready queue sizes. This shows that a power aware intra-core scheduling policy provides a better completion time for a *small ready queue size* scenario but a power agnostic intra-core scheduling policy is better suited for a *large ready queue size* scenario.

Furthermore, the best power aware stealer of this experiment is again the PAW_MaxEnergyInKSlices_MaxEnergyTask policy with about 8% reduction in completion time compared to PAG_NoSteal policy. The PAG_MaxMin_HRRN_Task policy shows slightly better completion time compared to the PAW_MaxEnergyInKSlices_MaxEnergyTask policy but it cannot be appreciated since it

fails to be the best in terms of efficiency in the energy consumption variance and turnaround time metrics.

Experiment 4: Multi-core Task Scheduling for Fixed Time

This experiment is a special case study of the previous experiment. The main intent is to analyze the performance of the policies during the *steady state phase* of experiment 3 and determine if power efficiency is better achieved by the scheduling policies during this particular phase. The following figures, derived from this experiment, show the processor's energy consumption variance, the average turnaround time, and average wait time of tasks.

Figure 18 shows the processors' energy consumption variance.

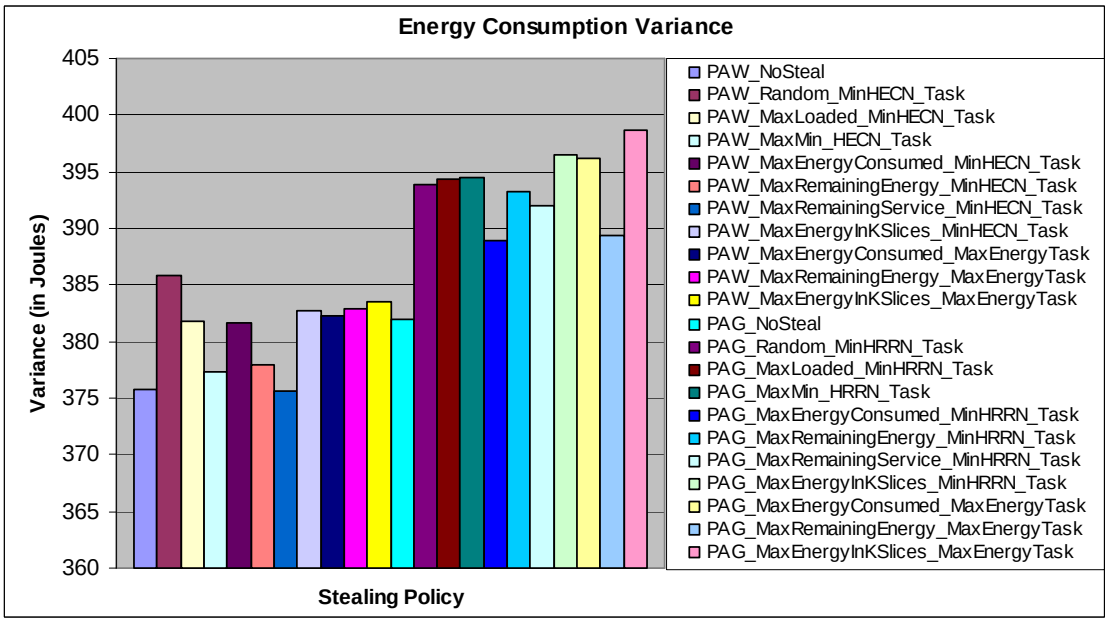


Figure 18: Energy Consumption Variance of Experiment 4

According to the graph, the PAW_NoSteal policy is the best policy. It reduces the energy consumption variance by about 2% compared to the PAG_NoSteal policy. This implies that a simple power aware intra-core scheduling policy with no stealing can reduce the energy consumption variance. But it important to note that this occurs during the steady state phase of the entire simulation implying that all processors keep selecting the low energy consuming tasks first and keep the high energy consuming tasks waiting thereby reducing the overall variance in energy consumption during this period. Consequently, as seen, the stealing tends to increase the variance since more high energy tasks are now available in ready queues for stealing.

Figure 19 displays the turnaround time.

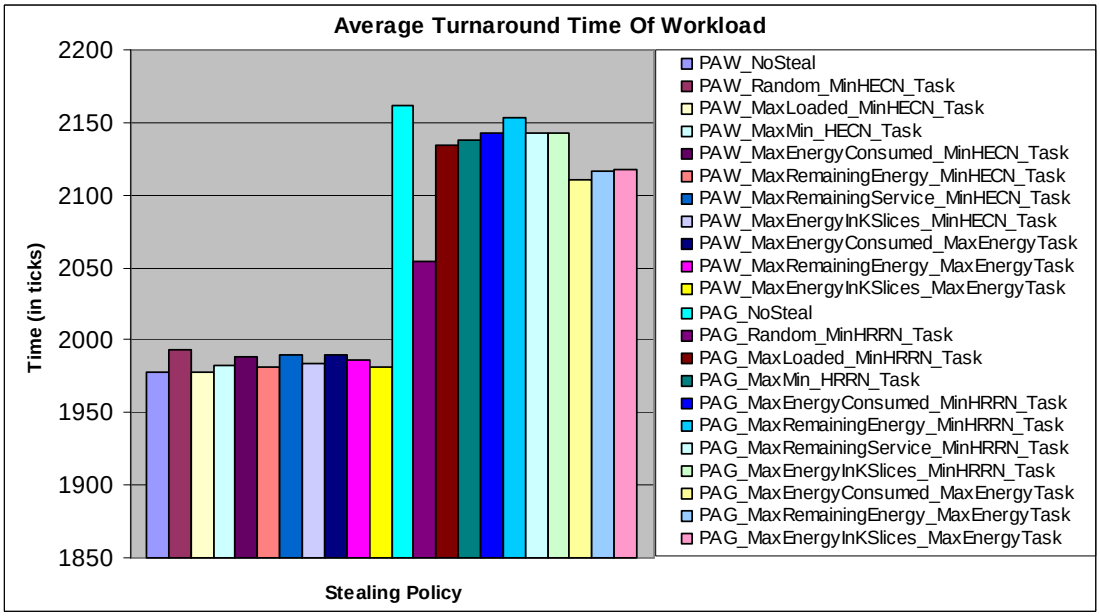


Figure 19: Average Turnaround Time of Experiment 4

It is observed that the PAW_NoSteal is again the best policy. This policy reduces the turnaround time by about 9% compared to the PAG_NoSteal policy. The processor's task selection criterion gives tasks with low execution time and/or power more priority thereby the completed tasks have lower turnaround time. But since this is only during the steady phase, it is important to note that while the low power short tasks may be getting more priority, the lower priority tasks continue to wait causing a high waiting time in the ready queue. This reasoning is validated in the next figure.

Figure 20 presents the Ready-Queue Average Wait Time.

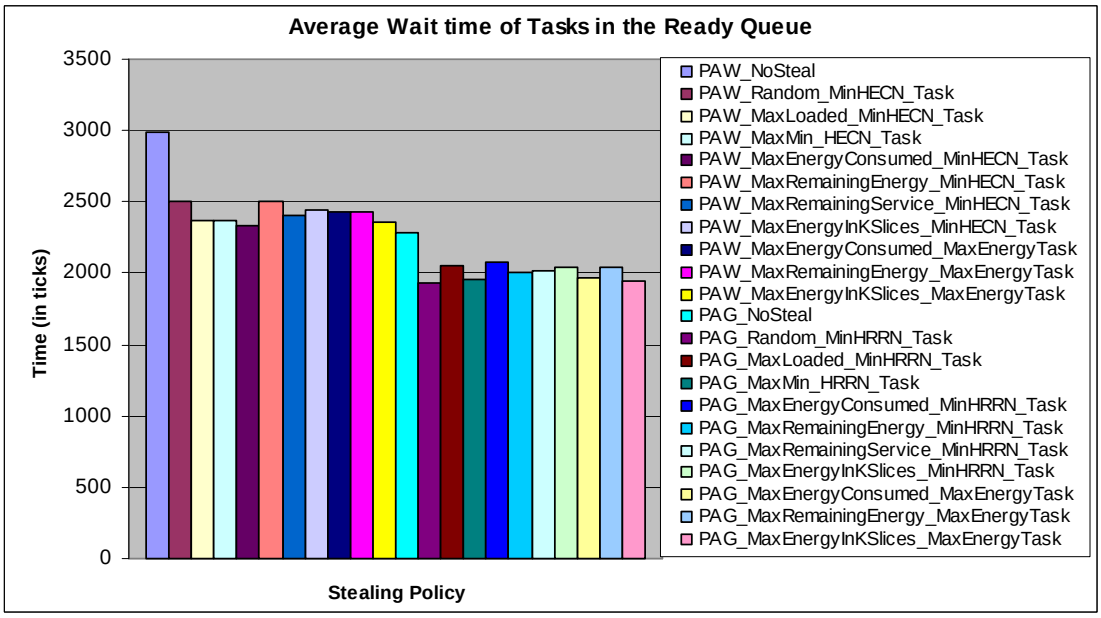


Figure 20: Ready-Queue Average Wait Time of Experiment 4

As expected, this graph illustrates that the PAW_NoSteal policy increases the ready queue wait time of tasks by about 21% compared to PAG_NoSteal policy.

From this experiment result it is seen that for a scenario with tasks arriving in a steady continuous pattern, introducing stealing during the steady phase does not impact the energy consumption variance much. It is observed that the PAW_NoSteal policy is more efficient during the steady phase while the PAW_MaxEnergyInKSlices_MaxEnergyTask policy improves energy consumption variance during the transient phase of the simulation.

VI. RESULT EVALUATION

The previous section provided the performance results of all the individual experiments conducted for this study. This section analyses the combined results of all the experiments to identify the overall effect of introducing power aware intra-core task scheduling and inter-core stealing in a multi-core system. The behavior of the devised intra-core task scheduling and inter-core task stealing policies has been studied under different workload scenarios.

The four types of experiments that have been conducted are:

1. Single core task scheduling simulations for a fast task arrival rate system.
2. Multi-core task scheduling simulations for a fast task arrival rate system.
3. Multi-core task scheduling simulations for a slow task arrival rate system.
4. Multi-core task scheduling analysis during the steady state phase of a simulation.

Table 4 provides the summary of the experiment results.

Table 4: Summary of Experiments' Results

	Experiment Type	Best Scheduling Policy	% Reduction Compared to PAG_NoSteal		
			Energy Consumed/ Variance	Turnaround Time	Completion Time
1	Single Core, Fixed Time	HECN	28 %	3%	N/A
2	Multi-Core, Fixed Workload, Fast Task Arrival	PAW_MaxMin_HECN _Task	18%	31%	17%
3	Multi-core, Fixed Workload, Slow Task Arrival	PAW_MaxEnergyInK Slices_MaxEnergyTask	5%	17%	8%
4	Multi-Core, Fixed Time, Slow Task Arrival	PAW_NoSteal	2%	9%	N/A

The classical HRRN intra-core task scheduling policy has been utilized as the basis of all newly formulated power efficient policies. The power aware HECN intra-core task scheduling policy has been devised by extending this basic HRRN policy. Furthermore, inter-core task stealing policies have been introduced by incorporating power characteristics. Hence, the *power agnostic HRRN policy with no stealing*, also referred to as PAG_NoSteal is used as the base policy to compare the performance of all the other policies.

According to the data shown in Table 4, in every experiment, a *power aware* policy emerges as the policy that successfully reduces energy consumption variance, turnaround time and completion time concurrently. In addition to accomplishing energy efficiency, the performance time has been improved as well. The *PAW_MaxMin_ HECN_Task* policy shows the highest potential with 18% reduction in energy consumption variance,

31% improvement in turnaround time, and 17% more efficiency in completion time compared to the PAG_NoSteal policy.

Furthermore, the key points noted from the combined results of all the experiments are listed next.

1. The *HECN* intra-core scheduling policy outperforms *Round Robin* and *Shortest Remaining Time First* policies in Experiment 1. This policy provides the best power efficiency and turnaround time. This can be attributed to the fact that the HECN cost function gives priority to tasks with low power and service time and also ensures low priority tasks are not waiting for long.

2. The *PAW_MaxMin_ HECN_Task* policy emerges as the best policy in Experiment 2. The reason for this might be because the *MaxMin* policy is the only policy that directly selects a task to steal by choosing the least power consuming task among the high power consuming tasks of all potential victim processors. All the other stealing policies first select a potential victim processor and then select a task from that chosen processor. Therefore, for scenarios that have the system flooded with tasks, the ready queues of processors are large, and the policies have to choose from a large set of tasks, the stealing policy that considers all the tasks in the system such as the *MaxMin* policy outperforms other policies.

3. The *PAW_MaxEnergyConsumedInKSlices_MaxEnergyTask* policy is the most efficient policy in Experiment 3. This can be best explained by the following analysis. Excluding the *MaxMin* policy, all the stealing policies first consider the power properties related to a processor to determine a victim. Most properties are related to the number of

tasks (like the PAW_*MaxLoaded_MinHECN_Task* policy) or type of tasks in the ready queue (like the PAW_*MaxRemainingEnergy_MinHECN_Task* policy) but only two of the policies consider the past history of the processor, namely, the *MaxEnergyConsumedInKSlices* policy which uses recent past data and the *MaxEnergyConsumed* policy which uses all the past data. If there is a steady continuous stream of incoming tasks in the system then the ready queues are reasonably small. Hence, if the selection policy has to choose from a small set of tasks in the queue, the policy that considers properties related to the recent past history of potential victim processors rather than the processor's tasks is most promising like the *MaxEnergyConsumedInKSlices_MaxEnergyTask* policy.

4. The *PAW_NoSteal* policy which is simply the *HECN intra-core task scheduling policy* outperforms all other policies during the steady phase of the simulation in Experiment 4. This policy schedules the low power consuming and low service time tasks first to ensure the power consumption variance across processors is reasonably low but pays the price by making low priority tasks wait for increased amounts of time. For situations that consider the power consumption level and variance of a multi-core system to be more critical then the waiting time of tasks, the intra-core power aware *HECN* policy with no stealing shows good potential for power efficiency during the steady phase of the simulation whereas task stealing performs well during the transient phase.

5. Based on the experiment results, the *PAW_MaxMin_HECN_Task* procedure is the policy with the most potential for power efficiency even if the task arrival rate is unknown. From experiments 2 and 3, it is observed that this policy performs the best for

cases with fast task arrival rate and also performs reasonable well in situations with steady task arrival rate.

6. In all the experiments, there is no significant difference in performance amongst many of the stealing policies. This could be attributed to the fact that the variations in work stealing are very minute and have subtle differences as explained in the following examples:

(i) The *MaxLoaded*, the *MaxRemainingService*, and the *MaxRemainingEnergy* policies can pick the same victim processor because a processor with a huge queue is most likely the one with the most remaining energy or service as well.

(ii)The *MaxEnergyConsumed_MinHECN_Task* and the *MaxEnergyConsumed_MaxEnergyTask* policy pick the same victim processor since the policy is the same in that regard. However, the former steals a task with lowest HECN but the latter steals a task with the maximum power consuming rate. If the wait times of the task are almost the same like in experiment 2, where all tasks arrive nearly at the same time, then the task with lowest HECN task is most likely the task with the maximum power consuming rate since lower the HECN cost function, the higher the power of the task and vice-versa.

7. The turnaround time is improved much more than the power efficiency level in all the experiments. This implies that the *Energy-Delay-Product (EDP)* metric integrated into the *HECN* policy might be giving more consideration to the *task time* rather than the *task power* attribute as seen below.

$$\text{Energy Delay Product (EDP)} = s \times s \times P; \text{ where } s = \text{service time of a task,}$$
$$P = \text{power consumption rate of task.}$$

In the EDP formula, clearly the *service time* of a task is used more than the power of a task thereby giving more importance to *task service time* than task power.

Having reviewed all the results generated from the four experiments, it is evident that the energy consumption variance, a key indicator of load balancing has been improved in every experiment by one of the devised power aware policies. In addition to accomplishing energy efficiency, the performance time has been improved as well. This validates the initial hypothesis that it is possible to devise power aware task scheduling policies by incorporating power characteristics such that energy efficiency and performance time is improved. It is observed that the improvement in energy efficiency level varies from marginal to significant depending on the experiment scenario. However, this research has been successful in devising power management techniques at the OS scheduling level and opening prospective avenues for further advancement.

VII. CONCLUSION

The growing concern of the semi-conductor industry with regard to efficient power management within the processor chips is addressed at the OS level via power aware OS intra-core task scheduling and inter-core task stealing. The primary goal of this research work is to develop power aware intra-core task scheduling and inter-core task stealing policies. In an attempt to achieve the desired goal, the following steps have been implemented.

First, three classical intra-core task scheduling policies, namely, Round Robin(RR), Shortest Remaining Time First (SRTF) and Highest Response Ratio Next (HRRN) have been considered. This leads to the creation of a power aware intra-core task scheduling policy referred to as HECN that extends the HRRN policy to include power characteristics of tasks in the system.

Next, building on the new intra-core HECN policy, various inter-core work stealing policies have been explored. Several different power aware variations of work stealing have been formulated that consider power features of the processors and its tasks before identifying the task to steal. Finally, an in-house simulator has been developed solely to evaluate the potential of the policies devised. Single core simulations have been

conducted to determine the viability of HECN with respect to power management. With this result being positive, further extensive multi-core experiments have been performed to study the effect of coupling power aware intra-core task scheduling (HECN) with power aware inter-core task stealing. The outcome suggests that the *PAW_MaxMin_HECN_Task* procedure is the most promising policy that attains power efficiency and manages minimal effect to performance.

The main conclusion drawn from this research is that there has been success in identifying potential OS based power management methods and provoking further study into OS level power management techniques. The next section provides recommendations for future work.

Recommendation for Future Work

The following are proposals for future work related to this study.

1. In the intra-core power aware HECN policy, the HECN derivative to determine task priority is calculated as

   $$HECN\ \_\ PRIORITY\ = \frac{(w^{\alpha} + (s^{\beta} \times p^{\gamma})]}{(s^{\beta} \times p^{\gamma})}$$ where variables w = task wait time, s= task service time, and p = task power consumption rate. The constants $\alpha = 1$, $\beta = 2$, and $\gamma = 1$. The values for α, β, and γ can be tested with several combinations of values to vary the importance of the task properties such as task service time, wait time of tasks, and power consumption rate of tasks.

2. The experiments can be conducted using a test workbench with realistic task data such as task execution time and power consumption rate of a task. The power aware policies can then make task selections based on actual task information.
3. The work stealing policy can consider an affinity model. The central unit can be extended to have knowledge of task's affinity to a processor and a stealing policy can be devised based on the task's affinity information.
4. Several inter-core scheduling policies require global system knowledge such as the most energy consuming processor and power consumption level of the system. Such parameters of power and performance available at the hardware/firmware level can be exposed to the operating system. They can be utized by the simulator by using vendor boards. This can enable more realistic intra-core task scheduling and inter-core task stealing, and possibly further improve power/performance.
5. The experiments can explore DVFS based scheduling within a simulated environment.
6. The experiments can consider shutting down idle processors by interacting with the firmware.

VIII. REFERENCES

[1] A.Silberschatz, P.B. Galvin and G. Gagne."CPU Scheduling," in Operating System Concepts,8th ed.,John Wiley and Sons, 2008, pp. 183-223.

[2] D. Tam, R.Azimi and M.Stumm. "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007, pp. 47-58.

[3] S. Boyd-Wickizer, M.F. Kaashoek and R. Morris. "Reinventing scheduling for multicore systems," in Proceedings of the 12th conference on Hot topics in Operating Systems, 2009, pp. 21-21.

[4] M. Rajagopalan, B.T. Lewis and T.A. Anderson."Thread scheduling for multi-core platforms," in Proceedings of the 11th USENIX workshop on Hot topics in operating systems, 2007.

[5] S. Chen, P.B. Gibbons, M.Kozuch et al. "Scheduling threads for constructive cache sharing on CMPs" in Proceedings of the nineteenth annual ACM Symposium on Parallel Algorithms and Architectures, 2007, pp. 105-115.

[6] A. Merkel and F. Bellosa. "Balancing power consumption in multiprocessor systems," in Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, 2006, pp. 403-414.

[7] A.K. Coskun, R. Strong, D.M. Tullsen and T.S. Rosing. "Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors," in Proceedings of the eleventh international joint conference on Measurement and Modeling of Computer Systems, 2009, pp. 169-180.

[8] J. Donald and M. Martonosi. "Techniques for Multicore Thermal Management: Classification and New Exploration," in Proceedings of the 33rd International Symposium on Computer Architecture, 2006, pp. 78-88.

[9] M. Kashif, T. Helmy and E. El-Sebakhy. "A Priority-Based MLFQ Scheduler for CPU Power Saving," in Proceedings of the IEEE International Conference on Computer Systems and Applications, 2006, pp. 130-134.

[10] K. H. Kim, R. Buyya, and J. Kim, “Power aware scheduling of bag-oftasks applications with deadline constraints on dvs-enabled clusters,” in Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, ser. CCGRID ’07, 2007.

[11] G. Wu, Z. Xu, Q. Xia, J. Ren and F. Xia. “Task Allocation and Migration Algorithm for Temperature-constrained Real-time Multi-Core Systems,” in Proceedings of the IEEE International Conference on Cyber,Physical and Social computing, 2010, pp. 189-196.

[12] X. Zhou, J. Yang, M. Chrobak and Y. Zhang. ”Performance-Aware Thermal Management via Task Scheduling.”The Journal of ACM Transactions on Architecture and Code Optimization, vol. 7 issue 1, April. 2010.

[13] J. Quintin and F. Wagner. “Hierarchical Work-Stealing,” in EuroPar'10 Proceedings of the 16th International Euro-Par Conference on Parallel Processing, 2010, pp. 217-229.

[14] Y. Guo, J. Zhao, V. Cave and V. Sarkar. “SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler,” in Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2010, pp. 1-12.

[15] S. Agarwal, G.K. Mehta and Y. Li. “Performance- based Scheduling with Work Stealing.” Internet: http://www.cs.ucsb.edu/~gaurav_mehta/reports/cs290b.pdf, 2009 [Aug, 2011].

[16] D. Sudarshan and D. Pooja. “LIBRA:Client Initiated Algorithm for Load Balancing Using Work Stealing Mechanism,” in Proceedings of 2nd International Conference on Emerging Trends in Engineering and Technology, 2009, pp. 636-638.

[17] A. Robison, M, Voss and A. Kukanov. “Optimization via Reflection on Work Stealing in TBB,” in Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1-8.

[18] K. Faxén and J. Ardelius. “Manycore Work Stealing,”in Proceedings of the 8th ACM International Conference on Computing Frontiers ACM, 2011.

[19] Y. Guo, R. Barik, R. Raman, and V. Sarkar. “Work-first and helpfirst scheduling policies for async-finish task parallelism,” in IPDPS ’09: Proceedings of the 2009 IEEE International Symposium on Parallel& Distributed Processing. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[20] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. “Deadlock-free scheduling of x10 computations with bounded resources,” in SPAA ’07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures. New York, NY, USA: ACM, 2007, pp. 229–240.

VITA

Shwetha Shankar was born in Mumbai, India, on October 15, 1982, the daughter of Shubha Shankar and K Shankar. After completing her high school education at Clarence High School, Bangalore, India, in 2000, she was admitted to R.V College of Engineering, Bangalore, India. She received her Bachelor of Engineering degree in Industrial Engineering and Management in June, 2004. She was first employed as a Software Engineer and later as a Programmer Analyst and Technology Lead in Infosys Technologies Ltd, Bangalore, India, from July 2004 to March 2010 in the Banking and Capital Markets Group, where she was involved in all phases of financial software development from requirement analysis to software development and production support. In June 2010, she was admitted to Master's Program in Computer Science at Texas State University-San Marcos. She is currently a research student in the Computer Science Department.

Permanent Email: shwethashankar15@gmail.com

This thesis was typed by Shwetha Shankar.