Palestine Polytechnic University
Deanship of Graduate Studies and Scientific Research

Performance-Power Enhancement on High-Scale Heterogeneous Multi Processors (HMP) using Genetic Algorithms (GA)

Submitted By:
Ahmad N. M. Aljabari

Thesis submitted in partial fulfillment of requirements of the degree
Master of Science in Informatics
June, 2012

The undersigned hereby certify that they have read and recommend to the Deanship of Graduate Studies and Scientific Research at Palestine Polytechnic University the acceptance of a thesis entitled: **Performance-Power Enhancement on High-Scale Heterogeneous Multi Processors (HMP) using Genetic Algorithms (GA)** submitted by **Ahmad N Aljabari**, in partial fulfillment of the requirements for the degree of Master in Informatics.

Graduate Advisory Committee:

Committee Chair Name (Supervisor), University:

Dr. Mohammed Aldasht, Palestine Polytechnic University

Signature:[illegible] Date: 3/7/2012

Committee Member Name , University:

Dr. Hashem Tamimi, Palestine Polytechnic University

Signature:[illegible] Date: 3/7/2012

Committee Member Name, University:

Dr. Labib Arafeh, Alquds University

Signature:[illegible] Date: 4/7/2012

Approved for the Deanship:

Dean of Graduate Studies and Scientific Research - Palestine Polytechnic University

Signature:[illegible] Date: 4-7-2012

Prof. Dr. Karim Tahboub

Palestine Polytechnic University
جامعة بوليتكنك فلسطين

نبـــذة الأطروحـــــة

عملية المقايضة بين مقدار الطاقة المستنفذة وجودة الأداء تعتبر من القضايا المهمة والحرجة في أنظمة المعالجة والتي تضم عدة كبير من المعالجات الغير متجانسة. و خصوصا في الأنظمة الحديثة والمتنقلة التي تضم عدد كبير من المعالجات. ومن الجدير بالذكر أن عملية الجدولة على أنظمة تضم عدد من المعالجات الغير متجانسة ما زالت غير مكتملة وقيد البحث (NP–complete problem). وهذا يعني أن أفضل قيمة للطاقة المستنفذة و وقت التنفيذ لم يتم تمثيله بخوارزمية على شكل معادلات رياضية. و يجدر الذكر هنا إلى أن الأنظمة التي تضم عدد من المعالجات الغير متجانسة تحتاج إلى نظريات وخوارزميات جدولة معقدة تمكننا من الحصول على أداء جيد لهذه الأنظمة عند إسناد أي عمل عليها.

وفي هذه الأطروحة تم تقديم منهجية يعتمد عليها لاكتشاف تركيبة جيدة من المعالجات الغير متجانسة في أنظمة معالجة تضم عدد كبير جدا من المعالجات الغير متجانسة بهدف الحصول على مقايضة مناسبة بين الطاقة والأداء. المنهجية المقدمة تطبق خوارزمية الجينات لاستكشاف تركيبة جيدة من المعالجات في مجال بحث كبير ومعقد لتنفيذ أي مهمة تسند لهذا النظام.

النتائج التي تم التوصل إليها من خلال التطبيقات والتجارب تظهر الأهداف المراد تحقيقها، بالإضافة إلى عمل تقدير جيد للوقت الذي يحتاجه النظام لتنفيذ مهمة كبيرة على نطاق واسع جدا من المعالجات قبل البدء بعملة التنفيذ الفعلي. وفقا لذلك تم التوصل الر تركيبة مناسبة من المعالجات يتم من خلالها الحصول على أداء جيد للنظام وفق قيود مطبقة على الطاقة المستهلكة باستخدام خوارزمية الجينات.

DECLARATION

I declare that the master thesis entitled "Performance-Power Enhancement on High-Scale Heterogeneous Multi Processors (HMP) using Genetic Algorithms (GA)" is my own original work, and hereby certify that unless stated, all work contained within this thesis is my own independent research and has not been submitted for the award of any other degree at any institution, except where due acknowledgment is made in the text.

Ahmad N. M. Aljabari

Signature:..........[illegible]..........

Date: ..8..../..7..../..2012

STATMENT

In presenting this thesis in partial fulfillment of the requirements for the master degree in Informatics at Palestine Polytechnic University (PPU), I agree that the library shall make it available to borrowers under rules of the library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made. Permission for extensive quotation from, reproduction, or publication of this thesis may be granted by my main supervisor, or in his absence, by the dean of graduate studies and scientific research when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Ahmad N. M. Aljabari

Signature:[illegible]

Date: 8/7/2012

ABBREVIATIONS

ARM	Acorn RISC Machine
CPU	Central Processing Unit
EP	Execution Power
ET	Execution Time
GA	Genetic Algorithm
HARD	History-Aware, Resource-Based Dynamic Scheduling
HMP	Heterogeneous Multi Processor
IC	Instructions Count
IPC	Instruction Per Clock
IP	Information Policy
ISF	Instruction Set Format
LP	location policy
MCP	Multi Core Processor
minGW	Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications
MIPS	Millions of Instructions Per Second
MSYS	collection of GNU utilities
MUTP	Mutation Probability
NEC	Nippon Electric Company
NP	Number of Processor
OVPsim	Open Virtual Platform Simulator
POPSIZE	Population Size (Number of Individual Population)

PSD	Processor Sharing Density
PVP	Primary Virtual Processor
RP	Real Processor
RRDA	Round Robin Dynamic Assignment
SP	Selection Policy
SD	Sharing Data
TLP	Thread Level Parallelism
TP	Transfer Policy
VP	Virtual Processor
VLSI	Very Large-Scale Integration
XOVERP	Crossover Probability

Dedication

I dedicate this study to my parents, my wife, my children, my brothers, my sisters, and my friends for their encouragement, support, endurance, and patience.

Acknowledgment

I bow my head to ALMIGHTY ALLAH for the help, guidance and blessing HE has bestowed me.

I am indebted to all who encouraged me to produce this research study. So many people helped during the process of writing the study. I am especially grateful for the encouragement given to me by my supervisor, Dr. Mohammad Aldasht, at various stages of the production of the research study.

To all of you, once again, thank you very much.

Ahmad N M Aljabari

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces a general overview of the present research, its problem statement, objectives, motivations and finally its major contributions to the field. This chapter is organized as follows: section 1.1 is an overview about the study or the thesis; section 1.2 describes the problem statement; section 1.3 introduces the objectives of the thesis; section 1.4 lists the motivations of the study; and section 1.5 includes the thesis contributions.

1.1 Overview

The computer system consists of variety components (CPU, Memory, I/O device, etc.) Each component in the system is responsible for a particular job during the execution such as, memory to load code, I/O device to input/output data, system bus to connect between components, etc. The primary component in the system is the central processing unit (CPU). The original CPU consists of a single processing core that executes the program code in a sequential mode (instruction by instruction). Computing performance is improved by increasing the execution speed to the CPU, for example; by increasing the clock frequency for the processor unit. In the recent years, this approach was abandoned due to some challenges (increased core complexity, energy and heat cost, limiting the rate at which the clock speed can be increased, etc.) [18] .

Parallel programming is another technique which has been developed to increase the performance of a single processor. Parallel programming is applied by using different methods like pipelining, VLSI, and superscalar. However, parallel programming methods have limitations. For example, the pipeline method throughput is generally less than the number of pipeline stages. In addition, there are limitations in parallel level in the program itself, and the parallel programming technique is still weak to meet the needs for high performance in the computing system [10, 43].

For these challenges, new trends were adopted for multiprocessor architecture approach by either adding more core in the chip or adding more processor units to increase computer system performance. Each processor unit is responsible for executing its own sequence of instruction to increase the overall computing system performance. The multiprocessor architecture is a good method to improve such performance. Amdahl's Law reveals the maximum speed up that can be expected from parallel algorithms given the proportion of parts that must be computed sequentially [21] .

There are two types of multiprocessor architecture: homogeneous and heterogeneous. The first one is also called symmetric. In this type, the platform consists of identical processors. The second type, heterogeneous multiprocessors (HMP) architecture, is called asymmetric; HMP architecture consists of variety processors on one chip. There are two different types of HMP architecture: performance heterogeneity, which uses the same instruction set architecture "ISA" and functional heterogeneity where each core has an instruction set architecture.

Homogeneous type provides a uniform platform on which parts of the computer programs are executed equally on all processors. However, the homogeneous platform may not provide the best possible level of performance. To execute different applications of the different parts from the same application, we need different processing requirements. The benefits of multiprocessors can be achieved by executing different program parts on different processors capability or more specialized processors. The implementation on HMP system might be more complicated. This needs more information of program behavior and processor types when matching between the processors and the threads to achieve high performance.

While the HMP has the potential to increase performance, the designed any program on HMP could be difficult. The programmer not only deals with concerns implicit in concurrent programming on HMP such as scalability, synchronization, consistency, and deadlock prevention, but also with different processor types, instruction set architecture, and thread behavior characteristics so as to schedule threads on processors and achieve high performance [30] .

In multiprocessors architectures, parallel programming has different methods when compared to a single processor. Multiprocessors technique can exploit thread level parallelism (TLP) methods. The program or application will decompose to different parts depending on decomposing methods. There are two methods applicable to decompose the application into a small part: functional (task) decomposition and data (domain) decomposition. Each part (thread) will be executed on a processor. With Parallel programming and TLP, we can exploit multiprocessors system architecture, especially HMP architecture when executing threads with different processing requirements [43].

The present study aims to exploit high scale HMP architecture by applying genetic algorithm scheduler on different processor types without any intervention from programmers, to achieve optimized performance with minimal concumption power, and to make the assignment of threads to the processors responsible for the runtime system. Scheduling algorithm on high scale HMP architecture will be implemented by using OVPsim simulators and Matlab [3, 42]. The OVPsim simulator supports many types of processors such as mips32, arm, PowerPC 32, and NEC v850. These processor types are used to build the computer processing environments. The benchmarks selected in the implementation support multiprocessor platform environment.

1.2 Problem Statement

In general, laptops, desktop computers, and servers use homogeneous multiprocessor architecture to move away from complexity. Some applications focus on using HMP architecture, particularly the applications that have a special purpose such as image/video processing applications.

Addressing the difficult methods used to decide which thread will be assigned to processors during the runtime to achieve high performance occurs by exploiting the HMP system. The scheduler relies on the program characteristic behavior in the runtime environment and the processor's processing capability to make the system decide which thread-processor matching is better to achieve high performance with minimal power.

To achieve this, the study is developed through the following stages:

- Step1. Selecting a simulation environment and a set of benchmark applications to achieve the experimental work of the research.

- Step2. Developing cost function algorithm that will be able to collect program behavior characteristic information for each processor, processing requirements, execution behavior and inter-thread communication.

- Step3. Developing an evolutionary scheduling algorithm that uses the program behavior characteristics and processor characteristics to optimized performance and decrease power consuming.

- Step4. Analyzing the results and achieving comparison to decide the quality of the proposed algorithm and realize the required enhancements which lead to a better algorithm.

Problem example:

By using different methods to schedule thread-processor, we can achieve different performance. We use an architecture system that consists of 4 heterogeneous processors (C1, C2, C3, and C4) as shown in figure 1.1. The application that needs to be executed consists of 4 threads (T1, T2, T3, and T4). Each thread consists of 1 million instructions and has an instruction per clock (IPC) behavior characteristic for each processor as shown in Table 1.1.

Table 1.1: Threads Behavior Characteristics

Thread	C1	C2	C3	C4
T1	1.4	2.4	0.5	1.2
T2	0.5	1.5	1	0.5
T3	2	0.5	2.4	3.2
T4	2.6	1	1.6	1.4

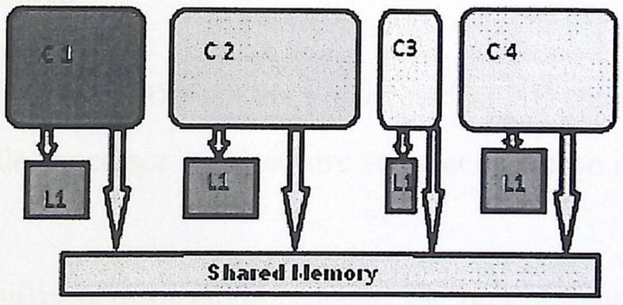


Figure 1.1: Processors Interconnection

By using 4 processors to execute 4 threads, we have 16 different combinations to execute the program. In the example, we will show 4 different combinations. The results are shown in figure 1.2 by executing the programs using 4 scheduling methods as explained in table 1.2.

Table 1.2: Scheduling Methods

Methods	C1	C2	C3	C4
1	T1	T2	T3	T4
2	T2	T3	T4	T1
3	T3	T4	T1	T2
4	T4	T1	T2	T3

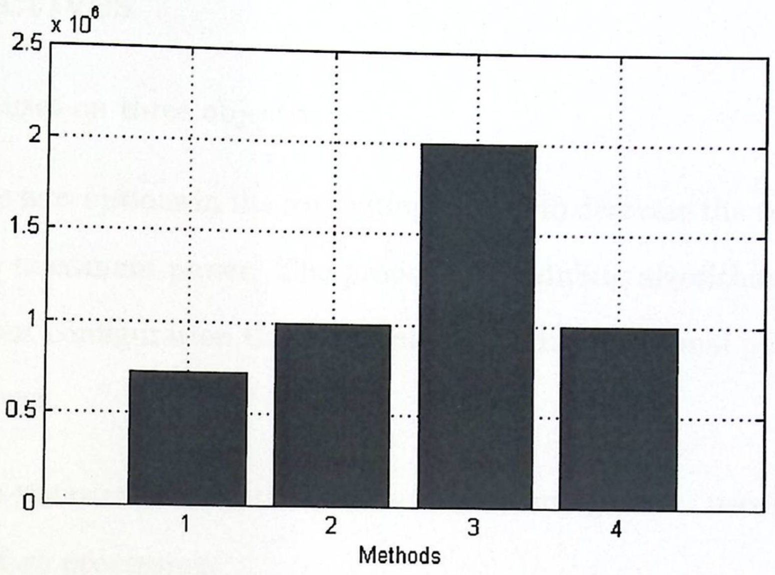


Figure 1.2: Execution Results

1.3 Motivations

Scheduling on heterogeneous multiprocessors is defined as NP-complete problem [23, 31] . The work on high scale processor architecture is expected to be more complicated.

In heterogeneous multiprocessor environments, the programmer must deal with high scale HMP architecture characteristic to write a program that uses the heterogeneity of the system to achieve high performance. In this work, the scheduling algorithm will assist the programmer to write a program without dealing with HMP architecture to build a program that can observe the system during the run time to perform thread-processor scheduling in high scale HMP architecture to achieve high performance. In addition to developing the computing system by integrating a new option that controls the processing performance according to power status; this option will work through two methods:

First: manual control method: the user increases or decreases the overall performance to save power.

Second: automatic control method: the system observes the power source status or peak time to change the CPUs performance according to the observation information.

1.4 Objectives

This research focuses on three objectives:

- To integrate new options in the computing system to decrease the value of consuming power from maximum power. The proposed scheduling algorithm is responsible to find processor configuration that guarantees to achieve optimal performance within minimal power.
- To estimate the overall execution time and consuming power needed to execute the problem before processing.
- To distribute the any problem to all processors according to processor's capability.

1.5 Contribution

Estimating the power consumption and time needed to execute programs on high scale HMP architecture before executing takes place by:

- Presenting the program by small sample (S) that specify the real phenomena of the program.
- Determining the program problem size.
- Executing sample (S) on each processor and measuring the time and power needed for each processor.
- Estimating the consumption power and execution time when using all processors.
- Setting the processor configuration of the high scale HMP architecture by using GA.

- Estimating the overall power and time needed to execute the program according to the processor configuration, sample power and time for each processor, and program size.

This thesis presents information about how to assist the system to exploit high scale HMP architecture by carrying out dynamic thread scheduling that increases the performance with minimal power in a number of ways:

- *Abstracting the program behavior characteristic* during the run time execution.
- *Developing a cost function* that uses a program's behavior characteristic to inform thread placement across heterogeneous processor.
- *Developing an evolutionary scheduling algorithm* that exploits both the program behavior characteristics and processor characteristics to optopmize performance and minimize power consuming.

Chapter 2

Background and Literature Review

The purpose of this chapter is to provide a theoretical background about the scheduling algorithms which are implemented on multi / many processor architectures. In addition, this chapter reviews the literature about the development of scheduling algorithms for heterogeneous multi-processor architectures. The chapter is divided into sections as follows: Section 2.1 presents an overview about two general types of scheduling in HMP architecture. Section 2.2 presents the general scheduling methods. Section 2.3 presents the recent method proposed to schedule HMP architecture.

In general, processor performance enhancement focuses on increasing clock frequency rate and parallel processing by applying instruction level parallelism technique in single processor (improving single thread performance). However, these techniques are stalled due to the limitations in the degree of parallel processing that can be extracted from sequential processing [46] and the clock frequency issues (energy, heat, and complexity) [39]. The idea is to increase computing system performance by designing a platform which consists of multi processors with multicores [35]. The multi-core processor is designed by exploiting available transistors on a given size of processor die. With this generation of multi-core processors, we can exploit thread level parallelism [21].

Multi-processors platform can be symmetrical (homogeneous); in this type all processors in the chip are identical. Most computing systems are homogeneous. Asymmetrical (heterogeneous) HMP architectures consist of different types of processors; each processor has different function (ISA), performance, and capabilities [8].

HMP architecture has better performance than the homogeneous type [9]; the disadvantage of using HMP architecture is limited to the complexity of exploiting the heterogeneity in the implementations. There are different methods used to schedule threads on processors. These methods can be classified into static and dynamic scheduling algorithms [12].

In order to exploit Thread Level Parallelism (TLP), the application can be decomposed by two methods: function decomposition and data decomposition. In order to use function decomposition, the task splits into small tasks. Each task is run by a special processor depending on the efficiency and the executing thread on any processing type (in HMP architecture) or on any processor (in symmetric architecture). In order to using data decomposition, when the executing application has large data, we split the database boundary into small parts and distribute these parts to the processors; each processor will execute the same function [27, 32, 38, 43].

When the executing application uses any type of decomposition, dependency must be controlled to save the consistency and accuracy. There are many methods used to save implicit execution property like share memory, massage passing, and mailbox. The present study proposes some methods and techniques that can be used to schedule threads in HMP architecture in order to increase throughput and system performance.

2.1 Heterogeneous multi-processor architecture

Computer systems that use multi-processors (heterogeneous resources) are not new. The IBM System/709 incorporates a processor to process I/O operation. In this case, the main CPU executes another operation during I/O operation [19]. The same technique can be found in another system such as IBM System/370, System/360 [13], and Control Data CDC 6600 [42]. These processors are generally limited to performing special purpose operation (transfer and signaling operations) and not sharing the computer system capabilities. There is another specific purpose processor which has been used to string matching [15], accelerate floating points [14], encryption [47] and many other applications. [30] proposes that HMP architecture consists of general purpose processors (CPU), unlike the superscalar processors that have specific purpose processors.

HMP architecture has been recently used in specialist applications such as image/video applications, network processing, and low power embedded systems. some primary implementations of HMP architecture:

- Network processing equipment uses a number of processors like, the IBM Power NP [7], Intel IXP [5], and Motorola C-Port [20].
- Multimedia workloads and scientific computation use IBM Cell processors [1,22].
- Low power embedded devices, such as mobile phones, use Intel PXA800F [29].

In this research, we are not talking exclusively about HMP archetecture. However, the research extends a wide range of high scale HMP archetecture. Some models of high scale HMP archetecture are shwon in table 2.1 [1].

When the transistors count are increased, the expected performance will be improved. In general, performance increase is governed by Pollack's Rule [11]. This means that the expected performance will be increasing by square root of increasing the transistor count. For example, doubling the number of transistors in a single processors will increase the

Table 2.1: high Scale HMP Model

Processor	NP	Manufacturer
TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670	73278	NEC/HP
K computer, SPARC64 VIIIfx 2.0GHz	705024	Fujitsu
Cielo - Cray XE6, Opteron 6136 8C	142272	Cray Inc

performance about 40% ; on other hand, doubling the number of processors will increase the performance about 100% in case of parallel workload. Figure 2.1 explains how we can achieve different performance from the same area of silicon when the processor design is improved. Figure 2.1(a) presents a single processor, whereby the overall execution performance for a sequential program equals 4. As shown in Figure 2.1(d), if we can exploit all processors at the same time with the executing parallel program, we can achieve performance which is equal to 8. With high degree of complexity to design more processors from the same area of silicon die, we can increase the performance as shown in 2.1(b, c, and d).

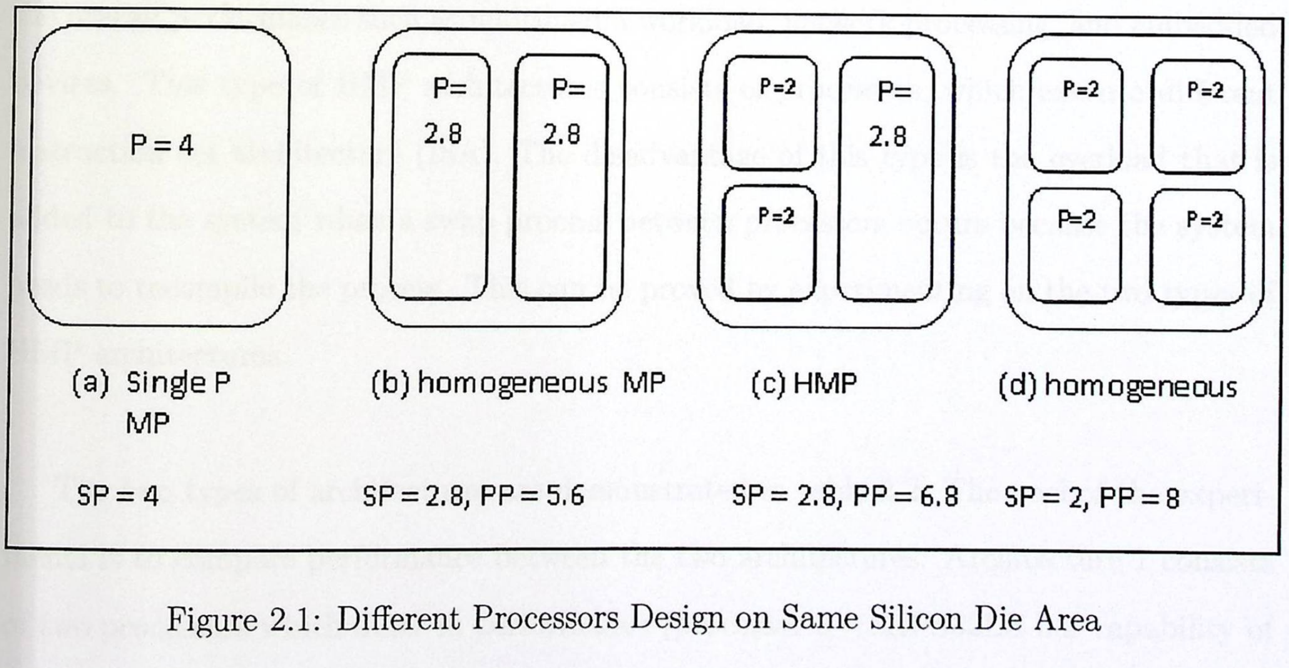


Figure 2.1: Different Processors Design on Same Silicon Die Area

As we noted earlier, processor architectures can be divided into two types: homogeneous and heterogeneous multi-processor architectures. Homogeneous multi-processor architectures, also called symmetric, consist of identical processors. In general, implementation depends on homogeneous type, specially the computers, which have general purpose programs, like laptops, servers, and desktops [36]. Heterogeneous multi-processor architectures, also called asymmetric HMP architecture, consist of a variety of processors. The following is an explanation of the two main domains of heterogeneity:

Processors with different performance: these are processors with high performance and are used to process big threads and processors with low performance and used to process small threads to reduce consuming power and exploit thread level parallelism. In this type of variety, all processors on the platform use the same instruction set architecture (ISA), and there is no need to recompile threads in case of swapping between the processors.

Processors with different ISA (function): these processors can be found in the applications that have specific purpose. The developer enhances the system by adding processors that can execute some function with high degree of performance to improve the overall performance such as multimedia workload, network processing, and embedded devices. This type of HMP architectures consists of processors, which execute different instruction set architecture (ISA). The disadvantage of this type is the overload that is added to the system when a swap process between processors occurs because the system needs to recompile the process. This can be proved by experimenting on the two types of HMP architectures.

The two types of architectures are demonstrated in table 2.2. The goal of the experiments is to compare performance between the two architectures. Architecture 1 consists of two processors which differ in performance (processor 0 =MIPS32LE has capability of 300 MIPS; processor 1=MIPS32LE has capability of 100 MIPS). Architecture 2 consists of two processors which differ in function (processor 0 =MIPS32LE has capability of 300 MIPS; processor 1=ARM7 has capability of 100 MIPS). The source code used to create architecture 1/2 are shown in Appendix A.1 and A.2.

Table 2.2: Processors Specification

Architecture 1 (performance HMP)			Architecture 2 (functional HMP)		
Processor	**Type**	**MIPS**	**Processor**	**Type**	**MIPS**
Processor0	MIPS32LE	100	Processor0	MIPS32LE	300
Processor1	MIPS32LE	300	Processor1	ARM7	100

This experiment will implemented by using OVPsim simulator environment. Two processors will execute the same program shown in Appendix B.1. The program consists of two threads, each of which is used to multiply two matrices. The program will execute 9 times. In each iteration, the size of matrix will increase, and the size of the matrix in each thread will be equal.

Figure 2.2 and Figure 2.3 shown the results for architecture1 and architecture 2 respectively. In each iteration, we can observe the number of instructions executed by each processor and the time needed to execute these instructions for each processor without overhead. We can also measure the overhead needed for each platform, and measure the total time (overhead + execution).

The difference between the two processor architectures in processor1; replacement of processor1 in architecture1 from ARM7 100 to MIPS32LE 100.

Figure 2.2 shows the cooperation execution time (performance) between two processors with the same capability. As can be seen from Figure 2.2, the ARM7 processor is faster than MIPS32LE. The difference in speed for each processor is due to the instruction set architecture format for each processor.

In figure 2.3(a), we can compare the execution time for two architectures without overhead. Architecture1, which consists of processors different in function, is faster than architecture 2, which consists of processors different in performance, because architecture1 contains an ARM7 processors that have capabilities better than MIPS32LE which is used in architecture2.

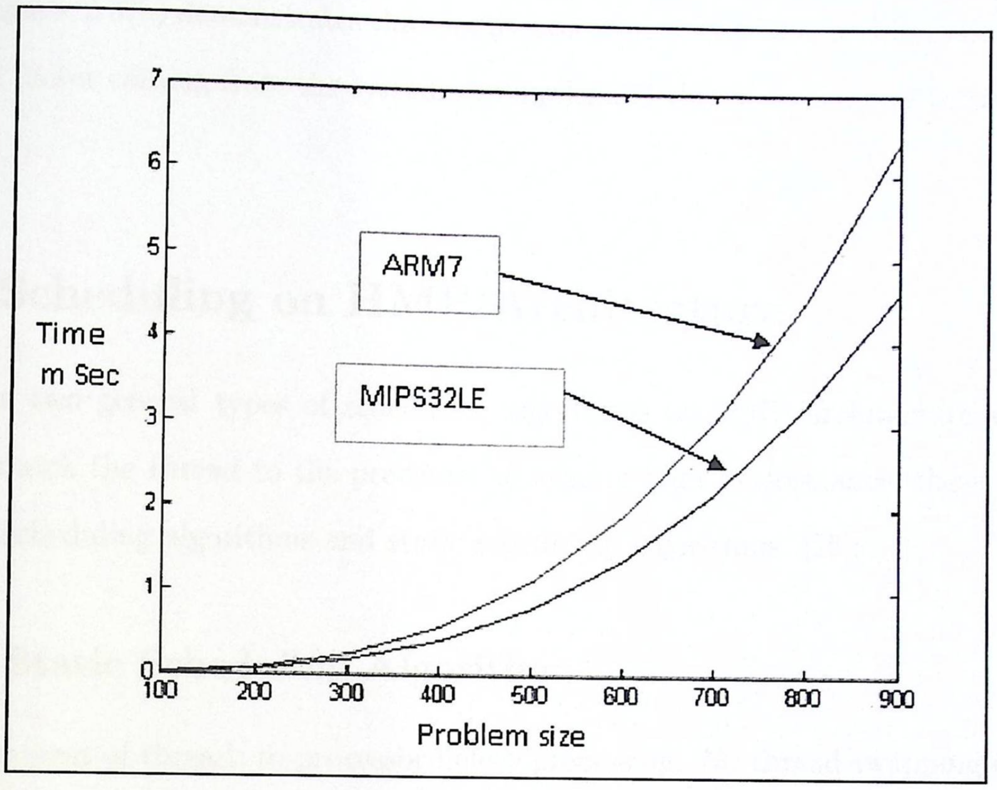


Figure 2.2: Performance of Separate Processors

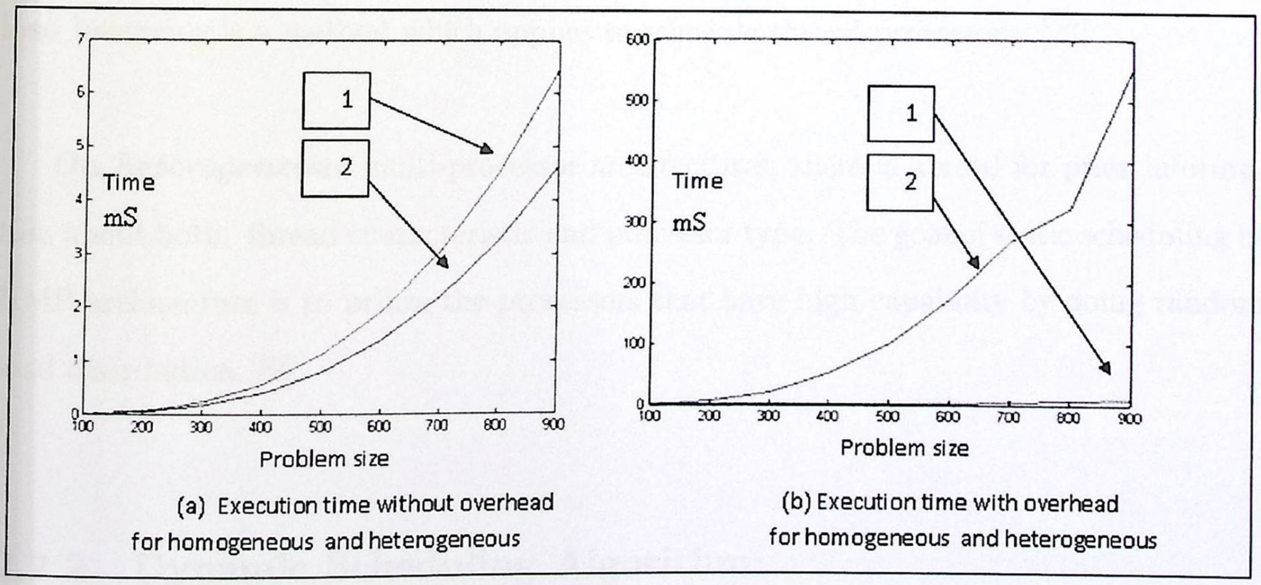


Figure 2.3: Execution Time for Homogeneous and Heterogeneous

However, in overall execution time (processor processing time + system overhead time), Figure 2.3(b) demonstrates that architecture 2 is better than architecture1 because overhead factor coming from the system recompiles the thread to the new instruction set format.

2.2 Scheduling on HMP Architecture

There are two general types of scheduling algorithms on HMP architecture which are used to match the thread to the processor to achieve high performance; these types are dynamic scheduling algorithms and static scheduling algorithms [25]:

2.2.1 Static Scheduling Algorithm:

The assignment of threads to processor before processing. No thread swapping or assignment change between processors during runtime.

On homogeneous multi-processor architectures, there is no need for prior information about thread characteristic and processor type needs to schedule. In general, the load balancing is a method which applies to schedule thread processors [27].

On heterogeneous multi-processor architectures, there is a need for prior information about both: thread characteristic and processor type. The goal of static scheduling in HMP architecture is to utilize the processors that have high capability by doing random load distribution [9].

2.2.2 Dynamic Scheduling Algorithm:

Re-assignment of threads to the processor during processing according to thread behavior characteristics.

There are many policies which must be considered to decide which method or algorithm is better to dynamic schedule threads in multi-processor architectures. Dr. Aldasht proposes using genetic algorithm to explore the overall policies that define the dynamic load balancing strategies [6].

The first policy, **the information policy** (IP), is used to determine how the information exchange between nodes. If IP=1, the information will exchange according to load balancing frequency. If IP=0, the information will exchange on demand.

The second policy, **the transference policy** (TP), is used to specify the case of nodes. If TP = 0, the node is overloaded and needs to transfer the tasks (sender initiator). If TP=1, the node is ready to receive the tasks (receiver initiator). If TP = 2, we have a sender initiator and a receiver initiator at the same time (symmetric initiators).

The Third policy, **the location policy** (LP), is used to determine which node is incorporated in load balancing operation. If LP=0, that means the information collected from the node is compared with the threshold to decide to which location the node will send or receive tasks. If LP=1, that means the decision will be according to two-thresholds. If LP=2, the node will send tasks to the shortest-path based node, with the lower load. If LP=3, the node will select a random location node as a receiver or sender.

The fourth policy, **the selection policy** (SP), refers to the task selected to transfer the preemptive (SP=0) or non-preemptive (SP=1) from the sender node. These policies, which are defined by Aldasht, cover all dynamic and static scheduling methods for multi-core processers. By tuning the policies during the runtime and checking the overall system performance, THE scheduling algorithm set the value for each policy according to program behavior characteristics.

2.3 Literature review

At first glance, when we talk about the distribution of tasks on a multi-processor architecture, we propose using the theory of load balancing on a multi-processor architecture as David W. Holmes does by applying H-Dispatch algorithm to a wide range of numerical simulation problems by using spatial decomposition to create "orthogonal computational tasks." This algorithm makes efficient use of memory resources by limiting the need for garbage collection and taking optimal advantage of multiple processors through employing a "hungry pull strategy" [16]. This algorithm is flexible and efficient when it is applied to a numerical domain, depending significantly on data sharing, regardless of the type of multi-processor: homogeneous or heterogeneous. Nevertheless, if the database domain is used on a heterogeneous multi-processor architecture, it is necessary to develop this algorithm to do data decomposition and distribution depending on the specification and processing type of the processor.

Many recent works have tried to handle the above mentioned problem. Jian [16], tries to exploit the efficiencies of the heterogeneous multi-processor architectures by executing the processes on a smart scheduler which obtains a high performance and consumes power . There are many proposed scheduling algorithms; each depends on some properties from the system to achieve dynamic scheduling such as History-Aware Resource-Based Dynamic Scheduling (HARD). As for the heterogeneous multi-processor architectures proposed by Jooya, this algorithm depends on recording application resources, utilization and throughput, and analyzing the information to decide how to adaptively change the matching between the processes and the processors during the runtime [26]. The above authors wonder how it would be possible to develop the scheduling algorithm by mixing new information to make the algorithm more efficient.

Jaejin proposed scheduling algorithm for simultaneous multithreading environment to obtain good matching between the processes and the processors. Furthermore, the

History-Aware Resource-based Dynamic (HARD) scheduling algorithm proposes and implements in Open Solaris which uses single-ISA heterogeneous multi-processors architecture [24]. It is comparatively simple and scalable, but the resulting Algorithm does not rely on dynamic profiling [18, 34, 40]. The scheduling Algorithm is based on the idea of architecture signatures "a compact summary of architectural properties of an application. It may contain information about memory-boundedness, available Instruction Level Parallel (ILP), sensitivity to variations in clock speed and other parameters" [40].

M. Becchi and P. Crowley proposed scheduling algorithm to heterogeneous multiprocessor architecture scheduler. The algorithm has the ability to dynamically rescheduling processes at a runtime based on periodically collect performance statistics [9]. This Algorithm collects information on the performance produced by rotating all processes periodically between cores. After that, the information is used to assign the processes to the processors in the optimal way that guarantees to achieve high performance. The main disadvantages of this algorithm are: the overhead is required to update the information about the performance statistics in addition to the loss caused by rotating processes periodically; the second disadvantage is the increase of complexity when the numbers of processors are increasing, down to the case of unreasonable.

As a result, the best and most recent methods are used to dynamically schedule threads in HMP architecture to maximize the overall performance and reduce the overall consumption of power during the execution time:

2.3.1 Round Robin Dynamic Assignment (RRDA) on HMP architecture

The main of objective of the RRDA algorithm on HMP is to assign the threads to the processors which better exploit the hardware resources in order to improve the performance with the simple technique of the procedure and policy [9, 37].

There is no need for the cost function in the RRDA algorithm to observe the program behavior characteristics during the runtime. The procedure of the Algorithm is simple as shown in figure 2.4: by periodically rotating the threads to processors in a round robin fashion.

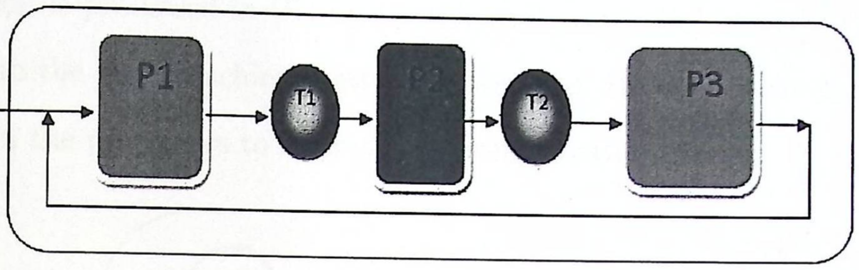


Figure 2.4: Threads Swap In RRDA

This routine ensures that the available processors are equally shared among the running programs.

Algorithm drawbacks, first, the round robin strategy is blind, that This means that the runtime system is not aware of the thread behavior characteristics and does not use the runtime execution information to drive threads assignment. Second, the overhead has a high value for a swap period parameter (the frequency of the rotation).

2.3.2 IPC-Driven Dynamic Assignment on HMP Architecture

This Algorithm depends on Instruction per Clock (IPC) behavior characteristic of threads during the execution. The assignment of threads to core is driven by IPC for each thread to improve the overall IPC. The scheduler observes the runtime system and collects information by cost function from the core for each thread. By comparing IPC threads for all processors, the scheduler decides which thread needs migration to another core.

The IPC must be available for all threads. In fact, the IPC value is available for the cores that execute threads. If the processor does not execute any thread, we need to refresh the information by carrying out the round robin method on the system or swap the threads for the processor that doesn't execute these threads [9].

Figure 2.5 shows how a simplified IPC-driven guides the system. In (A), the cost function collects information (IPC for each thread executing in processor) from the processors. The information is processed in (B) by the scheduler controller to drive the assignment of the thread to the core to achieve better performance. In (C), the controller swaps the thread between the processors to improve the overall performance.

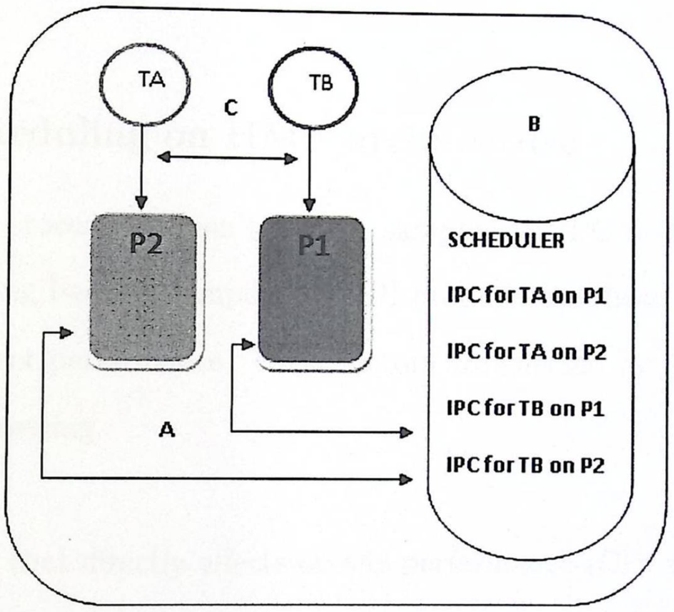


Figure 2.5: Simplify IPC-Driven Dynamic Assignment

2.3.3 History-Aware, Resource-Based Dynamic Scheduling on HMP architecture

Jooya proposed scheduling algorithm for dynamic scheduler on HMP architecture. The main method in this algorithm is recording the resources utilization and throughput during the runtime to maximize performance by upgrading the sensitive threads which require more resources to a faster processor and reducing consumption of power by downgrading the insensitive threads to the weaker processor [26].

The HARD algorithm relies on two subsections: detection phase and reassignment phase. In the detection phase, the cost function records the thread behavior (throughput and processors utilization) during a T-clock cycle interval. A short T interval reduces the performance due to the switching overhead, and a long T interval may miss the thread behavior information, which means that the value of T must be chosen carefully. In the reassignment phase, the history of information which is collected in the detection unit to decide which threads need reassignment.

2.3.4 Bias scheduling on HMP architecture

The bias scheduling procedure doesn't include sampling of IPC or CPI on all processor type or offline profiling because sampling of CPI produces overhead [28]. IPC and CPI specify single processor performance. These factors are affected by internal and external stalls during the processing.

The other reason that directly affects on the performance (CPI value) of a processor can be divide into two categories. The first category specifies the performance differences which are caused by micro architectural choices in the processor such as, micro architectural execution algorithm (ILP, out-of-order, and in-order design), the size of resources allocated (cash unit, registers, TLP, etc.), and micro architectural feature (pipeline depth, branch predictors, or unit latency). The second category specifies the performance effects on the resources outside the core such as, access to shared caches and memory, and I/O operations.

Application bias means that the processor which operates the system is preferred to execute a thread at a particular time. Application bias is divided into two types. In first type, the thread has a small processor bias if the speedup of the execution thread from a big processor size to a small processor size is modest. In second type, the speedup is large.

The two types of stall (external and internal) are a strong predictor of the application bias. The scheduler determines the processor stall that limits the application performance. After that, the application bias guides the scheduling decisions that maximize the performance.

2.4 Summary

The main difference between the current research and the previous research; is that most proposed scheduling algorithms in the previous research attempt to enhance performance to multi processors computing systems through running the system with its all capability, which means, sharing all processors in execution. These algorithms also efficient if applied on low scale multi processors systems.

However, the current study proposed scheduling algorithms using genetic algorithm method. It also tries to run multi processors computing system in high performance through selecting a number of processors that exist in the multi processors system, this is to be done by a determined value of power. The efficiency of this algorithm increases as the number of processors increases in the system.

Chapter 3

Methodology

This chapter describes and explains the methodology used to achieve the thesis objectives. This chapter is organized as follows : section 3.1 describes and explains the scheduling algorithm assumption; section 3.2 describes and summarizes the scheduling algorithm pseudo code; section 3.3 explains in detail and exemplifies the scheduling Algorithm pseudo code; section 3.4 describes genetic algorithm, presents an overview about algorithm and explains algorithm procedure.

The study proposes scheduling algorithms, which exploit an HMP platform to achieve a high performance. In general these algorithms are efficient for a limited number of processors, and they observe the system during the runtime that needs to process all tasks in the application. The relationship between the number of processors in the platform and the overheads is proved. The study proposes that GA based scheduling algorithm runs in an offline mode and is efficient for a large number of heterogeneous processors. By this algorithm, it is possible to estimate the overall execution time and consuming power which are very close to the real value.

3.1 Scheduling Algorithm Assumption

The proposed scheduling algorithm is supported to be efficient in a high scale HMP platform. However, the application types change the efficiency degree. To obtain a high degree of efficiency and make the GA based scheduling algorithm that we are developing more effective, we assume the following application properties:

3.1.1 Work Distribution:

We assume the domain decomposition type with loosely coupled properties. To exploit many processor environments, we need to decompose or split the application into a small number of tasks that can be executed in parallel on available processors. After that these tasks are matched to different processors by applying the efficient scheduling Algorithm. In general, there are two main types of application decomposition: domain decomposition and functional decomposition [32, 38]

Functional Decomposition: In this method, the original function can be reconstructed into relational constituent parts; each processor executes some parts from the original function for example:

If we have a function

$$f = \cos(x) + \sin(\sqrt{y}) \tag{3.1}$$

We can reconstruct f into three function:

$$f1 = \sqrt(y), f2 = \sin(f1), and f3 = \cos(x) \tag{3.2}$$

In this case the data x and y will be stored in a shared place and the functions ($f1$, $f2$, and $f3$) will be distributed to various processors.

Domain Decomposition: This method solves a boundary value problem by splitting it into smaller boundary value problems on sub domains. For example, the function 3.3 is filter (x and y will determine pixel positions) Domain (D) consists of images and the boundary = 1000 units. We can split the D into sub-domains like:

D1 = 200 images, D2 = 500 images, and D3 =300 image.

The filter function f executes on all processors, and the sub-domains $D1$, $D2$, and $D3$ are distributed to various processors.

$$f = a(x) + b(y) \tag{3.3}$$

3.1.2 Application/ Behavior Problems:

The application type has two properties: size and sample. In general, the computing system executes different types of applications, and each type has size I, and we can present the application size with a small sample like:

1. Relational Databases: The database domain is a structured collection of data; the structure presents relational data (table) which consist of a number of records. The size of the application is a number of record and the small sample is one record.
2. Image Processing: The domain consists of a number of images. The size of the application is a number of images, and the sample is one image.
3. Matrices Processing: The domain consists of a number of matrices, and the operation result is MxN matrix. The problem size is M or N, and the sample is M result for size M or N result for size N.

3.1.3 Processor Specifications

The scheduling algorithm deals with a a black box processor and doesn't focus on any properties of the processor specifications like ISA, cache memory, processing type (parallel or sequential), bus bandwidth, ALU, etc. The main part that the scheduler focuses on is the time needed to execute the sample.

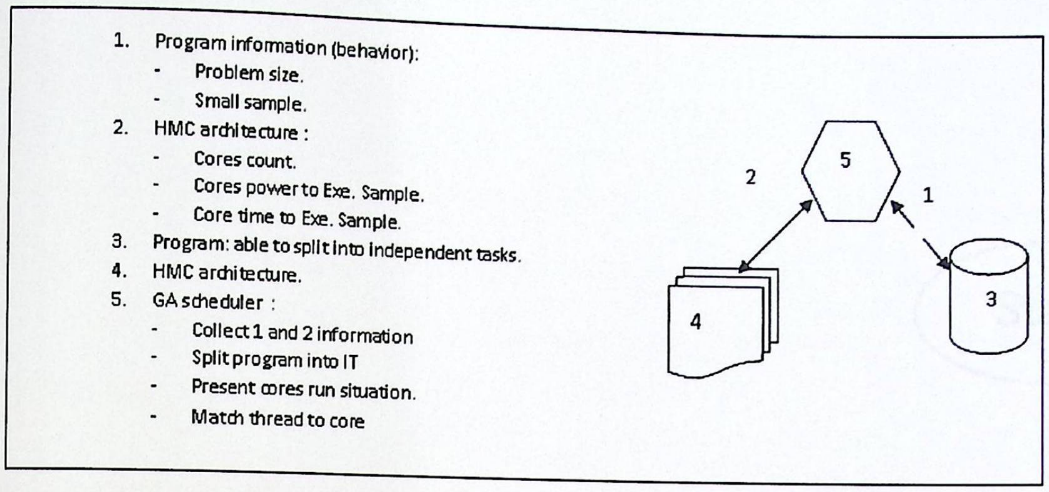
3.2 Scheduling Algorithm Pseudo Code:

This section summarizes the scheduling algorithm procedure as shown in algorithm 3.2 for all parts of the system shown in figure 3.1. Regardless of the implementation of any part of the procedure.

```
Begin
Step1.  Select a sample S that represents the program of size I
Step2.  Determine processors that will share in execution (overloaded
processor s will be excluded from the selection phase due to architectural
constraint)
Step3.  Execute S on every processor
Step4.  Collect execution information (Time (ETi)and Power (EPi)
Step5.  Determine the objective function( min ETi or min EPi)
Step6.  Use GA to find the optimum configuration of the processors
Step7.  Distribute I according to the processors configuration done in step 6
End
```

Algorithm 3.2

Figure 3.2 shows the proposed algorithm flowchart, all steps in the flowchart sequence explains in sections (3.3 and 3.4).



1. Program infomation (behavior):
   - Problem size.
   - Small sample.
2. HMC architecture :
   - Cores count.
   - Cores power to Exe. Sample.
   - Core time to Exe. Sample.
3. Program: able to split into independent tasks.
4. HMC architecture.
5. GA scheduler :
   - Collect 1 and 2 information
   - Split program into IT
   - Present cores run situation.
   - Match thread to core

Figure 3.1: Scheduling Algorithm Part

3.3 Scheduling Algorithm Details

In this section, we will explain our scheduling algorithm. In addition, we will show the implementation type of each step.

To illustrate the theories and methods in the algorithm, the Example 3-2 is developed as follows:

1. Create a platform, which consists of 4 heterogeneous processors, processors specification are shown in table 3.2. The source code is in Appendix A.3.

2. Create matrix-vector multiplication application program, using GA to distribute jobs among processors. The source code is in Appendix B.2.

3. Create a Matlab script file to analyze the information and show the results. The file is described in Appendix C.1.

4. Create a script file to compile the above files. The source code is described in Appendix C.2.

Selecting a sample S (which is a portion of the program of size I): According to the assumptions about the application type, we have two application properties: size (I) and sample (S). These properties are determined from the compiler. The domain boundary is determined from I and S as shown in formula 3.4.

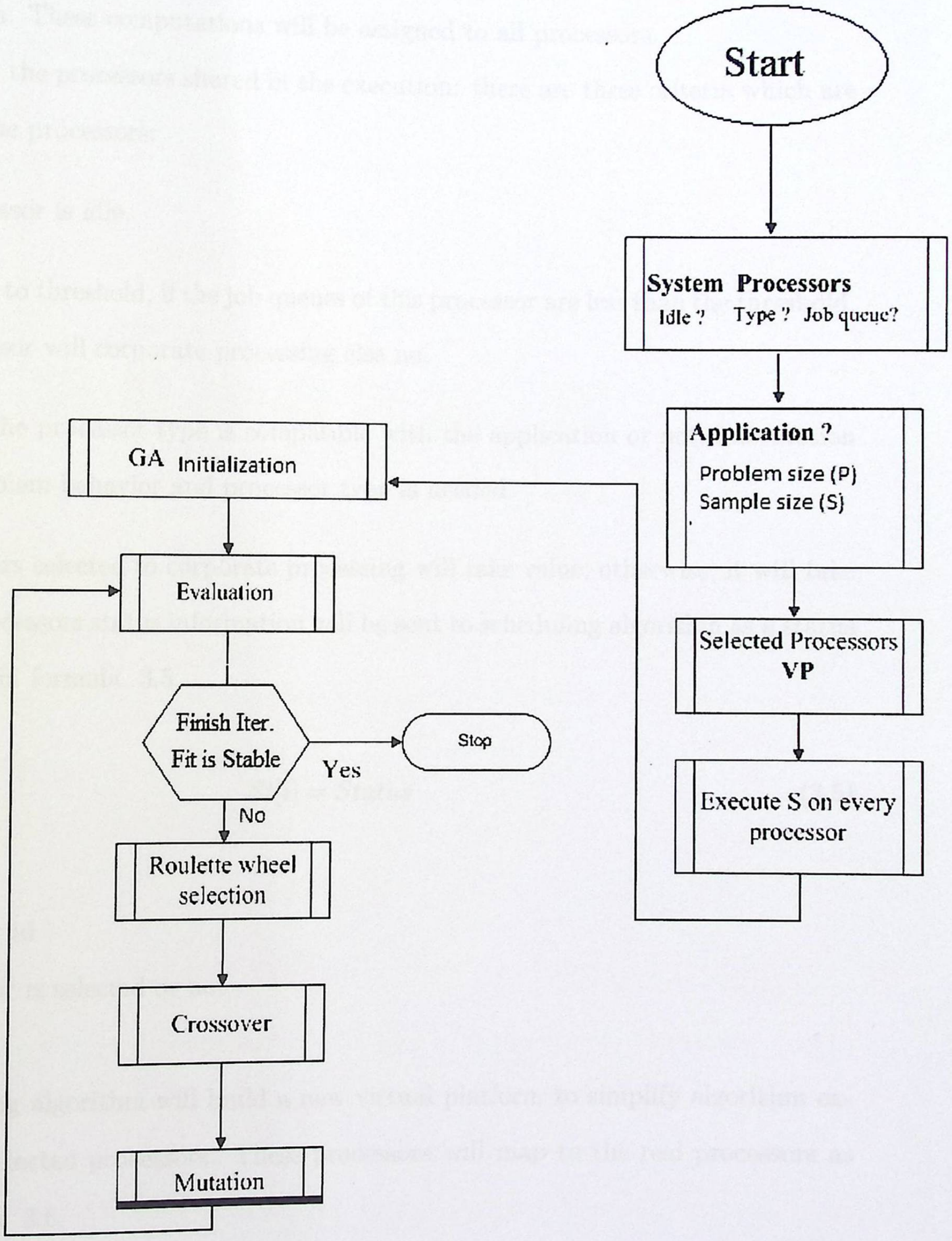


Figure 3.2: Proposed Algorithm Flowchart

$$Problem = \frac{I}{S} \times computation \tag{3.4}$$

The computation part in formula 3.4 represents the functions and operations that process the data. These computations will be assigned to all processors.

Determining the processors shared in the execution: there are three criteria which are used to select the processors:

1. The processor is idle.

2. According to threshold, if the job queues of this processor are less than the threshold, the processor will corporate processing else no.

3. Whether the processor type is compatible with the application or not, information about problem behavior and processor type is needed.

The processors selected to corporate processing will take value; otherwise, it will take value 0. The processors status information will be sent to scheduling algorithm as a status array, as shown in formula 3.5.

$$St[i] = Status \tag{3.5}$$

where :

i is the processor id

Status : processor is selected or not

The scheduling algorithm will build a new virtual platform to simplify algorithm execution on the selected processors. These processors will map to the real processors as shown in formula 3.6.

$$VP[i] = RP[p] \tag{3.6}$$

where :

i is virtual processor

p is real processor if status is 1

For example, figure 3.3 shows the virtual processor platform mapping to the real processor platform.

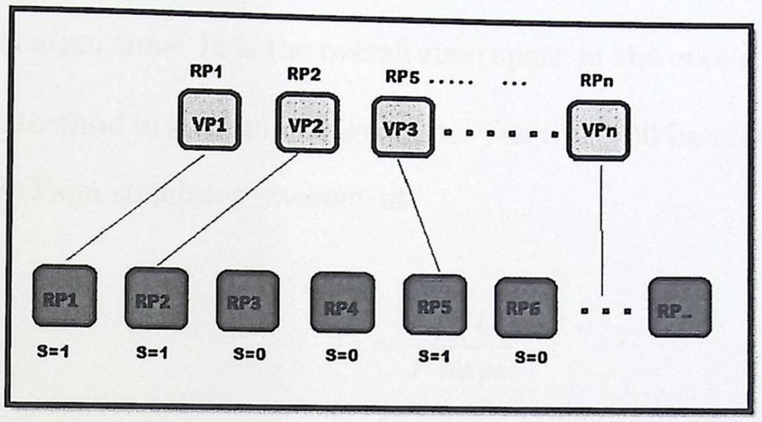


Figure 3.3: Mapping Virtual Platform to Real

The scheduling algorithm checks processors status; if the result is 1, this means that the processor will corporate the virtual platform. Table 3.1 shows the processor mapping.

Table 3.1: Virtual Platform Mapped to Real Platform

Virtual processor	Real processor	Position
VP1	RP1	1
VP2	RP2	2
VP3	RP5	5
VP4	RP7	7

Executing S on every processor: According to the collected information (about the problem sample S and virtual platform VP array), we can execute S on every selected processor. The overhead will be arising in this process; the scheduling algorithm must wait for all processors to execute because the next procedures depend on this execution information. There is only one way to decrease the overhead by decreasing the size of the sample S.

In the previous example, S will execute in the shared processors (RP1, RP2, RP5, and RP7), they incorporate to execute problem I.

Collecting execution information (Time (ETi), Power (EPi): According to the processor type and capacity, each processor will spend time and consume power to execute S. We can measure the execution time and consuming power in following steps:

1. Measure execution time: It is the overall time spent in the execution.

   Use the first method in scheduling algorithm. The time will be measured by formula 3.7 under OVPsim simulator inveroment.

   $$Ti = \frac{ICi}{P.mips} \tag{3.7}$$

   where :

   ICi is the count of instructions executed by processor i

   $P.mips$ is MIPS for Processor i

2. Measure execution power: In this process we measure the consuming power needed to execute S on every processor, as shown in formula 3.8 [48].

   $$Pi = C \times V^2 \times f \tag{3.8}$$

   where :

   Pi is the power for processor i

   C is switching capacitance

   V is processor i voltage

   f is processor i clock frequuency

For example: previous platform measured executing time for a sample by multiply

Vector (N) with square matrix (NxN) on platform consists of four processors. As shown in table 3.2, the processor specifications are not real. Table 3.3 shows the result of execution time and execution power according to formula (3.7, 3.8). IC presents the instruction count for each processor which needs to execute S. IC value depends on the processor ISA and compatibility of application with the processor type.

Table 3.2: Processors Specification Assumption

Processor	Type	MIPS X M	V	Cxk	f MHZ
RP1	ARM7	100	3.5	18	33
RP2	ARM7	200	3.7	20	36
RP5	NEC V850	100	4.1	45	50
RP7	NEC V850	150	1.7	5	20

Table 3.3: Execution Time and Power Information

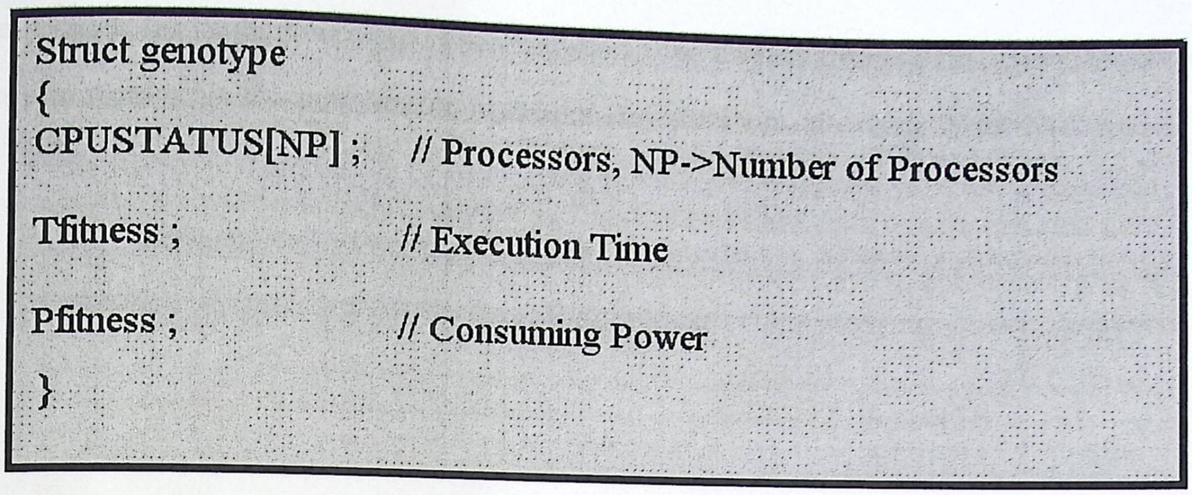
Processor	IC	Execution time micro S	Execution power volt
RP1	1881767	18817.67	7276.5
RP2	1881767	9408.835	9856.8
RP5	14514366	145143.66	37822.5
RP7	14514366	96762.44	289

Determining the objective function: We have two objective functions. The first one is to find the processor configuration s to achieve better performance according to consumed power or, the secound is find better energy according to the determined performance.

Using the GA to find the optimum processor configuration: In genetic Algorithm, we use virtual processor platforms to calculate processes, and search spaces in high-scale multiprocessors, which is very big (2^N , N number of processor). For example, if we use a platform consisting of 64 processors, the search space will be approx. ($18x10^{18}$) possible solutions.

To configure the genotype in genetic algorithm, the chromosome variable will be as shown in figure 3.4.

CPUSTATUS: present the virtual processor status: for any generation of GA, we have



```
Struct genotype
{
CPUSTATUS[NP] ;     // Processors, NP->Number of Processors

Tfitness ;          // Execution Time

Pfitness ;          // Consuming Power

}
```

Figure 3.4: Chromosome Structure

virtual processor configurations. Each processor presents one binary variable.

Tfitness: presents the overall execution time needed to execute problem I for any processor configuration. Here is the procedure to calculate Tfitness

1. Calculate processor sharing density PSD.

$$PSD[i] = S[i] \times \frac{1}{ETi} \sum (S[i] \times \frac{1}{ETi} \qquad (3.9)$$

   where :

   $PSD[i]$ is processor sharing density for Pi

   $S[i]$ processor status

2. Calculate P sharing data:

$$Ps[i] = I \times PSD[i] \qquad (3.10)$$

3. Calculate overall execution time

$$Tfitness = Max(Ps[i] \times ETi) \qquad (3.11)$$

In the previous example, table 3.4 shows the result of the processor sharing density (PSD), and table 3.5 shows the sub size (PS) which will be executed for each processor from the total problem size and the execution time needed to solve this sub size. These are the results for the sum virtual processor configuration.

Table 3.4: Processors PSD

VP config				ET m.s				PSD			
1	2	3	4	1	2	3	4	1	2	3	4
1	0	0	1	1.9	1.9	14.5	14.5	0.837189	0	0	0.162811
1	1	0	1	1.9	1.9	14.5	14.5	0.313041	0.626081	0	0.060878
0	1	1	0	1.9	1.9	14.5	14.5	0	0.939122	0.060878	0
1	0	1	1	1.9	1.9	14.5	14.5	0.755218	0	0.097913	0.146869

Table 3.5: PS and Executing Time for Each Processor

VP config				ET m.s				PSD			
1	2	3	4	1	2	3	4	1	2	3	4
1	0	0	1	837	0	0	163	15750390	0	0	15772278
1	1	0	1	313	626	0	61	5889931	5889931	0	5902509
0	1	1	0	0	939	61	0	0	8834896	8853763	0
1	0	1	1	755	0	98	147	14207341	0	14224079	14224079

The amount of consuming power needed by each processor (Pfitness) needed to execute problem I for any processor configuration is calculated by using formula

$$Pfitness = \sum (S[i] \times PEi) \tag{3.12}$$

where:

S[i] Processor Status (shair/not shair).

PEi Consuming Power for Processr i.

Distribute problem size 'I': according to the processor configuration done in step 6, The GA results present the virtual processor configurations that achieve objective fitness. Returning to the virtual processor map, we can present the real processor configuration and the shared value for each processor.

3.4 Genetic Algorithm

Genetic Algorithm was developed by john Holland [33] at University of Michigan. It directs search Algorithms based on the mechanics of biological evolution, and a subfield of artificial intelligence that involves combinatorial optimization problems based on heuristic search methods of exploring all the possible solution to get the optimal one (sub-optimal may be sufficient [17]. In many cases, it will be time consuming to get the optimal solution. For GA efficiency, it is widely used today in various fields like engineering, science, and business.

3.4.1 GA Details

We must represent a solution for any problem as a genome (or chromosome). Figure 3.4 represents a problem as a chromosome. The genetic algorithm then creates a population of solutions and applies genetic operators, such as mutation and crossover, to evolve the solutions in order to find the best one(s).
Several steps will be used: determining the value of population size, the minimum number of processors, the crossover probability (XOVERP) between (0.5 an 1), and the mutation probability (MUTP) between (0.1 and 0.2).

Step 1: ***Initializing Population***: enter random processors configuration for each population individually, as shown in pseudo code:

```
Begin
Step1.  Determine the number of individual population (POPSIZE) according to
population size and the number of processor (NOOFCPU = length(VP))
Step2.  For 1 to POPSIZE
Step3.  For 1 to NOOFCPU
Step4.  Enter random value (0 or 1) for each processor (S[i])
Step5.  If (S[PVP] = 0) set S[PVP] = 1 // constraint 1
End If
Step6.  If ($\forall S[i] = 0$) repeat step 3 to 5 // constraint 2
End If
End For
End For
End
```

Example (3-2) shows the effect of population size on executing matrix-victor multiplication by 8 processors (4 ARM with MIPS =100, 150, 200, and300, 4 NEC V850 with MIPS =100, 250, 250, and 350).

Note: Population size (POPSIZE parameter) affects the performance of GA Algorithm. Figure 3.5 shows GA performance according to the value of POPSIZE. This figure represents the average of the overall execution time for the previous Example (3-2) with different value of POPSIZE and fixed value of XOVERP (0.8) and MUTP (0.15).

Step 2: ***Evaluating Population***: Calculate the overall execution time (Tfitness) and the overall execution consuming power (Pfitness) for each individual, as shown in the pseudo code:

```
Begin
Step1.  Determine the number of individual population (POPSIZE) according to
population size and the number of processor (NOOFCPU = length(VP))
Step2.  For 1 to POPSIZE
Step3.  For 1 to NOOFCPU
Step4.  Enter random value (0 or 1) for each processor (S[i])
Step5.  If (S[PVP] = 0) set S[PVP] = 1 // constraint 1
End If
Step6.  If ($\forall S[i] = 0$) repeat step 3 to 5 // constraint 2
End If
End For
End For
End
```

Example (3-2) shows the effect of population size on executing matrix-victor multiplication by 8 processors (4 ARM with MIPS =100, 150, 200, and300, 4 NEC V850 with MIPS =100, 250, 250, and 350).

Note: Population size (POPSIZE parameter) affects the performance of GA Algorithm. Figure 3.5 shows GA performance according to the value of POPSIZE. This figure represents the average of the overall execution time for the previous Example (3-2) with different value of POPSIZE and fixed value of XOVERP (0.8) and MUTP (0.15).

Step 2: ***Evaluating Population***: Calculate the overall execution time (Tfitness) and the overall execution consuming power (Pfitness) for each individual, as shown in the pseudo code:

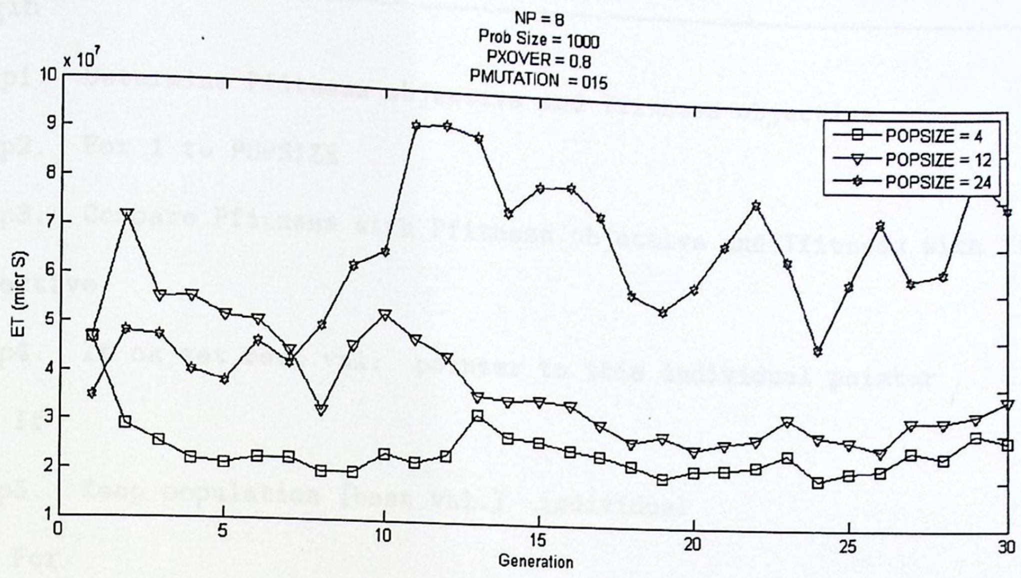


Figure 3.5: Effect of Population Size on Search Accuracy in GA

```
Begin
Step1.  Determine the execution time (ETi) and Execution Power (PEi) for each
processor corporate in processing problem size I.
Step2.  For 1 to POPSIZE
Step3.  For 1 to NOOFCPU
Step4.  Calculate sharing density:  as formula  3.9
Step5.  Calculate sharing size:as formula  3.10
Step6.  Calculate over all execution time:  as furmula  3.11
Step7.  Calculate over all consuming power (Pfitness)
End For
End For
End
```

Step 3: ***Keeping the best***: Determine the best individual for the population which is compatible with the objective, as shown in the pseudo code:

```
Begin
Step1.  Determine Pfitness objective and Tfitness objective
Step2.  For 1 to POPSIZE
Step3.  Compare Pfitness with Pfitness objective and Tfitness with Tfitness
objective
Step4.  If ok set best val.  pointer to this individual pointer
End If
Step5.  Keep population [best val.]  individual
End For
End
```

Step 4: ***Selection eliminates*** the individuals' population; these are far from objective fitness, as shown in the pseudo code:

```
Begin
Step1.  Determine objective search (fitness)
Step2.  For n= 1 to POPSIZE
Step3.  Calculate relative fitness
Step4.  Calculate cumulative fitness
End For
Step5.  For n= 1 to POPSIZE
Step6.  If (random(between(0.0 and 1.0)) < cfitness[n])
Step7.  New population = population[n]
Step8.  Population = new population
End If
End For


End
```

Step 5: ***Crossover***: It creates new individual population by selecting two individual parents from population and creating two new individual populations, as shown in the pseudo code:

```
Begin
Step1.  For n = 1 to POPSIZE
Step2.  Initialize X=0
Step3.  If (random(between(0 and 1)) < XOVERP)
Step4.  ++X End if
Step5.  If (X mod 2 = 0)
Step6.  Position = random(between 1 and NOOFCPU) End if
Step7.  For p = 1 to position
Step8.  Swap(S[n] and S[p]
End For
Step9.  If (S[PVP] = 0) set S[PVP] = 1 End if
Step10.  If (∀S[i] =0) repeat step 6 to 9 End if
Step11.  Else m= n
End For
End
```

Note: To show the effect of XOVERP, Figure 3.6 represents the average of the overall execution time for the previous Example (3-2) with a different value of XOVERP and fixed value of POPSIZE (8) and MUTP (0.15).

Step 6: ***Mutation***: this process changes the status of some processors of some individual population, as shown in the pseudo code:

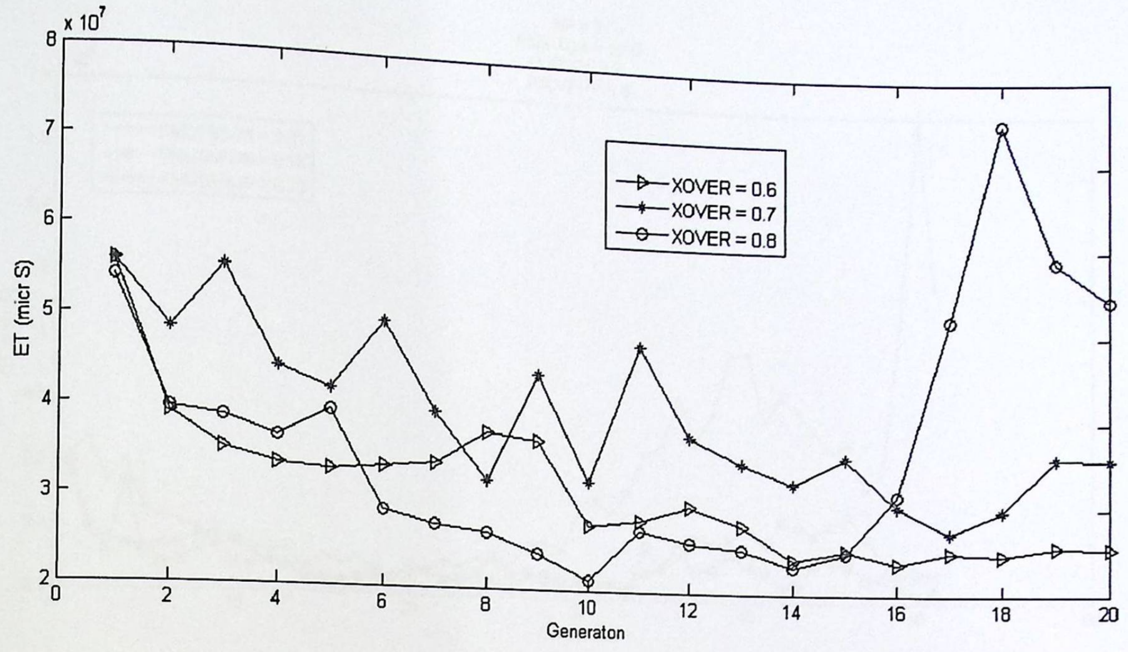


Figure 3.6: Effect of Crossover Size on Search Accuracy in GA

```
Begin
Step1.  For 1 to POPSIZE
Step2.  For n = 1 to NOOFCPU
Step3.  If (random(between(0.0 and 1.0)) < MUTP)
Step4.  If (S[n] = 0) S[n] = 1 else S[n] = 0 End If
Step5.  If (S[PVP] = 0) set S[PVP] End If
End If
End For
End For
End
```

Note: To show the effect of MUTP, Figure 3.7 represents the average of the overall execution time for the previous Example (3-2) with a different value of MUTP and fixed value of POPSIZE (8) and XOVERP (0.6).

Step7: ***Evaluating Population***: Calculate the overall execution time (Tfitness) and the overall consuming power (Pfitness) for each individual (the same procedure in step2).

Step 8: ***Keeping the best***: Determine the best individual for the population which

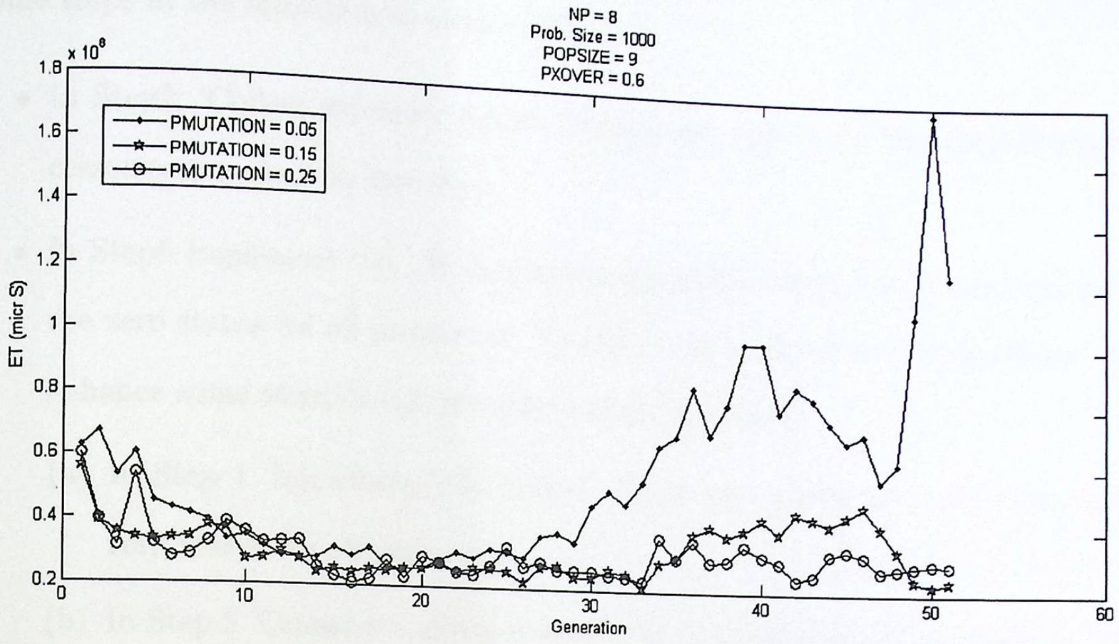


Figure 3.7: Effect of Mutation Size on Search Accuracy in GA

is compatible with the objective, as shown in the pseudo code:

```
Begin
Step1.  For population [best val.]  set prev.  best.  valu = best val.
Step2.  Same Procedure in step 3 end for
Step3.  If (population [best val.]  worst than population [prev.  best.
valu.])
Step4.  Population [best val.]  = population [prev.  best valu.]  end if
End
```

Step 9: ***New Generation***: Repeat steps 4 through 7 until you achieve the objective (Tfitness and Pfitness value).

3.4.2 Scheduling Algorithm Constraints

In general, the scheduling algorithm has two constraints:

1. It prevents the solution that produces virtual processor configuration with 0 status for all processors. This configuration means that the overall execution time is close to zero (optimal value). To avoid this status in the algorithm, we must introduce

some steps in the algorithm as shown below:

- In Step2, 'Collect processor status information', wait until (at least) one processor corporate is in execution.
- In Step6 Implement GA', It does not propose any configuration solution with the zero status for all processors. To avoid that problem in GA algorithm, we enhance some steps in GA procedure as shown below:
  (a) In Step 1 'Initializing Population', reject any individual population with zero status for all processors.
  (b) In Step 5 'Crossover', check new individual population. If the result is "individual population with zero status for all processors", repeat the process with a new crossover probability value.
  (c) In Step 6 'Mutation', check new individual population. If the result is "individual population with zero status for all processors", repeat the process with a new mutate probability value.

2. The existing primary processors must run all time. In this situation, the scheduling algorithm prevents any configuration solution that does not include the primary processors. To make sure of that process, we must make some enhancements in the scheduling algorithm as shown below:

   - In Step2 'Collect processor status information', the most important step make sure that the corporate of primary processors in processing platform:
     (a) Determine primary processor position.
     (b) Check virtual processor map onto real processors. If primary processor is present, save the virtual processor pointer in variable (PVP).
     (c) If primary processor is not present, add new virtual processor map with the primary processor and save the virtual processor pointer in variable (PVP).

- In Step6 Implement GA', it does not propose any configuration solution that does not consist of PVP pointer with status 1. To avoid that problem in GA Algorithm, we enhance some steps in GA procedure as shown below:
  (a) In Step 1 'Initialize Population': After initialize population, check the status of PVP position. If the status is 0, alter it to 1.
  (b) In Step 5 'Crossover': In new individual population, check the status of PVP position. If the status is 0, alter it to 1.
  (c) In Step 6 'Mutation': In the new individual population, check the status of PVP position. If the status is 0, alter it to 1.

Chapter 4

Experiments and results

This chapter implements some applications on our algorithm to validate it. The algorithm is split into a number of steps. Each step is to be validated by itself. After validating all steps, a number of experiments that use standard benchmarks will be implemented. The chapter is organized as follows: section 4.1 describes the implementation environment and explains the hardware and software used in the implementation, section 4.2 explains the tools used in the experiments, Section 4.3 describes the challenges and difficulties in the implementations, and finally, section 4.4 describes the experiments used to validate the proposed algorithm.

4.1 Environment

To validate the proposed algorithm, the researcher used special software that can simulate the real environment. In addition, the researcher used programs and tools to show the execution results and tools to analyze these results. Implementation and analysis were made using an environment with the following specifications:

- Personal computer: Compaq P4 dual core processor (Intel CoreDuo CPU T7300 2.00 GHz 2.00 GHz), RAM 3.00 GB.

- Operating system: Windows 7 Professional 32 bit.
- OVPsim simulator.
- MSYS / MinGW Environment to represent linux (upuntu) under windows7.
- Matlab R2007b.
- Excel 2003.

4.2 Tools

There are two types of tools used in the implementation. The first type is OVPsim simulator software to simulate high scale HMP architecture hardware; while the second type is Matlab to analyze and draw the results.

4.2.1 OVPsim Simulator

Open Virtual Platform simulator (OVPsim) software is used because no kit is available to create high scale HMP architecture hardware. The OVPsim simulator was developed by Imperas Company Partner with 30 famous companies and organizations like (Tensilica, MIPS Technologies, CircuitSutra, and Cadence) [41].

OVPsim Features: OVPsim is selected because it has the following properties [3]:

- Easy to create complex processors.
- Easy to create virtual platforms of many processors.
- Easy to create shared and local memories.
- Collect a large number of libraries for processors and peripheral modules.
- Simulations instructions are accurate and very fast.

- Used for applications, operating system and embedded software.
- Efficient and complete system environment.
- Interfacing by C language.

OVPsim Setup: The installation of OVPsim simulator on personal computer needed to do some steps:

- Registration with Imperas Company.
- Download OVPsim simulator and processors library.
- Request of license key from the company to activate the simulator for a certain period.

Processors Supported by OVPsim Simulator: The company provides new processors library from time to time. Until now, the available processors are:

- OpenCores OR1K
- ARM (Arm10, Arm11, Arm7, Arm9, ARM7TDMI, ARM1136J-S, ARM Cortex-m3, and ARM Cortex-A8/A9)
- MIPS32 (MIPS4KEm, MIPS24KEc, MIPS34Kc, MIPS74Kc, MIPS1004Kc, M14K, and MIPS32 1074Kc)
- ARC (Arc6xx and Arc7xx)
- NecV850
- PowerPc32
- SparcV8

4.2.2 MSYS / MinGW Environment

GNU Operating system is a default environment required to build model with OVPsim tools. To create models under windows operating system, it is needed to validate windows environment using minGW and MSYS tools.

4.2.3 Matlab R2007b

MATLAB is a highly tuned mathematical environment. It can execute simple, complex operations on matrices, vectors to solve different problems and visualize the results with very little code in an interactive development environment. This combination has made it a standard tool for scientists and engineers all over the world. MATLAB designed to support executables written in C or FORTRAN. Those executables are known as MEX-files, where MEX stands for MATLAB Executable. The MEX API is available by including special MATLAB header files in C. The resulting MEX-files are equally accurate and much more efficient than the corresponding MATLAB functions [45].

The header files define MATLAB specific functions for many built-in C functions such as malloc (memory allocation), free (memory de-allocation) and printf (print to standard out). These functions have names such as mxFree, mxMalloc and mexPrintf, respectively.

4.3 Challenges and Difficulties during Implementations

In general, the research in many processor fields is a difficult task. It requires high potential tools like many processors kit hardware and test program (benchmarks) and supercomputing environment. Many difficulties were faced. The most important ones are:

- Lack of computer labs. They do not have processors kit hardware or simulator software. This has been the cause for spending a long period of time for finding a simulator software and learning how to use it.

- No availability of heavy computers that can execute operations quickly. The researcher used his own personal computer for his study experiments. This takes long periods of time to do all the experiments.

4.4 Experiments and Results

Experiments were divided into three phases. Each phase used to validate a main part of the proposed algorithm:

- **Phase1**: Experiments to evaluate procedures that used to estimate overall execution time.
- **Phase2**: Experiments to show scheduling algorithm overhead.
- **Phase3**: Experiments to schedule problem on all processors and evaluate the proposed algorithm using standard benchmarks.

4.4.1 Phase 1. Accuracy of Estimation Execution Time

The goal of this experiment is:

- Estimate overall execution time and measure real execution time on different platforms.
- Calculate the difference (error) between the estimated and the execution time.
- Study the effect of number of processors in the platform on the error value.

Environment and Tools:

- OVPsim Simulator: It is used to build HMP architectures, and to run the application on the architectures platform.

- Matlab 2007b: It is used to calculate and estimate execution time and show the results by analyzing execution information.
- Excel Files: They are used to store execution information. The (xls) file passes information between the simulator and matlab2007.

Platform: The OVPsim simulator is the environment that used to build different HMP architecture platforms. Each platform consists of various processors. These processors are four main types as shown in table 4.1, each of which has its particular performance in the platform.

Table 4.1: Processors Specifications

Processor	Type	ISA
1	POWERPC	32 bit
2	ARM	32 bit
3	MIPS32	32 bit
4	OR1K	32 bit

Application Domain: Most of benchmark methods focus on a linear algebra equation that used LU decomposition technique as shown in formula (4.1,4.2), the built application used to solve LU decomposition on high scale HMC processors.

$$XA = Y \tag{4.1}$$

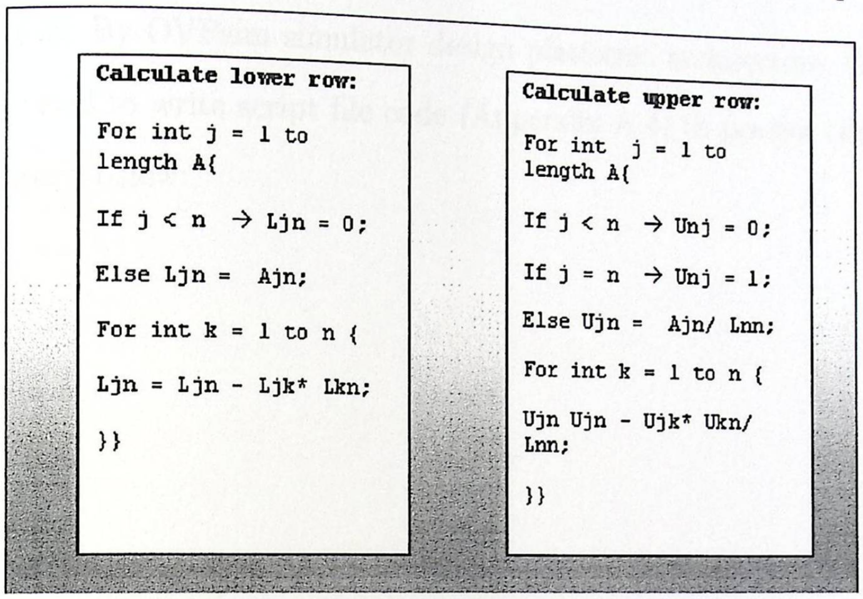
$$LU = A \tag{4.2}$$

The scheduling Algorithm will decompose matrix A. Distribute it on all chosen processors, and will estimate the time needed to solve LU decomposition. Matrix A is stored in share memory as a domain decomposition database. The database decomposition is based on rows distributed to processors.

Application Behavior:

- *Problem Size*: Size of database is presented as a matrix length (number of rows). In LU application the size of A is equal 1000.

- *Sample Size*: One row from matrix A is needed to calculate one row for both L and U at the same time.
- *Computation*: All processors on platform will execute the same process. Computation pseudo code shown in figure 4.1 is used to calculate LU decomposition:



```
Calculate lower row:

For int j = 1 to
length A{

If j < n  → Ljn = 0;

Else Ljn =  Ajn;

For int k = 1 to n {

Ljn = Ljn - Ljk* Lkn;

}}
```

```
Calculate upper row:

For int  j = 1 to
length A{

If j < n  → Unj = 0;

If j = n  → Unj = 1;

Else Ujn =  Ajn/ Lnn;

For int k = 1 to n {

Ujn Ujn - Ujk* Ukn/
Lnn;

}}
```

Figure 4.1: Pseudo Code to Calculate LU Decomposition

1. **Experiment Procedures:** The following steps are used to show the results:
   - Design application.
   - Design platform architecture.
   - Determine application behavior, size and sample.
   - For each platform
     (a) Execute the sample on all processors and collect runtime information.
     (b) Estimate overall execution time for all processors configuration.
     (c) Measure real overall execution time for all processors configuration.
     (d) Compare the estimation overall execution time to measurement execution time.
     (e) Calculate the error between two results.

- Study the effect of processor count on the result.

2. **Experiment Components and Program Files :** This section explains all script files that are designed to apply the experiment. These files are explained as run sequence:

Platform.c: By OVPsim simulator design platforms architecture, C language interface is used to write script file code (Appendix A.4) to present the pseudo code that is shown below:

```
Begin
Step1.  Initialization ; enabling verbose mode to get statistics at end
Step2.  For x =1 to Ps count
Step3.  Create Px; create a processor instance: name, type, id, attributes,
address bit Etc
Step4.  Create Busx; create the processor busses and determine, and address
bits
Step5.  Create Memx; create processors memories, and detriment size and
privilege(w ,r, wr, and wrx )
Step6.  Create Share Mem end for; create share memories, and detriment size
and privilege(w ,r, wr, and wrx )
Step7.  For x = 1 to Ps count
Step8.  Connect Px to Busx; connect the processors onto the busses
Step9.  Connect BusX to Memx; connect local memories onto individual processor
buses
Step10.  Connect Busx to Share Mem; connect the shared memory onto all the
local buses
Step11.  Load application to memory; load the processor object file
Step12.  Simulate platform; simulate the platform
Step13.  Terminate ; free processors and memories
End for
End
```

Application1.C: According to C language interface, design application script file code is used to execute the sample on all processors in platform. Below, shows pseudo code for the program that is described in (Appendix B.3).

```
Begin
Step1.  For x = 1 to Ps count...; loop to cover all processors in platform
Step2.  Load sample; Store sample onto processors memories
Step3.  Exec.  Sample; Execute sample on every processors
Step4.  Export info.; store execution information to excl file
End for
End
```

Application2.C: By C language, design application script file is used to decompose and distribute the problem on all processors in platform according to algorithm calculation that are described in chapter three section 3.3. This file also used to measure average overall execution time. Below, shows pseudo code for the program that is described in (Appendix B.4).

```
Begin
Step1.  For x = 1 to Ps count; loop to cover all processors in platform
Step2.  Load sample; Store sample onto processors memoreis
Step3.  Exec.  Sample;  Execute sample on every processors
Step4.  Measure Exec T ETx ; end for; measurement execution time for every
processor
Step5.  For x = 1 to No.  of Config.  ; all processors configuration
Step6.  Cal.  PSD; calculate sharing density according to ETx
Step7.  Disrepute data; decompose data and disrepute to processor according to
PSD and problem size (PSD x I)
Step8.  Measure Avg ET.; end for; measurement overall execution time for each
configuration
Step9.  Export info.; store execution information to excl file
End
```

Make File: Design compiler script file is used to compile platform script file and application script file. Below, shows pseudo code for a compiler script file that is described in (Appendix C.5). After compilation process under minGW environment, many files are created as shown in table 4.2.

```
Begin
Step1.  Compile platform; build the Platform executable file under OVPsim library
Step2.  Compile application; build the Application execution compatible with ARM7
Step3.  Compile application; build the Application execution compatible with MIPS32
Step4.  Compile application; build the Application execution compatible with POWERPC
Step5.  Compile application; build the Application execution compatible with OR1K
End
```

Table 4.2: LU Application Files

Before compile	After compile
Platform.c	Platform.dll Platform.windows32.exe
Application1.c	Application1.ARM.ELF Application1.MIPS32.ELF Application1.OR1K.ELF Application1.V850.ELF
Application2.c	Application2.ARM.ELF Application2.MIPS32.ELF Application2.OR1K.ELF Application2.V850.ELF

Runall.m: Design Matlab script file is used to analyze information that is collected when executing application1 and application2 files. This file is also used to compare and display the results in figures and xls file tables. Below, shows pseudo code that is used to create Matlab script file (Appendix C.4).

```
Begin
Step1. Import ETi; import execution time for the sample on processors from excel
Step2. Import overall ET; import overall execution time that measured from excel file
Step3. Determine No. Ps; extract the number of processors in platform
Step4. Create Ps configuration Space; from 1 to config. space
Step5. Cal. PSD; calculate sharing density (∑ PSD = 1) for each Processors.
Step6. Disrepute data; according to PSD and Size I for each processor (I x PSD)
Step7. Cal. Avg. Overall ET; calculate average overall execution time for each processors configuration
Step8. Comparisons; compare measurement overall execution time with estimation overall execution time
Step9. Show comparisons result
End
```

3. **Experiment parts**

In this section, the experiment is split into six parts according to a processors count in platform. This strategy for the splitting is adopted because the platform in real world presents a computer machine that has static and fixed number of processors. The experiment in each platform has the same procedure steps. The following parts show and execute these procedure steps:

(a) **Part 1: Create HMP Architecture Platforms:** According to Platforms Processors count shown in tables (4.3, 4.4, 4.5, 4.6, 4.3, 4.7, 4.8), the sex platform are created using OVPsim simulator.

(b) **Part2: Execution Information:** For each platform, the sample problem

Table 4.3: Platform Consists of 4 Processors

Processor	Type	MIPS
P1	ARM	100
P2	POWERPC 32	100
P3	MIPS32	100
P4	OR1K	100

Table 4.4: Platform Consists of 8 Processors

Processor	Type	MIPS
P1 - P2	ARM	100 - 200
P3 - P4	POWERPC 32	100 - 200
P5 - P6	MIPS32	100 - 200
P7 - P8	OR1K	100 - 200

Table 4.5: Platform Consists of 16 Processors

Processor	Type	MIPS+100
P1 - P4	ARM	100 - 400
P5 - P8	POWERPC 32	100 - 400
P9 - P12	MIPS32	100 - 400
P13 - P16	OR1K	100 - 400

Table 4.6: Platform Consists of 32 Processors

Processor	Type	MIPS+50
P1 - P7	ARM	100 - 400
P8 - P15	POWERPC 32	100 - 400
P16 - P23	MIPS32	100 - 400
P24 - P32	OR1K	100 - 400

Table 4.7: Platform Consists of 64 Processors

Processor	Type	MIPS+25
P1 - P15	ARM	50 - 425
P16 - P31	POWERPC 32	50 - 425
P32 - P47	MIPS32	50 - 425
P47 - P64	OR1K	50 - 425

Table 4.8: Platform Consists of 128 Processors

Processor	Type	MIPS+25
P1 - P31	ARM 7	2x (50 - 425)
P32 - P63	POWERPC 32	2x (50 - 425)
P64 - P95	MIPS32	2x (50 - 425)
P96 - P128	OR1K	2x (50 - 425)

is executed by implementing application1 program on all processors in the platform. Figures (4.2, 4.3) show the execution time for all processors in platform consist of 4 and 8 processors.

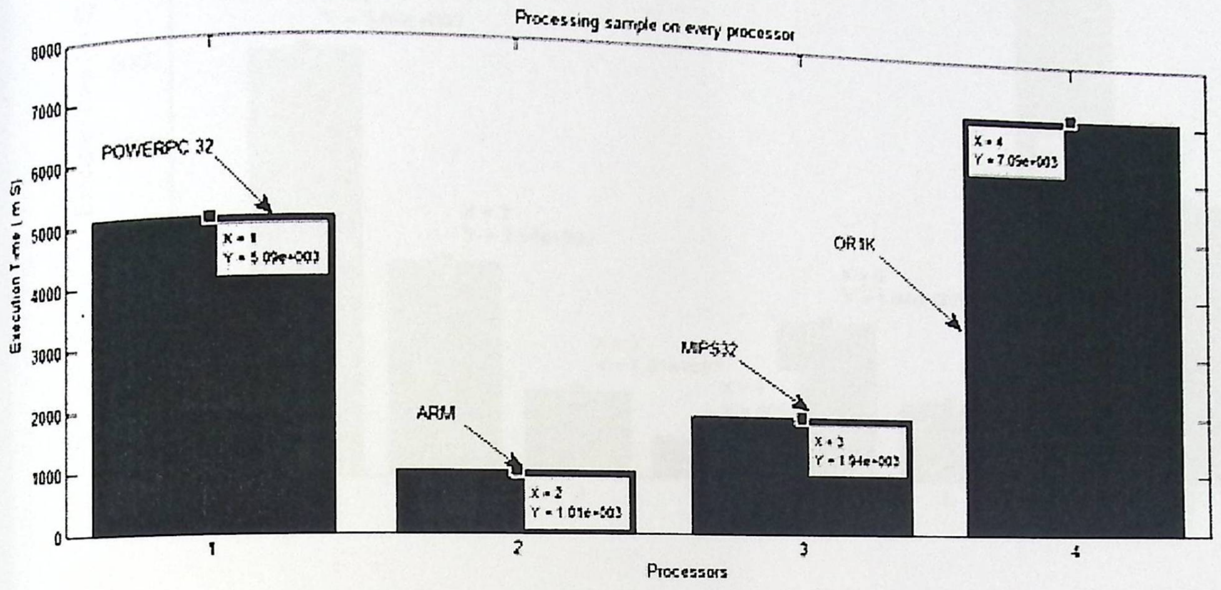


Figure 4.2: Execution Time for One Sample among 4 Processors

In figure 4.3, we notice that, the execution time differs from one processor to other, in spite that, some of these processors are same type (use same ISA), but it differs in execution time due to difference in performance (MIPS value). Also are similarities in information for the experiment that include platforms that consist of 16, 32, 64, and 128 processors.

The performance value (MIPS) as shown in the above figures is the effective factor of the execution time.

(c) **Part 3: Estimate and Measure overall Execution Time:**

In this part of the experiment, the applications responsible to estimate and measure overall execution time has two steps:

i. Run the application1 file on each platform that is responsible on measuring sample problem execution time. After execution is completed and the information is stored in excel file, the Matlab script file (runall.m) becomes responsible to calculate and estimate overall execution time for any pro-

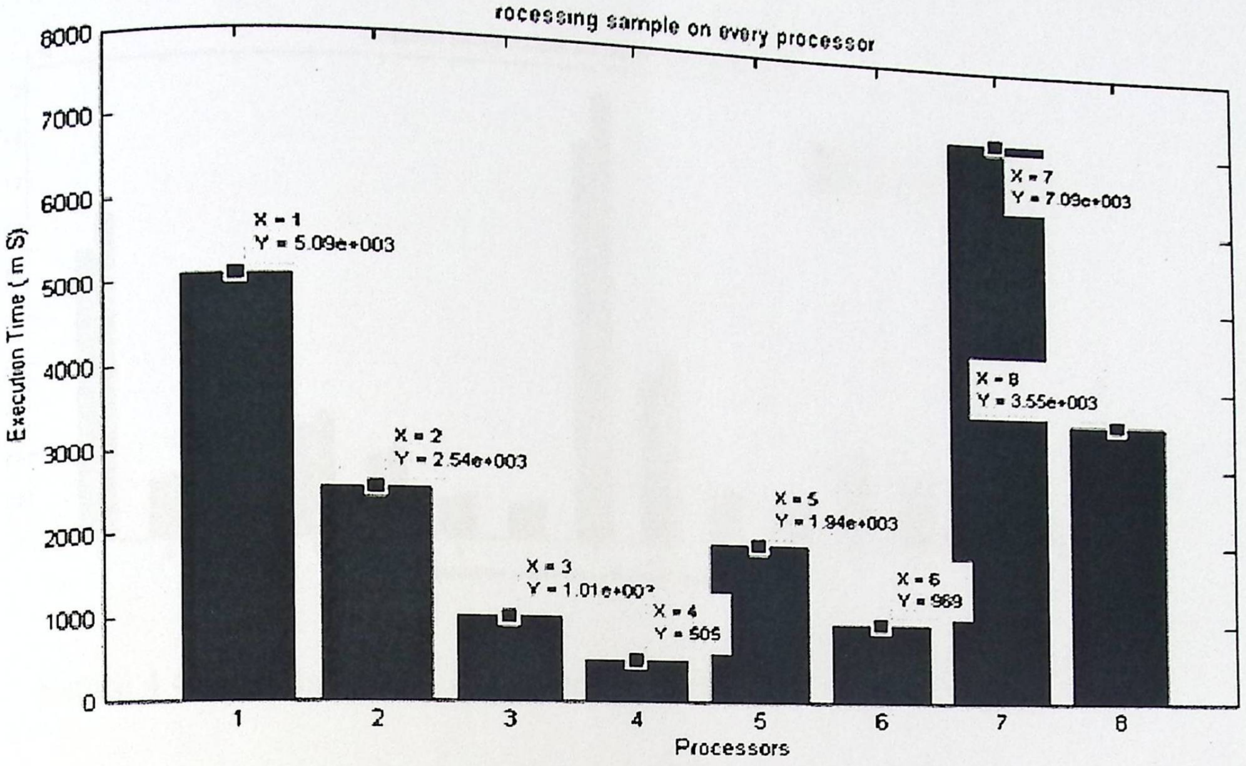


Figure 4.3: Execution Time for One Sample among 8 Processors

cessors configurations for the same platform according to formulas (3.7, 3.8, 3.9, and 3.10) in chapter three.

ii. Run application2 program that is responsible to measure real overall execution time to solve LU problem on the same processors configuration that is used in the previous step for each platform.

The processors configurations is presented as a digital stream. If the processor is shared in processing, it is presented as 1 value. If the processor is not shared in processing, it is presented as 0 value. To simplify the presentation of processors configurations shown in the figures, processors configuration stream will convert to decimal value.

Figure 4.4 shows overall execution time for all possible processors configuration ($2^4 - 1 = 15$) in platform that consists of 4 processors.

Figure 4.5 shows the overall execution time for all possible processors config-

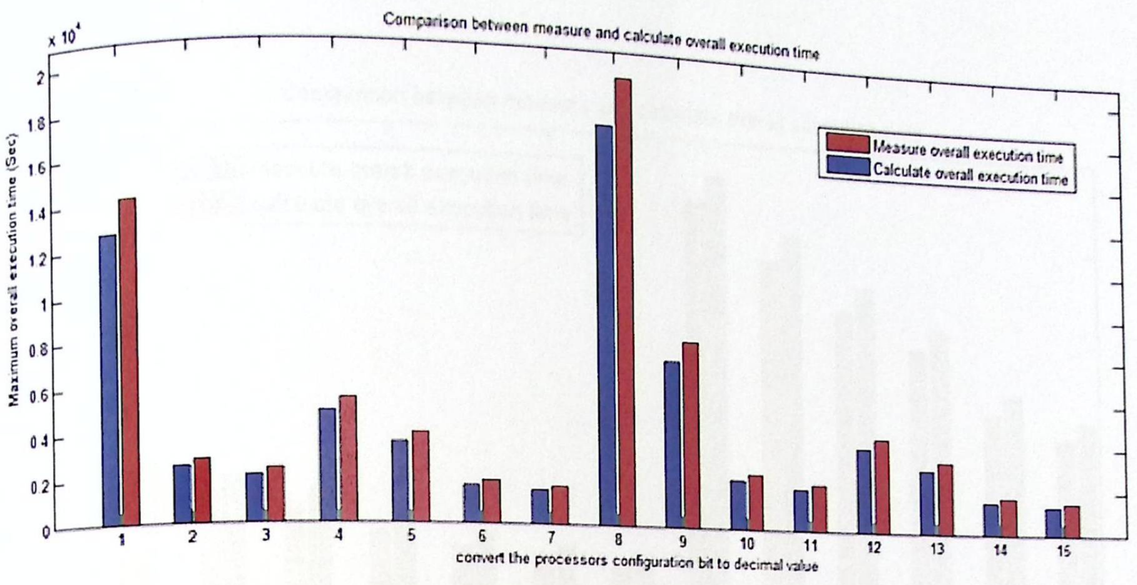


Figure 4.4: Estimated and Measured Execution Time for 4 Processors

urations ($2^8 - 1 = 245$) in platform consists of 8 processors.

In platform consist of 16 processors, the processors configurations is too large ($2^{16} = 65536$ combinations). To simplify the presentation, sample of processors configuration is chosen to show overall execution time as shown in figure 4.6. In platform consists of 32, 64, and 128 processors, the same presentation method is applied as shown in figures 4.6.

We notice that, in figure 4.6 there exist zigzag in the output information, since there are sharing or not sharing from processors in the execution, and these processors have good performance or bad. Execution time may vary from high to low if there are sharing from processors that have high performance, and vice versa.

(d) **Part 4: Calculate Percentage Error:** In this part of the experiment, the percentage error between the measured and calculated overall execution time will calculate for all platforms. This calculation uses formula 4.3.

$$Perc.Error = \frac{|CalculatedTime - MeasuredTime|}{MeasuredTime} \times 100\% \tag{4.3}$$

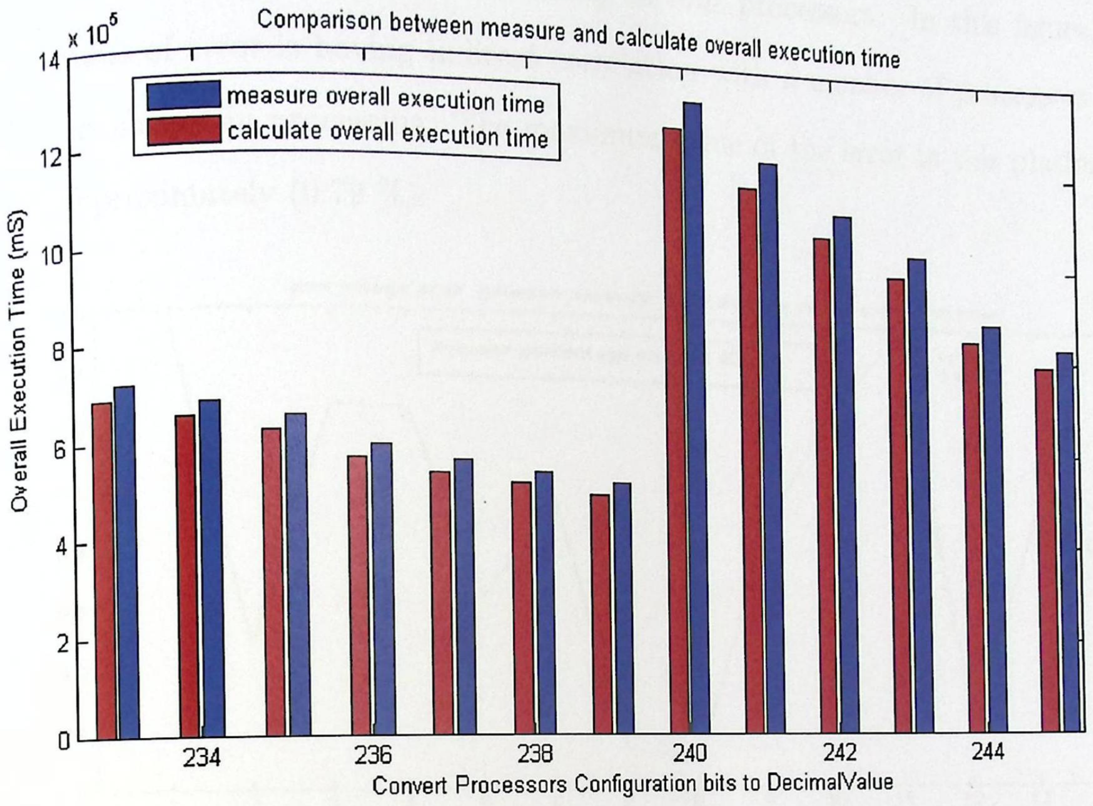


Figure 4.5: Estimated and Measured Execution Time for 8 Processors

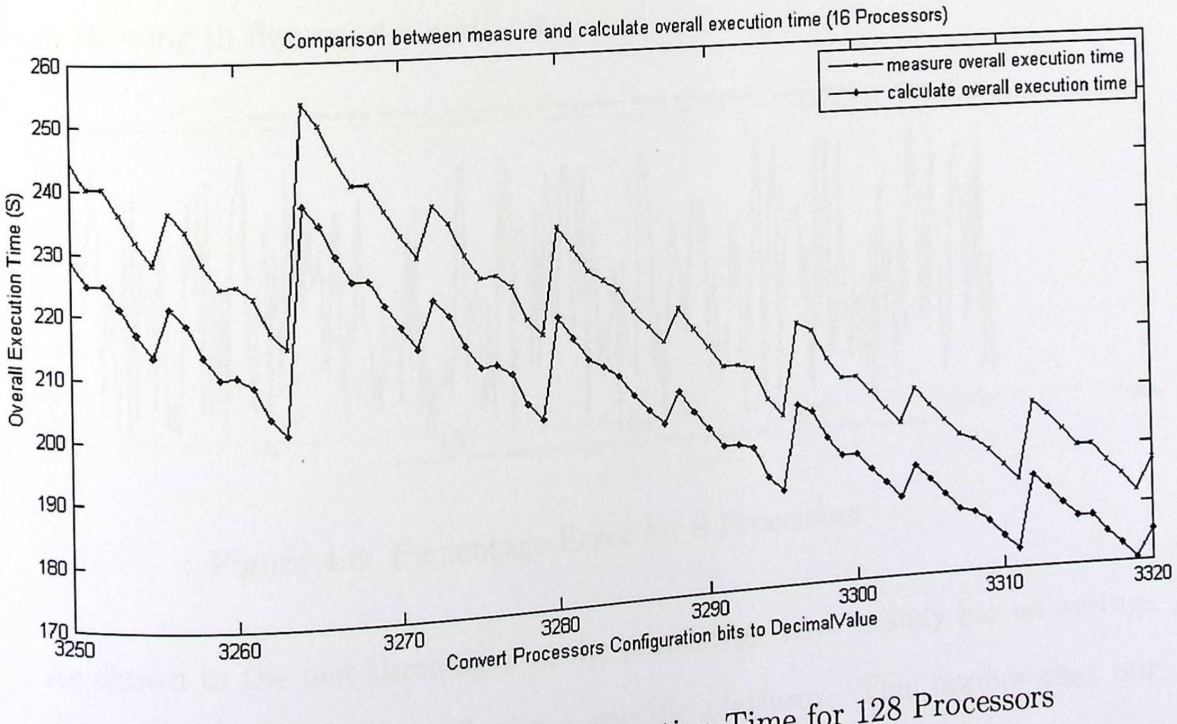


Figure 4.6: Estimated and Measured Execution Time for 128 Processors

Figure 4.7 shows the percentage error for all processors configurations, which are combined in platform consisting of four processors. In this figure, the value of error is having indirect correlation with a number of processors that are shared in processing. The maximum value of the error in this platform is approximately (0.79 %).

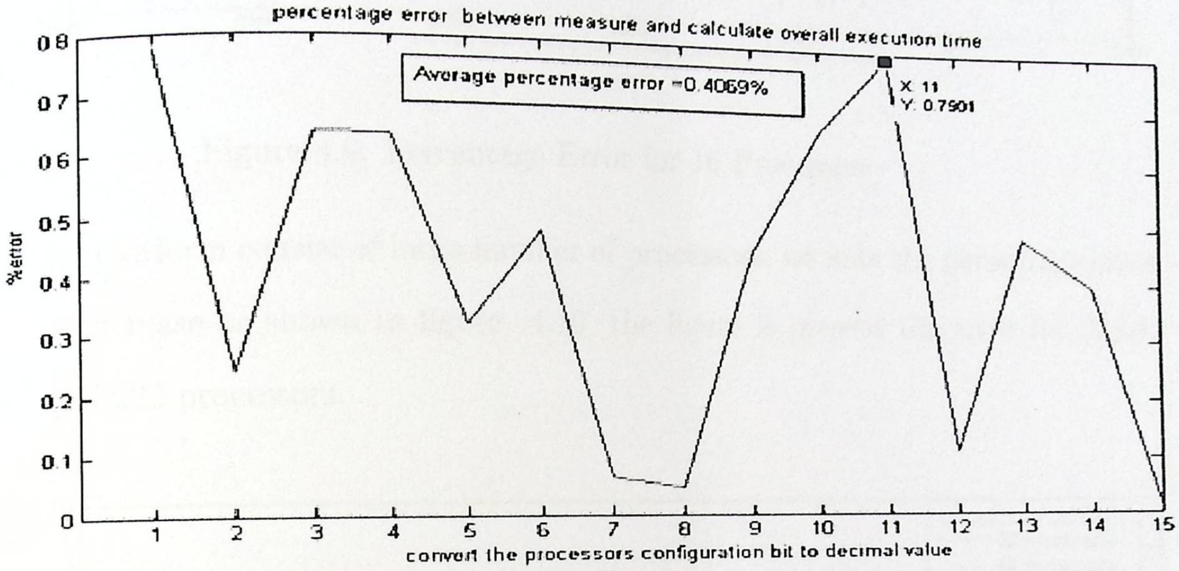


Figure 4.7: Percentage Error for 4 Processors

In platform that consists of 8 and 16, the percentage error for these platforms is showing in figures 4.8 and 4.9.

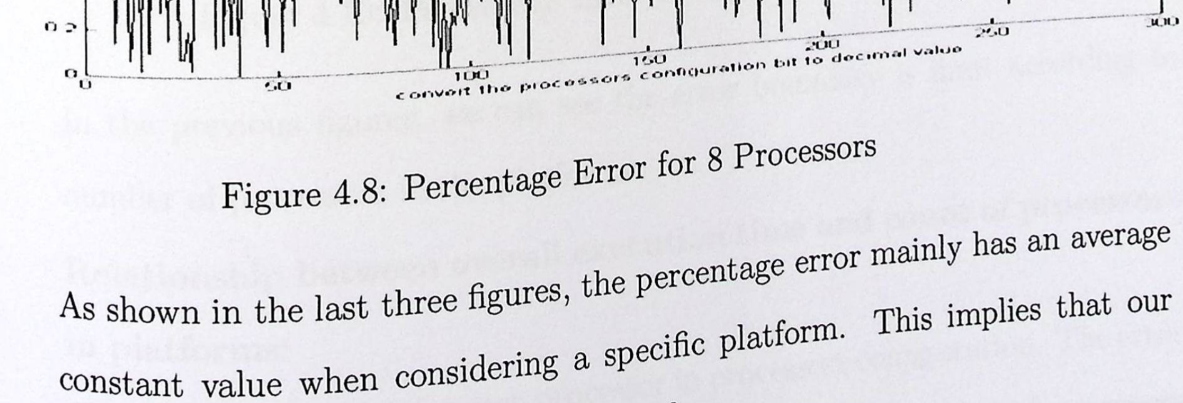


Figure 4.8: Percentage Error for 8 Processors

As shown in the last three figures, the percentage error mainly has an average constant value when considering a specific platform. This implies that our estimation of the execution time is reliable.

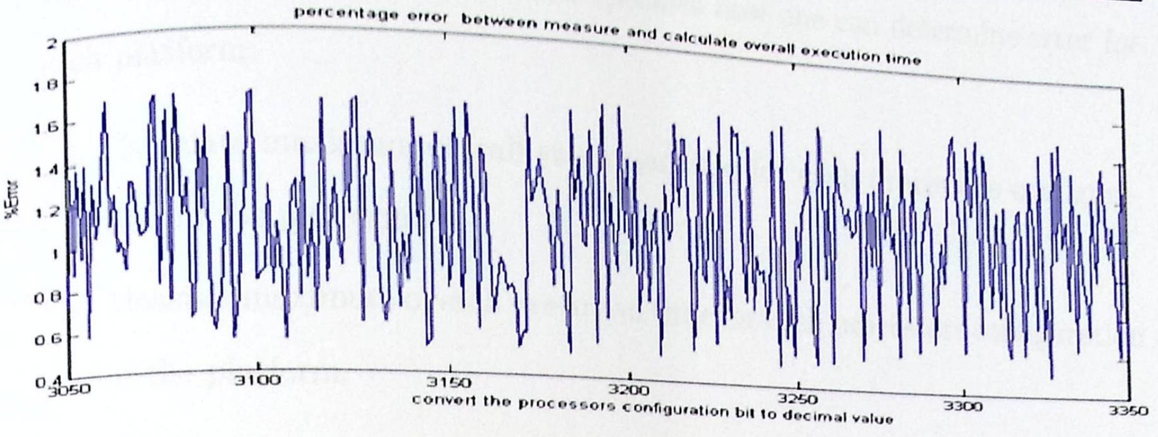


Figure 4.9: Percentage Error for 16 Processors

for platform consist of large number of processors, we note the persentage error is increase as shown in figure 4.10, the figure is present the error for 32,64, and 128 processors.

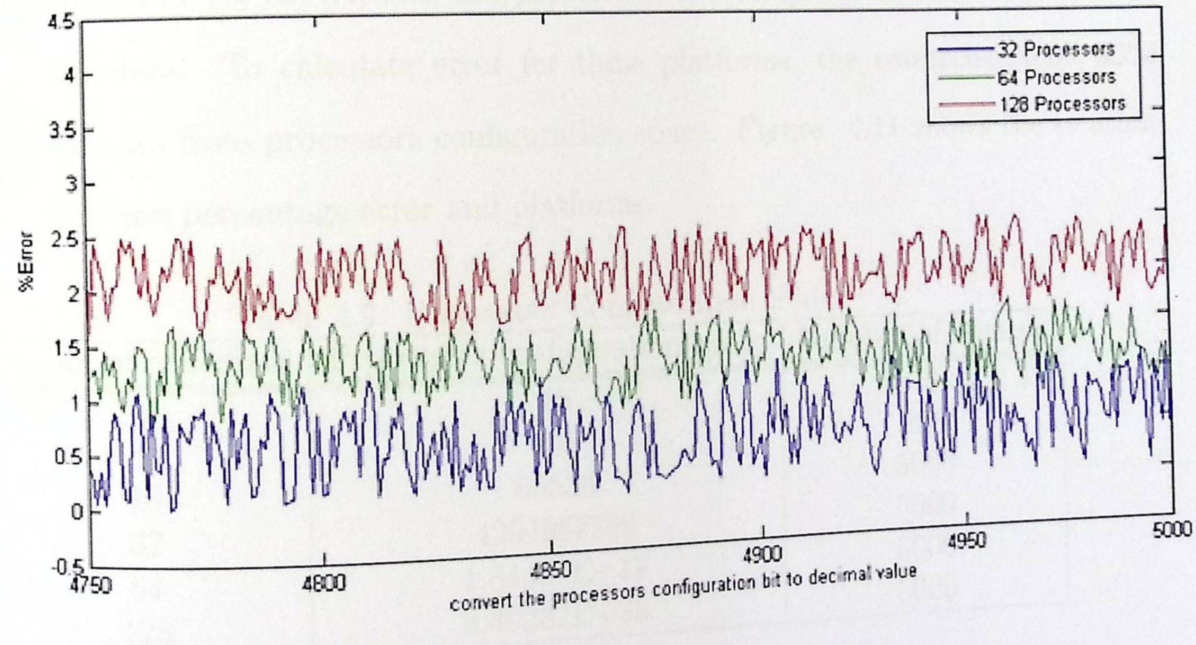


Figure 4.10: Percentage Error for 128 Processors

In the previous figures, we can see the error boundary is limit according to number of processors in the platforms.

Relationship between overall execution time and count of processors in platforms:

There is an error value for each processor in processors configuration. The error value for each platform needs to be determined to study the effect of processors scale to error. The procedure below specifies how one can determine error for each platform:

i. Calculate maximum overall execution time for each processors configuration in the platform.

ii. Measure maximum overall execution time for each processors configuration in the platform.

iii. Calculate percentage error for each processors configuration in the platform by using formula 4.3.

iv. Calculate the average and maximum percentage error for each platform.

Table 4.9 shows the number of processors configuration slides for all platforms. Because the processors configuration space is very large for platforms that consist of 16, 32, 64, and 128 processors, the computer memory cannot store the data. To calculate error for these platforms, the researcher took 5000 samples from processors configuration space. Figure 4.11 shows the relation between percentage error and platforms.

Table 4.9: Processors Configuration Slides

Count of processor	Processors configuration space	Number of Sample
4	15	16
8	255	256
16	65536	5000
32	4294967296	5000
64	1.84467E+19	5000
128	3.40282E+38	5000

From figure 4.11, we can conclude that there is an upper bound for the percentage error in estimating execution time.

4.4.2 Phase 2: Overhead

The goal of this phase is to study the effect of implementation behavior characteristic such as the sample size and problem size, to present the relation between problem sample

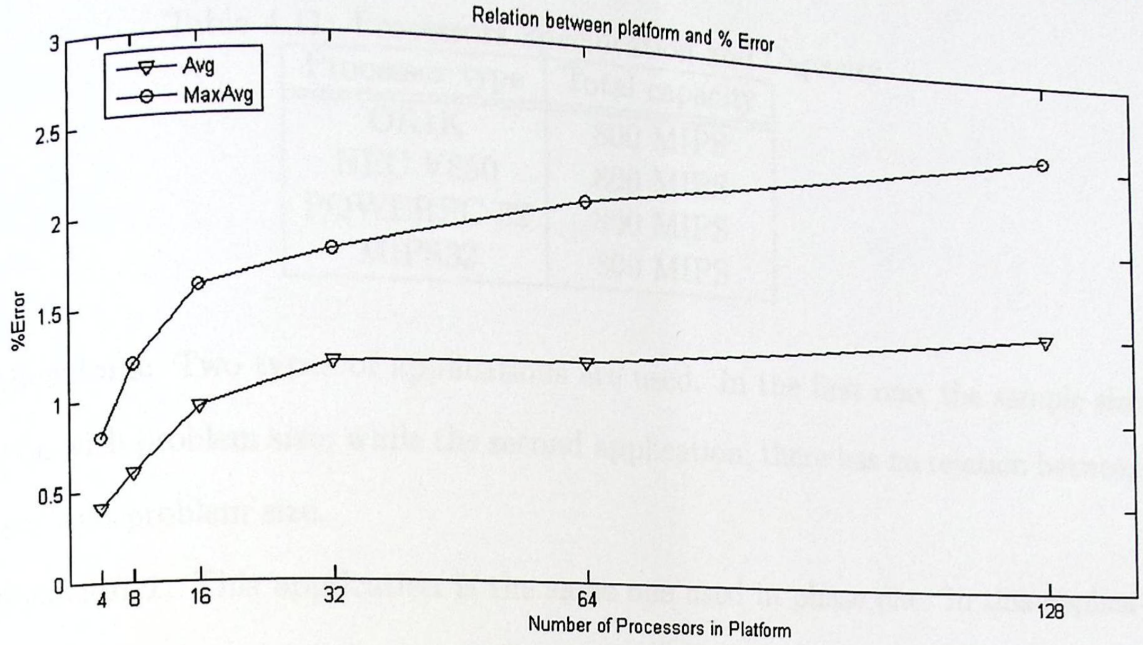


Figure 4.11: Percentage Error Relative to Number of Processors

size and problem size, and to study the effect of this relation on execution time overhead.

Tools:

1. OVPsim simulator.
2. Matlab 2007 b.
3. Excel.

Platform: In this experiment there are various platforms that are created. Each platform consists of a number of processors as shown in table 4.10. These processors are repeated from the four main types as shown in table 4.11 in the previous phase. All platforms have the same overall capacity for each processors types shown in table 4.11.

Table 4.10: Different Platforms Processor Count

Platform No.	Number of processor
Platform 1	4
Platform 2	8
Platform 3	16
Platform 4	32
Platform 5	64
Platform 6	128

Table 4.11: Processors Specification and Capacity

Processor type	Total capacity
OR1K	800 MIPS
NEC V850	800 MIPS
POWERPC 32	800 MIPS
MIPS32	800 MIPS

Applications: Two types of applications are used. In the first one, the sample size has relation with problem size; while the second application, there has no relation between sample size and problem size.

Application 1: This application is the same one used in phase one. In this application, when the problem size is increases the sample size will increases.

Application 2: This application executes program that solves images filtering. the sample size in this experiment is one image and the problem size is number of images (D) in data base. Each image in the database is presented as one matrix (800X600 = 480000 pixel) as shown in figure 4.12. All images in the database are stored in one matrix (480K x D). In this application, there are no relations between the sample size and the problem size. The filter function in this application multiplies each pixel in the matrix by factor as shown in formula 4.4. Source code for this application is shown in [Appendix B.5]

$$OutImage(x,y) = Image(x,y)^{\frac{2y}{p \times y}} + Image(x,y) \times (x \times y \times p) \tag{4.4}$$

where:

p is pixel value.

y and x are pixel position.

Image is imput image.

Experiment procedure:

1. OVPsim is used to write script file (Appendix A.5) that built all platforms shown in figure 4.13. processors specifications are shown in table 4.4.2.
2. c commands is used to write script file (Appendix B.3) that built application 1.

3. c commands is used to write script file (Appendix B.5) that built application 2.

4. Execute application 1 on each platform for problem size (from 500 to 3000).

5. Execute application 2 on each platform for problem size (from 500 to 3000).

6. Show result and analysis by Matlab script file (Appendix C.4).

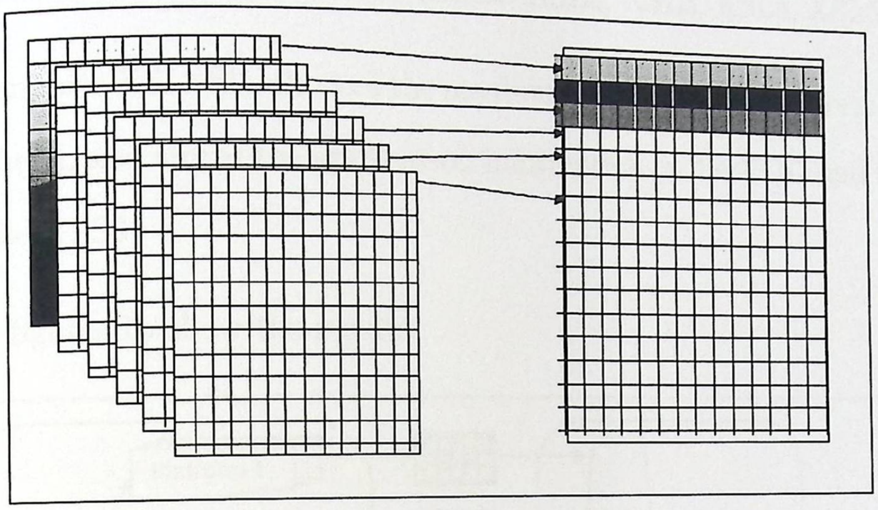


Figure 4.12: Image Data Base to Matrix

Table 4.12: Processors Specification

PLATFORM	POWERPC 32 MIPS	OR1K MIPS	MIPS32 MIPS	NEC V850 MIPS
4	P x 800	P x 800	P x 800	P x 800
8	2P x 400	2P x 400	2P x 400	2P x 400
16	4P x 200	4P x 200	4P x 200	4P x 200
32	8P x 100	8P x 200	8P x 200	8P x 200
64	16P x 50	16P x 50	16P x 50	16P x 50
128	32P x 25	32P x 25	32P x 25	32P x 25

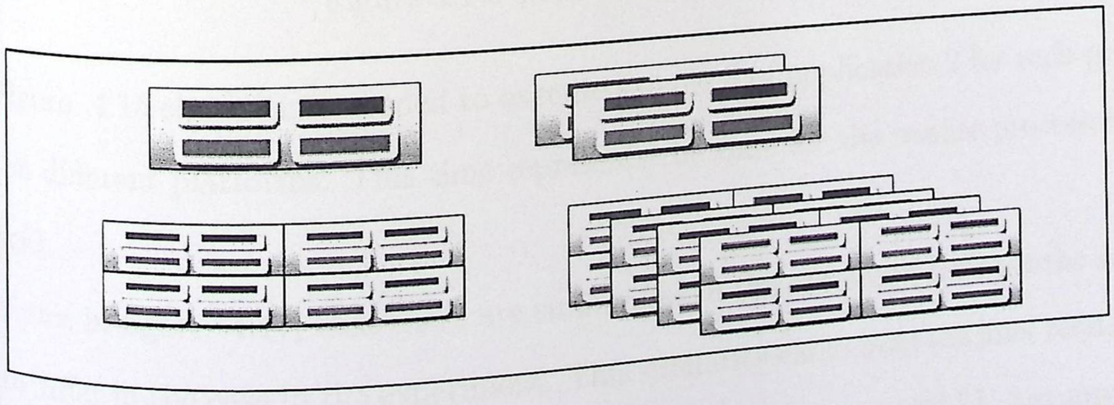


Figure 4.13: Processors Interconnection

Experiment Results of Application1: Figure 4.15 shows time needed to execute sample S of application1 on different platforms. This time represents the time for the weaker processor (bottleneck). This figure shows information that are collected as shown in figure 4.14 . The collection process contained the following steps:

1. OVPsim simulator is used to run application1 on each platform and export the execution information to excel files (XSL1, XSL2, XSL3, XSL4, XSL5, and XSL6).

2. Matlab imports XSLs file, chooses the maximum time for each platform that presents the overhead, and stores the application information in file XLS7 and time information in file XLS8.

3. Plot the figures to show the results.

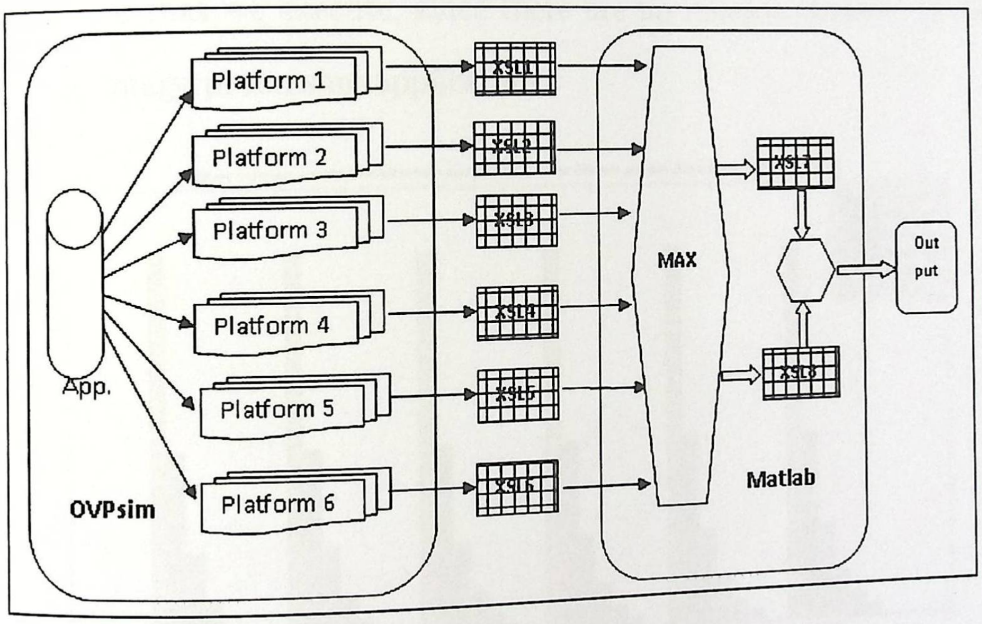


Figure 4.14: Processing Enviroment

Figure 4.15 shows time needed to execute sample S of application 2 for each problem size on different platforms. This time represents the time for the weaker processor (bottleneck).

We notice in figure 4.17, that there are similarities in the execution time for the sample, due to differ in the case in the experiment. This similarity came from the liner relation between the problem size and the sample size. In the experiment we used LU decomposition that present the previous relation.

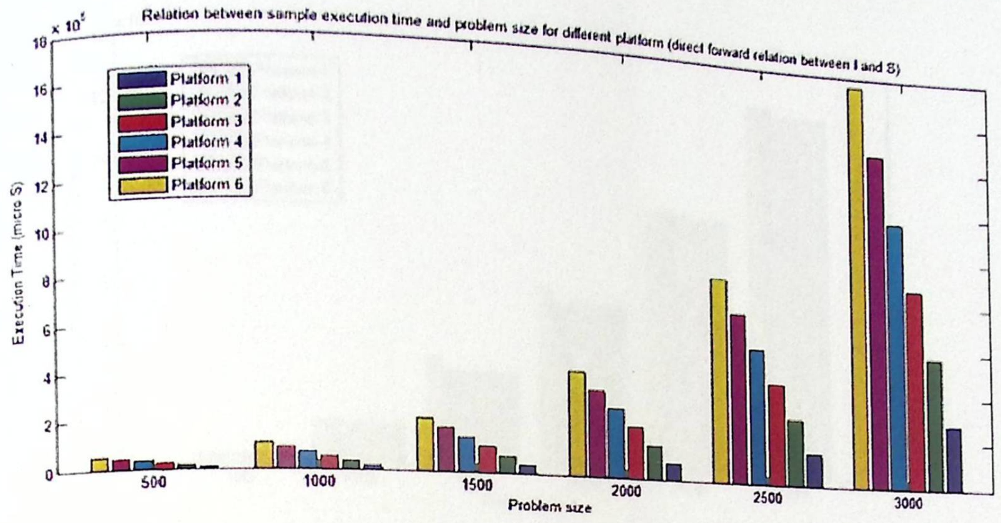


Figure 4.15: Sample Execution Time for LU Application

In figure 4.16, we see that, all execution time for the sample are the same, away from the problem size that we execute, since there are no relation between problem size and sample size in image processing application.

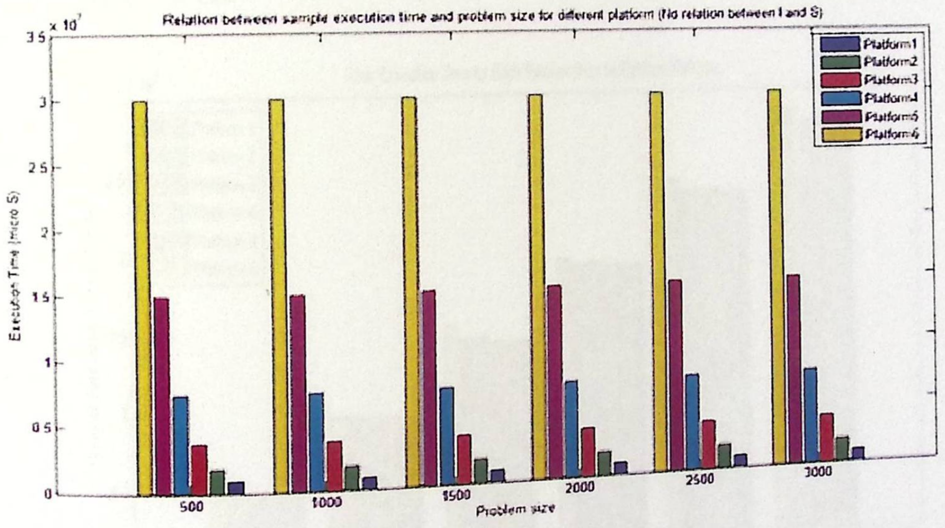


Figure 4.16: Sample Execution Time for Images Application

Figure 4.17 and figure 4.18 show maximum overall execution time for each problem size needed to execute application1 and application2 time on different platforms; all processors that exist in platform are sharing processing.

In figure 4.17 that show the execution time for LU decomposition application, we see that all platforms (regardless number of processors in the platform) needs (approximate)

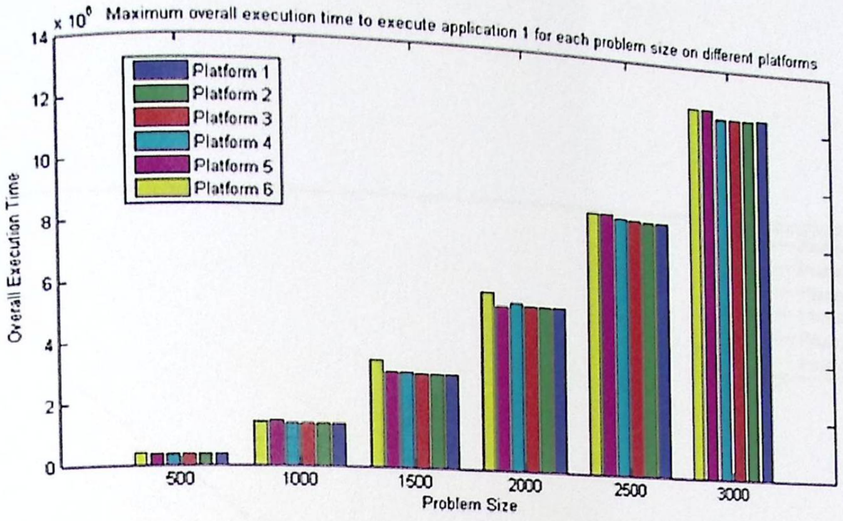


Figure 4.17: Execution Time for LU Application

same execution time for the same problem size, due that, all platforms in the experiment have the same total MIPS value. And from the figure, execution time increases where the problem size increases. These notes applies also for image processing application as it appears in figure 4.18.

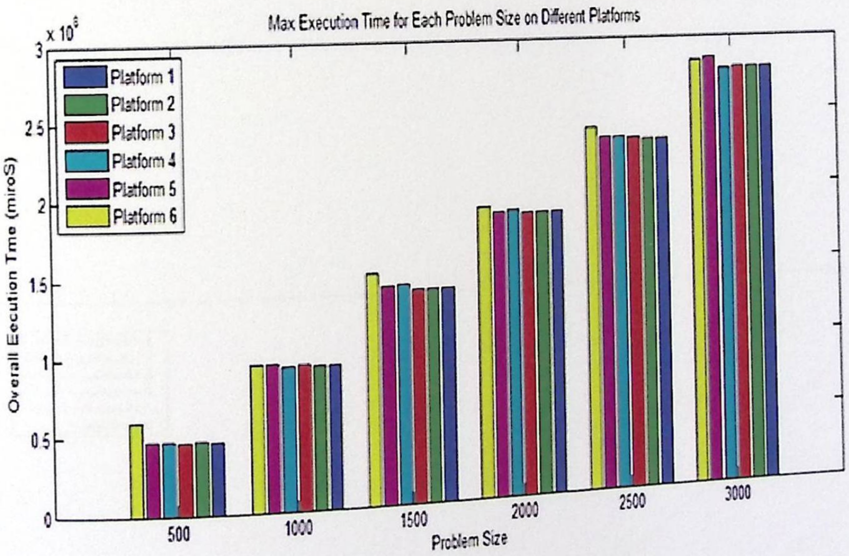


Figure 4.18: Execution Time for Images Application

Figure 4.19 and figure 4.20 shown the density of overhead to overall execution time for each problem size when execute application 1 and application 2 on different platforms.

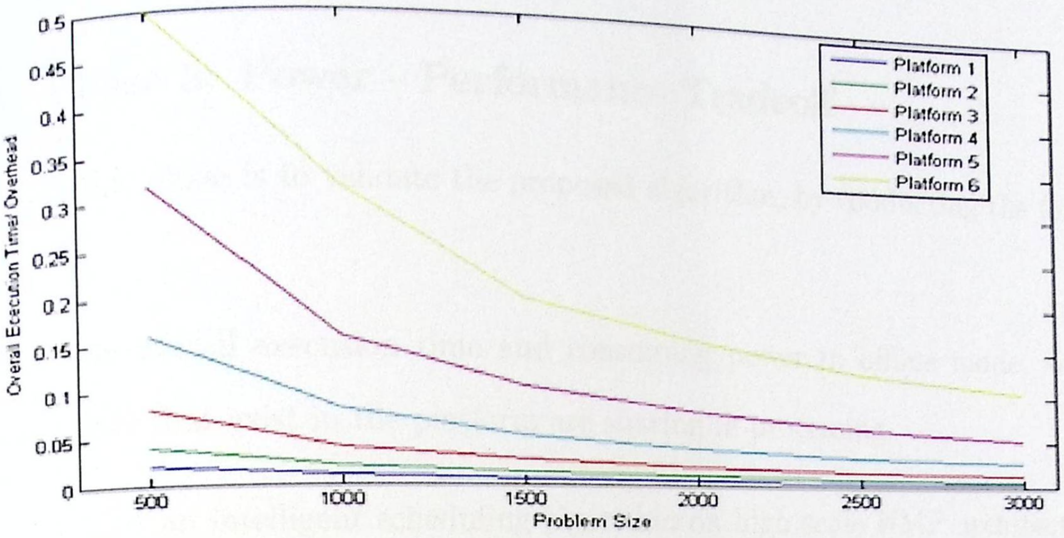


Figure 4.19: Overhead Ratio for LU Application

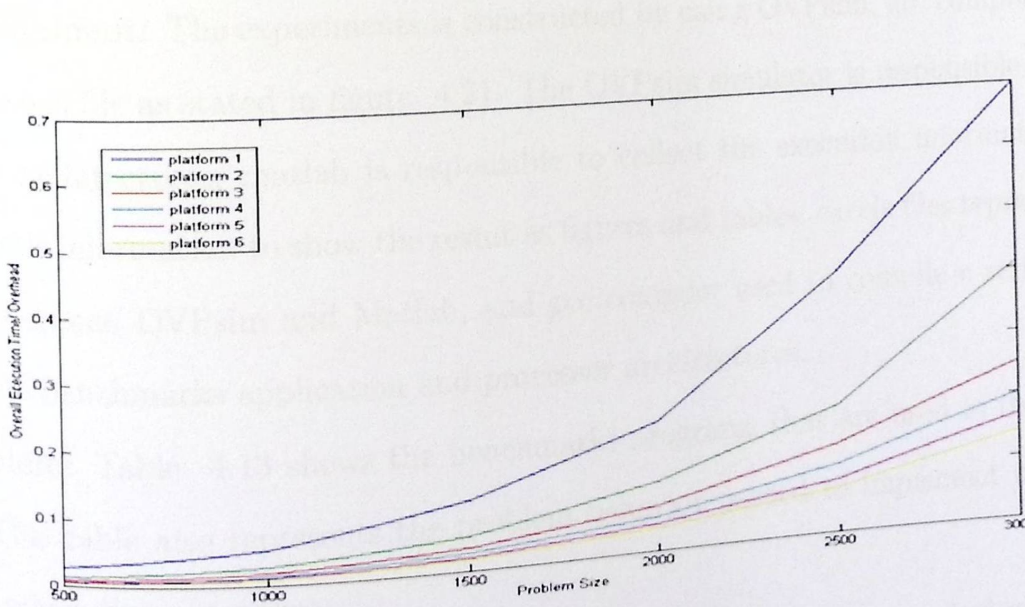


Figure 4.20: Overhead Ratio for Images Application

In the last figures if there is no relation between the sample size and the problem size good result can be achieved. If, however, there is relation between the sample size and the problem size, the overhead is affected by the problem size.

4.4.3 Phase 3: Power - Performance Tradeoff

The goal of this phase is to validate the proposed algorithm, by conducting the following steps:

- Estimate overall execution time and consuming power in offline mode, when all processors that exist in the platform are sharing in processing.
- Implement an intelligent scheduling algorithm on high scale HMP architecture using GA to set processors configuration that ensures to achieve performance closed to optimal value by designing a complex search space from the large number of combinations given by trying the share / not share state of each processor.
- Study the relation between performance and consuming power on different platforms.

Environment: The experiments is constructed by using OVPsim, gcc compiler, matlab, and excel file as stated in figure 4.21. The OVPsim simulator is responsible to built processor architectures; matlab is responsible to collect the execution information and analysis this information to show the result as figures and tables, excels files represent the medium between OVPsim and Matlab, and gcc compiler used to compile c scripts files that create benchmarks application and processor architectures.

Problem: Table 4.13 shows the benchmarks programs that are used in the experiment. This table also represents the problem behavior needed to implement using the research scheduling algorithm.

LINPACK: Develop by Jack Dongarra and Jim Bunch, this program is designed for supercomputers in the 1970s and early 1980s. LINPACK is a collection of FORTRAN subroutines that analyze and solve linear equations and linear least-squares problems.

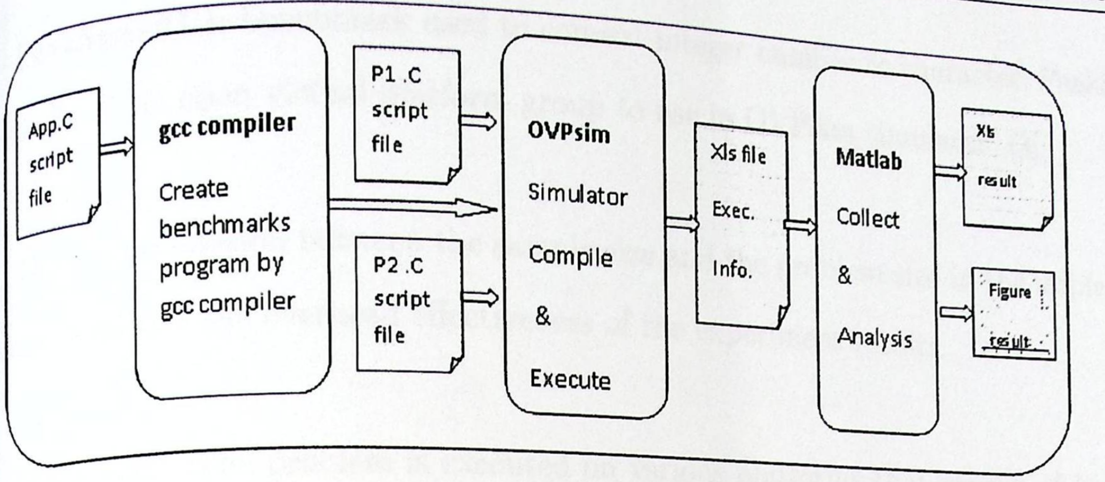


Figure 4.21: GA Environment System

Table 4.13: benchmarks behavior

Benchmark	Sample	Size
LINPAC	Solve 200X200	100000X S
PeakSpeed1	Solve 5000000 iteration	100000X S
Dhrystone	Solve 2000 run	100000X S

The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and triadiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems. LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference [2].

Dhrystone Benchmark is a general-performance benchmark test originally developed by Reinhold Weicker in 1984. Dhrystone benchmark used to measure and compares the performance of different computers or, the efficiency of the code generated for the same computer by different compilers. Dhrystone consists of standard code and concentrates on string handling. It uses no floating-point operations. It heavily influenced by hardware and software design, compiler and linker options, code optimizing, cache memory and integer data types [44].

PeakSpeed1 is benchmark used to convert integer number to character; PeakSpeed1 is developed by open virtual platform group to use in OVPsim simulator [3].

There is no relation between the sample size and the problem size in the implementation to eliminate the overhead effectiveness of the experiment results.

Platform: The problem is executed on various platforms that consist of high scale HMP architecture as shown in table 4.14. The platforms represents the similarity of real processors that are described in table 4.15 [4].

Table 4.14: Platforms Combination

Platform No.	Number of processors	Total MIPS
Platform1	32 P1	64000
	32 P2	64000
Platform2	64 P1	64000
	64 P2	64000
Platform3	128 P1	64000
	128 P2	64000

Table 4.15: Real Processors Specification

Processor	Type	Power	Similarity
P1	Intel® Xeon® processor X5675 3.07GHz	180W	ARM7
P2	AMD Opteron processor 6174 2.2GHz	230W	OR1K

Experiment Procedure: The experiment is split into three parts; in each part the researcher will execute all benchmarks on the same platform. Then he will analyze the information to show results.

1. Create script files (Appendix A.6) to simulate platforms architecture under OVPsim simulator. Then compile these script files to create executable files there shown in table 4.16. In this step the architecture represents one processor type. Processor1Platform is simulated ARM7 processor and Processor2Platform is simulated OR1K processor.

2. Compile application file (LINPACK, Dhrystone, and PeakSpeed1 benchmarks) to create exactable files as shown in table 4.16. These files can execute on simulated platforms.

3. Run the system. The information will store in xls files.

4. Design Matlab program (Appendix C.6) that collects the information produced when executing the benchmarks program on the platforms. This program is also used to build high scale HMP architectures as shown in table 4.14, and to apply the scheduler Algorithm to show the result.

Table 4.16: Experiment Program Files

Source code file .c	Executable file
Processor1Platform Processor2Platform	Processor1Platform.Windows32.exe Processor1Platform.dll Processor2Platform .Windows32.exe Processor2Platform .dll
LINPAC	linpack.ARM7.elf linpack.OR1K.elf
peakSpeed1	peakSpeed1.ARM7.elf peakSpeed1.OR1K.elf
Dhrystone	Dhrystone.ARM7.elf Dhrystone.OR1K.elf

In this experiment, all platforms have the same value of overall capacity (MIPS values). This result to eliminate the effect on the experiment result.

Implementation

Execute the benchmarks shown in table 4.13 on platform1 that is described in the table 4.17, platform2 that is described in table 4.18, and platform3 that is described in table 4.19.

Table 4.17: Platform Combination, Consist of 64 Processors

Processor type	Count	MIPS
P1	32	10P x 100 + 12P x 200 + 10P x 300 = 64000
P2	32	10Px 100 + 12P x 200 + 10P x 300 = 64000

Table 4.18: Platform Combination, Consist of 128 Processors

Processor type	Count	MIPS
P1	64	20P x 50 + 24P x 100 + 20P x 150 = 64000
P2	64	20Px 50 + 24P x 100 + 20P x 150 = 64000

Table 4.19: Platform Combination, Consist of 256 Processors

Processor type	Count	MIPS
P1	128	40P x 25 + 48P x 50 + 40P x 75 = 64000
P2	128	40P x 25 + 48P x 50 + 40P x 75 = 64000

The first step in the Algorithm, measure the execution time that needs to execute the benchmark sample on each processor in the platform that is corporated in processing. Figure 4.22 shows the execution information for platform1, figure 4.23 for platform2, and figure 4.24 for platform3. Therefore, all figures for the benchmarks are similar, because the platform combination consists of two processer types (heterogeneous in functional), and each processor was repeated in different capacity (heterogeneous on performance).

The second note about the processors execution time is that the processors that have the same performance (MIPS value) differ in execution time that needs to execute the same sample. For example in figure 4.22 group 1 of processors (1-10) has MIPS equal 100 that need approximate (0.9) seconds to execute the sample, and group 2 of processors (32-42) has the same MIPS and needs approximate (11.2) seconds to execute the same sample. Reason of the difference in execution time caused from the difference of the number of instruction that is produced from the compiler for each processor type.

The last note is the overhead (sample execution time for the slower processor) comparison between the three platforms. The reason caused from the total MIPS for all platforms are equal. In the platforms which contain more processors need to decrease the value of processors MIPS to keep the value of total MIPS.

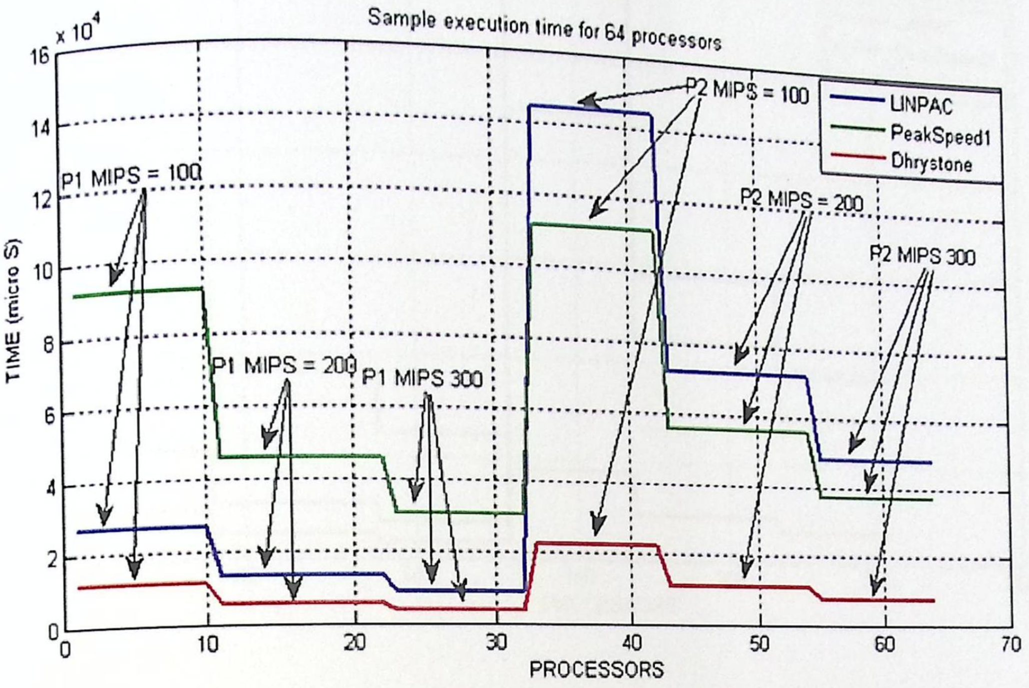


Figure 4.22: Benchmarks Execution Information among 64 Processors

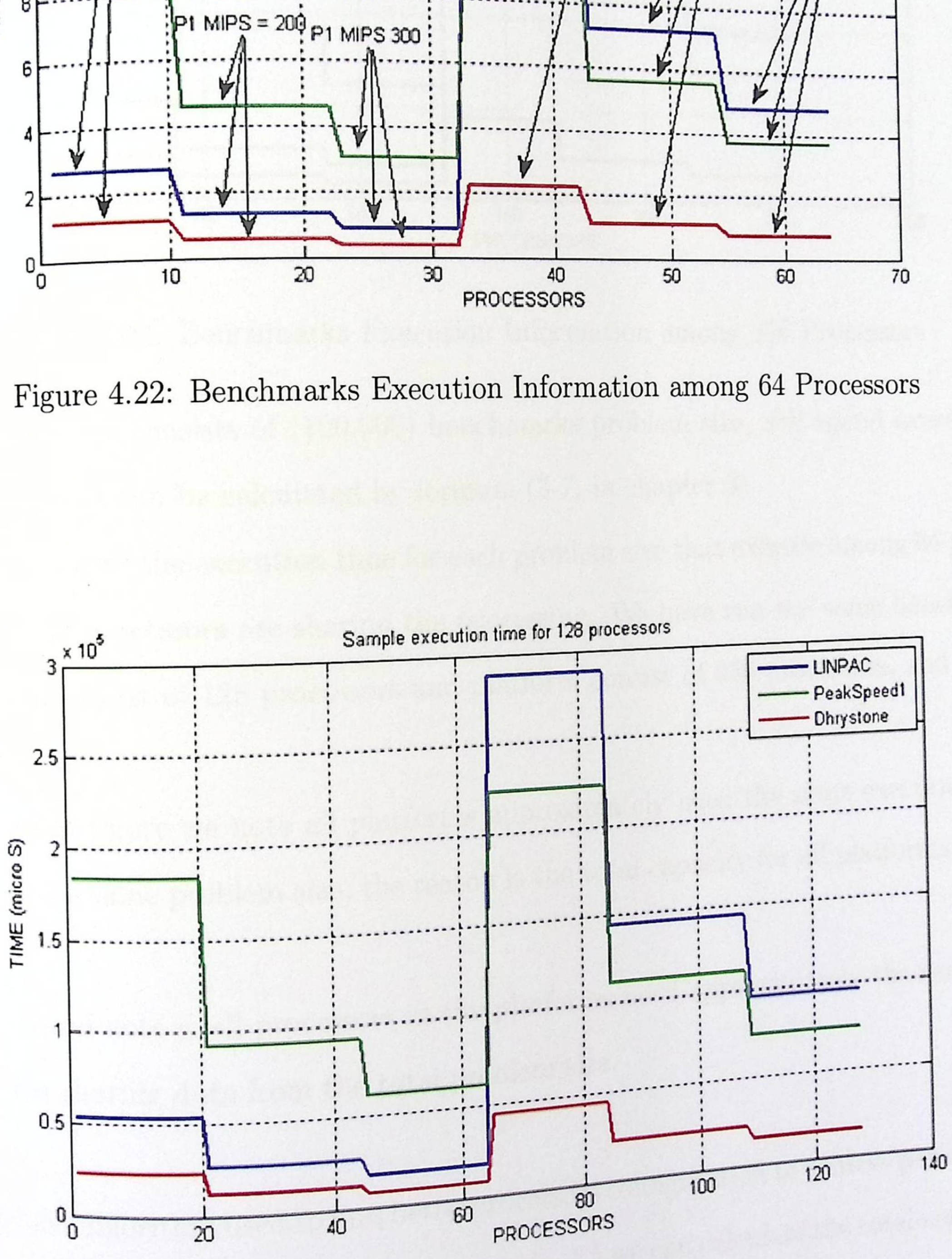


Figure 4.23: Benchmarks Execution Information among 128 Processors

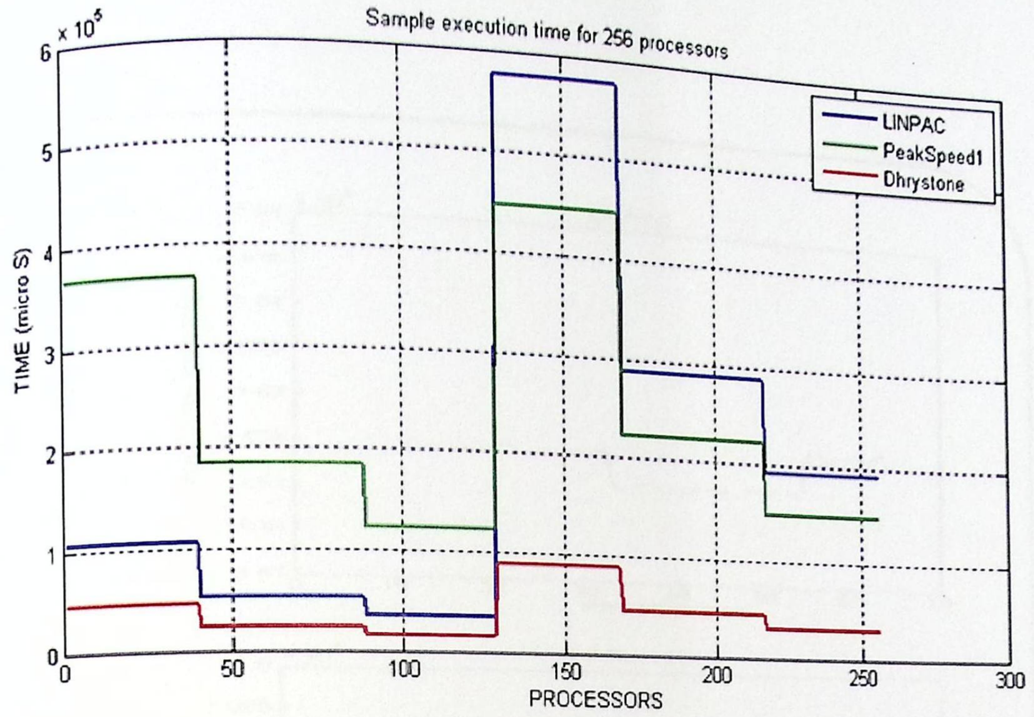


Figure 4.24: Benchmarks Execution Information among 256 Processors

Each problem consists of (100,000) benchmarks problem size, will spend overall execution time that can be calculated by formula (3-7, in chapter 3).
Figure 4.25 shows the execution time for each problem size that execute among 64 processors, when all processors are sharing the processing. We have run the same benchmarks on platform consist of 128 processors and platform consist of 256 processors, and we get similar results.

In the last figure we note all platforms approximately need the same execution time to execute the same problem size, the reason is the total capacity for all platforms are the same.

The second note is all processors in the platforms need approximately the same time to solve the sharing data from the total problem size.

Scheduler algorithm used to find betterprocessors configuration to achieve performance with determine consuming POWER. By Formula 4.5 we can calculate the total consuming power that is needed to solve the problem.

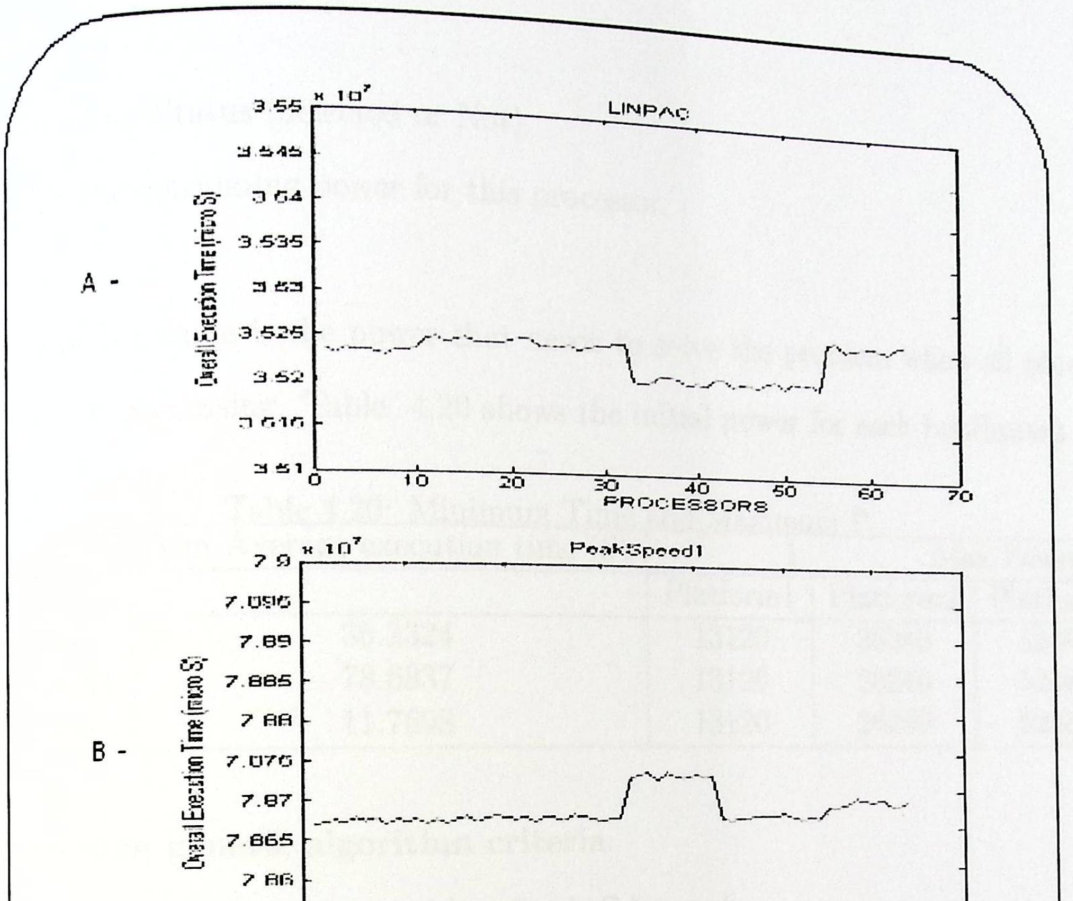


Figure 4.25: Benchmarks Execution Time among 64 Processors

$$TotalconsumingPOWER = \Sigma(S[i] \times Power[i]) \tag{4.5}$$

where:

$S[i]$ is Processor Status (Selected or Not).

$Power[i]$ is the consuming power for this processor.

Initial power value is the power that needs to solve the problem when all processors are sharing in processing. Table 4.20 shows the initial power for each benchmark.

Table 4.20: Minimum Time and Maximum P

Benchmark	Min Average execution time (S)	1		Max Power(W)
		Platform1	Platform2	Platform3
LINPAC	35.2324	13120	26240	52480
PeakSpeed1	78.6837	13120	26240	52480
Dhrystone	11.7698	13120	26240	52480

Concerning genetic algorithm criteria

There are two main criteria used to finish GA search:

- Number of iteration: set maximum number of iteration
- Stability of fitness value: stop the search when fitness value will be stable for number (n) of iteration.

This experiment set the maximum number of iteration which equals 50 and the count of iteration which makes fitness stable is 10.

Figure 4.26 shows algorithm result when running the system in maximum power (all processors sharing to solve the problem size) by using platform 1that consists of 64 processors.

We have run the same benchmarks on platform consists of 128 processors and platform consists of 256 processors. Similar results are gotten.

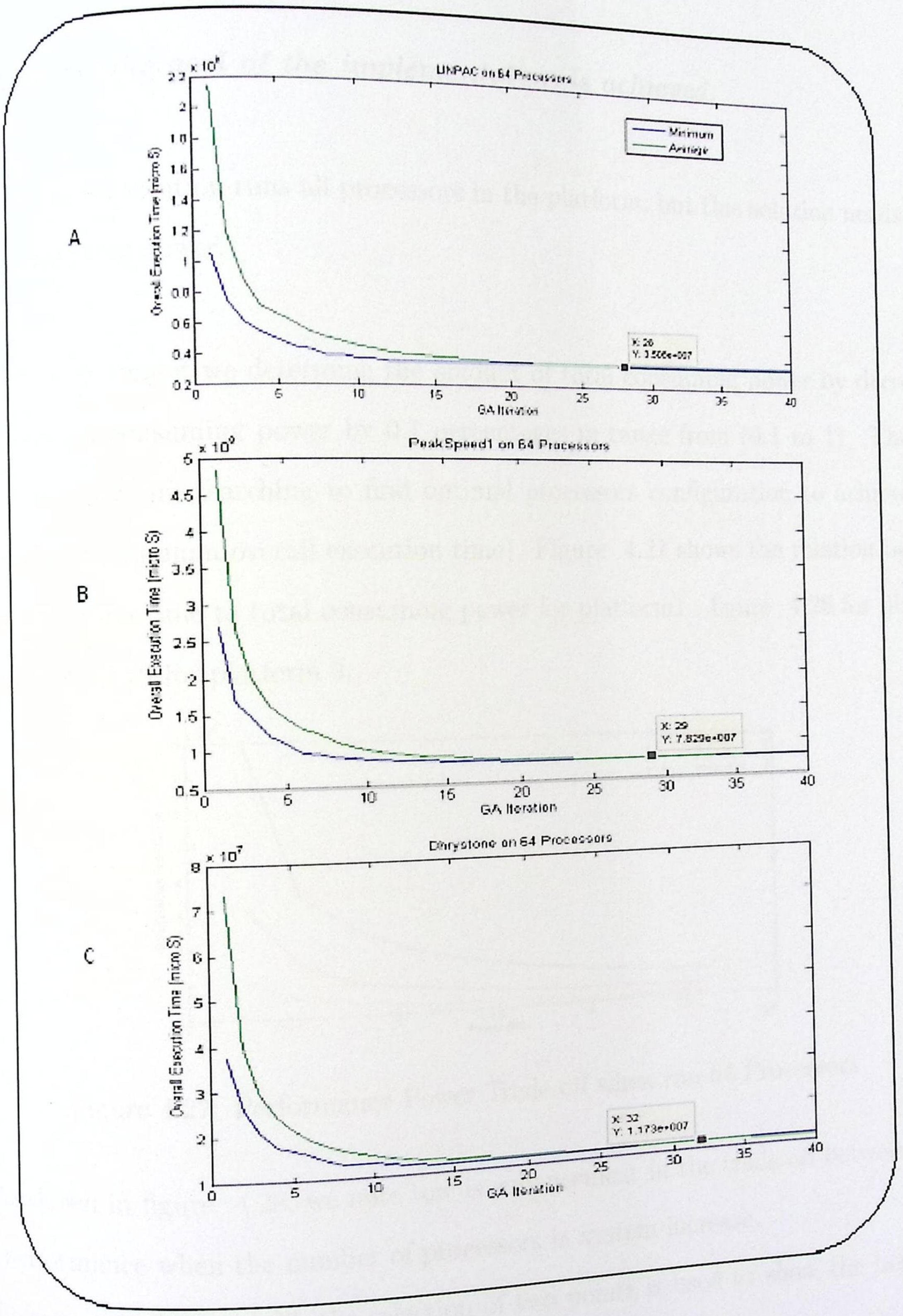


Figure 4.26: Max. Performance when Executed among 64 Processors

Genetic algorithm is fast to find better processors configuration to solve problem in optimal performance because the number of variables (processors) is compatible with GA.

However, the goal of the implementation is achieved.

The optimal solution runs all processors in the platform, but this solution needs maximum consuming power.

In the experiment we determine the amount of total consuming power by decreasing the maximum consuming power by 0.1 percentages in range from (0.1 to 1). Then the algorithm will begin searching to find optimal processors configuration to achieve high performance (minimum overall execution time). Figure 4.27 shows the relation between overall execution time to total consuming power for platform1 , figure 4.28 for platform 2, and figure 4.29 for platform 3.

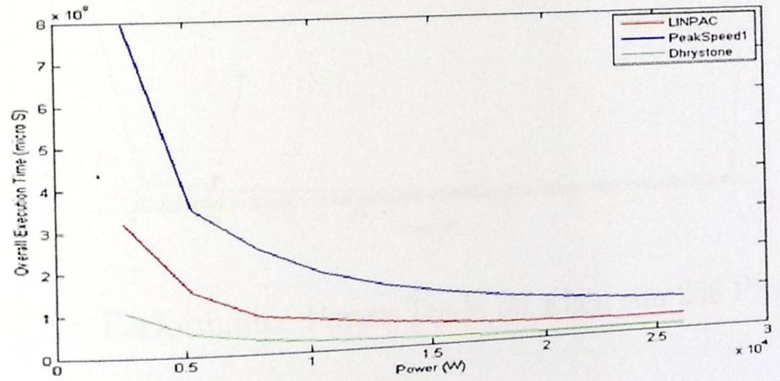


Figure 4.27: Performance Power Trade off when run 64 Processors

As shown in figure 4.28, we note tow is enhancement in the trade off between power and performance when the number of processors in system increase.

Reference to figure 4.29, the selection of two points is used to show the benefits of algorithm when applying our propsed algorithem:

Point A achieves max performance, but the system in this case needs full power. The result of this point is shown in table (4-20).

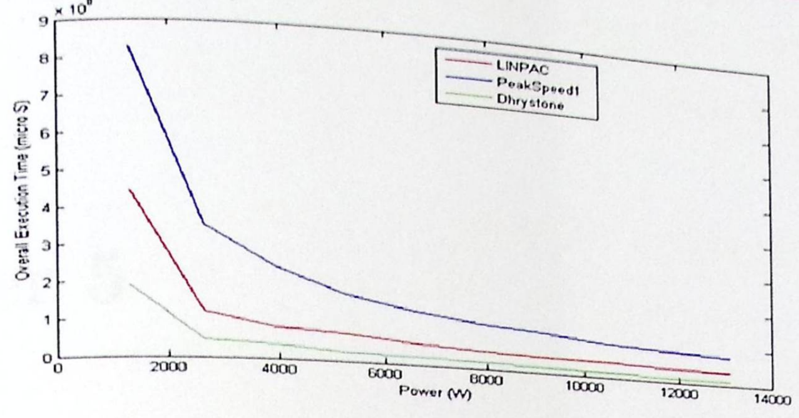


Figure 4.28: Performance Power Trade off when run 128 Processors

Point B saves large amounts of power by sacrifice a small part of the performance. For example: To execute linpak benchmark in 246 micr S (sacrifice 168 micro S), the system needs 15 k wat (saving is 40 k wat). Then if the processors is configured to point B, it can solve linpak problem three times in the same power needed in full performance.

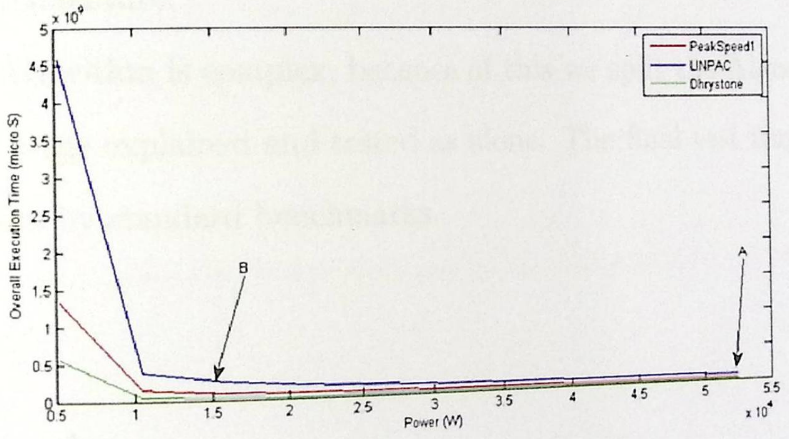


Figure 4.29: Performance Power Trade off when run 256 Processors

In the obove figures, to achive high degree of performance one has to implement the research scheduling algorithm on system contain high scale of processors.

Chapter 5

Conclusion and Future Work

The purpose of this research is developing scheduling algorithm that split data domain problem into independent parts and distribute these parts on a heterogeneous processors set in platform architecture.

The propsed Algorithm is complex, because of this we split the Algorithm procedures into Stages, each stage explained and tested as alone. The final test implemented on the scheduler algorithm by standard benchmarks.

5.1 Discussion

It is important to point that, the proposed scheduling algorithm on HMP platform focus on performing a tradeoff between the power and the performance, that is to reduce the power by constant value by stockholder (system or user), then the proposed algorithm searches for adequate group of processors in the system using GA method. After that, the proposed algorithm performs load balancing for the processes on the selected processors group. In addition, this algorithm performs time estimation which the system needs to execute the processes. Moreover, this algorithm is applied to high scale HMP platform.

Most of the previous work on this field focused on enhancing the computing system performance through sharing all the system processors executing processes or programs, moreover, these theories were applied to small or medium scale HMP platform.

As mentioned above, the proposed algorithm can not be fairly compared with most of the previous work which did not focus on trading off between power and performance. To prove the realistic and reliability of the proposed algorithm, a discussion and study of implementations results is done and the results shown in figure 5.1.

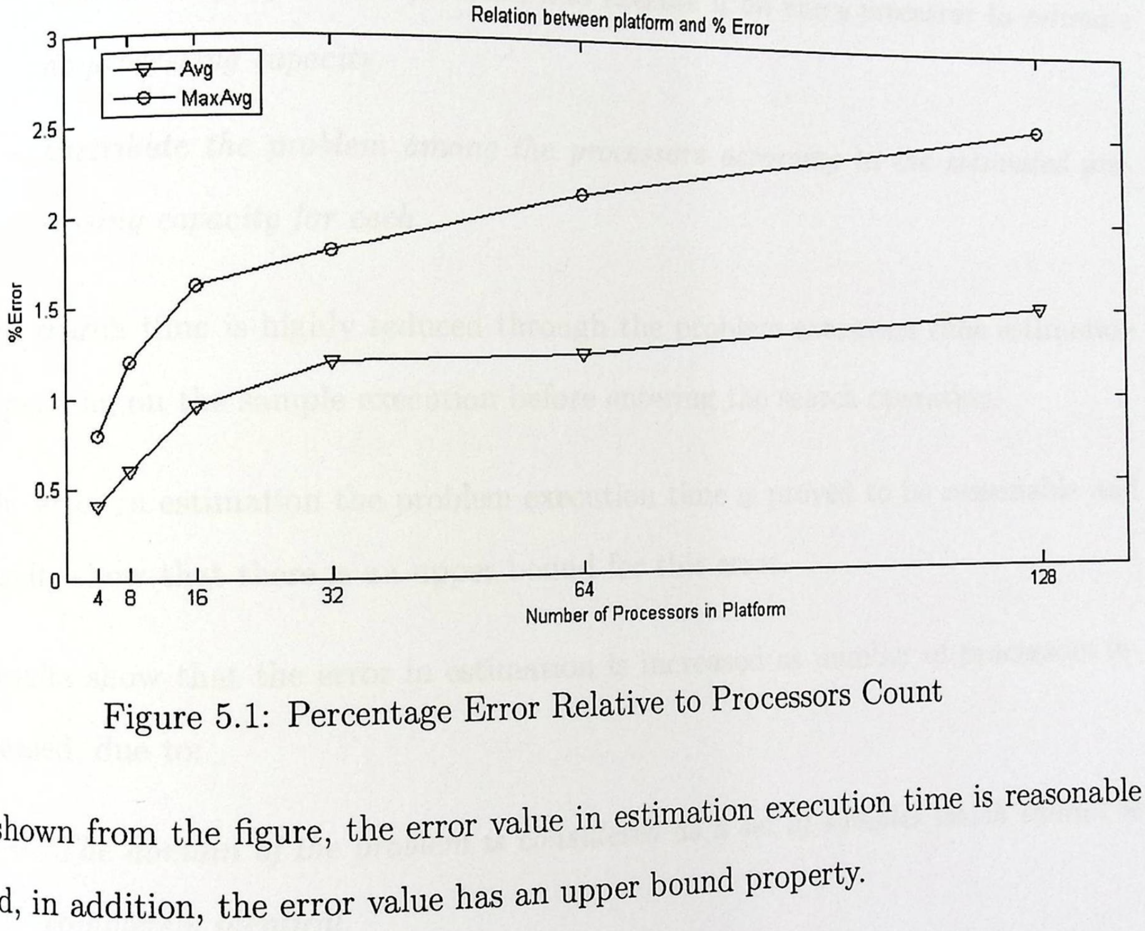


Figure 5.1: Percentage Error Relative to Processors Count

As shown from the figure, the error value in estimation execution time is reasonable accepted, in addition, the error value has an upper bound property.

5.2 Conclusions

We have proposed a powerful methodology for exploration of the valid combinations of processors in a large multicore platforms, to achieve the suitable power-performance trade off. Combinations of processors in a multicore platform means selecting a set of processors out of the rest to participate in execution.

Also, we have used a genetic algorithm to explore the search space of valid combinations of the processors to execute the problem. The key parameters of the GA are tested and shown that they are fixed on the correct values; mutation probability is equal to 0.15, cross over probability is equal to 0.75 and Roulette wheel selection have been used.

We can summarize a number of conclusions in the following points:

- In case the consumed power is not counted for, the scheduling algorithm can simply be work as follows:
  - *Take a sample from the problem, and execute it on every processor to estimate its processing capacity.*
  - *Distribute the problem among the processors according to the estimated processing capacity for each.*
- The search time is highly reduced through the problem execution time estimation depending on the sample execution before entering the search operation.
- The error in estimation the problem execution time is proved to be reasonable and results show that there is an upper bound for this error.
- Results show that the error in estimation is increased as number of processors increased, due to:
  - *The domain of the problem is considered as a set of samples which cannot be completely identical.*
  - *The sample does not contain the execution obstacles founded in the whole problem execution, such as exceptions, stalls, etc*
- The scheduling algorithm work efficiently when using domain decomposition, where the relation between the problem size and the sample size is simple.
- The sample execution before starting the search algorithm is considered as an overhead which increases when handling problems like matrix multiplication where the sample size has to increase as the problem size increase.

- The proposed methodology permits power saving through selecting an optimum processor configuration for executing the problem in a reasonable time with a fixed maximum power consumption.

- The power saving is highly noted and enhanced as the number of processors in the platform increases.

5.3 Future work

1. Generalize the methodology to work using real implementations on large distributed computing environments.

2. Another direction may be considered in the future is integrating our algorithm into a powerful compiler like gcc.

3. Study the possibility of employing the proposed scheduling algorithm on dependant domains and with functional decomposition.

Bibliography

[1] *TOP500 SUPERCOMPUTER*, 2011. http://www.top500.org/.

[2] *LINPACK*, 2012. http://www.netlib.org/linpack/,http://www.netlib.org/utk/people/JackDongarra/faqlinpack.html#_Toc27885709.

[3] *OVPpresentation*, 2012. http://www.ovpworld.org/presentation.php?slide=OVPINTRO2.

[4] *Measuring Processor Power, Intel, White Paper*, April, 2011. http://www.intel.com/content/www/xa/en/benchmarks/resources-xeon-measuring-processor-power-paper.html.

[5] M Adiletta, M Rosenbluth, and D Bernstein. Virtualization: A survey on concepts, taxonomy and associated security issuesthe next generation of intel ixp network processors. *Intel Tech. Journal*, pages 6–18, 2002.

[6] M Aldasht, J Ortega, and G Puntonet. A genetic exploration of dynamic load balancing algorithms. *IEEE: Evolutionary Computation,Congress on. ISBN: 0-7803-8515-2*, l.1:1158 – 1163, 2004.

[7] J Allen, B Bass, C Basso, and R Boivie. Ibm powernp network processor: Hardware, software, and applications. *IBM Journal of Research and Development, 47(2)*, pages 177–194, 2003.

[8] J Barker, K Davis, and A Hoisie. Entering the petaop era: the architecture and performance of roadrunner. *Proceedings of the Conference on Supercomputing (SC'08)*, 2008.

[9] M Becchi and P Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *ACM 1-59593-302-6 06 0005*, 2006.

[10] M Bernd. Introduction to parallel computing. computational nanoscience. *ISBN 3-00-017350-1*, 31, 491-505.

[11] S Borkar. Thousand core chips: a technology perspective. proc. design automation conference. *ACM press: SanDiego,CA,USA*, pages 746–749, 2000.

[12] F Bower, D Sorin, and L Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE: Micro-Institute of Electrical and Electronics Engineers, 28(3)*, pages 17–25, 2008.

[13] P Case and A Padegs. Architecture of the ibm system370. communications.

[14] P Chandra. Programming the 80387 coprocessor. *BYTE*, 13 Issue 3, 1988.

[15] Y Cho and W Mangione-Smith. A pattern matching coprocessor for network security. *In DAC 07: Proceedings of the 42nd annual Design Automation Conference*, pages 234–239, 2005.

[16] W David, R Williams, and P Tilke. An event based algorithm for distributing concurrent tasks on multi-core architectures. *ISSN 1879- 2944:Computer Physics Communications, 181 (2)*, pages 341–354, 2010.

[17] E Ephzibah. Cost effective approach on feature selection using genetic algorithms and ls-svm classifier. *IJCA Special Issue on Evolutionary Computation*, pages 16–20, 2010.

[18] D Fatima and C Tech. Computer hardware text book. *Saidabad-Hyderabad 500059 A.P.*, 2005.

[19] J Greenstadt. The ibm 709 computer. *ACM*, 1957:92–98.

[20] T Hamaguchi, T Komata, and T Nagai. A framework of better deployment for wlan access point using virtualization technique. *IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, page 968–973, 2010.

[21] J Hennessy. Computer architecture a quantitative approach fourth edition. *San Francisco: Denise E. M. Penrose*, 2007.

[22] H Hofstee. Power efficient processor architecture and the cell processor. *International Symposium on High-Performance Computer Architecture HPCA-11*, 2005:258–262.

[23] A Huisman. Heterogeneous multi-core processor scheduling using meta-heuristic techniques. *UMI-MR52237*, 2009.

[24] A Jaejin and A Jung-Ho. Adaptive execution techniques of parallel programs for multiprocessors. *Elsevier Inc*, 2009.

[25] L Jiong and J Niraj. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. *IEEE*, 2002.

[26] A Jooya, A Baniasadi, and M Analoui. History-aware, resource-based dynamic scheduling for heterogeneous multi-core processors. *Computers and Digital Techniques: IET*, pages 254 – 262, 2009.

[27] O Kiyarazm, M Moeinzadeh, and S Sharifian. A new method for scheduling load balancing in multi-processor systems based on pso. *IEEE ISBN: 978-1-4244-98093*, pages 71–76, 2011.

[28] D Koufaty, D Reddy, and S Hahn. Bias scheduling in heterogeneous multi-core architectures. *ACM*, 2010.

[29] D Krishnaswamy, R Stevens, and R Hasbun. The intel pxa800f wireless internet-on-a-chip architecture and design. *IEEE Custom Integrated Circuits*, 2003.

[30] D Kumar and N Tullsen R Jouppi. Heterogeneous chip multiprocessors computer. *IEEE*, 38(11):32–38, 2005.

[31] W Li and L Wang. Energy-considered scheduling algorithm based on heterogeneous multi-core processor. *IEEE.ISBN: 978-1-61284-719-1*, pages 1151–1154, 2011.

[32] J Mars, D Williams, D Upton, and S Ghosh. A reactive unobtrusive prefetcher for multicore and manycore architecture. *Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms 2008*, 2008.

[33] Melanie Mitchell. An introduction to genetic algorithm(complex adaptive system). *A Bradford Book, Third edition*, 1998.

[34] C Mogul, J Mudigonda, and N Binkert. Using asymmetric single-isa cmps to save energy on operating systems. *IEEE Micro, 28(3)*, pages 968–97326–41, 2008.

[35] K Olukotun, A Nayfeh, and L Hammond. The case for a single-chip multiprocessor. *SIGPLAN Not., 31(9)*, pages 84–952–11, 1996.

[36] A Pfaff. Synthesis algorithm for application specific homogeneous processor networks. *IEEE: 1063 8210*, 2009.

[37] V Rasmussen. Round robin scheduling - a survey. european journal of operational research. *European Journal of Operational Research*, pages 617–636, 2008.

[38] J Reinders. Intel thread building blocks: Outfitting c++ for multi-core processor parallelism. *O Reilly Media*, 2007.

[39] P Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4):72–72, 2008.

[40] D Shelepov, J Saez, and S Jeffery. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Operating Systems Review*, 43(2):66–75, 2009.

[41] Imperas Team. 19. imperas partners, 2012. `http://www.imperas.com/partners-memberships`.

[42] E Thornton. Design of a computer the control data 6600. *Foresman and Co.*, 1970.

[43] A Vajda. Programming many-core chips. *Springer*, 2011.

[44] VALKOMMEN TILL COGNITIVE. *Dhrystone Benchmark*, 2012. `http://www.ct.se/dhrystone/index.html`.

[45] M Volker, S Vasile, V Andras, and X Hongguo. A matlab mex-file environment of slicot. *TU Chemnitz, D-09107 Chemnitz, Germany*, 1999.

[46] W Wall. Limits of instruction-level parallelism. sigarch computer architecture. *News*, 19(112):176–188, 1991.

[47] B Yee. Heterogeneous chip multiprocessors computer. *Carnegie-Mellon University. Carnegie-Mellon University.*, 1994.

[48] W Yefu, M Kai, and W Xiaorui. Temperature-constrained power control for chip multiprocessors with online model estimation. *ACM*, 2009.

Appendices

- Appendix A: Source code to construct Processors Platform
- Appendix B: Source code to construct Application
- Appendix C: Source code to construct Matlab and Compiler

```
//////////////////////////
// Appendix A.1   /////////
//////////////////////////
#include <stdio.h>
#include <string.h>
#include <impTypes.h>
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
// Main routine
int main(int argc, char **argv) {
    // initialize OVPsim, enabling verbose mode to get statistics at end
    // of execution
    icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC|ICM_ENABLE_IMPERAS_INTERCEPTS, NULL, 0);
    // create an array of pointers to processor instances
    icmProcessorP processor[2];
    // Create a new attributes list
    icmAttrListP mipsUserAttr0 = icmNewAttrList();
    icmAddDoubleAttr(mipsUserAttr0, "mips", 300.0);
    //Link Processors With Library
    const char *mips32Model     = icmGetVlnvString(NULL, "mips.ovpworld.org",
    "processor", "mips32", "1.0", "model");
    const char *mips32Semihost = icmGetVlnvString(NULL, "mips.ovpworld.org",
    "semihosting", "mips32SDE", "1.0", "model");
    // Create First Processor
    processor[0] = icmNewProcessor(
        "CPU0_MIPS",            // CPU name
        "mips32",               // CPU type
        0,                      // CPU cpuId
        0,                      // CPU model flags
        32,                     // address bits
        mips32Model,            // model file
        "modelAttrs",           // morpher attributes
        SIM_ATTRS,              // attributes
        mipsUserAttr0,            // user-defined attributes
        mips32Semihost,         // semi-hosting file
        "modelAttrs"    );         // semi-hosting attributes
    // Create a new attributes list
    icmAttrListP mipsUserAttr1 = icmNewAttrList();
    icmAddDoubleAttr(mipsUserAttr1, "mips", 100.0);
    // Create Secound Processor
    processor[1] = icmNewProcessor(
        "CPU1_MIPS",            // CPU name
        "mips32",               // CPU type
        1,                      // CPU cpuId
        0,                      // CPU model flags
        32,                     // address bits
        mips32Model,            // model file
        "modelAttrs",           // morpher attributes
        SIM_ATTRS,              // attributes
        mipsUserAttr1,            // user-defined attributes
        mips32Semihost,         // semi-hosting file
        "modelAttrs"  );           // semi-hosting attributes
    create the processor busses
    icmBusP bus1 = icmNewBus("bus1", 32);
icmBusP bus2 = icmNewBus("bus2", 32);
// connect the processors onto the busses
icmConnectProcessorBusses(processor[0], bus1, bus1);
icmConnectProcessorBusses(processor[1], bus2, bus2);
// create memories
icmMemoryP local1 = icmNewMemory("local1", ICM_PRIV_RWX, 0x0000ffff);
icmMemoryP local2 = icmNewMemory("local2", ICM_PRIV_RWX, 0x0000ffff);
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x0000ffff);
// connect the memories onto the busses
icmConnectMemoryToBus(bus1, "mp1", shared, 0x00100000);
icmConnectMemoryToBus(bus2, "mp2", shared, 0x00100000);
icmConnectMemoryToBus(bus1, "mp1", local1, 0x001f0000);
icmConnectMemoryToBus(bus2, "mp1", local2, 0x001f0000);
// create the processor busses
// NOTE: One bus for each processor instantiation
icmBusP bus1 = icmNewBus("bus1", 32);
icmBusP bus2 = icmNewBus("bus2", 32);
// connect the processors onto the busses
icmConnectProcessorBusses(processor[0], bus1, bus1);
icmConnectProcessorBusses(processor[1], bus2, bus2);
// Load Program to Processors Memories
if(icmLoadProcessorMemory(processor[0], "Profram.MIPS32LE.elf", False, False,
 True &&
                        (processor[1], "Program.MIPS32LE.elf", False, False,
                       True)) {
} else {
   return -1;}
// run platform
icmProcessorP final = icmSimulatePlatform();
// say whether simulation was interrupted
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
   icmPrintf("*** simulation  interrupted\n"); }
 // Free Processors
int stepIndex;
for (stepIndex=0; stepIndex < 2; stepIndex++) {
   icmFreeProcessor(processor[stepIndex]);  }
// Free attributes list
icmFreeAttrList(mipsUserAttr0);
icmFreeAttrList(mipsUserAttr1);
return 0;}
```

```
///////////////////////
// Appendix A.2  ///////
///////////////////////
#include <stdio.h>
#include <string.h>
#include <impTypes.h>
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
// Main routine
int main(int argc, char **argv) {
  // initialize OVPsim, enabling verbose mode to get statistics at end
  // of execution
  icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC|ICM_ENABLE_IMPERAS_INTERCEPTS, NULL, 0);
  // create an array of pointers to processor instances
  icmProcessorP processor[2];
  // Create First Processor
  icmAttrListP mipsUserAttr0 = icmNewAttrList();// Create a new attributes list
  // Set the endian attribute for little endian
  icmAddDoubleAttr(mipsUserAttr0, "mips", 300.0);
  icmAddStringAttr(mipsUserAttr0, "endian", "little");
  //Link Processor With Library
  const char *mips32Model    = icmGetVlnvString(NULL, "mips.ovpworld.org",
  "processor", "mips32", "1.0", "model");
  const char *mips32Semihost = icmGetVlnvString(NULL, "mips.ovpworld.org",
  "semihosting", "mips32SDE", "1.0", "model");
  processor[0] = icmNewProcessor(
     "CPU0_MIPS",          // CPU name
     "mips32",             // CPU type
     0,                    // CPU cpuId
     0,                    // CPU model flags
     32,                   // address bits
     mips32Model,          // model file
     "modelAttrs",         // morpher attributes
     SIM_ATTRS,            // attributes
     mipsUserAttr0,        // user-defined attributes
     mips32Semihost,       // semi-hosting file
     "modelAttrs"  );        // semi-hosting attributes
  // Create Secound Processor
   icmAttrListP mipsUserAttr1 = icmNewAttrList();// Create a new attributes list
  // Set the endian attribute for little endian
  icmAddDoubleAttr(mipsUserAttr1, "mips", 100.0);
  icmAddStringAttr(mipsUserAttr1, "endian", "little");
  //Link Processor With Library
  const char *arm7Model    = icmGetVlnvString(NULL, "arm.ovpworld.org",
  "processor", "arm", "1.0", "model");
  const char *arm7Semihost = icmGetVlnvString(NULL, "arm.ovpworld.org",
  "semihosting", "armNewlib", "1.0", "model");
  processor[1] = icmNewProcessor(
     "CPU1_ARM",           // CPU name
     "arm",                // CPU type
     0,                    // CPU cpuId
     0,                    // CPU model flags
     32,                   // address bits
     arm7Model,            // model file
```

```
    "modelAttrs",               // morpher attributes
    SIM_ATTRS,                  // attributes
    icmAttr,                    // user-defined attributes
    arm7Semihost,               // semi-hosting file
    "modelAttrs"                // semi-hosting attributes
);
// create the processor busses
icmBusP bus1 = icmNewBus("bus1", 32);
icmBusP bus2 = icmNewBus("bus2", 32);
// connect the processors onto the busses
icmConnectProcessorBusses(processor[0], bus1, bus1);
icmConnectProcessorBusses(processor[1], bus2, bus2);
// create memories
icmMemoryP local1 = icmNewMemory("local1", ICM_PRIV_RWX, 0x0000ffff);
icmMemoryP local2 = icmNewMemory("local2", ICM_PRIV_RWX, 0x0000ffff);
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x0000ffff);
// connect the memories onto the busses
icmConnectMemoryToBus(bus1, "mp1", shared, 0x00100000);
icmConnectMemoryToBus(bus2, "mp2", shared, 0x00100000);
icmConnectMemoryToBus(bus1, "mp1", local1, 0x001f0000);
icmConnectMemoryToBus(bus2, "mp1", local2, 0x001f0000);
// Load Program to Processors Memories
if(icmLoadProcessorMemory((processor[0], "Program.MIPS32LE.elf", False, False
, True) &&
                            (processor[1], "Program.ARM7.elf", False, False,
                            True)) {
} else {
    return -1; }
// run platform
icmProcessorP final = icmSimulatePlatform();
// say whether simulation was interrupted
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
    icmPrintf("*** simulation  interrupted\n");}
//Free Processors
int stepIndex;
for (stepIndex=0; stepIndex < 2; stepIndex++) {
    icmFreeProcessor(processor[stepIndex]);}
// Free attributes list
icmFreeAttrList(mipsUserAttr0);
icmFreeAttrList(mipsUserAttr1);
return 0;}
```

```
/////////////////////
// Appendix A.3 ////
/////////////////////
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
#include "icm/icmCpuManager.h"
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
// Function Prototypes early declaration
//static void parseArgs(int argc, char **argv);
// valid command line
#define MIN_ARGS     1
const char *usage = "[P <GDB port>] [C <core name to debug e.g. CPU0_ARM>]";
// Variables set by arguments
Bool enableDebug  = False;             // set True when debugging selected
Uns32 portNum       = 0;               // set to a port number for a debug
connection
Bool selectCore    = False;            // set True when a specific core is selected
char coreName[32] = "CPU0_ARM";        // set default core name to debug
char *coreNameP    = coreName;
// Main routine
int main(int argc, char **argv) {
Uns64 IC0,IC1,IC2,IC3;*/
   //parseArgs(argc, argv);
   // initialize OVPsim, enabling verbose mode to get statistics at end of
   execution
   // and
   // Imperas Intercepts to utilise specific builtin simulator functions
   unsigned int icmAttrs = ICM_VERBOSE | ICM_STOP_ON_CTRLC |
   ICM_ENABLE_IMPERAS_INTERCEPTS;
   if(enableDebug) {
      icmInit(icmAttrs,"rsp", portNum);
   } else {
      icmInit(icmAttrs, 0 , 0);}
   // create an array of pointers to processor instances
   icmProcessorP processor[4];
   // create processors
   const char *arm7Model     = icmGetVlnvString(NULL, "arm.ovpworld.org",
   "processor", "arm", "1.0", "model");
   const char *arm7Semihost = icmGetVlnvString(NULL, "arm.ovpworld.org",
   "semihosting", "armNewlib", "1.0", "model");
   icmAttrListP icmAttr = icmNewAttrList();
   icmAddStringAttr(icmAttr, "variant", "ARM7TDMI");
   icmAddStringAttr(icmAttr, "endian", "little");
   processor[0] = icmNewProcessor(
      "CPU0_ARM",               // CPU name
      "arm",                    // CPU type
      0,                        // CPU cpuId
      0,                        // CPU model flags
      32,                       // address bits
      arm7Model,                // model file
```

```
    "modelAttrs",           // morpher attributes
    SIM_ATTRS,              // attributes
    icmAttr,                // user-defined attributes
    arm7Semihost,           // semi-hosting file
    "modelAttrs"  );        // semi-hosting attributes
    icmAddDoubleAttr(icmAttr, "mips", 200.0);
processor[1] = icmNewProcessor("CPU1_ARM", "arm",1,0, 32,arm7Model,
"modelAttrs",SIM_ATTRS,icmAttr,arm7Semihost,"modelAttrs");
const char *v850Model      = icmGetVlnvString(NULL, "necel.ovpworld.org",
"processor", "v850", "1.0", "model");
const char *v850Semihost = icmGetVlnvString(NULL, "necel.ovpworld.org",
"semihosting", "v850Newlib", "1.0", "model");
icmAttrListP icmAttr_v850 = icmNewAttrList();
icmAddStringAttr(icmAttr_v850, "endian", "little");
// create a processor instance
 processor[2] = icmNewProcessor(
    "CPU2_V850",                  // CPU name
    "v850",                // CPU type
    2,                     // CPU cpuId
    0,                     // CPU model flags
    32,                    // address bits
    v850Model,             // model file
    "modelAttrs",          // morpher attributes
    SIM_ATTRS,             // attributes
    icmAttr_v850,                    // user-defined attributes
    v850Semihost,          // semi-hosting file
    "modelAttrs"  );           // semi-hosting attributes
icmAddDoubleAttr(icmAttr_v850, "mips", 150.0);
processor[3]  = icmNewProcessor( "CPU3_V850","v850",  3,0,32, v850Model,
"modelAttrs",SIM_ATTRS,icmAttr_v850,v850Semihost,"modelAttrs");
// create the processor busses
// NOTE: One bus for each processor instantiation
icmBusP bus1 = icmNewBus("bus1", 32);
icmBusP bus2 = icmNewBus("bus2", 32);
icmBusP bus3 = icmNewBus("bus3", 32);
icmBusP bus4 = icmNewBus("bus4", 32);
// connect the processors onto the busses
icmConnectProcessorBusses(processor[0], bus1, bus1);
icmConnectProcessorBusses(processor[1], bus2, bus2);
icmConnectProcessorBusses(processor[2], bus3, bus3);
icmConnectProcessorBusses(processor[3], bus4, bus4);
// create memories
// the ARM processor toolchain sites code in lower memory and stack in
higher memory
// so we will use two memories
// NOTE: this is just a consequence of the default linker script used
icmMemoryP local1a = icmNewMemory("local1a", ICM_PRIV_RWX, 0x9fffffff);
icmMemoryP local1b = icmNewMemory("local1b", ICM_PRIV_RWX, 0x0fffffff);
icmMemoryP local2a = icmNewMemory("local2a", ICM_PRIV_RWX, 0x9fffffff);
icmMemoryP local2b = icmNewMemory("local2b", ICM_PRIV_RWX, 0x0fffffff);
icmMemoryP local3 = icmNewMemory("local3", ICM_PRIV_RWX, 0x9fffffff);
icmMemoryP local4 = icmNewMemory("local4", ICM_PRIV_RWX, 0x9fffffff);
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x1fffffff);
// connect the memories onto the busses
// connect local memories onto individual processor buses
icmConnectMemoryToBus(bus1, "mp1", local1a, 0x00000000);
icmConnectMemoryToBus(bus1, "mp1", local1b, 0xf0000000);
icmConnectMemoryToBus(bus2, "mp1", local2a, 0x00000000);
icmConnectMemoryToBus(bus2, "mp1", local2b, 0xf0000000);
icmConnectMemoryToBus(bus3, "mp1", local3, 0x00000000);
icmConnectMemoryToBus(bus4, "mp1", local4, 0x00000000);
// connect the shared memory onto all the local buses
icmConnectMemoryToBus(bus1, "mp1", shared, 0xa0000000);
icmConnectMemoryToBus(bus2, "mp2", shared, 0xa0000000);
icmConnectMemoryToBus(bus3, "mp3", shared, 0xa0000000);
icmConnectMemoryToBus(bus4, "mp4", shared, 0xa0000000);
// load the processor object file
if(icmLoadProcessorMemory(processor[0], "application.ARM7.elf", False, False,
 True) &&
   icmLoadProcessorMemory(processor[1], "application.ARM7.elf", False, False,
    True) &&
   icmLoadProcessorMemory(processor[2], "application.V850.elf", False, False,
    True) &&
   icmLoadProcessorMemory(processor[3], "application.V850.elf", False, False,
    True)  ) {
 } else {
    return -1; }
icmSetSimulationTimeSlice(0.00001);
// run platform
icmProcessorP final = icmSimulatePlatform();
//free processors
  icmTerminate();
  return 0;}
```

```
//////////////////////
// Appendix A.4  ////////
//////////////////////
//////////////////////
#include <stdio.h>
#include <string.h>
#include <impTypes.h>
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
#define P = 128  // Determine the number of processors these used in  the
platform
// Main routine
int main(int argc, char **argv) {
   int MIPSVALUE = 50; // TO CHANGE MIPS VALUE
   int stepIndex;
   int CHVLAUE = 0;
   char cpuName[16];//to store processor name
   // initialize OVPsim, enabling verbose mode to get statistics at end
   // of execution
   icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC|ICM_ENABLE_IMPERAS_INTERCEPTS, NULL, 0);

   // create an array of pointers to processor instances
   icmProcessorP processor[P];
   // Create group one of processors, the same type butt different in
   performance
   // Link to Library
   const char *powerpc32Model     = icmGetVlnvString(NULL, "power.ovpworld.org",
   "processor",  "powerpc32",         "1.0", "model");
   const char *powerpc32Semihost = icmGetVlnvString(NULL, "power.ovpworld.org",
   "semihosting", "powerpc32Newlib", "1.0", "model");
   // Create Attribute
   icmAttrListP cpu1_attr = icmNewAttrList();
   icmAddStringAttr(cpu1_attr, "endian","big");

   for (stepIndex = 0; stepIndex < P/4; stepIndex++)
   {
    CHVLAUE++;
    if (P==4) MIPSVALUE = 100;
    else if (P == 8)  MIPSVALUE = 100*CHVLAUE;
    else if (P == 16) MIPSVALUE = 100*CHVLAUE;
    else if (P == 32) MIPSVALUE = 100+50*(CHVLAUE-1);
    else if (P == 64) MIPSVALUE = 50+25*(CHVLAUE-1);
    else if (P == 128)
    {if (CHVLAUE == 15) MIPSVALUE = 50;
    MIPSVALUE = MIPSVALUE + 25; }
   sprintf(cpuName, "CPU_powerpc32-%d", stepIndex);
   icmAddDoubleAttr(cpu1_attr, "mips", MIPSVALUE);
   processor[stepIndex] = icmNewProcessor(
       cpuName,    // name
       "powerpc32",           // type
       stepIndex,                        // cpuId
       0,                   // flags
       32,                  // address bits
       powerpc32Model,        // model
       "modelAttrs",          // symbol
```

```
  SIM_ATTRS,                // procAttrs
  cpu1_attr,                // attrlist
  powerpc32Semihost,  // semihost file
  "modelAttrs"  );          // semihost symbol
// Create group two of processors, the same type butt different in
performance
// Link to Library
const char *arm7Model     = icmGetVlnvString(NULL, "arm.ovpworld.org",
"processor", "arm", "1.0", "model");
const char *arm7Semihost = icmGetVlnvString(NULL, "arm.ovpworld.org",
"semihosting", "armNewlib", "1.0", "model");
// Create Attribute
icmAttrListP icmAttr = icmNewAttrList();
icmAddStringAttr(icmAttr, "endian","big");
char cpuName[16];
sprintf(cpuName, "CPU_ARM-%d", stepIndex+P/4);
icmAddStringAttr(icmAttr, "mips", MIPSVALUE);
processor[stepIndex+P/4] = icmNewProcessor(
   cpuName,                 // CPU name
   "arm",                   // CPU type
   stepIndex+P/4,                          // CPU cpuId
   0,                       // CPU model flags
   32,                      // address bits
   arm7Model,               // model file
   "modelAttrs",            // morpher attributes
   SIM_ATTRS,               // attributes
   icmAttr,                 // user-defined attributes
   arm7Semihost,            // semi-hosting file
   "modelAttrs" );            // semi-hosting attributes
// Create group three of processors, the same type butt different in
performance
//Link Processors With Library
const char *mips32Model     = icmGetVlnvString(NULL, "mips.ovpworld.org",
"processor", "mips32", "1.0", "model");
const char *mips32Semihost = icmGetVlnvString(NULL, "mips.ovpworld.org",
"semihosting", "mips32SDE", "1.0", "model");
// Create Attribute
icmAttrListP mipsUserAttr0 = icmNewAttrList();
icmAddDoubleAttr(mipsUserAttr0, "endian","big");
char cpuName[16];
sprintf(cpuName, "CPU_MIPS32-%d", stepIndex+P/2);
icmAddDoubleAttr(mipsUserAttr0, "mips", MIPSVALUE);
processor[stepIndex+P/2] = icmNewProcessor(
   cpuName,                 // CPU name
   "mips32",                // CPU type
   stepIndex+P/2,                         // CPU cpuId
   0,                       // CPU model flags
   32,                      // address bits
   mips32Model,             // model file
   "modelAttrs",            // morpher attributes
   SIM_ATTRS,               // attributes
   mipsUserAttr0,           // user-defined attributes
   mips32Semihost,          // semi-hosting file
   "modelAttrs"      );       // semi-hosting attributes
// Create group four of processors, the same type butt different in
```

```
performance
//Link Processors With Library
const char *or1kModel     = icmGetVlnvString(NULL, "ovpworld.org", "processor"
, "or1k", "1.0", "model");
const char *or1kSemihost = icmGetVlnvString(NULL, "ovpworld.org",
"semihosting", "or1kNewlib", "1.0", "model");
// Create Attribute
icmAttrListP or1kUserAttr0 = icmNewAttrList();
icmAddDoubleAttr(or1kUserAttr0, "endian","big");
char cpuName[16];
sprintf(cpuName, "CPU_OR1K-%d", stepIndex+stepIndex+3*P/4);
icmAddDoubleAttr(or1kUserAttr0, "mips", MIPSVALUE);
processor[stepIndex+3*P/4] = icmNewProcessor(
  cpuName,                 // CPU name
   "or1k",                     // CPU type
   stepIndex+3*P/4,
                                         // CPU cpuId
   0,                          // CPU model flags
   32,                         // address bits
   or1kModel,                  // model file
   "modelAttrs",               // morpher attributes
   SIM_ATTRS,                  // attributes
   or1kUserAttr0,              // user-defined attributes
   or1kSemihost,               // semi-hosting file
   "modelAttrs"    );}         // semi-hosting attributes
//create the processor busses
iicmBusP = bus[P]
for (stepIndex = 0; stepIndex < P; stepIndex++)
   bus[stepIndex] = icmNewBus("stepIndex", 32);
// connect the processors onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectProcessorBusses(processor[stepIndex], stepIndex, stepIndex);
// create memories
icmMemoryP local[P];
for (stepIndex = 0; stepIndex < P; stepIndex++)
local[stepIndex]  = icmNewMemory("stepIndex", ICM_PRIV_RWX, 0x0000ffff);
// create share memories
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x0000ffff);
// connect the share memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
{
char PortName[16];
sprintf(PortName, "mp%d", stepIndex+1);
icmConnectMemoryToBus(bus[stepIndex], PortName, shared, 0x00100000);}
// connect the  memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectMemoryToBus(bus[stepIndex], "mp1", local[stepIndex], 0x001f0000);
// Load Program to Processors Memories
for (stepIndex = 0; stepIndex < P/4; stepIndex++)
if(icmLoadProcessorMemory((processor[stepIndex], "application.POWERPC32.elf",
 False, False, True)     &&
                         (processor[stepIndex +P/4], "application.ARM7.elf",
                          False, False, True)       &&
                         (processor[stepIndex +P/2],
                         "application.MIPS32.elf", False, False, True)     &&
                         (processor[stepIndex +3*P/4],
                         "application.OR1K.elf", False, False, True)       &&
                        ) {
} else {
    return -1;}
// run platform
icmProcessorP final = icmSimulatePlatform();
// say whether simulation was interrupted
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
    icmPrintf("*** simulation  interrupted\n");}
// Free Processors
int stepIndex;
for (stepIndex=0; stepIndex < P; stepIndex++) {
    icmFreeProcessor(processor[stepIndex]);}
return 0;}
```

```
//////////////////////////
// Appendix A.5  /////
//////////////////////////
#include <stdio.h>
#include <string.h>
#include <impTypes.h>
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
#define P = 4  // Determine the number of processors these used in  the platform
// Main routine
int main(int argc, char **argv) {
    // initialize OVPsim, enabling verbose mode to get statistics at end
    // of execution
    icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC|ICM_ENABLE_IMPERAS_INTERCEPTS, NULL, 0);
    // create an array of pointers to processor instances
    icmProcessorP processor[4];
    // Create group one of processors, the same type butt different in
    performance
    // Link to Library
    const char *powerpc32Model    = icmGetVlnvString(NULL, "power.ovpworld.org",
    "processor",  "powerpc32",         "1.0", "model");
    const char *powerpc32Semihost = icmGetVlnvString(NULL, "power.ovpworld.org",
    "semihosting", "powerpc32Newlib", "1.0", "model");
    // Create Attribute
    icmAttrListP cpu1_attr = icmNewAttrList();
    icmAddStringAttr(cpu1_attr, "endian","big");
    icmAddDoubleAttr(cpu1_attr, "mips", 100);
    processor[0] = icmNewProcessor(
        cpu1,   // name
        "powerpc32",          // type
        0,                    // cpuId
        0,                  // flags
        32,                  // address bits
        powerpc32Model,       // model
        "modelAttrs",         // symbol
        SIM_ATTRS,            // procAttrs
        cpu1_attr,            // attrlist
        powerpc32Semihost,  // semihost file
        "modelAttrs"   );      // semihost symbol
     // Create group two of processors, the same type butt different in
     performance
    // Link to Library
    const char *arm7Model     = icmGetVlnvString(NULL, "arm.ovpworld.org",
    "processor", "arm", "1.0", "model");
    const char *arm7Semihost = icmGetVlnvString(NULL, "arm.ovpworld.org",
    "semihosting", "armNewlib", "1.0", "model");
    // Create Attribute
    icmAttrListP icmAttr = icmNewAttrList();
    icmAddStringAttr(icmAttr, "endian","big");
    icmAddStringAttr(icmAttr, "mips", 100);
    processor[1] = icmNewProcessor(
        cpu2,              // CPU name
        "arm",                // CPU type
        1,                    // CPU cpuId
    0,                   // CPU model flags
    32,                  // address bits
    arm7Model,           // model file
    "modelAttrs",        // morpher attributes
    SIM_ATTRS,           // attributes
    icmAttr,             // user-defined attributes
    arm7Semihost,        // semi-hosting file
    "modelAttrs"  );        // semi-hosting attributes
// Create group three of processors, the same type butt different in
performance
//Link Processors With Library
const char *mips32Model    = icmGetVlnvString(NULL, "mips.ovpworld.org",
"processor", "mips32", "1.0", "model");
const char *mips32Semihost = icmGetVlnvString(NULL, "mips.ovpworld.org",
"semihosting", "mips32SDE", "1.0", "model");
// Create Attribute
icmAttrListP mipsUserAttr0 = icmNewAttrList();
icmAddDoubleAttr(mipsUserAttr0, "endian","big");
icmAddDoubleAttr(mipsUserAttr0, "mips", 100);
processor[2] = icmNewProcessor(
    cp3,              // CPU name
    "mips32",              // CPU type
    2,                    // CPU cpuId
    0,                     // CPU model flags
    32,                    // address bits
    mips32Model,           // model file
    "modelAttrs",          // morpher attributes
    SIM_ATTRS,             // attributes
    mipsUserAttr0,         // user-defined attributes
    mips32Semihost,        // semi-hosting file
    "modelAttrs"      );     // semi-hosting attributes
// Create group four of processors, the same type butt different in
performance
//Link Processors With Library
const char *or1kModel    = icmGetVlnvString(NULL, "ovpworld.org", "processor"
, "or1k", "1.0", "model");
const char *or1kSemihost = icmGetVlnvString(NULL, "ovpworld.org",
"semihosting", "or1kNewlib", "1.0", "model");
// Create Attribute
icmAttrListP or1kUserAttr0 = icmNewAttrList();
icmAddDoubleAttr(or1kUserAttr0, "endian","big");
icmAddDoubleAttr(or1kUserAttr0, "mips", 100);
processor[3] = icmNewProcessor(
   cp4,              // CPU name
    "or1k",                    // CPU type
    3,                         // CPU cpuId
    0,                         // CPU model flags
    32,                        // address bits
    or1kModel,                 // model file
    "modelAttrs",              // morpher attributes
    SIM_ATTRS,                 // attributes
    or1kUserAttr0,             // user-defined attributes
    or1kSemihost,              // semi-hosting file
    "modelAttrs"      );}      // semi-hosting attributes
//create the processor busses
iicmBusP = bus[P]
```

```
for (stepIndex = 0; stepIndex < P; stepIndex++)
    bus[stepIndex] = icmNewBus("stepIndex", 32);
// connect the processors onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectProcessorBusses(processor[stepIndex], stepIndex, stepIndex);
// create memories
icmMemoryP local[P];
for (stepIndex = 0; stepIndex < P; stepIndex++)
local[stepIndex]  = icmNewMemory("stepIndex", ICM_PRIV_RWX, 0x0000ffff);
// create share memories
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x0000ffff);
// connect the share memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
{
char PortName[16];
sprintf(PortName, "mp%d", stepIndex+1);
icmConnectMemoryToBus(bus[stepIndex], PortName, shared, 0x00100000);
}
// connect the  memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectMemoryToBus(bus[stepIndex], "mp1", local[stepIndex], 0x001f0000);
// Load Program to Processors Memories
if(icmLoadProcessorMemory((processor[0], "application.POWERPC32.elf", False,
False, True) &&
                              (processor[1], "application.ARM7.elf", False, False
                              , True)      &&
                              (processor[2], "application.MIPS32.elf", False,
                              False, True)    &&
                              (processor[3], "application.OR1K.elf", False, False
                              , True)      &&
                              ) {

} else {
    return -1;
}
// run platform
icmProcessorP final = icmSimulatePlatform();
// say whether simulation was interrupted
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
    icmPrintf("*** simulation  interrupted\n");
}
 // Free Processors
int stepIndex;
for (stepIndex=0; stepIndex < P; stepIndex++) {
    icmFreeProcessor(processor[stepIndex]);}
return 0; }
```

```
///////////////////
// Appendix A.6  ///
///////////////////
#include <stdio.h>
#include <string.h>
#include <impTypes.h>
#include "icm/icmCpuManager.h"
// enable relaxed scheduling for maximum performance
#define SIM_ATTRS (ICM_ATTR_RELAXED_SCHED)
#define P = 2 // Determine the number of processors these used in  the platform
// Main routine
int main(int argc, char **argv) {
   // initialize OVPsim, enabling verbose mode to get statistics at end
   // of execution
   icmInit(ICM_VERBOSE|ICM_STOP_ON_CTRLC|ICM_ENABLE_IMPERAS_INTERCEPTS, NULL, 0);
   // create an array of pointers to processor instances
   icmProcessorP processor[2];
   // Create group one of processors, the same type butt different in
   performance
   // Link to Library
   const char *powerpc32Model    = icmGetVlnvString(NULL, "power.ovpworld.org",
   "processor",  "powerpc32",        "1.0", "model");
   const char *powerpc32Semihost = icmGetVlnvString(NULL, "power.ovpworld.org",
   "semihosting", "powerpc32Newlib", "1.0", "model");
   // Create Attribute
   icmAttrListP cpu1_attr = icmNewAttrList();
   icmAddStringAttr(cpu1_attr, "endian","big");
   icmAddDoubleAttr(cpu1_attr, "mips", 100);
   processor[0] = icmNewProcessor(
      cpu1,   // name
      "powerpc32",          // type
      0,                    // cpuId
      0,                   // flags
      32,                   // address bits
      powerpc32Model,       // model
      "modelAttrs",         // symbol
      SIM_ATTRS,            // procAttrs
      cpu1_attr,            // attrlist
      powerpc32Semihost,  // semihost file
      "modelAttrs"    );    // semihost symbol
   // Create group two of processors, the same type butt different in
   performance
   // Link to Library
   const char *arm7Model    = icmGetVlnvString(NULL, "arm.ovpworld.org",
   "processor", "arm", "1.0", "model");
   const char *arm7Semihost = icmGetVlnvString(NULL, "arm.ovpworld.org",
   "semihosting", "armNewlib", "1.0", "model");
   // Create Attribute
   icmAttrListP icmAttr = icmNewAttrList();
   icmAddStringAttr(icmAttr, "endian","big");
   icmAddStringAttr(icmAttr, "mips", 100);
   processor[1] = icmNewProcessor(
      cpu2,                 // CPU name
      "arm",                   // CPU type
      1,                       // CPU cpuId
```

```
    0,                          // CPU model flags
    32,                         // address bits
    arm7Model,                  // model file
    "modelAttrs",               // morpher attributes
    SIM_ATTRS,                  // attributes
    icmAttr,                    // user-defined attributes
    arm7Semihost,               // semi-hosting file
    "modelAttrs" );                // semi-hosting attributes
//create the processor busses
iicmBusP = bus[P]
for (stepIndex = 0; stepIndex < P; stepIndex++)
    bus[stepIndex] = icmNewBus("stepIndex", 32);
// connect the processors onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectProcessorBusses(processor[stepIndex], stepIndex, stepIndex);
// create memories
icmMemoryP local[P];
for (stepIndex = 0; stepIndex < P; stepIndex++)
local[stepIndex]  = icmNewMemory("stepIndex", ICM_PRIV_RWX, 0x0000ffff);
// create share memories
icmMemoryP shared = icmNewMemory("shared", ICM_PRIV_RWX, 0x0000ffff);
// connect the share memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
{
char PortName[16];
sprintf(PortName, "mp%d", stepIndex+1);
icmConnectMemoryToBus(bus[stepIndex], PortName, shared, 0x00100000);}
// connect the  memories onto the busses
for (stepIndex = 0; stepIndex < P; stepIndex++)
icmConnectMemoryToBus(bus[stepIndex], "mp1", local[stepIndex], 0x001f0000);
// Load Program to Processors Memories
if(icmLoadProcessorMemory((processor[0], "application.POWERPC32.elf", False,
False, True) &&
                            (processor[1], "application.ARM7.elf", False, False
                            , True)
                            ) {

} else {
    return -1; }
// run.platform
icmProcessorP final = icmSimulatePlatform();
// say whether simulation was interrupted
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
    icmPrintf("*** simulation  interrupted\n");
}
// Free Processors
int stepIndex;
for (stepIndex=0; stepIndex < P; stepIndex++) {
   icmFreeProcessor(processor[stepIndex]);}
return 0;}
```

```
////////////////////////
// Appendix B.1   ///////
////////////////////////
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
int main(int argc, char **argv) {
  int id = impProcessorId(); // to define witch processor are process this file
  printf("CPU %d starting...\n", id);
  FILE *fp;
  // to avoid overlapping in output information create to file to store information
  if (id==0) fp = fopen("DataP0.xls", "w+"); // if processor 0  process store
  information into DtatP0 file
  if (id==1) fp = fopen("DataP1.xls", "w+"); // if processor 1  process store
  information into DtatP0 file
  fprintf(fp, " Processor ID \t Matrix Size\t Instruction Count \n ");
  int I; // Matrix Size
  // begin
  for(I=100;I<1001;++I) {
  int a[I][I],b[I][I],c[I][I]; // Define three matrix, a and p are input matrix,
  c is output matrix
  int i,j,k; // Loop Pointer
  // Input Matrex Data
  for(i=0;i<I;++i)
  for(j=0;j<I;++j)
  {b[i][j] = rand()%255;  // value from 1 to 255
   a[i][j] = rand()%255;}  // value from 1 to 255
  // Muliplication Matrix
  for(i=0;i<I;++i)
  for(j=0;j<I;++j)
 { c[i][j]=0;
  for(k= 0;k<I;++k)
  c[i][j]=c[i][j]+b[i][k]*b[k][j];}
   fprintf(fp,"%d\t,%d\t,%u\n",id,I,impProcessorInstructionCount());// output
   information to excel file
  return 0;}}
```

```
//////////////////////
// Appendix B.2  /////
//////////////////////
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
#include "mmmulticore4_hetero.h"
#define NOOFCPU 4 // number of CPU in platform
// GA parameters
#define POPSIZE 8            // Population size
#define I 1000               // problem size
#define PXOVER 0.6           // probability of crossover
#define PMUTATION 0.25       // probability of mutation
#define MAXGENS 50           // Maximum generations
volatile int *flag = (volatile int *)FLAG;  // LOCK
volatile int *IM = (volatile int *)A;        // INPUT MATRIX
volatile int *OM = (volatile int *)C;        // OUTPUT MATRIX
volatile int *f = (volatile int *)F;         // VECTOR TO CHECK ROW
MULTIPLICATION MATRIX
volatile int *cpu_s = (volatile int *)CPU_S;// PROCESSOR STATUS
volatile long double *cpu_t = (volatile long double *)CPU_T;   // time that need
to execute sample for each CPU
volatile long double *cpu_tt = (volatile long double *)CPU_TT; //
volatile long double *cpu_td = (volatile long double *)CPU_TD; //
volatile long double *cpu_pd = (volatile long double *)CPU_PD; //cpu performance
density
volatile long double *cpu_p = (volatile long double *)CPU_P;   // cpu performance
volatile long double *cpu_pp = (volatile long double *)CPU_PP; //
volatile int *s = (volatile int *)S;//
volatile long double *p = (volatile long double *)P;//
// Create GA gene type
struct genotype {
long double Tfitness;   // execution time
long double Pfitness;   // execution Power
int cpustus[NOOFCPU];   // runing cpu's
long double rfitness;   // relative fitness
long double cfitness; };  // cumulative fitness
struct genotype population[POPSIZE+1];
struct genotype newpopulation[POPSIZE+1];
int mips1[4];
long double fitp,fitt;
// program functions
long double CallPFitness();
long double MinTime();// Calculate reference (minimum) overall execution time
when all processors are sharing
long double MaxPOWER ();
int bin2dec();// Function used to convert processors status to decimal number
void dec2bin(int decimal);  // Function used to convertdecimal number to
processors status
void InputMatrix(int id);   // Enter random value to created input matrix
long double CallTFitness(); // Calculate overall execution time
long double CallPFitness(); // Calculate overall consuming power
void init(int id); // Processors Initializations and repair output excel files
void initialize (); //GA initialization step
void evaluate (void); // GA evaluation step
void select1(void); // GA Selection step
void keep_the_best(); // Keep the best individual in the population
void crossover(void); // GA Crossover step
void Xover(int one, int two); // apply Crossover between two individuals
void mutate(void); // GA Mutation step
void elitist(); // GA Accepting step
void calCPUPerformance (int id); // Function that used to execute sample on each
processor in the platform
void PopPrint(); // Store output information in excel files
FILE *fp,*fp1;
int V[I],V1[I];
int generation,I1;
long double x;
int id;// processor id
long double IIC,iiic;// used in performance time calculation
int ,n=0,ii,z=0,ig,jj,l; // loop pointer
long double fitT,fitP;
int RunP;
int cur_best;  // Pointer to the best individual population
long double MinTime() {
long double sum=0.0;
for ( i=0; i < NOOFCPU; i++)
sum = sum + cpu_p[i];
for ( i=0; i < NOOFCPU; i++){
cpu_td[i]=cpu_p[i]/sum;
}
sum = I*cpu_td[1]*cpu_t[1];
return sum;}
long double MaxPOWER () {int i;
long double sum=0;
for ( i=0; i < NOOFCPU; i++)
sum = sum +p[i];
return sum;}
int bin2dec() {
int i,sum=0;
for (i = 0; i < NOOFCPU; ++i)
sum = sum + s[i]*pow(2,i);
return sum;
}
void dec2bin(int decimal) {
 int i = 0,j;
  int  remain;
 do
 {
   s[i]    = decimal % 2;
   decimal   = decimal / 2;
   ++i;
 } while (decimal > 0);
 for ( j=i+1; j < NOOFCPU; j++)
 s[j] = 0;}
void InputMatrix(int id) {
int i,j;
for(i=0;i<I;++i) V[i] = 2.23;//// vector matrix load on each processor
if (id==1) {
```

```
cpu_tt[0]=0;cpu_pp[0]=0;
for(i=0;i<I;++i){f[i] = 0;}
for(i=0;i<1;++i)
for(j=0;j<I;++j)
IM[j+i*I]=1.23;
printf("\nEnterbMatrex Size = %2d X %d Successful\n",I,I);
do {} while (flag[2] != (NOOFCPU-1));
flag[1] =2;}}
long double CallTFitness() {
long double sum=0.0;
int i;
for ( i=0; i < NOOFCPU; i++)
sum = sum + s[i]*cpu_p[i];
for ( i=0; i < NOOFCPU; i++){
cpu_td[i]=s[i]*cpu_p[i]/sum;}
for ( i=0; i < NOOFCPU; i++)
if (s[i] == 1)
{
sum = I*cpu_td[i]*cpu_t[i];
break;}
return sum;}
long double CallPFitness() {
long double sum=0.0;
int i;
for ( i = 0; i < NOOFCPU; i++)
sum = sum +s[i]*mips1[i];
return sum;}
void init(int id) {
if (id==0) {
 printf(" Starting .......");
 fp = fopen("ph1.xls", "w+");
 fp1 = fopen("ph2.xls", "w+");
 cpu_tt[0]=0; cpu_pp[0]=0;
 mips1[id] = 100;
 p[id] = 155;}
 if (id==1) {
 mips1[id] = 200;
 p[id] = 290;}
 if (id==2) {
 mips1[id] = 100;
 p[id] = 120; }
 if (id==3) {
 mips1[id] = 150;
 p[id] = 95;}}
void initialize (){
int i,j;
population[j].rfitness=0.0;
population[j].cfitness=0.0;
for (j = 0; j < POPSIZE; j++)
        {dec2bin((abs(rand())%170)+1);
   for (i = 0; i < NOOFCPU; i++)
         {population[j].cpustus[i]=s[i];}
       population[j].Tfitness = CallTFitness();
       population[j].Pfitness = CallPFitness();}
       population[POPSIZE].Tfitness =1.0/0.0;
       population[POPSIZE].Pfitness =MaxHz();}
```

```
void evaluate (void){
int i,j;
for (j = 0; j < POPSIZE; j++){
        for (i = 0; i < NOOFCPU; i++)
           s[i] = population[j].cpustus[i];
        population[j].Tfitness = CallTFitness();
        population[j].Pfitness = CallPFitness();
         if (bin2dec() ==0)
         {
         dec2bin((abs(rand())%208)+1);
         population[j].Tfitness = CallTFitness();
         population[j].Pfitness = CallPFitness();}}}
void select1(void) {
int mem, i, j, k;
long double sum = 0;
long double p;
// find total fitness of the population
for (mem = 0; mem < POPSIZE; mem++)
     {sum += population[mem].Tfitness; }
// calculate relative fitness
for (mem = 0; mem < POPSIZE; mem++)
     {population[mem].rfitness =  population[mem].Tfitness/sum;}
population[0].cfitness = population[0].rfitness;
// calculate cumulative fitness
for (mem = 1; mem < POPSIZE; mem++)
     {population[mem].cfitness =  population[mem-1].cfitness +
                            population[mem].rfitness;}
// finally select survivors using cumulative fitness.
for (i = 0; i < POPSIZE; i++)
     {p = rand()%1000/1000.0;
     if (p < population[0].cfitness)
          newpopulation[i] = population[0];
     else
          {
          for (j = 0; j < POPSIZE;j++)
                 if (p >= population[j].cfitness &&
                             p<population[j+1].cfitness)
                      newpopulation[i] = population[j+1];}}
// once a new population is created, copy it back
for (i = 0; i < POPSIZE; i++)
     population[i] = newpopulation[i];              }
void keep_the_best() {
int mem;
int i;
cur_best = 0;  stores the index of the best individual
for (mem = 0; mem < POPSIZE; mem++)
     {
     if (population[mem].Tfitness < population[POPSIZE].Tfitness && population[
     mem].Pfitness <= fitP   )
          {
          cur_best = mem;
          population[POPSIZE].Tfitness = population[mem].Tfitness;
          population[POPSIZE].Pfitness = population[mem].Pfitness; }}
// once the best member in the population is found, copy the genes
for (i = 0; i < NOOFCPU; i++)
     population[POPSIZE].cpustus[i] = population[cur_best].cpustus[i];}
```

```
void calCPUPerformance (int id) {
int ii,i;
IIC= impProcessorInstructionCount();
  for (ii=0;ii<I;ii++)
  V1[ii]=IM[ii];
  cpu_t[id]= impProcessorInstructionCount()/mips1[id];
  for(i=0;i<I;++i)
  { OM[i]=V[i]*cos(V1[i]);}
  cpu_t[id]=impProcessorInstructionCount()/mips1[id]- cpu_t[id];
  cpu_p[id]=1/cpu_t[id];
++flag[3] ;}// all CPU,s are execute the sample
void PopPrint(long double tavge,int g) {i
nt i,j;
printf("gen. mintime-> %d\n",g);
for (j = 0; j < POPSIZE; j++)
          {printf("\n");
          for (i = 0; i < NOOFCPU; i++)
          {s[i]=population[j].cpustus[i];
          fprintf(fp," %d\t %u\t d\t %u\t %u\t"
          ,s[i]  /*CPU STSTUS*/
          ,cpu_td[i] /*CPU SHD*/
          ,round(I*cpu_td[i]) /*CPU SD*/
          ,cpu_t[i] /*CPU SAMPLE EXECUTION TIME */
          ,I*cpu_td[i]*cpu_t[i]); /*CPU OVERALL EXECUTION TIME */
          }
          fprintf(fp,"%d\t%u\t%u\n",j,population[j].Tfitness,population[j].
          Pfitness);}
fprintf(fp1,"%d\t%u\t%u\t%u\n",g,population[POPSIZE+1].Tfitness,tavge,population[
POPSIZE+1].Pfitness);}
void Xover(int one, int two) {
int i,temp;
int point; /* crossover point */
/* select crossover point */
if(NOOFCPU > 1)
  {if(NOOFCPU == 2)
       point = 1;
  else
       point = (rand() % (NOOFCPU - 1)) + 1;
  for (i = 0; i < point; i++)
  {temp = population[one].cpustus[i];
   population[one].cpustus[i] = population[two].cpustus[i];
   population[two].cpustus[i]= temp;}}}
// Crossover: performs crossover of the two selected parents.
void crossover(void) {
int i, mem, one;
int first  =  0; /* count of the number of members chosen */
long double x;
for (mem = 0; mem < POPSIZE; ++mem)

    {x = rand()%1000/1000.0;
     if (x < PXOVER)

          {++first;
          if (first % 2 == 0)
               Xover(one, mem);
          else
```

```
            one = mem;}}}
void mutate(void) {
int i, j;
long double x;
for (i = 0; i < POPSIZE; i++)
    for (j = 0; j < NOOFCPU; j++)
        {
        x = rand()%1000/1000.0;
        if (x < PMUTATION)
            {
            if (population[i].cpustus[j]==1)
                population[i].cpustus[j]=0;
            else
            population[i].cpustus[j]=1;
            }}}
void elitist() {
int i;
long double bestT, worstT; /* best and worst fitness values */
long double bestP, worstP;
int bestT_mem, worstT_mem; /* indexes of the best and worst member */
bestT = population[0].Tfitness;
worstT = population[0].Tfitness;
bestP = population[0].Pfitness;
worstP= population[0].Pfitness;
for (i = 0; i < POPSIZE - 1; ++i)
    {
    if(population[i].Tfitness < population[i+1].Tfitness && population[i].
    Pfitness <= fitP )
        {
        if (population[i].Tfitness <= bestT  && population[i].Pfitness <=
        fitP)
            {
            bestT = population[i].Tfitness;
            bestP = population[i].Pfitness;
            bestT_mem = i;
            }
        if (population[i+1].Tfitness >= worstT)
            {
            worstT = population[i+1].Tfitness;
            worstP = population[i+1].Pfitness;
            worstT_mem = i + 1;}}
    else
        {
        if (population[i].Tfitness >= worstT)
            {
            worstT = population[i].Tfitness;
            worstP = population[i].Pfitness;
            worstT_mem = i;}
        if (population[i+1].Tfitness <= bestT && (population[i+1].Pfitness <=
         fitP ))
            {
            bestT = population[i+1].Tfitness;
            bestP = population[i+1].Pfitness;
            bestT_mem = i + 1;}}}
/* if best individual from the new population is better than */
/* the best individual from the previous population, then    */
/* copy the best from the new population; else replace the
/* worst individual from the current population with the      */
/* best one from the previous generation                       */
if (bestT <= population[POPSIZE].Tfitness && (bestP <= fitP))  */
   {for (i = 0; i < NOOFCPU; i++)
    population[POPSIZE].cpustus[i] = population[bestT_mem].cpustus[i];
   population[POPSIZE].Tfitness = population[bestT_mem].Tfitness;
   population[POPSIZE].Pfitness = population[bestT_mem].Pfitness;}
else
    {for (i = 0; i < NOOFCPU; i++)
     population[worstT_mem].cpustus[i] = population[POPSIZE].cpustus[i];
    population[worstT_mem].Tfitness = population[POPSIZE].Tfitness;
    population[worstT_mem].Pfitness = population[POPSIZE].Pfitness;} }
long double calcPOPSIZE(int g)
{ int i;
long double sum = 0.0, Tavg = 0.0;
for (i = 0; i < POPSIZE ; ++i)
sum = sum +  population[i].Tfitness;
Tavg = sum/POPSIZE;
return Tavg;}
int main(int argc, char **argv) {
int i;
long double mintime ;
id = impProcessorId(); // PROCESSOR ID
flag[1] = 1;// used to enter matrix
cpu_t[id] = 0.0;// used to initial performance time
cpu_p[id] = 0.0;
init(id);//creat output files, and enter MIPS valu for each matrix
InputMatrix(id); // Input Matrex Data and load it in share memory
do {} while (flag[1] ==1);// all processors wait while Enter input matrix to
share memory
calCPUPerformance(id); // Measured a time that need to execute the sample for
each processor
do {} while (flag[3] !=NOOFCPU);//wait for sample processing from all
processors
if (id == 1) // Execute the algorithm on one processor
fitP = 0.9*MaxPOWER;// Determine the value of power
// begin GA procedure
   initialize ();
   evaluate();
   keep_the_best();
   while(generation <= MAXGENS)
     {
   generation++;
   select1();
   crossover();
   mutate();
   evaluate();
   elitist();
   Tavge    =calcPOPSIZE(generation);
   PopPrint(Tavge,generation);   }
fclose(fp)  ;
fclose(fp1)  ;
return 0;}
```

```
///////////////////////////
// Appendix B.3  ///////////
///////////////////////////
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
#include "mmmulticore4_hetero.h"
FILE *fp;
volatile float *a = (volatile float *)A; // INPUT MATRIX
volatile float *l = (volatile float *)L; // LOWER MATRIX
volatile float *u = (volatile float *)U; // UPPER MATRIX
void InputMatrix(int id); // ENTER THE INPUT MATRIX IN SHARE MEMORY
void lu(int id);// MEASURED SAMPLE EXECUTION TIME FOR EACH PROCESSOR
void InputMatrix(int id) {
   int i=0,j=0,k=0;
    for (i=0;i<I;i++)
    for (j=0;j<I;j++)
     a[i][j]=i%(j+1)/3;

   printf("\nEnterbMatrex Size = %2d X %d Successful\n",I,I);}
void lu(int id)
{
fprintf(fp,"%u\t",impProcessorInstructionCount());
for(i=2000;i<2001;i++)
   {
       for(j=0;j<I;j++)
       {
           if(j<i)
               {l[j][i]=0;
               }
           else
           {
               l[j][i]=aa[j][i];
               for(k=0;k<i;k++)
               {
                   l[j][i]=l[j][i]-l[j][k]*u[k][i];
               }}          }
       for(j=0;j<I;j++)
       {
           if(j<i)
               {u[i][j]=0;}
           else if(j==i)
               u[i][j]=1;
           else
           {
               u[i][j]=aa[i][j]/l[i][i];
               for(k=0;k<i;k++)
               {
                   u[i][j]=u[i][j]-((l[i][k]*u[k][j])/l[i][i]);}}}}
   fprintf(fp,"%u\n",impProcessorInstructionCount());   }
int main(int argc, char **argv) {
int i;
id = impProcessorId(); // PROCESSOR ID
// EXPORT INFORMATION IN A SEPARATE FOLDER TO AVOID OVERLAPPING
if (id == 0) fp = fopen("POWERPC32/ph1.xls", "w+");
if (id == 0) fp = fopen("ARM/ph1.xls", "w+");
if (id == 0) fp = fopen("MIPS32/ph1.xls", "w+");
if (id == 0) fp = fopen("OR1K/ph1.xls", "w+");
InputMatrix(id);
lu(id);
fclose(fp) ;
printf("\nfinish--------------\n");
 return 0;}
```

```
///////////////////////////
// Appendix B.4  //////
///////////////////////////
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
#include "mmmulticore4_hetero.h"
#Define I = 10000 // PROBLEM SIZE
#Define NOOFCPU = 8 // NUMBER OF PROCESSORS IN PLATFORM

FILE *fp;
volatile float *f = (volatile float *)FLAG; // FLAGE, ALL PROCESSOR  WAIT DURING
CALCULATE SHARING FOR EACH ONE
volatile float *a = (volatile float *)A; // INPUT MATRIX store in share memory
volatile float *l = (volatile float *)L; // LOWER MATRIX store in share memory
volatile float *u = (volatile float *)U; // UPPER MATRIX store in share memory
volatile float *Time = (volatile float *)TIME; // TOTAL EXECUTION TIME
volatile float *Performance = (volatile float *)PERFORMANCE; // PERFORMANCE FOR
EACH PROCESSOR
volatile float *cpu_sh = (volatile float *)CPU_SH; // SHARING DATA FOR EACH
PROCESSOR
double long IC; // INITIAL POINT USE TO CALCULATE SAMPLE EXECUTION TIME AND
TOTAL EXECUTION TIME
int SH[NOOFCPU]; // processor sharing (=1)in processing or no (=0)
void InputMatrix(int id)// SUB ROUTINE TO INPUT A MATRIX
void samplelu(int id) //SUB ROUTINE TO MEASURE SAMPLE EXECUTION TIME, CALCULATE
LOWER AND UPPER FOR ONE ROW
void ClacSharing(int conf,int id) // SUB ROUTINE TO CALCULATE DATA SHARING FOR
EACH PROCESSOR
void Distributelu(int conf,int id) // SUB ROUTINE TO DISTRIBUTE DATA SHARING TO
PROCESSORS AND MEASURE TOTAL EXECUTION TIME FOR EACH ONE
void PrintResult(in id, FILE *fp,int conf) // SUB ROUTINE TO OUTPUT THE RESULT
void dec2bin(int decimal) {
 int i = 0,j;
  int  remain;
  do
  {
    SH[i]    = decimal % 2;
    decimal   = decimal / 2;
    ++i;
 } while (decimal > 0);

  for ( j=i+1; j < NOOFCPU; j++)
  SH[j] = 0;}
void InputMatrix(int id) {
    int i=0,j=0,k=0;
     for (i=0;i<I;i++)
     for (j=0;j<I;j++)
      a[i+I*j]=rand()/255; //RANDOM VALUE FROM 0 TO 255
    printf("\nEnterbMatrex Size = %2d X %d Successful\n",I,I);}
void samplelu(int id)
{
IC = impProcessorInstructionCount();
```

```
for(i=2000;i<2001;i++)
  {     for(j=0;j<I;j++)
        {
           if(j<i)
               {l[j][i]=0;
               }
           else
           {
               l[j][i]=aa[j][i];
               for(k=0;k<i;k++)
               {
                   l[j][i]=l[j][i]-l[j][k]*u[k][i];
               }}}
       for(j=0;j<I;j++)
       {
           if(j<i)
               {u[i][j]=0;}
           else if(j==i)
               u[i][j]=1;
           else
           {
               u[i][j]=aa[i][j]/l[i][i];
               for(k=0;k<i;k++)
               {
                   u[i][j]=u[i][j]-((l[i][k]*u[k][j])/l[i][i]);}}}}
   Time[id] = (impProcessorInstructionCount()-IC)/mips(id));
   Performance[id] = 1/Time[id];}
void ClacSharing(int config,int id)
{
long double  sum=0.0;
int i;
dec2bin(config)
if (id==0)
{
for ( i=0; i < NOOFCPU; i++)
sum = sum + SH[i]*Performance[i];
for ( i=0; i < NOOFCPU; i++){
cpu_sh[i]=round(SH[i]*I*Performance[i]/sum);
f[1] =1;}}}
void Distributelu(int id)
{
IC = impProcessorInstructionCount();
for(i=0;i<cpu_sh[id];i++)
   {
       for(j=0;j<I;j++)
       {
           if(j<i)
               {l[j][i]=0;
               }
           else
           {
               l[j][i]=aa[j][i];
               for(k=0;k<i;k++)
               {
                   l[j][i]=l[j][i]-l[j][k]*u[k][i];}}}
```

```
    for(j=0;j<I;j++)
    {
        if(j<i)
            {u[i][j]=0;}
        else if(j==i)
            u[i][j]=1;
        else
        {
            u[i][j]=aa[i][j]/l[i][i];
            for(k=0;k<i;k++)
            {
                u[i][j]=u[i][j]-((l[i][k]*u[k][j])/l[i][i]);}}}}
 Time[id] = (impProcessorInstructionCount()-IC)/mips(id));
flag[2]++;}
id PrintResult(in id, FILE *fp, int config)

t i;
f (id==0)

rintf(fp,"CPU ID\t Config \tTOTAL EXECUTION TIME\n");
r (i=0; NOOFCPU; i++)
rintf(fp,"%d\t%d\t%u\n",i,config,Time[i]);}}
t main(int argc, char **argv) {
t i;
ng double config;
nfig = pow(2,NOOFCPU)
har file_name[16];
  sprintf(file_name, "platform%d", NOOFCPU);
p = fopen(file_name, "w+");
d = impProcessorId(); // GET PROCESSOR ID
nputMatrix(id);
amplelu(id);
[1] = 0;
or (i = 0 ; i < config; i++)
{
lacSharing(i,id);
o {} wile(f[1] == 0);
istributelu(id);
o {} wile(f[2] < NOOFCPU);
rintResult(id,fp,i);}
close(fp) ;
rintf("\nfinish--------------\n");
return 0;}
```

```
/////////////////////////
// Appendix B.5  ///////
/////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <impTypes.h>
#include <string.h>
#include "mmmulticore4_hetero.h"
#define NOOFCPU 4
FILE *fp;
volatile float *aa = (volatile float *)A; // INPUT IMAGES MATRIX
volatile float *ll = (volatile float *)L; // OUTPUT IMAGES MATRIX
// sub rotine to execute one image (sample)
void FilterImage(int id)
{
for (I=  500; I <3001; I=I+500)
{printf("\n begin I = %d-------------\n",I);
float aa[I][Im],ll[I][Im];
int i=0,j=0,k=0;

for (i=0;i<1;i++)
for (j=0;j<Im;j++)
   aa[i][j]=j%255;
    fprintf(fp,"%d\t%u\t",I,impProcessorInstructionCount());
 for(i=0;i<Im;i++)
    {
    ll[0][i]= (aa[0][i]*(2*i))/(aa[0][i]*(i+1)) * aa[0][i];
    }
    fprintf(fp,"%u\n",impProcessorInstructionCount());  }}
 int main(int argc, char **argv) {
 int i;
 // TO AVOID OVERLAPPING
 if (id == 0) fp = fopen("P1/ph1.xls", "w+");
 if (id == 1) fp = fopen("P2/ph1.xls", "w+");
 if (id == 2) fp = fopen("P3/ph1.xls", "w+");
 if (id == 3) fp = fopen("P4/ph1.xls", "w+");
 id = impProcessorId(); // PROCESSOR ID
 FilterImage(id);
 fclose(fp) ;
 printf("\nfinish-------------\n");
  return 0;
 }
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Appendix C.1: return to chapter three implementation %%
%% explain source code .m that use to analysis Ei       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Dp0=xlsread('d:\TP\GAMULC4\ph1');
Dp1=xlsread('d:\TP\GAMULC4\ph2');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% ONE GENERATION    %%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
CPU=[Dp0(:,1),Dp0(:,6),Dp0(:,11),Dp0(:,16)];%CPU,s STATUS
SHD=[Dp0(:,2),Dp0(:,7),Dp0(:,12),Dp0(:,17)];% CPU,s SHD
SD=[Dp0(:,3),Dp0(:,8),Dp0(:,13),Dp0(:,18)];% CPU,s SD
SET=[Dp0(:,4),Dp0(:,9),Dp0(:,14),Dp0(:,19)];%CPU SAMPLE EXECUTION TIME
OVALLET=[Dp0(:,5),Dp0(:,10),Dp0(:,15),Dp0(:,20)];%CPU OVERALL EXECUTION TIME
% SHOW RESULT
xlswrite('d:\TP\GAMULC4\SHD.xls',[CPU,SHD] );
xlswrite('d:\TP\GAMULC4\SET.xls',[CPU,SET] );
xlswrite('d:\TP\GAMULC4\SET.xls',[CPU,SET] );
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MANY GENERATION   %%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
GENERATION = Dp1(:,1);%GENERATIONS
MINET = Dp1(:,2);%MINIMUM EXECUTION TIME
AVGET= Dp1(:,3);%AVARAGE EXECUTION TIME
CONPOWE = Dp1(:,4);%CONSUMING POWER
```

```
##########################################################################
## Appendix C.2: return to chapter three implementation
## explain source code .make that use to compile experiment files      ##
##########################################################################
IMPERAS_HOME := $(shell getpath.exe "$(IMPERAS_HOME)")
# Build locally without using a VLNV library
NOVLNV=1
# Build using two passes so that each pas includes separate Makefiles
MAKEPASS?=0
ifeq ($(MAKEPASS),0)
all:
  $(MAKE) MAKEPASS=1
  $(MAKE) MAKEPASS=2
  $(MAKE) MAKEPASS=3
  $(MAKE) MAKEPASS=4
  $(MAKE) MAKEPASS=5

clean:
  $(MAKE) MAKEPASS=1 clean
  $(MAKE) MAKEPASS=2 clean
  $(MAKE) MAKEPASS=3 clean
  $(MAKE) MAKEPASS=4 clean
  $(MAKE) MAKEPASS=5 clean
endif
# Pass 1 build the Platform
ifeq ($(MAKEPASS),1)
SRC=platform.c
include $(IMPERAS_HOME)/ImperasLib/source/buildutils/Makefile.platform
endif
# Pass 2 build the Application
ifeq ($(MAKEPASS),2)
OPTIMISATION?=-O2
CROSS=ARM7
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
endif
ifeq ($(MAKEPASS),3)
OPTIMISATION?=-O2
CROSS=OR1K
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
endif
ifeq ($(MAKEPASS),4)
OPTIMISATION?=-O2
```

```
CROSS=V850
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
ifeq ($($(CROSS)_CC),)
   IMPERAS_ERROR := $(error "Error : $(CROSS)_CC not set. Please check
   installation of toolchain for $(CROSS)")
endif
endif
# Pass 2 build the Application
ifeq ($(MAKEPASS),5)
OPTIMISATION?=-O2
CROSS=MIPS32
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
ifeq ($($(CROSS)_CC),)
   IMPERAS_ERROR := $(error "Error : $(CROSS)_CC not set. Please check
   installation of toolchain for $(CROSS)")
endif
endif
applicationFiles: $(EXE)
%.$(CROSS).elf: %.$(CROSS).o
   $(V)  echo "Linking $@"
   $(V)  $(IMPERAS_LINK) -o $@ $< $(IMPERAS_LDFLAGS) -lm
%.$(CROSS).o: %.c
   $(V)  echo "Compiling $<"
   $(V)  $(IMPERAS_CC) -c -o $@ $< -O2
```

```
%%%%%%%%%%%%%%%%%%%%%%%
%% Appendix C.3   %%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%
%%determine max Measurements  over all execution time (finish execution time)
for i=1:NOOFCPU*power(2,NOOFCPU(1))-1
    if (TimeP4(i,2) == TimeP4(i-1,2))
      if ( TimeP4(i,3) > TimeP4(i-1,3)
           max4 = TimeP4(i,3);

      end
      MesurTime4(TimeP4(i-1,2)) = max4;

    end
for i=1:NOOFCPU*power(2,NOOFCPU(2))-1
    if (TimeP8(i,2) == TimeP8(i-1,2))
      if ( TimeP8(i,3) > TimeP8(i-1,3)
           max8 = TimeP8(i,3);

      end
      MesurTime8(TimeP8(i-1,2)) = max8;

    end
for i=1:NOOFCPU*power(2,NOOFCPU(3))-1
    if (TimeP16(i,2) == TimeP16(i-1,2))
      if ( TimeP16(i,3) > TimeP16(i-1,3)
           max16 = TimeP16(i,3);

      end
      MesurTime16(TimeP16(i-1,2)) = max16;

    end
    for i=1:NOOFCPU*power(2,NOOFCPU(4))-1
    if (TimeP32(i,2) == TimeP32(i-1,2))
      if ( TimeP32(i,3) > TimeP32(i-1,3)
           max32 = TimeP32(i,3);

      end
      MesurTime32(TimeP32(i-1,2)) = max32;

    end
for i=1:NOOFCPU*power(2,NOOFCPU(5))-1
    if (TimeP64(i,2) == TimeP64(i-1,2))
      if ( TimeP64(i,3) > TimeP64(i-1,3)
           max64 = TimeP64(i,3);

      end
      MesurTime64(TimeP64(i-1,2)) = max64;

    end
for i=1:NOOFCPU*power(2,NOOFCPU(6))-1
    if (TimeP128(i,2) == TimeP128(i-1,2))
      if ( TimeP128(i,3) > TimeP128(i-1,3)
           max128 = TimeP128(i,3);

      end
      MesurTime128(TimeP128(i-1,2)) = max128;
    end
%Calculate error between calculation and measurement
ERROR4 = ((MesurTime4(:)-MAXTIME4(:)) / MesurTime4(:)) *100;
ERROR8 = ((MesurTime8(:)-MAXTIME8(:)) / MesurTime8(:)) *100;
ERROR16 = ((MesurTime16(:)-MAXTIME16(:)) / MesurTime16(:)) *100;
ERROR32 = ((MesurTime32(:)-MAXTIME32(:)) / MesurTime32(:)) *100;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Appendix C.4  %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%inport sample execution time information tables
data1=xlsread('D:\TP\IMP1\POWERPC32\ph1');
data2=xlsread('D:\TP\IMP1\ARM\ph1');
data3=xlsread('D:\TP\IMP1\MIPS32\ph1');
data4=xlsread('D:\TP\IMP1\OR1K\ph1');
D = =xlsread('D:\TP\IMP1\MIPS');
%Extract instructions count for each processor
IC1 = data1(:,2) - data1(:,1);
IC2 = data2(:,2) - data2(:,1);
IC3 = data3(:,2) - data3(:,1);
IC4 = data4(:,2) - data4(:,1);
%Collect mips values for every platform processors
MIPS4 = D(4/4:4/4+4/4-1,2:5);
MIPS8 = D(8/4:8/4+8/4-1,2:5);
MIPS16 = D(16/4:16/4+16/4-1,2:5);
MIPS32 = D(32/4:32/4+32/4-1,2:5);
MIPS64 = D(64/4:64/4+64/4-1,2:5);
MIPS128 = D(128/4:128/4+128/4-1,2:5);
%Calculate the time need to execute sample for each processor in the platforms
TIME4 = [IC1./MIPS4(:,1);IC2./MIPS4(:,2);IC3./MIPS4(:,3);IC4./MIPS4(:,4)];
TIME8 = [IC1./MIPS8(:,1);IC2./MIPS8(:,2);IC3./MIPS8(:,3);IC4./MIPS8(:,4)];
TIME16 = [IC1./MIPS16(:,1);IC2./MIPS16(:,2);IC3./MIPS16(:,3);IC4./MIPS16(:,4)];
TIME32 = [IC1./MIPS32(:,1);IC2./MIPS32(:,2);IC3./MIPS32(:,3);IC4./MIPS32(:,4)];
TIME64 = [IC1./MIPS64(:,1);IC2./MIPS64(:,2);IC3./MIPS64(:,3);IC4./MIPS64(:,4)];
TIME128 = [IC1./MIPS128(:,1);IC2./MIPS128(:,2);IC3./MIPS128(:,3);IC4./MIPS128(:,4)];
I = 1000;% detemine problem size
% Collect number of processors for all platforms in one matrix
NOOFCPU = [length(TIME4);length(TIME8);length(TIME16);length(TIME32);length(TIME64);
length(TIME128)];
%Building a matrix of all possibilities for the participation of processors
CONFIG4 = de2bi([1:power(2,NOOFCPU(1))-1]);
CONFIG8 = de2bi([1:power(2,NOOFCPU(2))-1]);
CONFIG16 = de2bi([1:power(2,NOOFCPU(3))-1]);
CONFIG32 = de2bi([1:5000],32);
CONFIG64 = de2bi([1:5000],64);
CONFIG128 = de2bi([1:5000],128);
%Calculate over all execution time for every platform, C for sharing density, K for
sharing Data, and Fit_T for overall execution time
C4 =CallFitTP(TIME4,CONFIG4,I);
K4 = round(I*C4(:,:));
for r=1:length(K4)
    for i=1:NOOFCPU(1)
        fit_T4(r,i)=K4(r,i)*TIME4(i);
    end
end
C8 =CallFitTP(TIME8,CONFIG8,I);
K8 = round(I*C8(:,:));
for r=1:length(K8)
```

```
    for i=1:NOOFCPU(2)
        fit_T8(r,i)=K8(r,i)*TIME8(i);
    end
end
C16 =CallFitTP(TIME16,CONFIG16,I);
K16 = round(I*C16(:,:));
for r=1:length(K16)
    for i=1:NOOFCPU(3)
        fit_T16(r,i)=K16(r,i)*TIME16(i);
    end
end
C32 =CallFitTP(TIME32,CONFIG32,I);
K32 = round(I*C32(:,:));
for r=1:length(K32)
    for i=1:NOOFCPU(4)
        fit_T32(r,i)=K32(r,i)*TIME32(i);
    end
end
C64 =CallFitTP(TIME64,CONFIG64,I);
K64 = round(I*C64(:,:));
for r=1:length(K64)
    for i=1:NOOFCPU(5)
        fit_T64(r,i)=K64(r,i)*TIME64(i);
    end
end
C128 =CallFitTP(TIME128,CONFIG128,I);
K128 = round(I*C128(:,:));
for r=1:length(K128)
    for i=1:NOOFCPU(6)
        fit_T128(r,i)=K128(r,i)*TIME128(i);
    end
end
%%determine max calculations over all execution time (finish execution time)
MAXTIME4  = max(fit_T4');
x4 = bi2de(CONFIG4);
MAXTIME8  = max(fit_T8');
x8 = bi2de(CONFIG8);
MAXTIME16  = max(fit_T16');
x16 = bi2de(CONFIG16);
MAXTIME32  = max(fit_T32');
x32 = bi2de(CONFIG32);
MAXTIME64  = max(fit_T64');
x64 = bi2de(CONFIG64);
MAXTIME128  = max(fit_T128');
x128 = bi2de(CONFIG128);
%inport Measured overall execution time
TimeP4 = xlsread('D:\TP\IMP1\platform4');
TimeP8 = xlsread('D:\TP\IMP1\platform8');
TimeP16 = xlsread('D:\TP\IMP1\platform16');
TimeP32 = xlsread('D:\TP\IMP1\platform32');
TimeP64 = xlsread('D:\TP\IMP1\platform64');
TimeP128 = xlsread('D:\TP\IMP1\platform128');
```

```
#################################################################
## Appendix C.5  return to chapter four implementation 1       ##
## explain source code .make that use to compile experiment files  ##
#################################################################
IMPERAS_HOME := $(shell getpath.exe "$(IMPERAS_HOME)")
# Build locally without using a VLNV library
NOVLNV=1
# Build using two passes so that each pas includes separate Makefiles
MAKEPASS?=0
ifeq ($(MAKEPASS),0)
all:
   $(MAKE) MAKEPASS=1
   $(MAKE) MAKEPASS=2
   $(MAKE) MAKEPASS=3
   $(MAKE) MAKEPASS=4
clean:
   $(MAKE) MAKEPASS=1 clean
   $(MAKE) MAKEPASS=2 clean
   $(MAKE) MAKEPASS=3 clean
   $(MAKE) MAKEPASS=4 clean

endif
# Pass 1 build the Platform
ifeq ($(MAKEPASS),1)
SRC=platform.c
include $(IMPERAS_HOME)/ImperasLib/source/buildutils/Makefile.platform
endif
# Pass 2 build the Application to execute on ARM processor
ifeq ($(MAKEPASS),2)
OPTIMISATION?=-O2
CROSS=ARM7
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
endif
# Pass 3 build the Application to execute on OR1K processor
ifeq ($(MAKEPASS),3)
OPTIMISATION?=-O2
CROSS=OR1K
SRC?=application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
endif
# Pass 4 build the Application to execute on POWERPC32 processor
ifeq ($(MAKEPASS),4)
```

```
OPTIMISATION?=-O2
CROSS=POWERPC32
SRC?=D:/TP/IMP1/application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
ifeq ($($(CROSS)_CC),)
   IMPERAS_ERROR := $(error "Error : $(CROSS)_CC not set. Please check
   installation of toolchain for $(CROSS)")
endif
# Pass 5 build the Application to execute on MIPS32 processor
ifeq ($(MAKEPASS),5)
OPTIMISATION?=-O2
CROSS=MIPS32
SRC?=D:/TP/IMP1/application.c
SRCINC?=
INCOBJ=$(patsubst %.c,%.$(CROSS).o,$(SRCINC))
EXE=$(patsubst %.c,%.$(CROSS).elf,$(SRC))
-include $(IMPERAS_HOME)/bin/Makefile.include
-include $(IMPERAS_LIB)/CrossCompiler/$(CROSS).makefile.include
ifeq ($($(CROSS)_CC),)
   IMPERAS_ERROR := $(error "Error : $(CROSS)_CC not set. Please check
   installation of toolchain for $(CROSS)")
endif
applicationFiles: $(EXE)
%.$(CROSS).elf: %.$(CROSS).o
   $(V) echo "Linking $@"
   $(V)  $(IMPERAS_LINK) -o $@ $< $(IMPERAS_LDFLAGS) -lm
%.$(CROSS).o: %.c
   $(V)  echo "Compiling $<"
   $(V)  $(IMPERAS_CC) -c -o $@ $< -O2
```

```
%%%%%%%%%%%%%%%%%%%%%
%% Appendix C.6  %%%%
%%%%%%%%%%%%%%%%%%%%%
%Initialization
clear all;
clc
I = 100000; % Problem size
%Import the time need to execute the sample  for each processor type.
       %information from Pprocessor1
       P1IMP1=xlsread('D:\TP\IMP3\ARM\linpack');
       P1IMP2=xlsread('D:\TP\IMP3\ARM\peakspead');
       P1IMP3=xlsread('D:\TP\IMP3\ARM\dhrystone');

       %information from Pprocessor2
       P2IMP1=xlsread('D:\TP\IMP3\OR1K\linpack');
       P2IMP2=xlsread('D:\TP\IMP3\OR1K\peakspead');
       P2IMP3=xlsread('D:\TP\IMP3\OR1K\dhrystone');
%The number of instructionss needed by processor1 to execute the benchmark sample
ICP1(1) = P1IMP1(:,2) - P1IMP1(:,1);
ICP1(2) = P1IMP2(:,2) - P1IMP2(:,1);
ICP1(3) = P1IMP3(:,2) - P1IMP3(:,1);
%The number of instructionss needed by processor2 to execute the benchmark sample
ICP2(1) = P2IMP1(:,2) - P2IMP1(:,1);
ICP2(2) = P2IMP2(:,2) - P2IMP2(:,1);
ICP2(3) = P2IMP3(:,2) - P2IMP3(:,1);
%Import MIPS and Consuming power information for each processor type.
DMIPS=xlsread('D:\TP\IMP3\mips');
DPOWER=xlsread('D:\TP\IMP3\POWER');
%Create vector matrix that represent MIPS information
MIPS4 = [DMIPS(:,1),DMIPS(:,2)];
[RP CP ] = size(MIPS4);
%Create vector matrix that represent consumed power information for all processor
PR = [DPOWER(1)* ones(RP,1);DPOWER(2)* ones(RP,1)];
%Calculate execution time for the samples on platform (all processors)
for i = 1:3
TIME(:,i) = [ICP1(i)./MIPS4(:,1);ICP2(i)./MIPS4(:,2)];
end
% Begin GA
G=10;% max generation
MinT=zeros(3,G);%Initialized minimum overall execution time
Avg=zeros(3,G);%Initialized average overall execution time
PWR=zeros(3,G);%Initialized overall consuming power
%Run GA for each benchmark separately  (z=1 for linpack, z=2 peakspead, and z=3
dhrystone)
for z = 1:3
T=TIME(:,z);
SH = CallSHD(T,ones(1,length(T)));% Calculate sharing density when all processors
are sharing
TSH(:,z)= SH;%Calculate sharing density for each benchmark separately
SD = round(I*TSH(:,:));%Calculate sharing data for each benchmark separately
%Initialized GA parameter
```

```
NOOFCPU = length(T); %Number of CPUs (gene variable)
POPSIZE = NOOFCPU+3;%Population size
PXOVR = 0.80;%Cross over probability
PMUT = 0.1;%Mutate probability
%Calculate reference (minimum) overall execution time when all processors are
sharing
%Calculate reference (maximum) overall consuming power when all processors are
sharing
MinFitness =  INIFTCallFitT(T,PR,ones(1,NOOFCPU),I);
%Decrease the value of power 10% from the maximum for each step
for PHA = 1:10
MODPOW = MinTimePower(2,z)*PHA*0.1;
%GA initialization step
X = init(POPSIZE,NOOFCPU);
%GA evaluation step
C =CallFitT(T,PR,X,I);
%Keep the best individual in the population
K = Keep_The_Pest(MODPOW,C);
% GA loop
for ii=1:G
S = select(K);%Selection step
XV = crossover(S,PXOVR);%Crossover step
M =mutate(XV,PMUT);%Mutation step
CF =CallFitT(T,PR,M,I);%evaluation step
K= elitist(MODPOW,CF);%Accepting step
%Keep information (minimum execution time, average execution time, and consuming
power) for each generation
[r c] = size(K);
MinT(z,ii)=K(r,c-3);
PWR(z,ii) = K(r,c-2);
Avg(z,ii)=sum(K(1:r-1,c-3))/(length(K)-1);
%Termination criteria
if (ii>5)
if ( MinT(z,ii)=MinT(z,ii-1)&&MinT(z,ii)=MinT(z,ii-2)&&MinT(z,ii)=MinT(z,ii-3)&&MinT
(z,ii)=MinT(z,ii-4)&&MinT(z,ii)=MinT(z,ii-5))
break
end
end
end
%Keep information (minimum execution time, average execution time, and consuming
power) for each consuming power value
POWER(z,PHA) =MODPOW;
AVARAGE(z,PHA)= Avg(z,end);
MINIMUM (z,PHA)= MinT(z,end);
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%show result   %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Plot(1:NOOFCPU,TIME) % show the time that need to execute the sample for each
processor
plot(1:NOOFCPU,TSH)  % show sharing density for each processor
plot(1:NOOFCPU,SD) %show sharing data for each processor
plot(POWER,AVARAGE,POWER,MINIMUM)%Show relation between the power and over all
execution time (average and minimum)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Explain the source code for the function that used to%%
%calculate reference time and power values           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function INIFT = INIFTCallFitT(t,pr,x,I)
INIFT = zeros(1,2);
[r c] = size(x);
sum = 0
%Calculate processors sharing density
   for j=1:c
       sum = sum+ x(j)*1/t(j);
   end
   for j=1:c
       fitP(j) = x(j)*(1/t(j))/sum;

   end
%calculate over all execution time
   for j=1:c
       if x(j)==1
       INIFT(1) = I*fitP(j)*t(j);
       end
   end
%calculate over all consuming power
INIFT(2)=0;
   for j=1:c
       INIFT(2) = INIFT(2)+ pr(j);
   end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Explain the source code for the function that used to Initial population%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function population = init(p,n)
B =zeros(1,n+4);
%Random population
population=round(rand(p+1,n+4));
%Validate 0,s constraint. Avoid any population not share all processors
for i = 1 : p
    if bi2de(population(i,:)==0)
       B = de2bi(i,n+4);
       population(i,:)=B(:);
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Explain the source code for the function that used to Evaluate the population%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function FT = CallFitT(t,pr,x,I)
FT = x;
[r c] = size(x);
fitP = zeros(r,c-4);
sum = zeros(r);
%Calculate processors sharing density for each population
```

```
function MUT = mutate(x,MP)
MUT = x;
[r c] = size(MUT);
for i=1:r-1
    for j=1:c-4
        if rand < MP
            if ~MUT(i,j);
                MUT(i,j)=1;


            end
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Explain the source code for the function that used to %
%Find and save the pest population                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function EL = elitist(MODPOW,x)
p=1/0;
EL=x;
[r c] = size(EL);
mem =r;
for i=1:r-1
    if (x(i,c-3)< p && x(i,c-2)< MODPOW)
        p = x(i,c-3);
        mem = i;
    end
end
if x(mem,c-3) < x(r,c-3)
    for i=1:c
    EL(r,i)=x(mem,i);
    end
end
```