A DYNAMIC RECONFIGURABLE COMPUTER WITH A DYNAMIC GENETIC ALGORITHM

By

KAZUNORI NISHIMURA

Associate of Arts

Central Christian College of Kansas

McPherson, Kansas

2002

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

2006

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2008

A DYNAMIC RECONFIGURABLE COMPUTER WITH

A DYNAMIC GENETIC ALGORITHM

Thesis Approved:

Dr. Sohum Sohoni

Thesis Advisor

Dr. Louis Johnson

Dr. Weihua Sheng

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGEMENT

First of all, I thank my parents, Mr. Kazuo and Ms. Reiko Nishimura, for their support. Without their help, I would not be able to come to U.S.A and continue studying for a long time. Also I thank my sister and brother for their patience and my host family for their support. I appreciate all faculty members that I took classes from. Among the faculty members, I give special remarks to my thesis committee members, Dr. Sohum Sohoni, Dr. Louis Johnson, and Dr. Weihua Sheng. Dr. Sohoni offered me the opportunities to use research resources and advice to complete my research. Without his help, I would not be able to complete my research. Also I thank to people in Writing Center for their help in the thesis edition and the lab assistants in CEAT labs and Alvin Ng for special accommodation on labs resources.

Kazunori Nishimura

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURE

LIST OF FIGURE

Table **Page**

LIST OF FIGURE

LIST OF FIGURE

Table **Page**

LIST OF TABLES

Table **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

As a result of the major technology boost after World War II, some of the things that we had not even imagined have come true. Examples of such kinds of dreams are space stations, robots, digital cameras, mobile phones, handhelds, portable PCs, and portable music players. Many technological improvements and realization of dreams came from the invention of the transistor, and continuing improvement of transistor technology following Moore's Law, which predicts the growth of number of transistor in a single chip. This law predicts that the total number of devices in a chip will double every 12 months in the 1970s and the number of transistors will grow slower in the 1980s (it would be double every 24 months) [1].

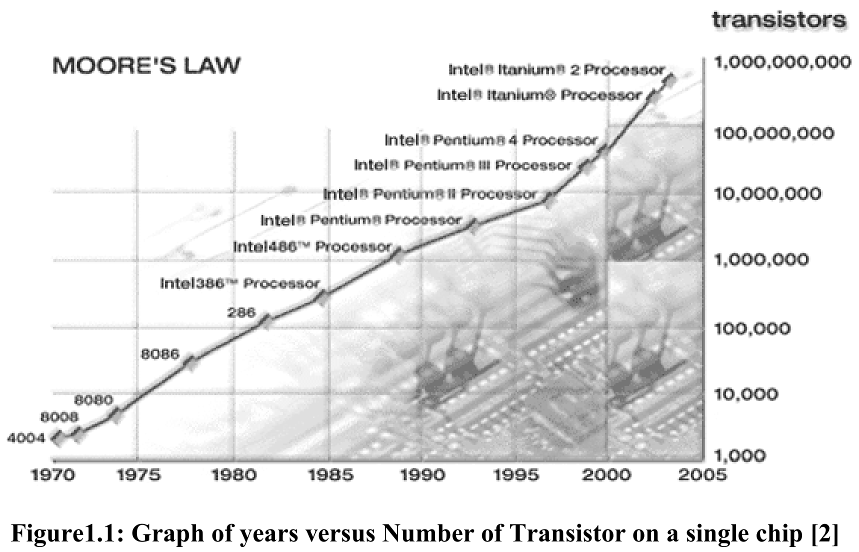


Figure1.1: Graph of years versus Number of Transistor on a single chip [2]

The dramatic increase in the number of transistor in a single chip (depicted in Figure 1.1 above), and the reduction in gains from aggressive superscalar techniques has led to multi-core architecture for the CPU. Examples of multi-core CPU architectures are Intel® Core 2™ Duo, Core 2™ Quad or cell processor that was invented by IBM and Toshiba [3] – [5]. Since multi-cores superscalar architecture can have more processing power compared to single core [3] – [5], it is natural trend to change the computer architecture to obtain higher processing performance.

Also there is another trend in the area of computer architecture research. Many computer researchers set their focus on the reconfigurable computer since the reconfigurable computer has the potential to achieve higher processing performance compared to the processing performance of single core CPU architecture [6] – [8]. Even though we are using the same terminology *reconfigurable* computer, it has different meaning and represents different system for different researches. In other words, there is no clear single definition for the *reconfigurable* computer, and the meaning depends on the purpose of the research. Therefore we clarify the definition of reconfigurable computer in this study with several examples.

1.0 Definition of reconfigurable computer

Among different definitions of *reconfigurable* computer, there is one common property that we can easily find. The *reconfigurable* computer has the ability to adjust functionalities or architecture to achieve the correct functionalities or the superior performance. Keeping in mind this common property, we use the term *reconfigurable* computer as the computer system that has the abilities to adjust the architecture and functionalities to achieve a specific objective. In our definition of *reconfigurable* computer, we can identify several different types of *reconfigurable* systems. One example of a reconfigurable system is the Central Processing Unit (CPU) of a general purpose computer. The current CPUs in the market have the multiple functionalities to implement the different logical operations and arithmetic operations. Examples of logic operations are AND, OR, XOR and NOT. Examples of arithmetic operations are addition, subtraction and multiplications. Among the different operations, the control command in the instruction set reconfigure the CPU to achieve the correct functionalities to process information [9]. Another example of *reconfigurable* computer is the current computer system. Since we have sufficient chips on the current high-end motherboards, we can operate the computer systems without any expansion cards. Although it is not necessary to install expansion cards, we usually enhance the performance of the computer system by installing graphic cards and sound cards and so on. In this case, we modify the system configuration by adding more resources to improve the performance.

In the previous paragraph, the first case also shows the example of run-time or *dynamic reconfigurable* computer. The *dynamic reconfigurable* computer changes its system configuration or architecture during the time when operations of the system are executing. The second example displays the off-line or *static reconfigurable* computer. The *static reconfigurable* computer changes its characteristics during the time when the system is not active or is turned off. In the other words, the reconfiguration is not only happened when the system is on. Table 1.1 summarizes the types of reconfigurable computer we describe in this section.

Table 1.1 Different types of *Reconfigurable c*omputer in our definition

	Types
Purpose	Performance or Correctness of the functions
Method	Static or Dynamic

1.1 Motivation

In the area of computer architecture research, we know that specialized computer architecture has better performance than general computer where the specific application or purpose is concerned. There is no doubt for this statement since this fact is very clear from results of almost all previous researches in the area of computer architecture research. The better performance is obtained from the *optimized* architecture that is corresponding to the purpose of the system, such as mathematical operation, logical operation, graphical operations, encoding or decoding and so on. (In this paper from this point, the term *optimized* stands for "specialized" to obtain the current best performance configuration as far as we find so far.)

However, the specific computer architectures also have several disadvantages in the purpose that the system is not specialized for. This characteristic of computer in general is similar to the characteristics of human brain. Most people definitely have some specific fields that they are better in, than other areas, even though these superiorities and inferiorities in performance are different for each person. Some people might be good at memorizing mathematical equations, but they might be not good at playing music from music score without any practice. Other people might have superior capability in computer research and programming but they might not be good at writing papers. Examples that we described are limited to academic subjects, but we can find these performance differences all over human life from daily life to business world. These characteristics are based on the environmental causes, such as what we had more interest in, what we studied, how we grew up, and so forth. In other words, we optimized ourselves or our brain for the characteristics that we need or what we use more often. The proof of this performance optimization is clear on where we are concerned about the acquisition of languages. The children who grew up with English in their schools can speak English fluently compared to the children who did not speak English at all. This accommodation of language abilities does not only appear in speaking, but also in listening, writing, and reading. For example, children who grew up with Japanese can distinguish between the meaning of words, which can have several different meanings, even though they might sound similar or exactly same.

There is one significant characteristic difference between conventional computer architecture and human brain, even though we usually use analogy of human brain to explain computer operations. Human brains optimize performance or improve performance up to certain age, but normal computer does not change architecture after the production stage. Some *reconfigurable* computer changes architecture to obtain better performance for the specific tasks, but this specialization does not work in all situations.

Due to the technological improvements in the recent era, the complex *reconfigurable* computer is not just a dream any more. With continuous changing motivation towards the technological front, we try to emulate the adjustability of human brain in a computer system using the *reconfigurable* computer. More specifically, we try to implement the flexibilities of human brain that is adjusting architecture for what we process currently. Therefore the heterogeneous *reconfigurable* computer that we want to propose has the three essential properties as summarized in Figure 1.2.

- Automatically adjustable architecture
- *Dynamic* or run-time Reconfiguration
- Optimization for specific objectives.

Figure1.2: List of three Essential Characteristics for the Proposed System

1.2 Thesis Organization

The remainder of the thesis is consisted of 6 Chapters. We start to discuss the reasons and problems statement for our proposed system in Chapter 2. In Chapter 3, we discuss the general idea of proposed system and the background concepts for the proposed system. In Chapter 4, we will do simulation to get the proof that the genetic algorithm can be used for simulation of computer architecture at high level. Also in this chapter, we briefly go over stochastics to introduce new assumption. The simulations in this chapter are implemented with static reconfiguration of computer architecture. Chapter 5 describes simulations of the proposed system and observations from the results of simulations. In Chapter 6, we finally conclude this thesis and offer suggestions for future research.

CHAPTER 2

REASONS AND PROBLEM STATEMENT

2.0 Reasons for our proposal

In this study, we propose a multi-core *reconfigurable* CPU as emulating the processing power of human brain. In this section, we go over several observations of the general *reconfigurable* computer to describe reasons of our choice. Additionally, we mention about problems that need to solve to create the system with the three properties listed in Figure 1.2: automatic reconfiguration, *dynamic* reconfiguration, and optimization for the specific target.

Assume we have a single *reconfigurable* core which implements the CPU, and our system has control unit which calculates the *optimized* architecture or configuration. Each time we try to reconfigure the system we cannot process the information during the reconfiguration time. We call this interval the *reconfiguration penalties*. If we try to implement the *static reconfigurable* CPU, the system does not try to process any information during reconfiguration. Therefore these penalties are not so important for the *static* implementation of the *reconfigurable* computer. As we describe in Chapter 1, we attempt to implement a system with the *dynamic* reconfiguration. The insignificance of the *reconfiguration penalties* is not same for our system. For the *dynamic* single core *reconfigurable* system, we definitely add one more term to calculate the total time needed to execute all information. We define the time unit we use for calculation of total Cycle Time (CT) as Cycle Time, which is time necessary to execute the specific instruction or the set of instruction. Relationship between total CT for *static reconfigurable* CPU and total CT for *dynamic reconfigurable* CPU for single core case is shown in expression (2.1) – (2.4) (Expression (2.2) is calculation for *static reconfigurable* CPUs and expression (2.3) is calculation for *dynamic reconfigurable* CPUs). We remark on one important fact of CT before moving to multi-core case. The CT would not be steady for both *dynamic* and *static reconfigurable* CPU. In other words, the time is dependent on the type of architecture we implement, and information we process during the period we are interested in. Also CT is measured as the average time obtained from tremendously large samples, since the instruction is not always executed in the same length of time.

$$CTSI = \sum_{\#\, of\, dif\, inst\, type} (\#\, of\ the\ specific\ inst\ to\ execute * CT) \tag{2.1}$$

$$TCT_{Static} = CTSI \tag{2.2}$$

$$RP = \sum_{\#\, of\, recon} t_{recon} = \sum_{\#\, of\, recon} t_{Recon\_Process} + ex\_t_{find\_optimized} \tag{2.3}$$

$$TCT_{dynamic} = \sum_{\#\, of\, recon} CTSI + RP \tag{2.4}$$

where CT is Cycle Time, CTSI is Cycle Time for the Specific Instruction,
dif is different, inst is instruction, TCT is Total Cycle Time,
RP is *reconfiguration Penalties*, recon is reconfiguration,
t is time, and ex is extra

As we see in expression (2.1) - (2.4), we have several different terms in *reconfiguration penalties*. Extra time to find the *optimized* configuration in the controller unit is usually longer than the benefit of reconfiguration. Therefore the *dynamic* system might have several cases that end up with worse performance in term of total CT compared to the *static* system, due to the *reconfiguration penalties*. As a result, the average total amounts of information that the system can process in the given time interval cannot produce outstanding benefit from reconfiguration. For the single core operation, the *dynamic* reconfiguration would not have sufficient motivation to implement, since it might not produce enough benefits in term of processing power as described above.

Next we evaluate multi-core *reconfigurable* CPU briefly. If reconfiguration of cores in systems has dependencies in terms of processing information, which means that all cores in systems cannot process information during reconfiguration, the result of simple observation would be similar to the conclusion from observation of single core case. Therefore we need to develop a new system whose reconfiguration of each core is independent of each other. In other words, the remainder of cores, which would not reconfigure, can still execute information during the process of reconfiguration. In this situation, the *dynamic* system equation in expression (2.1) - (2.4) cannot be used to determine total CTs. We have to determine *reconfiguration penalties* with more complex equations such as expression (2.5) and (2.6). The complexity of calculation increases tremendously, even though expression (2.5) and (2.6) look like simple equations. So we cannot determine benefit of reconfiguration as easy as evaluation of single core case if we use CT as performance measurement.

$$TCT_{Static} = \sum_{\#\,core\,used} CTSI \tag{2.5}$$

$$TCT_{dynamic} = \sum_{\#\,core\,used} \Big(\sum_{recon} CTSI\Big) + \sum_{\#\,core\,used} \Big(\sum_{Static\,Operation} CTSI\Big) \tag{2.6}$$

where CT is Cycle Time, CTSI is Cycle Time for the Specific Instruction, dif is different, inst is instruction, TCT is Total Cycle Time, RP is *reconfiguration Penalties*, recon is reconfiguration, t is time, and ex is extra

2.1 Problem Statements

From the previous argument, we need to develop new performance measurement, which we can easily use to determine the characteristics of *dynamic* systems and to compare the results with several different systems to find a better one. Also new measurement should be able to apply for *static* systems to compare performance between *dynamic* systems and *static* systems. As we discuss about performance of architecture, we not focus on the silicon area that is necessary for a whole system. We set our focus on processing power of systems. To determine processing power of systems, we cannot forget about one fact: results of performance measurements are dependent on benchmark programs we choose. For example, a result from one performance measurement shows outstanding benefits for a specific architecture, but the other performance measurement displays poor abilities for the same architecture. These differences come from the fact that different benchmark programs have different sequences of instructions, different measurement techniques, and different purposes of measurements which they are specialized for [10] – [13]. As a result of such specialization, we usually need to use several different performance measurements to determine benefits for implementing a specific architecture. Also most benchmark programs are developed for conventional computer system, which might not be as dynamic as we propose. Some of the bench mark programs might be developed for *reconfigurable* computers, but we would not measure performance of brain-like computers easily with them. This is because our proposed system has more flexibility to adjust system configurations and architectures dynamically to purpose of systems. As we describe previously, our brain-like computer is changing architectures according to the information we need to process while the machine executes the information. Therefore a result of performance measurement might not be always the same, since processing of information would change as we run the system. From this point of view, we need to develop the new measurement method that we could use for our purpose.

In the previous paragraph, we emphasize the importance of developing new performance measurements for reconfigurable systems to compare each individual system configurations. We also need to develop a performance measurement for reconfigurations or *performance prediction*. *Performance prediction* has the huge impact on final overall system performance measurement. The reason of the importance of *performance prediction* comes from the fact that the optimization technique decides timings for automatic reconfiguration and candidates for next configuration based on the information obtained from *performance prediction*. Also the optimization technique is critical for our system. The relationship between control algorithm and performance of systems can be explained with analogy in a branch prediction. If branch prediction has great precision, the performance benefit from branch prediction becomes more obvious compared to systems without branch prediction. In automatic *reconfigurable* system, the system with poorly developed optimization algorithm demonstrates only poor performance compared to the *static* systems. On the other hand, we can observe superior performance of *dynamic* automatic *reconfigurable* system from systems with well-developed optimization algorithm. Therefore control algorithm, its decision criteria, and performance prediction need to be developed carefully to obtain the sufficient results.

As we close this section, we summarize the problems obtained from our observation in Figure 2.1. With these problems in mind, we go over proposed brain-like system and its assumption in the next chapter.

- What kinds of evaluation technique or bench mark will we use?
- What kind of performance prediction will the optimization algorithm use?
- What kind of optimization algorithm will we implement for the system?
- What kind of performance measurement will we use for the system?

Figure2.1 List of Problem Statement we will solve to create the proposed system

CHAPTER 3

RECONFIGURABLE MULTI-CORE SYSTEM WITH GENETIC CONTROL

3.0 Assumptions

Before describing the general idea of our system with optimization technique and simulation technique we implement, we introduce several assumptions we use for this study. These assumptions are used efficiently to decrease the number of small problems and to reduce the complexity of problems in simulation of our system.

3.0.0 Configuration constrains

There are two methods that we can use to find architecture configuration or design in *reconfigurable* computers. One of the methods is that we start designing the computer configuration from scratch. We design the candidate architecture with all aspects, such as number of gates logics and functions implemented in the architecture, without any previous information and any design constrains except the maximum number of gate available in a reconfigurable core. This method can find the best architecture candidate in terms of performance for a specific purpose. Even for the single *static* core system, the search space of architecture configuration would be tremendously large since we have infinite choices. As a result, the search time that is necessary to find the best architecture would take more than a life time if we implement any search algorithm without blueprints. This search time issue would cause more serious problems for *dynamic* multi-core system. The search time that is necessary to find next configuration of each cores would take longer than a life time of production. The time needs for the search become too long for any numbers of cores if we do not use appropriate design constrains or pre-designs. Since all system change processing information as time goes, the "best" configuration of the past moment would not produce sufficient performance benefit if the search time is too long. In the other words, there are some opportunities that performance of system after reconfiguration would be worse than the performance of system prior to reconfiguration. This is caused by the "inappropriate" change of system architectures. As we describe about *reconfiguration penalties* in expression (2.3) - (2.4), search time to find the next configuration for all cores should include evaluation of performance measurement. In *dynamic* system, each core would be redesigned to find better performance for each chance for reconfiguration. Therefore if search time is too long such as the time necessary to find the next configuration from scratch, the performance benefits from reconfiguration also diminishes.

To reduce search time, we might be able to use the current designs or configurations of architecture as the start point of design. Such kinds of information might help to reduce search time that is necessary to find next configurations, but we cannot guarantee that the previous configuration would be efficient starting point for the architecture designs if we adjust systems for processing information in current time. We can verify this fact with a simple example. Assume a simple system which is processing “information A” for a long time and current configuration optimized to process “information A” as “configuration M”. If “information A” changes to “information B” which require similar set of instructions to process, “configuration M” would be a sufficient starting point to improve performance. However, the opposite case would cause a different result. “Configuration M” would not be worthy of use if “information A” changes to “information C,” which requires completely different pattern of instructions compared to the instruction pattern necessary to process “information A”.

In the two previous methods, we cannot have any characteristic information for each candidate we evaluate to find the *optimized* candidate beforehand. Therefore we should measure performance of architectures after the design is completed. To compare the candidates of next architecture configuration in the optimization algorithm, we need to wait until several other candidates are designed and measured. We can reduce the time necessary for multiple candidate designs by making the design process as simultaneous operation instead of sequential single design process. However, even with such kind of systems we cannot reduce the time necessary to complete any single candidate. Therefore these methods are not appropriate for our dynamic system.

The *reconfigurable* computer with pre-defined configuration is the method we use. This method has several benefits that increase the performance of our system. The first benefit is that the performance increases from reduction of the time necessary to design core architecture. Since we only use architectures that are pre-configured, we do not have to spend time for designing better architecture from the beginning. This idea is similar to the method of hierarchical design of Very Large Scale Integrated chip (VLSI) [14]. In the VLSI, we use the predesigned cells to create a larger and more complicated circuit. The cells used in the VLSI designs are extensively measured and well-designed to be specialized for certain objectives. In our system, we use well-designed architectures which are implemented in *reconfigurable* cores.

As we describe at the end of the previous paragraph, we only use well-defined core architecture which we know all performance characteristics such as power consumption, processing power, and Silicon area necessary to create. This fact generates several other benefits. We can reduce *reconfiguration penalties* due to the performance measurements of the next configuration candidates. We would know maximum performance of each architecture configurations without any assumptions since we can evaluate performance prior to use. We can also calculate the maximum performance of a multi-core system with simple arithmetic from the single core specifications. This well-established knowledge of performance can reduce the complexity of the optimization algorithm and the work load to find better configurations in the algorithm. In other words, the optimization algorithm is too complicated and time consuming without any prior knowledge of characteristic information since performance information is necessary to find better candidates. The reduction of *reconfiguration penalties* with preconfigured and well-defined systems are related to improvement in the processing performance. We go over one more benefit that is related to cost of implementation. Since we only use preconfigured architectures that we have the possibility to use, we can reduce the size of each *reconfigurable* core to the minimum requirement which is necessary to implement the largest architecture we have. This benefit is related to the cost of Silicon area that is required to implement each core. If we are designing the architecture of a core from scratch, we cannot obtain the information for the minimum size requirement which might be used in our system. Therefore we have to prepare overhead areas up to the limit of the Silicon area we can use, and sizes of the *reconfigurable* core would be more than the necessary area, even though the size of the final configuration might be much smaller than what we prepare as the overheads. Table 3.1 displays comparison between two methods: designing from scratch and using the preconfigured designs.

Table 3.1 Comparison of two methods of reconfiguration

RP stands for *reconfiguration penalties*

	With Scratch	With Preconfigured Designs
Performance Without RP (*Static*)	Better	Worse
Time Necessary to Create Next Candidate	Longer	Shorter
Time Necessary to Evaluate Next Candidate	Longer	Shorter
Complexity of Optimization Algorithm	More Complicated	Less Complicated
Performance With RP (*Dynamic*)	Worst	Better
Silicon Areas Used for each Design	Cannot be determined prior to complete designs	Can be determined with characteristics of designs

3.0.1 Data centric approach

One of the performance measurements we can use is processing power of computers. This is one of the traditions we use in research of computer architecture. In general, we use throughput which described the number of instructions we can process in a given interval, the Instruction per Cycle (IPC), the Cycle per Instruction, and time necessary to complete specific instruction sets [9]. These measurements are focused on the number of instructions and the time needed to use the resources.

There is other approach which uses data instead of individual instructions [15], [16]. This performance measurement uses the number of data or information processed in a given interval. Our brains always process several different sets of information in our daily lives. For example, the brain processes information from eyes to create what we feel to "see" such as colors, dimensional aspects, textures, and distances of objects. Also our brains process multiple sounds and identify the necessary information at the same moments. This identification comes from frequencies, amplitude, and distance from sources. If you think about motion of hands to grab something, human brains also process several different sets of information to move hands as we think. Therefore as we emulate flexibilities of the human brain, it is natural to use data centric approach for our system to find better candidates. If we consider more details of such information sets, we might consider them as millions of small instructions that have many dependencies. To reduce the effects of dependencies, we not break them down to individual instructions; instead we treat them as a set, which we define as *inforuction* from this point. Since we use *inforuction* for finding the next candidate architecture configuration, our data centric approach needs to define several different types of *inforuction* to create the performance measurement details. Using these types of information, all preconfigured architectures are measured with their characteristics such as the amount of *inforuction* they can handle in a given time interval. In other words, we know all performance measurements in term of the processing power of *inforuction*.

3.0.2 Inforuction Buffer

If we refer to architecture of superscalar processors in [9], we have an instruction buffer which stores instructions temporary till they feed to each individual pipeline. This mechanism allows us to control the flow of instructions and to utilize each pipeline as much as possible. We implement a similar mechanism in our system, which is identified as *inforuction buffer*. As we decide to use data centric approach, *inforuction buffer* stores each type of *inforuction* in different buffers. So each time *inforuction* is pre-fetched, the *inforuction* is separated with types of *inforuction* and feed into the corresponding *inforuction buffer*. The *inforuction* in the *inforuction buffer* is processed in the order they are fed in, which is the same order as first-in and first-out (FIFO) operation. This order reduces probabilities that we have dependencies among information. Therefore the *inforuction buffer* controls the flow of information and utilizes each core as much as possible.

3.0.3 Brief idea of architecture of proposed system

The General concept of our system would be similar to a cell processor [5] or similar to a tree. Therefore we explain our system architecture with an analogy to a tree. For a tree, we have one big trunk which holds minerals, some nutrients, and water obtained from the roots. Then the trunk sends these substances to the branches. The branches have different number of leaves that can implement photosynthesis, which uses sunlight to convert some nutrients and water and carbon dioxides into oxygen and some useful nutrients. Each leaf has also different amounts of chlorophyll which determines the capability of photosynthesis in a given day. In our system, the trunk corresponds to the *inforuction buffer*, the number of leaves on a branch corresponds to the number of cores in a system, and different amounts of chlorophyll corresponds to different architecture configurations we implement. Therefore our system has one big information buffer which stores and sends the *inforuction* to each core at the certain time and each core has input and output port at the same location in the architecture design. Therefore we can switch configurations of cores without any connection problems.

3.1 Optimization Technique

In this section, we introduce the genetic algorithm as the optimization technique. The genetic algorithm is one of the newly developed field and one of the hottest topics in research of computational intelligence. Application of genetic algorithm to research of computer architecture is not new. The algorithm is used for research of VLSI design to find the *optimized* area and *optimized* number of VIAs for the situation [17] – [21]. These references and [22] give more details for genetic algorithm which we do not go over deeply. In this study, we only provide the minimum knowledge to understand operations of the genetic algorithm.

3.1.0 General Genetic Algorithm: Background information

The genetic algorithm is inspired from mechanism in the natural world [22]. As we try to emulate flexibilities of a human brain with multi-core processors, the genetic algorithm emulates the optimization mechanism of species such as natural selection, evolution, and mutation. In the natural world, we know that there is natural selection and theory of evolution, which are proposed by Charles Darwin. The natural selection theory tells us that the species with characteristics more suitable to environment will prosper, and the species which cannot survive in the environment will decline in number, and will be terminated eventually [23]. The evolution theory tells us that the species would change its characteristics based on environment through generations [23]. These two theories can explain as we go over the history of Earth. For example, the dinosaurs prospered in a certain time in the ancient earth, but they do not survive in the current era. There might be several different hypotheses for reasons of termination, such as climate changes due to strike of large meteor, and survival races with the small size Mammal species that started to prosper. All of those hypotheses tell us that there might have been dramatic environmental changes in the ancient era and the dinosaurs could not adjust their characteristics to the changes in the environment. There is another mechanism which keeps the varieties of species. In the natural selection mechanism, each species would converge to the optimized characteristics, but the other mechanism generates the diversity in the species. This mechanism is called mutation. With mutation, the genes of the offspring generation would have different traits from the parent generation.

We introduce several terms which are commonly used in the genetic algorithm prior to going over the operations of the genetic algorithm. Most definitions that we use come from [22]. To implement the genetic algorithm, we need to decide the targets of optimization and how we evaluate systems with the objective we define. The targets of optimization are anything that can be evaluated with numeric values from distance of travels in the Traveling Salesman problems (TSP) to areas and numbers of VIAs in VLSI designs [21] [24] [25]. The method of evaluation is called the *fitness function* and numerical values obtained from *fitness functions* are called the *fitness values*. The numeric data of *fitness values* represents quality of measurement of the objective. To establish the *fitness function*, we also need to decide how we represent systems with some DNA like combinations, which is called *chromosome*. In other words, *chromosome* is representation for a possible candidate configuration of a whole system. Each entry in *chromosomes* represents some traits of a system, as each set of entries in DNA represent some kind of characteristics. For example of TSP, *chromosome* is the traveling path which travelers will follow to visit necessary cities, and the entry in *chromosome* or *gene* corresponds to a specific city which he has to visit. Set of multiple *chromosomes* is called *population*.

At the initial stage, general genetic algorithm produces a set of *chromosomes* up to *population* size which is defined by designer, and would not change the *population* size for the entire algorithm. We usually use random generation method, in which each *gene* in *chromosomes* is set randomly to include various *chromosomes* in *population*. We identify these initially generated candidates as *population* of the *first generation*. After we set *population*, we pick up parents of offspring with some methods such as random, weighted random, and other methods. With chosen parents, we use some methods to create set of offspring. One of the commonly used methods in process of offspring generation is called *crossover*. With the *crossover* operation, children have some common pattern of genes from both parents. As the name implies, *crossover* generates the offspring by exchanging some *gene* pattern between parents which is similar to mechanism of gene pattern succession in the offspring. Example of crossover is displayed in Figure 3.1. After generating a candidate or candidates of next *population*, we implement the mutation operation. This changes part of *gene* pattern randomly. Example of mutation is also displayed in Figure 3.1. Then we evaluate generated offspring and old generation with *fitness functions.* With *fitness values* and superiorities of *fitness values* that we decide, we sort entire set of *chromosomes* which contains both offspring and the entire *population* of previous generation. Then we implement termination mechanism to adjust the number of *chromosomes* in the *population* to the size of *population* we decide to use. After creating the new *population*, we designate this set as *population* of the second generation. In other words, we increment generation number each time we create new *population.* The processes after production of *population* of the first generation are repeated continuously till certain conditions are achieved. This condition is identified as *stopping criteria*. Examples of *stopping criteria* are *optimized* candidate have sufficient *fitness value* or maximum generation we define is passed. Figure 3.2 shows a flowchart of the genetic algorithm. The term *iteration* is used to count the number of repetitions for the entire flowchart in Figure 3.2.

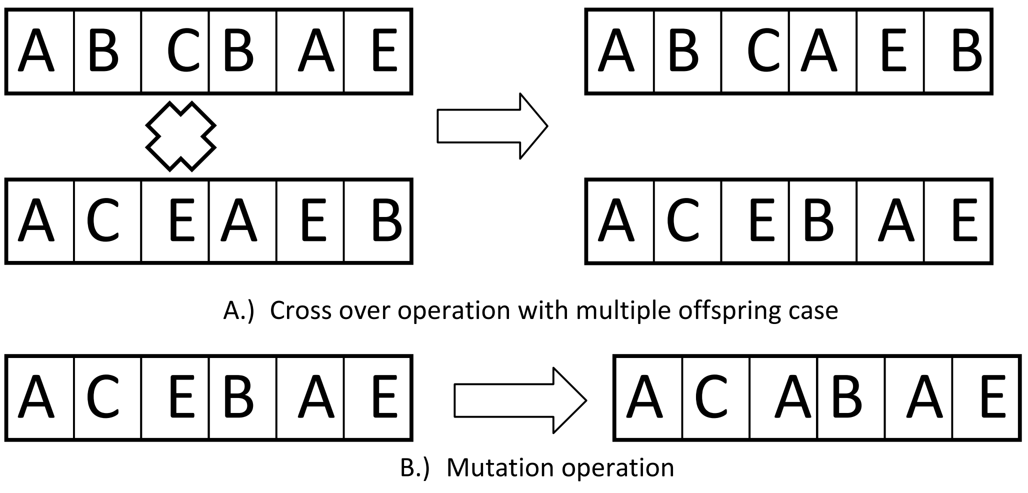
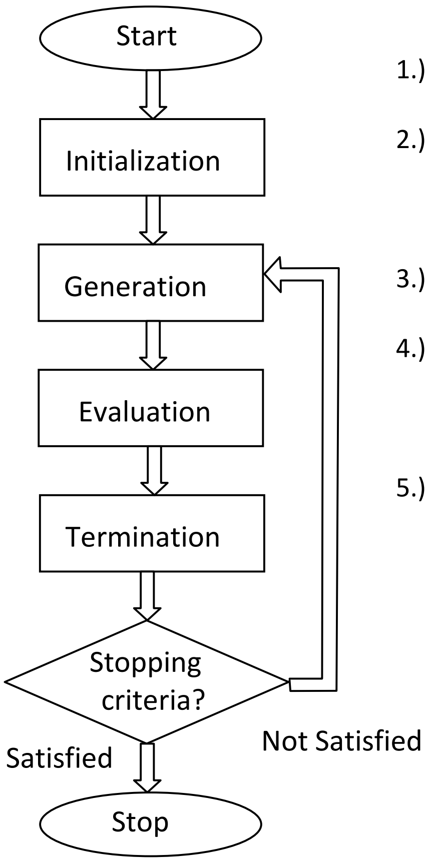


Figure3.1 Operation of the genetic Algorithm



1.) Initialization Stage
   a. Create the chromosome randomly
2.) Generation Stage
   a. Choose the parents
   b. Implement Crossover
   c. Implement Mutation
3.) Evaluation Stage
   a. Evaluate with Fitness function
4.) Termination Stage
   a. Sort the population with result of 3-a.)
   b. Choose the chromosome to terminate
   c. Generate new population
5.) Check Stage
   a. Check for stopping Criteria

Figure3.2 Operation flowchart for the genetic algorithm

There is significant benefit for applying the genetic algorithm as the optimization technique. The genetic algorithm is an optimization algorithm which has both global search and local search abilities. With the *crossover* operation, we implement the local search. With mutation operation, we implement the global search, which checks randomized candidate other than similar candidates which we produce with *crossover*. In conventional optimization algorithm, most algorithms have only one search method, not both. Also the conventional optimization technique uses the sequential evaluation, in which the algorithm generates only single candidate, evaluates and compares with the current system. Example of sequential optimization is simulated annealing [25]. For the genetic algorithm, we use the parallel optimization, which generates multiple candidates, evaluate, and compares with the previous *population*. We can find better candidates more efficiently since we check more candidates simultaneously and choose better candidates.

In this section, we discuss the general genetic algorithm. As we close this section, we define two terms: *global optimum* and *local optimum* according to [23]. The *local optimum* is better candidate than all other candidates in terms of *fitness values* among results of the current search. The *global optimum* is best candidates among all other *local optimum* in terms of *fitness values*. Relationship between these optimums is similar to cost of gas in a certain state. We can find cheapest price of gases in a city when we compare prices inside a city. This cheapest price or *local minimum* might not be cheapest price all over the state or *global minimum* since we might find better result from another city. This terminology would be used in this section. In next section, we go over the genetic algorithm that we implement in our system

3.1.1 Specialized General Genetic Algorithm

In the previous section, we introduce background information for the genetic algorithm. The actual genetic algorithm that we implement in our system has more functions and is slightly different from the general genetic algorithm. In this section, we describe the characteristics of the specialized genetic algorithm.

3.1.1.0 Dynamic population

As we explained in the previous section, the basic genetic algorithm has a fixed size of *population* which is not changed for the entire algorithm. The choices of appropriate size of *population* are one of the hottest topics in research of genetic algorithms. The reason is that the size of the *population* is deeply related to abilities of the *optimization* and time necessary to complete the search algorithm. We can explain this with a simple example. Assume we have the genetic algorithm which implements a fixed number of generations. If we increase the size of the *population*, the possibilities to find *optimized* candidate or the best candidate would be greater than with smaller sizes, but calculation costs of each generation would also expand. On the other hand, if we reduce the size of the *population*, calculation costs will get smaller but, the possibilities to find the best candidate will also decrease. In our *dynamic* system case, we want to reduce the calculation cost as much as possible, but at the same time we want to increase possibilities to find better candidates as much as possible. So the determination of a good *population* size is too sensitive of an issue since the choice of the *population* changes the functionality of algorithm dramatically. To overcome issues related to finding the best *population* size, we use the dynamic *population* size approach instead of the fixed *population* size as [26]. With this approach we start from the relatively smaller initial *population* size which we define. After the genetic algorithm starts, we do not know the *population* size since the size is automatically adjusted according to the current condition of the *optimization* process.

There are several characteristics we have to define for the dynamic *population* approach. Those characteristics are what would be the trigger of change the population size, how we will change it, and how much we will change it. If we do not choose each category carefully, the dynamic *population* algorithm does not work correctly or simply works as the fixed size *population* approach.

The first question we tackle is what would start the process of changing the size of the *population*. Since the dynamic *population* is part of the genetic algorithm which uses the *fitness values* to determine whether candidates are better or not, we can use the *fitness values* of the *population* to trigger changes of the *population* size. There are several different statistical data of the *fitness values*. Examples are the best *fitness value*, the worst *fitness value*, the mean *fitness value*, and the median *fitness value* of the *population*. Within these values, we use the average or mean *fitness value* of the current *population* and the mean *fitness value* of the group which consisted of both the current *population* and the candidates which we generated. This idea came from a question that was asked by then Ph.D. Student Wen-Fung Leong during one of my course presentations. Since the average *fitness value* displays the performance of the optimization process as a whole in a given moment, the dynamic *population* would ensure proper resource management for the search ability of the genetic algorithm. Each time we obtain the new *fitness values* of candidates and the new *fitness values* of the current *population*, we initiate the process of the dynamic *population* algorithm.

As we decide when we will change the size of the *population*, we need to decide how we will change it according to the difference between two average *fitness values*. There are two different resource management strategies we can use to determine how we will change the *population* size. These two methods of resource management are similar to methods that students commonly use in their studying for final exams. Some students prefer spending more time for subjects that they are very good at, and spending less time for topics they hate. As a result, students would answer the question extremely well for the subject they studied and cannot answer the questions they did not study well. Or more simply, students become specialized in specific subjects they like. On the other hand, other types of students would spend more time on subjects which they are not good at and spend less time for the subjects which they feel strong in. These people use time where it is necessary to spend it. Comparing these two types of students, the second type of student has more chances to have better grade point average (GPA). This example is true among different disciplines.

Back to method of resource management, we describe and evaluate two types of strategies for increasing the chance of generating better average *fitness values*. The first type of resource management technique uses more computational resources when we have better average *fitness values* and uses less when we have poor results. In other words, we spend more time where we find better average *fitness values* and less time for where we cannot find good average *fitness values*. This strategy might find a *local optimum* quicker than the other method, but one problem of this method is that the *local* *minimum* we find is not always the *global minimum* as we describe at the end of the previous section. For the second strategy, we utilize more resources when we have a hard time to find the better average *fitness values* and less when we can easily find better *fitness values*. This method spends more time where we cannot find better average *fitness values* and less time where we find better average *fitness values*.

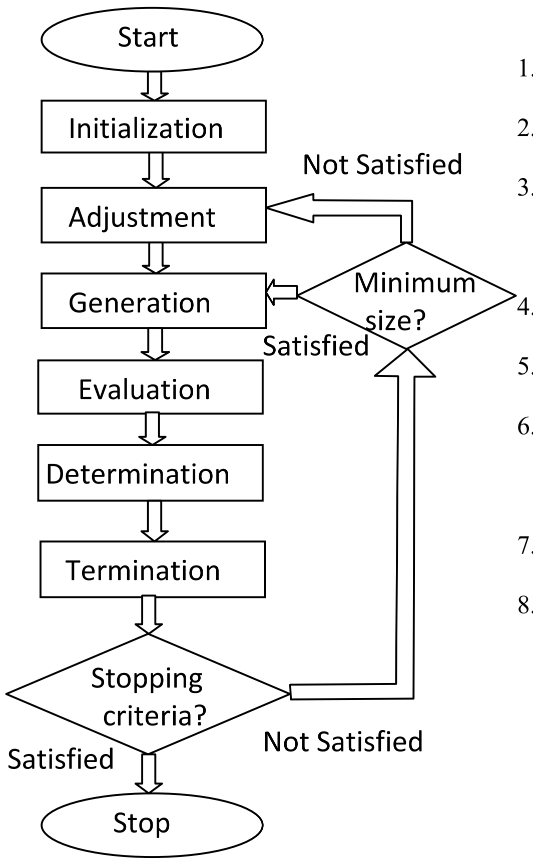
The comparison of these two strategies in a real situation can be explained with an analogy of open-book/open note exam. Assume we try to solve two different types of questions: questions that we have enough knowledge to solve and questions that we do not have any idea how to solve. To solve the first type of questions, we only need to check the correctness of our memory with books to obtain better scores. In this case, we just have to use resources which relate to the concepts we need. If we just go over each topic in the text books, it is just wasting time and we will end up running out of time to solve the other questions. To solve the second type of questions, we cannot review specific topics since we do not have any idea how we can solve them. We have to go over each topic briefly to find any related concepts which can be used to solve the questions. If we randomly choose specific chapters in books to read in detail, the exam time is too short to find necessary information. The first method of taking exams demonstrates a similar situation as the case when we know the *fitness value* of the *global optimum*. This is not the situation where we are during the search, since we do not know what the *global optimum* is. Therefore, we have to apply the second strategy instead of the first method. In other words, we will increase *population* size when we have a hard time to find better average *fitness values* or very close to *optimum* (either *global* or *local*) and reduce the size when improvement in average *fitness values* is sufficient. In this method, we would have better opportunities to find the *global optimum*. There is one problem while applying the second strategy: the *population* size might get too small to keep operation of the algorithm correct if we constantly improve the *average fitness values*. We discuss the solution for this issue in the latter section of this chapter.

We have already decided the timing of change and the method of change, but we have not yet finished the argument for the degree of change in the *population* size. This question has also several different strategies such as the constant fixed change method, dynamic fixed change method and proportional change method. We briefly go over each strategy with problems and benefits. The first strategy is simple enough to implement, since the size of change in the population is defined at the beginning of the algorithm. Therefore, we do not have to change it and also we do not have to calculate the degree of change according to the improvement in the average *fitness value*. The problem of this approach is very obvious, the size always changes constantly no matter what the current size of the *population* we have, and how much we improve the average *fitness values*. In extreme example, a result of twenty-five percent improve in the average *fitness value* cause the same degree of change in the *population* size as a result of one percent decrease in the average *fitness value*. Also the impact of change does not take into consideration the change in the *population* size. If we only use a fixed number of candidates to change the *population* size, the impact of change would not be efficient when the size of *population* is large. For example, we would double the size of the *population* if we change its size from 10 to 20. However, there will be only ten percent of increase in the size of *population* if we change the size from 100 to 110. The impact of the changes in the average *fitness value* in the previous two examples is not the same. The second approach requires several conditional statements to implement the algorithm, so the calculation cost will increase slightly compared to the first approach. The degree of changes in the *fitness values* are reflected in the second approach, but the impact of changes that is dependent on the *population* size is not included in this algorithm. The third approach treats both the degree of changes and the impact of changes. Instead of a fixed size algorithm implemented in the previous two approaches, it uses the proportional changes which are dependent on both the *population* size and the degree of changes in the average *fitness values*. The calculation cost for implementing the third method is slightly greater than the second approach. Even with the disadvantage in the calculation cost, the third approach is worth implementing, since we can change the *population* more dynamically compared to the other approaches.

3.1.1.1 Off by one theory

We now explain the method we implement to avoid errors in the algorithm due to the reduction of the *population* size. Since we implement *crossover* operation, we definitely require at least more than two *chromosomes* in the *population*. The method is that we insert new sets of the *chromosomes* into the current *population* whenever the *population* size does not meet the minimum size requirement of the algorithm. New sets of the *chromosomes* are generated with some randomization mechanism.

Before we explain randomization mechanism to fix violation of the minimum size requirement, we go back to several observations we discuss previously. In section 3.0.1, we describe that the current adapted core architecture configuration in the *dynamic* system would not guarantee good performance in the future. This statement is true when we observe processing performance of computer systems. For short time observation, we can say that current configuration would produce sufficient performance to use as seeds of randomized *chromosomes,* since the contents of *inforuction buffer* changes slowly unless it is completely empty. Therefore the current configuration would offer several hints to find the next configuration. If we only add the same configuration to the *population*, the abilities of both global and local search would decrease. Therefore we need to generate randomized candidates from the current configuration. In the randomization process, we generate candidates according to the *off by one theory*. This theory tells us that it would be better to change the architecture of only one *reconfigurable* core in the randomization process. This theory is derived from the careful observation of the assumption we made earlier. We assume that during the process of reconfiguration the reconfiguring core cannot process any *inforuction*. Therefore, if we increase the number of core which reconfigure, it implies the number of cores that cannot process *inforuction* in several intervals. As a result, performance of the system would decrease significantly during reconfiguration. Hence, candidates that only change one core would yield similar performance as the current configuration of the system. In most cases, generated candidates would have less performance in the processing power due to the *reconfiguration penalties*. From this observation, we create candidates which randomize only single core from the current configuration, and we insert generated *chromosomes* into the *population* to prevent failure of the algorithm and to maintain diversity in the *population*. Figure 3.3 illustrates the flow of the modified genetic algorithm.



1.) Initialization Stage
    a. Create the chromosome randomly
2.) Adjustment Stage
    a. Change the population size
3.) Generation Stages
    a. Choose the parents
    b. Implement Crossover
    c. Implement Mutation
4.) Evaluation Stage
    a. Evaluate with Fitness function
5.) Determination Stage
    a. Evaluate the next population size
6.) Termination Stage
    a. Sort the population with result of 4-a.)
    b. Choose the chromosome to terminate
    c. Generate new population
7.) Check Stage
    a. Check for stopping Criteria
8.) Stop Stage
    a. The best candidate is picked up as the output of algorithm

Figure3.3 Operation flowchart for the modified genetic algorithm

3.1.1.2 Multiple Crossover Operation with multiple parents

There are several papers which study about the effect of multiple *crossover operations* in the genetic algorithm. Examples of such studies are [27] [28] [29]. The multiple *crossover* operators have one important benefit. This method enhances the global search abilities of the genetic algorithm. Even though the *crossover* operation implements local search, each different *crossover* operator would search through different search spaces. The variety of children in each generation would increase compared to the implementation of single *crossover* operation. Also, the multiple parents would increase the global search ability, since we choose several different sets of parents for each type of *crossover* operator. So many traits of *chromosomes* are effectively used to create the next *population*.

3.1.1.3 Fitness function

In this section, we talk about the most important topic of the genetic algorithm. This is not an exaggerated expression since the result of *fitness function* is used for control of reconfiguration and control of the *dynamic* population. We go over several facts that are used to develop our *fitness function*. To compare each configuration of the system, we can use the processing performances of configurations after we complete the process of reconfiguration since they are simple enough to calculate. Even though we know the processing characteristics of each core which we obtain from measurements, the actual performance of cores and systems might be lower than what we calculate. There are three types of barriers that make system performance lower. One of the barriers is the *stall*. The *stall* is the time we cannot have the full abilities of processing power since some *infoructions* are not ready to process [9] or we have too much stock of *inforuction* in the *inforuction buffer*. This is caused by dependencies in *inforuction*. In the *stall* condition, systems cannot obtain any new *infoructions*.

Another barrier has a *similar* effect as the *stall*, but the cause of this barrier is slightly different. We name this barrier *empty-running*. As the name implies, the system does not store sufficient amounts of specific types of *infoructions* in the *information buffer* compared to the maximum performance we have. We explain the problem of this situation with an example. We feed each type of *infoructions* into the corresponding *inforuction buffer* each time we pre-fetch. Each *reconfigurable* core in the system processes *infoructions* from this set of *inforuction buffers*. If we have sufficient prefetched *infoructions* for all types of *inforuction buffers*, the system can process as much as possible with its maximum potential and there is no problem for this situation. However, if the *inforuction buffer* contains fewer amounts of some types of *inforuction*, cores cannot utilize the full-processing power as we calculate. It can only process the amounts of *inforuction* in the *inforuction buffer*. After it completes processing, it becomes idle or runs with the emptied *inforuction buffer* till the next set of infoructions is ready.

Another barrier is *reconfiguration penalties*. During reconfiguration, the processing power of systems decreases compared to full specification since several cores are isolated or excluded from our system. After reconfiguration, the isolated cores become active to process *infoructions*. Therefore these performance changes during the process of reconfiguration also make the processing power difficult to use as the *fitness values* of our system. With all three of these barriers together, we cannot use processing power directly to determine the *fitness values* of a given system configuration.

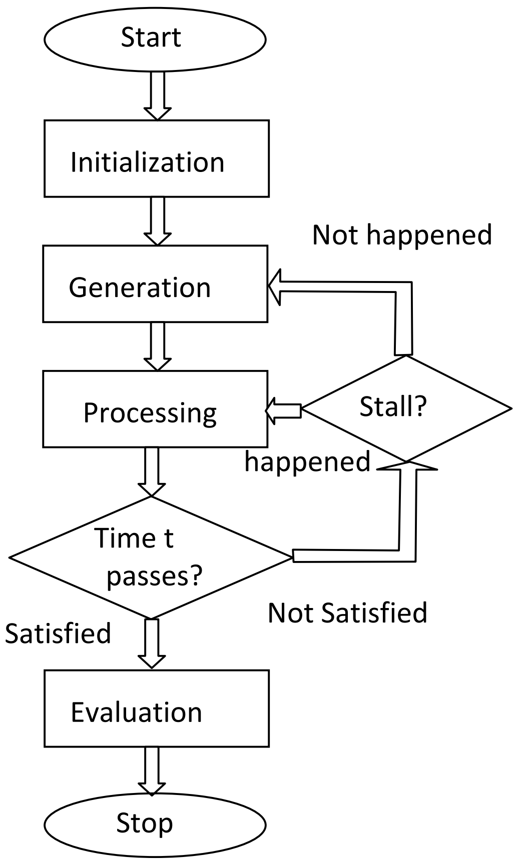
The appropriate *fitness function* should treat all three barriers. We use the predicted amounts of *infoructions* we can process in a certain time interval as our *fitness values*. The detail of *fitness function* is described in expression (3.1). The time parameter t is used in expression (3.1). This time parameter should be longer than the *reconfiguration penalties* since we know the performance of a *dynamic* system during reconfiguration is lower than the system which stays the same as the current configuration. If we take a longer time interval for the parameter t, calculation costs of reconfiguration would increase dramatically since the *fitness function* emulates behaviors of systems to predict amounts of information we might process. To predict the performance of dynamic systems, we have to predict the amount of *infoructions* which will be generated in the time interval t since the *inforuction* we need to process in the future time is unknown.

$$
\begin{aligned}
&Fitness\ Value = \#\ of\ inforuction\ that\ predict\ to\ process\ in\ time\ t \\
&= (\#\ of\ inforuction\ in\ the\ inforuction\ buffer\ \ after\ time\ t) \\
&+ (\#\ of\ inforuction\ that\ assume\ to\ generate\ in\ the\ given\ time\ t\ ) \\
&- (\#\ of\ inforuction\ in\ the\ inforuction\ buffer\ \ before) \qquad (3.1)
\end{aligned}
$$

The approach we take for prediction of the future *infoructions* uses the similar concept in the neural network system [30]. In the neural system, we train the network with some sample patterns which model the operation of the system. The neural network trains to obtain the correct result or desirable results in terms of purpose of the system with the differences of simulated results, and preferable results. For our system, we will train our prediction mechanism with the data of the generated *infoructions* which are fed into our *inforuction buffer* whenever we do not reconfigure the system. Our training method is straightforward. We take the data for occurrence of each *inforuction* for a specific time interval. This time interval is related to the time parameter (t) we use in the *fitness function*. As a result of the *fitness value* evaluation, we conclude that we do not have to reconfigure our system, since the current system configuration might have better performance within the time interval (t). Therefore we might not have to reconfigure our system during the time interval. At the same time, we do not have to evaluate the candidates with the genetic algorithm. We use the time interval in which the optimization process does not operate, and resources which is usually used in the optimization algorithm. We discuss the meaning of the data of *inforuction* occurrences in the later chapter.

We will evaluate the impact of misprediction. The misprediction occurres when the data for occurrence does not correspond to the actual behavior of systems. It happens when sets of *inforuction* produced are changed dramatically from what we observe. The impact caused by the misprediction would not be severe since it is softened by the *inforuction buffers*. This is because the sets of *inforuction* in the *inforuction buffer* should be processed prior to the sets of *inforuction* predicted with the prediction mechanism. In other words, we always have some portion of *infoructions* that we will definitely process since they are the stored *infoructions* in the *inforuction buffer*. Therefore, the ratio of amounts of mispredicted *inforuction* handled in the *fitness function* to the *infoructions* that are predicted to be handled would be always less than 1. Even though there is a possibility of misprediction, our prediction mechanism represents the actual behavior of systems more accurately than the simplest prediction method, which assumes to have exactly the same number of each *inforuction* during our prediction.

As we close this section, we show the flowchart of the *fitness function* in the Figure 3.4. Our *fitness function* is a short time simulation of the system since we need to determine the performance in the future for both types of systems: the system with the current configuration and the system with the candidate configurations during the process of *optimization* for the multi-core system.



1.) Initialization Stage
   a. Memorize the current inforuction buffer and current pattern of inforuction generation
2.) Generation Stage
   a. Generate the inforuction pattern the same as 1-a.)
3.) Processing Stage
   a. Increment the internal clock
   b. Process the # of inforuction for the current processing power
4.) Check Stage
   a. Time check
      i. Check whether the time interval t is passed?
   b. Stall check
      i. Check whether the stall is happened or not
5.) Evaluation Stage
   a. Evaluate the fitness function

Figure3.4 Operation flowchart for the fitness function

3.1.1.4 Running BEST

The genetic algorithm will converge to the *optimum* when we use algorithms with a large number of generations. Hence, the algorithm needs long time to obtain the *optimum*. Even if we use large number of generations, the *optimum* obtained is either *local* or *global*. There is no guarantee we can get *global optimum* without prior knowledge. As we implement the genetic algorithm as part of *dynamic* system, the large generation for convergence would be problematic, since the larger generation needs more time to compute the *fitness values*. Therefore we cannot wait the algorithm to converge, and we should set the generation we want to use as the *stopping criteria*. Instead, we use the running best or *optimized* candidate as our next candidate, which is the candidate with best *fitness values* that we evaluate through our algorithm. In other words, we stop the genetic algorithm with a specific generation, and obtain the best candidate from the given population. Since the chosen candidate is the best candidate in the population, we can use this approach for the best candidate that we can find within the limited resources.

CHAPTER 4

STATIC RECONFIGURABLE SYTEM: GENETIC ALGORITHM EVALUATION

4.0 Reason for static reconfigurable computer implementation

Even though we have several examples which apply the genetic algorithm to research of computer architecture [17] – [21], we do not have an example of the genetic algorithm applied as the optimization algorithm to find better configuration candidates in terms of processing power of CPUs. We can prove the performance of the genetic algorithm to find the *optimized* candidates through the implementation of a *static reconfigurable* computer, which is similar to a cell processor. Also we can develop the framework of simulation that we can use for the dynamic system. In addition to these benefits, the results of simulation can be used as the performance of specialized *static* computer, which can be used to compare with our proposed system. Therefore we should implement a *static reconfigurable* computer.

4.1 Assumptions for Simulation

We describe the important assumptions that are necessary for the simulation for the *dynamic* system in Chapter 3. We will discuss assumptions that we will use for the *static reconfigurable* systems and will introduce the new mechanism with a new assumption that would be used in both simulations of *static* and *dynamic* systems. Assume we have the multi-core system described in section 3.0.3. Each of architecture configurations of cores we use in our system is well designed. The characteristics of each architecture configuration are well measured in terms of amounts of possible *inforuction* each core can process during a certain time unit. This measurement is called *Inforuction* Per Clock Cycle (IPCC). Since we implement superscalar architecture in each core, IPCC for each core has amounts of each types of *inforuction* that a single core can process simultaneously during a given time unit. Before discussing the details of genetic algorithm in the next section, we will introduce the new concept: *stochastic inforuction generator*.

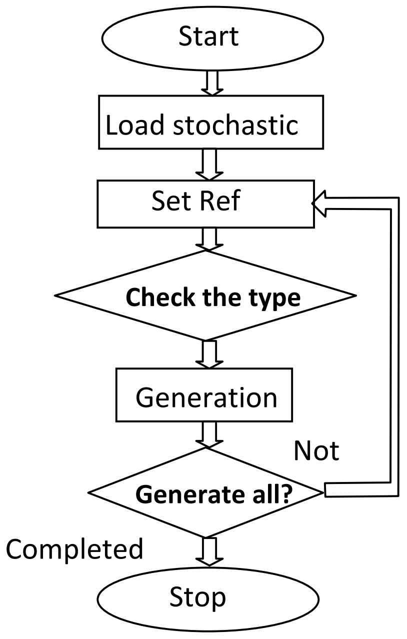
4.1.0 Stochastic Inforuction Generator

Before introducing the *stochastic inforuction generator*, we will go over the basic idea of stochastics. Most definitions come from [31] – [33]. When we observe result of a fair-six-faced dice roll, we get faces from 1 to 6. For short time observations or from small number of samples of a single dice roll, occurrence of each faces dynamically change as we roll, and we cannot find steady result in frequencies of occurrence for each face. When we observe fairly large numbers of dice roll or we have sufficient numbers of samples of dice roll, there would be steady data for occurrence of each face, which is one sixth for each face in this case. In other words, if we observe some system for a long time, we would find stochastic data or probability information in behaviors of the observed system. In the previous example, the stochastic information we get is frequencies of occurrence of each face. Since this is the representation of system behavior, we can predict the behavior of the given system with probabilities. The example of prediction is probability of fair-coin toss. We know probabilities of occurrence for each side of the coin would be one half. Therefore, if we toss a coin significant number of times, we can predict that we will have about half of toss as heads and the other half as tails. In real situations, there are dependencies among occurrence of each instance. Since the dependencies change the probabilities of occurrence, our prediction of occurrence is more difficult. However, if we find out all dependencies, the predictions would correspond to the actual system behavior.

If we observe a single computer system with a certain simulation for a long time, or if we observe multiple computes with same settings, there should be stochastic information for occurrence of each type of instructions. We prove this statement with contradiction. Assume we do not have any stochastic information or probabilities for occurrence of each instruction. In other words, the instructions we process is decided randomly. For such situation, we cannot reproduce a result of simulation, even if we use the exactly same setting. However, the normal simulations implemented in some benchmark programs definitely have steady results that we can reproduce with the same settings. With reproducible results, we justify our accomplishments. Simulators or benchmark programs are designed to reproduce the data if we use the exactly same setting since they produce the same sets of instructions for the same settings. On other hand, if we have different sets of instruction, the result of simulation would not be reproducible. This is contradictory to what we assume: we cannot reproduce the same result since the instruction is randomly generated. Therefore there exists the stochastic information for patterns of instruction generation in a simulator, which is dependent on the type of application. This assumption is based on the observation of a specific purpose application. For example, the mathematical application software would produce more instruction related to mathematical operation compared to other software. If we are watching a movie on the computer, the instruction related to graphics and sounds would be more than the normal operation of the system. For our system, we would have stochastic patterns in *inforuction* generation that are different for each types of application. With stochastic information for occurrence of each types of *inforuction*, we can generate *infoructions* to simulate real behavior of systems. We call this stochastic method of generating *inforuction* as *stochastic inforuction generators*. We will remark one more fact for the *stochastic inforuction generator*. The stochastic information should obtain from data that is consisted of large numbers of samples since validity of the stochastic information increases with larger amounts of data due to the reduction of the effect from the noise or unexpected data. Also we need to use *stochastic inforuction generator* for longer cycle to increase accuracy and precision of the simulation.

The actual process of *inforuction* stream generation is illustrated in Figure 4.1. As Figure 4.1 displays, we use a random number generator to generate the actual *inforuction* in each cycle. We cannot match the stochastic information used to the generate *infoructions* with the stochastic information from actual generated *infoructions* in each cycle. We can only observe the random generation of *infoructions* in a short time.

However, if we observe the *inforuction* generation pattern for a long time, we can get similar stochastic information as information used to generate *inforuction* stream. From this point, this method can be used for a short cycle operation, which does not have any clear stochastic information for a generation pattern.



1.) Load Stage
    a. Load stochastic data of inforuction generation
2.) Set Ref Stage
    a. Use random generator to set the reference point
3.) Check Type Stage
    a. Check what type of inforuction will be generated (Find the value k which satisfies the following expression: $Ref \le \sum_{i=1}^{k} \mathrm{P(ith\ type)}$)
4.) Generation Stage
    a. Generate the inforuction according to check stage result
5.) Check # of Generated Stage
    a. Check the number of inforuction generated for this cycle is the same as the number we defined or not

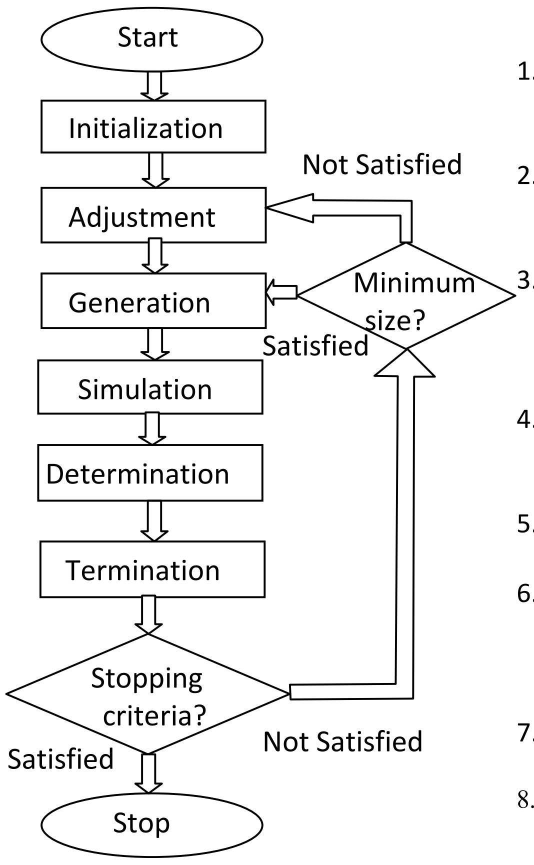
Figure4.1 Operation flowchart for the inforuction generation for each cycle

4.1.1 Dependencies and inforuction stream

Up to this point we do not have any mechanism to implement for dependencies of *infoructions*. For our simulation, instead of implementing the actual dependency mechanism, we randomly insert an extra cycle in which no new data is fed into the *inforuction buffer*. This corresponds to NO OP instruction in regular computer terminology. To model system accurately, the numbers of dependencies correspond to the numbers of total *inforuction* we will process in our simulation. To compare the systems with the same number of cycles fairly, we use the exact same *inforuction* stream. This means we have the same stochastic data, the same number of *inforuction*, the same numbers of dependencies, and the same sequence for each *inforuction*.

4.2 Genetic algorithm for the static operation

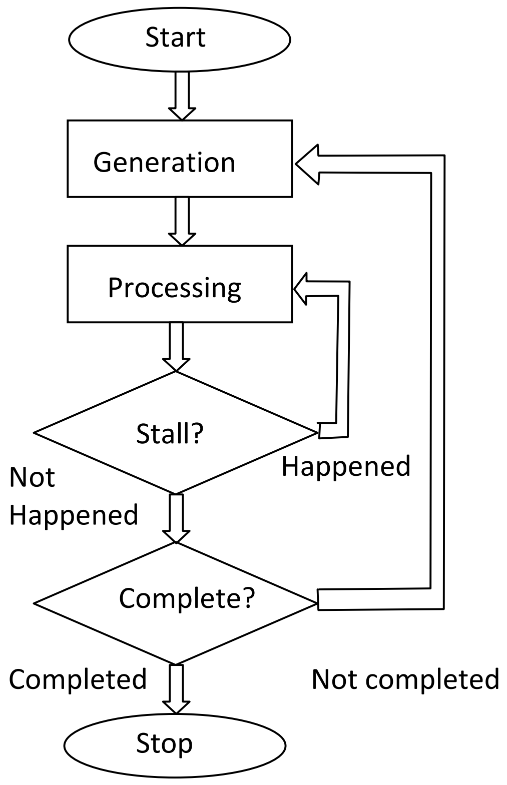
The genetic algorithm we implement for the *static* system has the same characteristic as *dynamic* algorithm expect the several differences that come from system properties. For the *static* application, we do not change system configuration while the system processes *infoructions*. Therefore we do not have to worry about *reconfiguration penalties* and reduction of performance due to reconfiguration. The objective of the *static reconfigurable* computer is to find the *optimized* configuration for the system which has the "best" performance for given application or application sequence. So *fitness values* of the *static reconfigurable* computer are the total time unit necessary to complete all *inforuction*. As we describe previously, we have to worry about the reduction of processing performance due to *stall* and *empty-running,* since *stall* and *empty-running* will increase the necessary time unit to complete all *inforuction*. The flowchart of the *static* genetic computer is described in Figure 4.2. We go over several stages which is different from the *dynamic* operation we described at the previous chapter.



1.) Initialization Stage
   a. Create the configuration of cores randomly
2.) Adjustment Stage
   a. Insert randomly generated candidates to satisfy the minimum population size
3.) Generation Stages
   a. Choose the parents
   b. Implement Crossover
   c. Implement Mutation
4.) Simulation Stage
   a. Simulate the time necessary to complete the inforuction sets
5.) Determination Stage
   a. Evaluate the next population size
6.) Termination Stage
   a. Sort the population with result of 4-a.)
   b. Choose the chromosome to terminate
   c. Generate new population
7.) Check Stage
   a. Check for stopping Criteria
8.) Stop Stage
   a. The best candidate is picked up as the output of algorithm

Figure4.2 Operation flowchart for the genetic algorithm for static operation

Most significant difference between *dynamic* operation and *static* operation is location of the genetic algorithm or the stage of simulation. For *dynamic* systems, we are running simulation while we use the genetic algorithm to find better candidates of the next configuration. For this application, we run the miniature of simulation in the process of *fitness function* evaluation. For the *static* operation, we use the genetic algorithm after we simulate a system to find the *fitness values*. The details of simulation flow are shown in Figure 4.3.



1.) Generation Stage
   a. Generate the inforuction pattern which is pre-configured for the purpose for the system
2.) Processing Stage
   a. Increment the internal clock
   b. Process the # of inforuction for the current processing power
3.) Stall Check Stage
   a. Check whether the stall is happened or not
4.) Information check
   a. Check whether all inforuction is completed or not

Figure4.3 Operation flowchart for the simulation stage of genetic algorithm for static application

This figure is similar to Figure 3.4 which displays the flow of *fitness function* in *dynamic* systems. The differences between these two flowcharts are at the last conditional or check statement. For the *fitness function* of *dynamic* system, the amount of *infoructions* processed is variable, and the cycle in which we simulate remains constant. The simulation for *static* systems is targeted to obtain the difference in time cycle for given amounts of *inforuction*s we need to process. From these reason the *stall* condition is implemented in the different location in the two flowcharts.

Another difference between the two flowcharts is the method we use to generate *chromosomes* to adjust the *population* size. Remember for the *dynamic* system, we introduce the *off-by-one theory*, which tries to change configuration of only one core in the process of generating the extra candidate for adjusting *population* size. This theory is to avoid the severe *reconfiguration penalty* from reconfiguring multiple cores at once. As we described previously, we do not have to worry about the *reconfiguration penalty* in the *static* system. Therefore we insert *chromosomes* which are randomly generated to ensure the varieties in the *population*. Since there are *local optimum* issues for the optimization process, the randomly chosen configuration would give more chances to find *global optimum*.

One more remark for simulation method we can use for both *dynamic* and *static* system as we close this section, which is how we implement the *stall*. As we describe, the *stall* is time necessary to "catch up" with the current *inforuction* flow to avoid the hardware hazard. Therefore the *stall*s are detected when any types of *inforuction buffers* have greater amounts of *infoructions* than the dimension we pre-defined. During the *stall*, the internal clock is incremented without generating new *inforuction* sets. The *stall* will be continued until amounts of *infoructions* in all types of the *inforuction buffers* are smaller than the limitation we defined. In the next section, we will go over the simulation setting and results of the simulation with observations.

4.3 Simulation settings and simulation results

In the previous section, we go over details of the genetic algorithm for the *static reconfigurable* CPUs. Before we show the results of simulations, we go over the details of the simulation setting.

We simulate our programs with 77 high end computers simultaneously to reduce the simulation time. Performance of each individual computer which runs simulation is given in Table 4.1. In our simulation, we have two types of settings. One of them is common for all *static* computer simulations and the other is unique for each individual simulation. We go over the details of common characteristics and then describe details of unique settings.

Table4.1 Performance of Simulation Stations

	Performance
Cumulated # of computer	77
CPU	Intel® Core™2 Duo Processor 6700 @ 2.66 GHz
Memory	2 GB

We choose the following setting as common configuration for all static computers: number of different types of *inforuction*, characterization of each type of *infoructions*, amounts of *infoructions* generated in a time cycle, number of predefined architectures we use, characterization of each core architectures, number of generations in the genetic algorithm, and number of initial *population*. Also we set the number of maximum population we will use in the genetic algorithm as constant to improve the simulation time. We identify 4 different types of *infoructions* that we use to characterize the performance of core architecture and stochastic data in *inforuction* generation pattern for a certain application. The types of *infoructions* are similar to types of functional units in the superscalar processor [9]. The types of *inforuction* are logic, mathematic, floating point, and memory. We use 4 different pseudo applications that have unique stochastic information pattern for each *inforuction*. Each of application is identified as General, Logic/Math, Floating Point, and Memory, which corresponds to the intensity of the *infoructions* for a given application. Characteristics of each application in terms of stochastic data of *inforuction* generation are summarized in Table 4.2. We define the characteristics of 5 different specialized cores. We name each type of architecture such as GENeral, LOGic, MATh, FLoating Point, and MEMory, which corresponds to their specialization. The performances of each type of architecture in terms of IPCC are listed in Table 4.3. The total amounts of *inforuction* each processor can handle are set to 8 which is the realistic number since it corresponds to the number of functional units in the superscalar processor [9]. To emulate changes of application in the systems, we randomly choose three applications from the repeated sample which consisted of the four applications we defined. The list of all *inforuction* generation patterns is displayed in Table B.1 and Table B.2. The common characteristic of systems we simulate is set as Table 4.4. Each *inforuction* stream is unique for the same length of cycle for each application we want to simulate.

Table4.2 Stochastic data of Inforuction generation in the percentage

where L is Logic inforuction, MA is Mathematic infoructions, FLP is Floating point infoructions, and ME is Memory infoructions.

	L	MA	FLP	ME
General	35	30	20	15
Logic/Math	70	18	7	5
Floating Point	5	12	80	3
Memory	15	15	10	60

Table4.3 Amounts of inforuction that each architectures can process in the given time unit

where L is Logic inforuction, MA is Mathematic infoructions, FLP is Floating point infoructions, and ME is Memory infoructions.

	L	MA	FLP	ME
General	3	3	1	1
Logic	4	2	1	1
Math	2	4	1	1
Floating	1	1	5	1
Memory	1	1	1	5

Table4.4 Common setting for each configuration in simulation

	Setting
# of total simulation	64 (= 4^3)
Total # of Inforuction prefetched	64
Size of Initial population	15
# of generation	200
Maximum size of population	450 (= 15*30)
Dependencies factor	.05

The rest of the settings, such as the number of reconfigurable cores we have in a system, the number of minimum cycles for each application, the number of iteration we implement, and the amounts of *inforuction* that trigger stall, are chosen to be variables to observe the properties of the *static reconfigurable* CPUs. Each individual setting for these variables is shown in Table 4.5 and the complete set of actual setting for each system we will simulate is in Table B.3 through Table B.6.

Table4.5 Variables and their possible settings

	Candidates of setting we will setting
# of cores in the system	2, 4, 6, 8, 10, 12, 14
Length of each application	50, 100, 200, 400, 600
# of iterations we implement	10, 15
Stall trigger	30, 50

4.4 Simulation Results and Observations

Our simulation time to complete individual configuration over 64 different application sequences are less than 5 minutes with the computer we used. Before discussing about any further observation of results, we discuss how we will compare the results of simulations among the different settings. We use a PITCGEN (Percentage of the performance Improvement in terms of Time necessary to process all *infoructions* Compared with the performance of GENeral core only systems) to describe the performance of our systems. Remember, our raw simulation result is time necessary to complete the *inforuction* streams. To obtain the performance improvement compared with the general core only system, we use the expression (4.1). We identify these ratios as a Specific PITCGEN (SPITCGEN) since these measurements are dependent on the types of application sequences in the simulation. To find the Average PITCGEN over all application sequences (APITCGEN), we use the expression (4.2). The result of the calculation of APITCGEN is shown in Table 4.6. We should not forget that SPITCGEN and APITCGEN are both dependent on the setting of the systems.

$$PITCGEN = 1 - \frac{Performance\ of system\ we\ want\ to\ compare}{Perfomance\ of\ general\ core\ only\ system} \tag{4.1}$$

$$APITCGEN\ for\ a\ given\ setting = \left(\frac{\sum_{all\ application\ pattern} SPITCGEN}{number\ of\ application\ pattern}\right) \tag{4.2}$$

Figure 4.4 and Figure 4.5 (Enlarged figure is in Appendix: Figure B.1 through Figure B.6) displays the APITCGEN for BEST (BEST performance we find in the whole algorithm), AVER (AVERage performance of the best performance we find in each *iteration*), LOG (LOGic specialized core only systems), MAT (MATh specialized core only systems), FLP (FLoating Point operation specialized core only systems), and MEM (MEMory operation specialized core only systems).

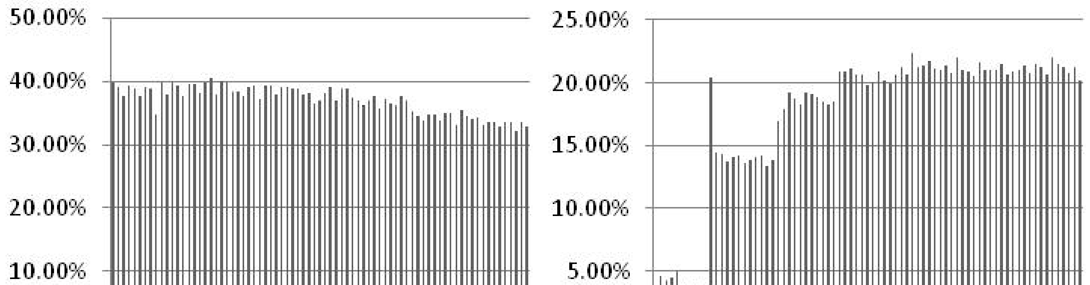


Figure4.4 APITCGEN for different types of systems

Left graph: result of BEST and Right graph: result of AVER. The horizontal axis displays different settings: S001 through S077 which is depicted in Table B3 through Table B6.

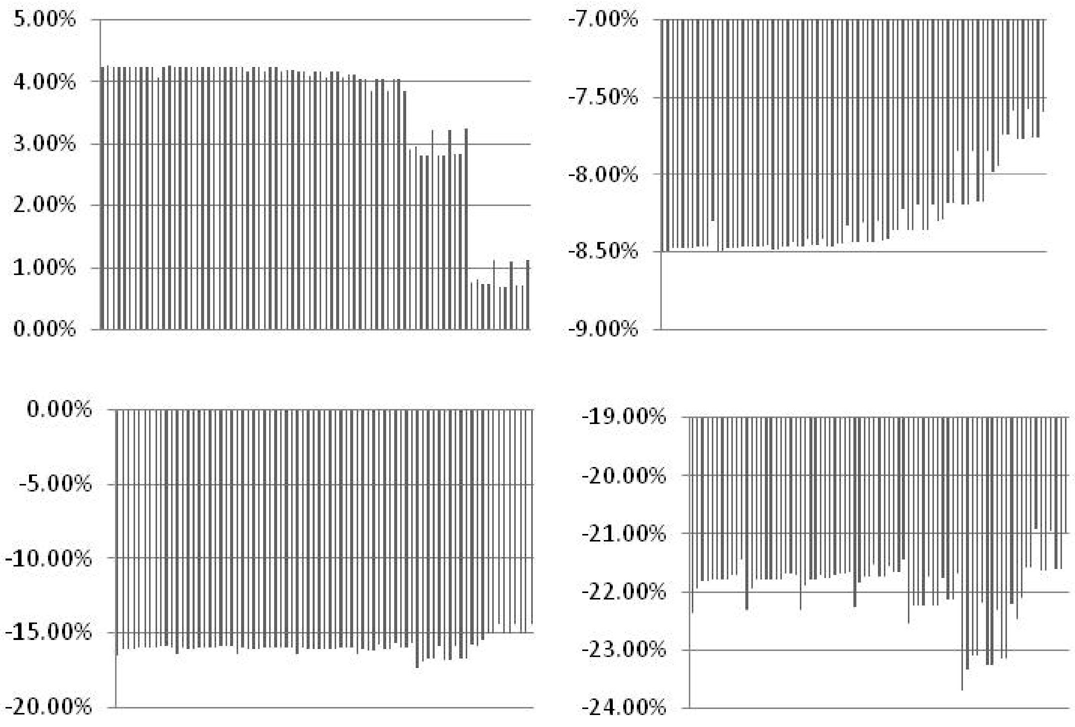


Figure4.5 APITCGEN for different types of system

Top Left graph: result of LOG, Top Right: result of MAT, Bottom Left: result of FLP, and Bottom Right: result of for MEM. The horizontal axis displays different settings: S001 through S077, which are depicted in Table B3 through Table B6.

Figure 4.4 and Figure 4.5 demonstrate that we seem to have about 30 % improvement in overall performance compared to GEN. This performance increase comes from the result of the *static* reconfiguration according to the individual *inforuction* streams. Since we have a different configuration for each *inforuction* stream, we can compare our result with the best configuration for single design core only systems. To compare the performance of our system with the best performance for single design core only systems, we use expressions (4.3) and (4.4). Since GEN systems do not produce the best PITC for all settings, this comparison provides more information about the optimization capability of our system.

$$PITCALL = 1 - \frac{Performance\ of our\ system}{Perfomance\ of\ best\ single\ design\ core\ only\ system} \tag{4.3}$$

$$APITCALL\ for\ a\ given\ setting = \left(\frac{\sum_{all\ application\ pattern} SPITCALL}{number\ of\ application\ pattern}\right) \tag{4.4}$$

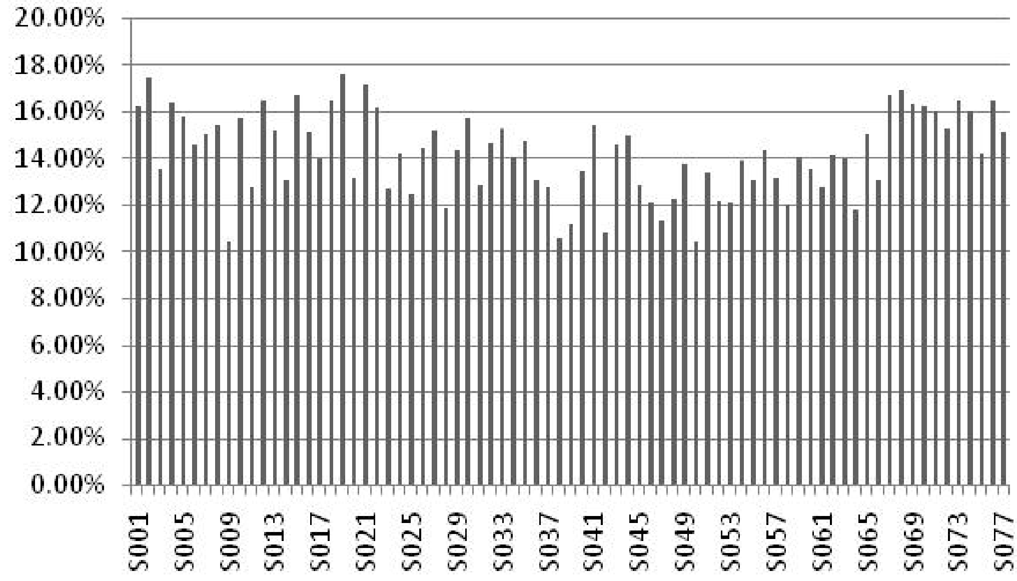


Figure4.6 APITCALL with BEST for different types of systems

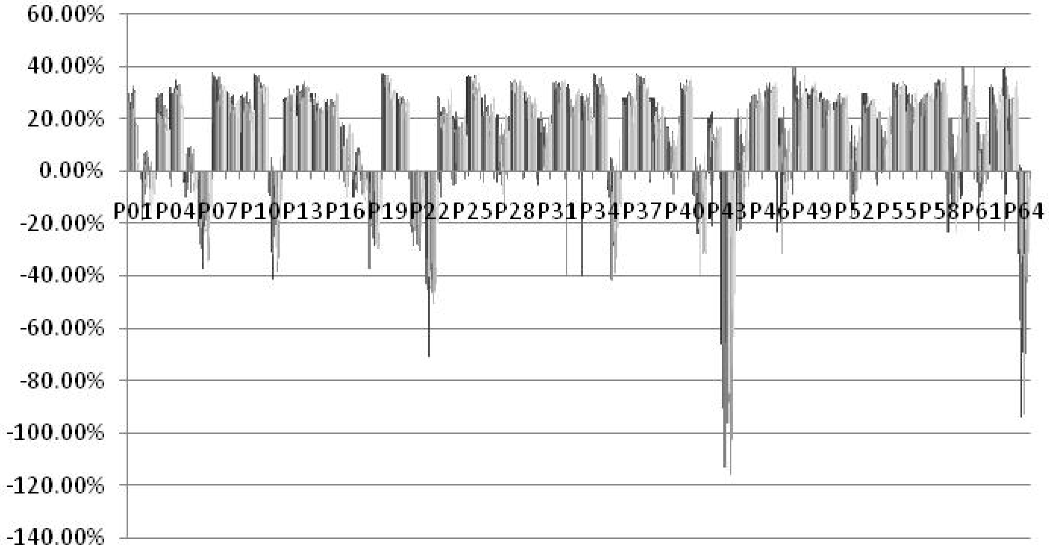


Figure4.7 SPITCALL with BEST among systems

Each entry in the graph corresponds to a specific setting (S001 – S077) in Table B.3 through B.6.

Figure 4.6 and Figure 4.7 demonstrate the performance of our system that is calculated from expressions (4.3) and (4.4). As we focused on Figure 4.6, the average performance improvements we achieve with our systems compared to the best performance of the single design core only systems is about 14 % and we have only positive average improvement for all settings. The 14 % increase might not seem outstanding. We look for details about this 14 % increase. Figure 4.7 demonstrate the interesting results and problems of our *sta*tic systems. The problem is also one of the problems of the genetic algorithm. Each individual PICTALL is ranged from about -120 % to 40 %. Most parts of settings in Figure 4.7 demonstrate the positive performance improvement or the same performance as the best performance for single design core only systems. As we describe, the magnitude of negative performance improvement or performance reduction is momentous because it reduces our average of APICTALL, even though the number of simulations which end up with negative result is about 20 % of the entire simulation results. Table 4.6 demonstrates the several data of our simulation.

Table4.6 Statistical data of SPITCALL with BEST

	# of simulation	% of Occurrence	Average
Total Number	4928	100 %	14.25 %
Positive PITCALL	3919	79.53 %	22.33 %
Zero PITCALL	69	1.40 %	0 %
Negative PITCALL	940	19.07 %	-18.39 %

In the theory and our simulation setting, we should not get any result with a negative SPITC since all single core only configurations, which are GEN, LOG, MAT, FLP, and MEM, are one of the subsets in our search space. Therefore, if our optimization method uses the exhaustive search, we could identify these configurations as a superior configuration whenever it is appropriate. The reason why we cannot find these configurations can be explained with the one of the weaknesses of a genetic algorithm. The genetic algorithm uses a stochastic search method. During the generation of the set of candidate *chromosomes*, we have a certain probability to generate each candidate based on a choice of parents. There are certain probabilities that we do not check a certain candidate. Therefore, there is always non-zero probability that we cannot find the truly optimized candidate. What we end up with in our optimization process is *local optimums* that are different from the *global optimum*. This result is well displayed when we refer to Figure 4.8 which shows the APITCALL with AVER and Figure 4.9 which shows SPITCALL with AVER. AVER, which is the average of the "best" values we find over the individual iteration, is much less than the BEST, which is the "best" we find over all iterations (Data for SPITCALL with AVER is displayed in Table 4.7). The reason is that there are some probabilities that the algorithm could not find the true "optimum" within the single iteration.

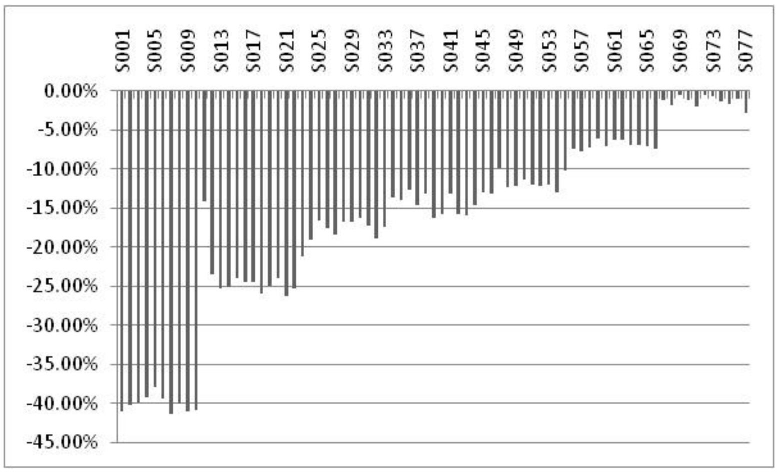


Figure4.8 APITCALL with AVER for different types of system

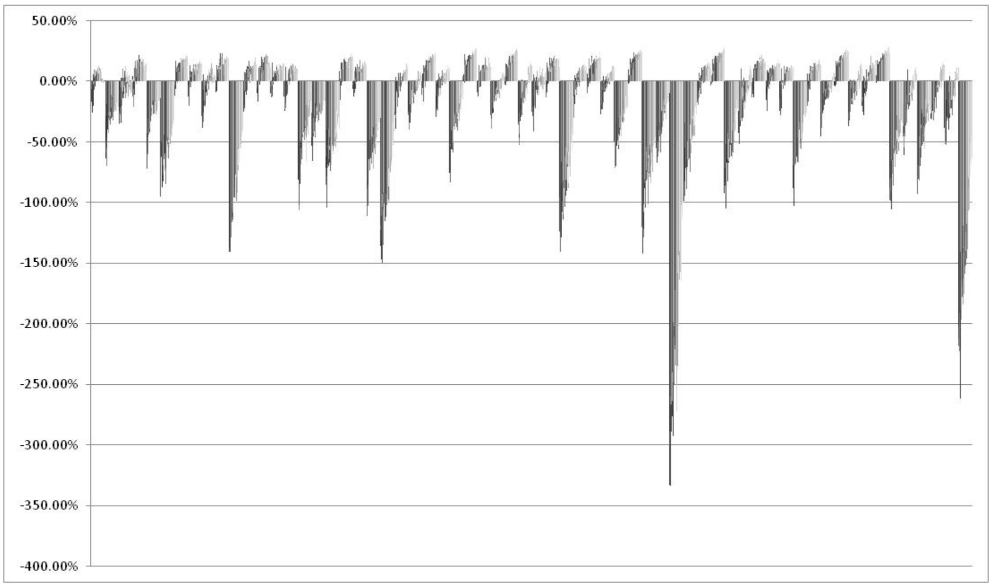


Figure4.9 SPITCALL with AVER among systems

Each entry in the graph is a specific setting (S001 – S077) in Table B.3 through B.6. X Axis is for the sequence pattern of applications from P01 to P64 in Table B.1 and Table B.2;

Table4.7 Statistical data of SPITCALL with AVER

	# of simulation	% of Occurrence	Average
Total Number	4928	100 %	-16.44 %
Positive PITCALL	2220	45.05 %	11.90 %
Zero PITCALL	1	0.02 %	0 %
Negative PITCALL	2707	19.07 %	-20.68 %

We can easily improve our algorithm by introducing 5 single design core only systems into the first generation of our search. With this method, we are sure the search space of simulation includes the single design core only cases. Also this modification improves the search area of the algorithm since the single design core only systems are not similar to each other. As a result, our search algorithm can produce better results. Simulation results of the modified algorithm are displayed in Figure 4.10 through Figure 4.12. The APITCGEN and SPITCGEN with BEST are in Figure B.14 through Figure

B.21. Compared with Figure 4.10 through Figure 4.13, our algorithm shows improvement in the optimization abilities. We have only positive performance improvements in both SPITCALL with BEST and SPITCALL with AVER. The negative SPICTALL is replaced with the zero or the positive SPICTALL since we increase the variety of systems in the first generation and we are forced to evaluate the 5 different types of single core only systems.

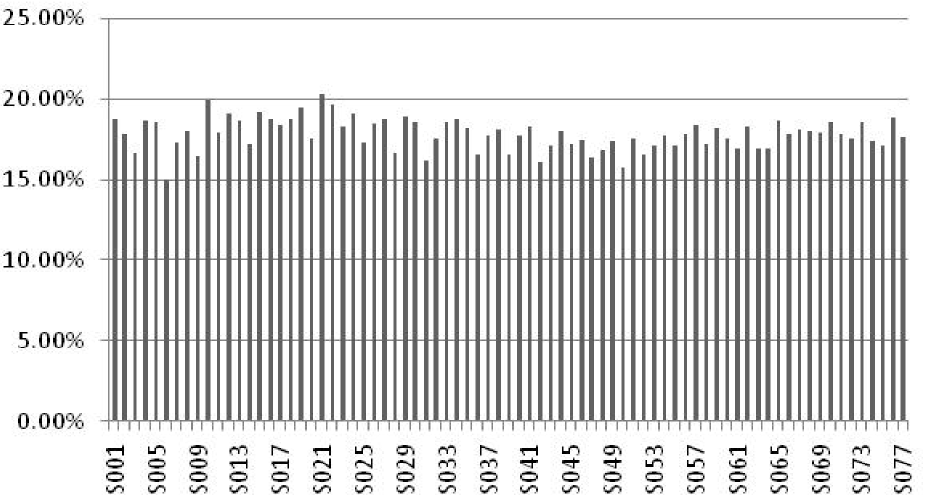


Figure4.10 APITCALL with BEST for different types of system with modified algorithm

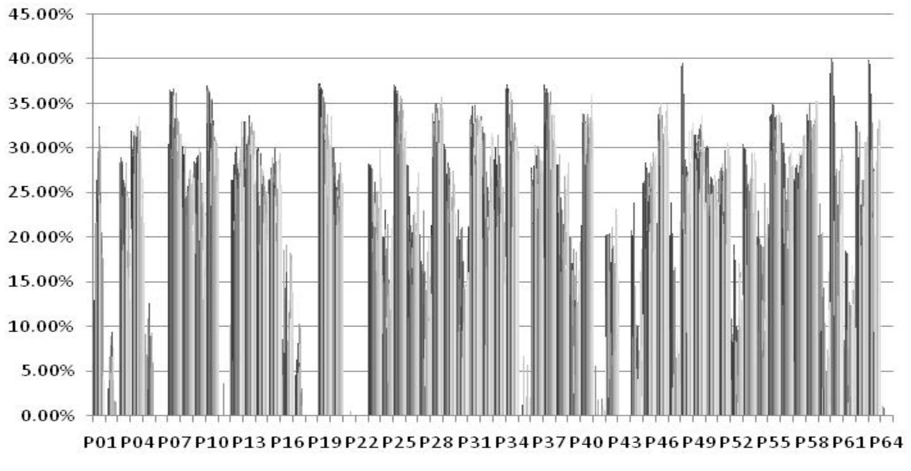


Figure4.11 SPITCALL with BEST among systems with modified algorithm

Each entry in the graph is a specific setting (S001 – S077) in Table B.3 through B.6.

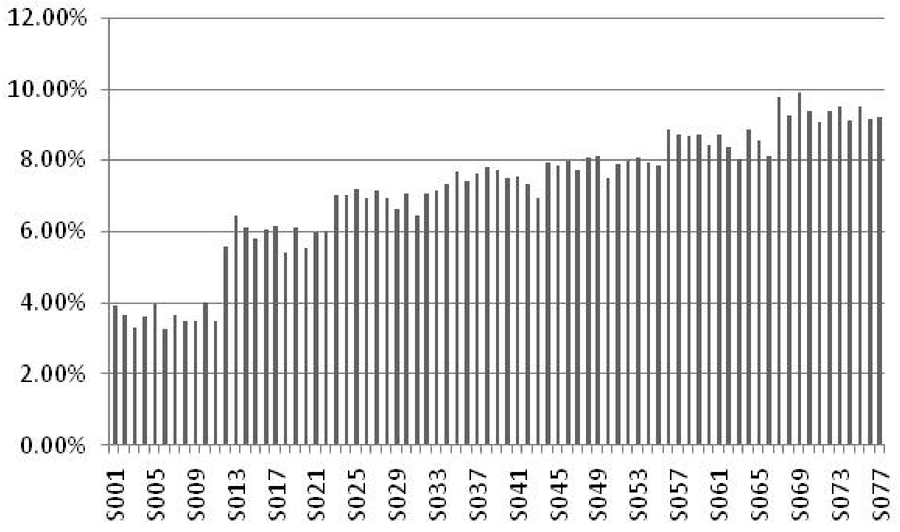


Figure4.12 APITCALL with AVER for different types of system with modified algorithm

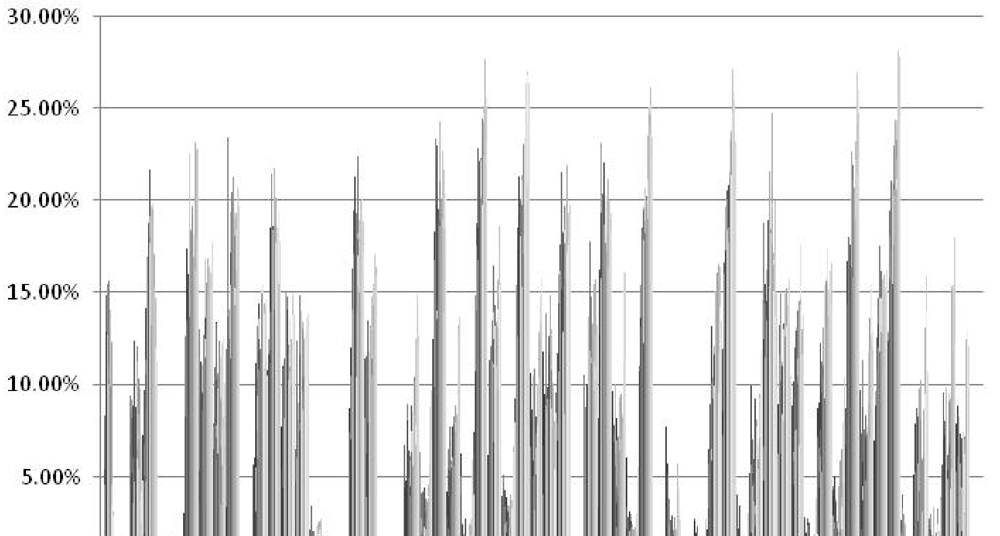


Figure4.13 SPITCALL with AVER among systems with modified algorithm

Each entry in the graph is a specific setting (S001 – S077) in Table B.3 through B.6.

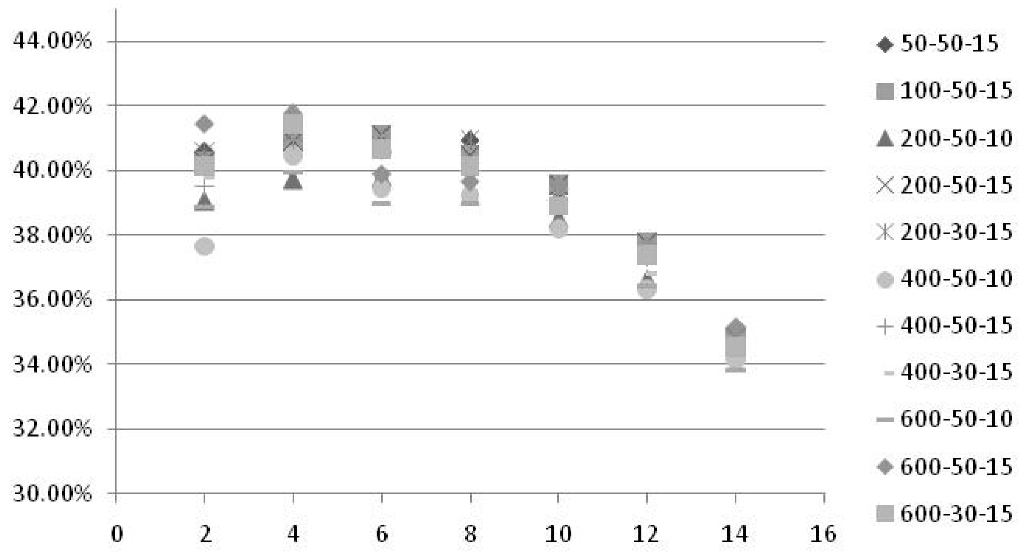


Figure4.14 Changes on number of cores in APITCGEN for BEST

Legend explains the rest settings of simulation with the following format: Cycles of each application: Stall trigger levels: Number of iterations.

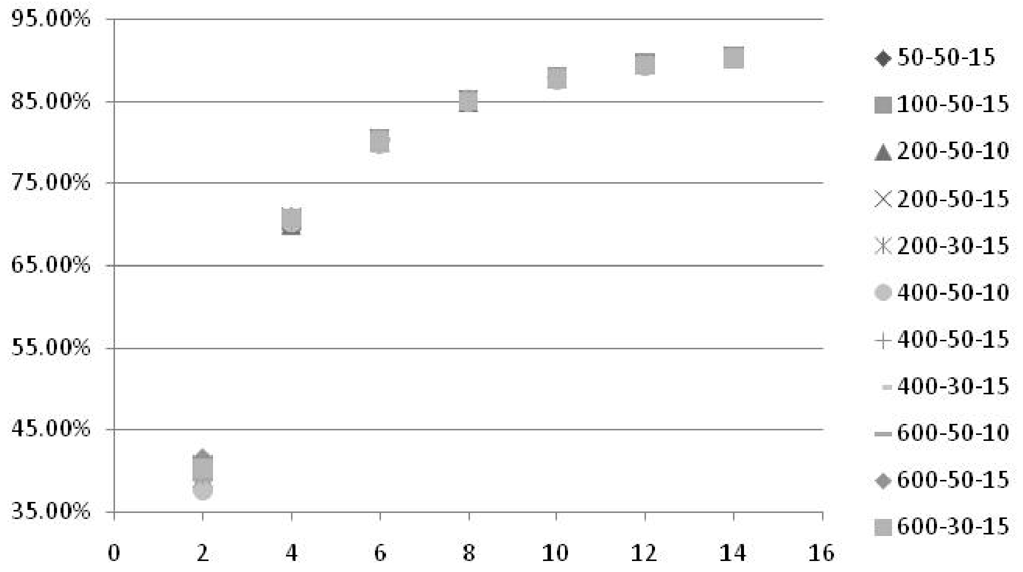


Figure4.15 Changes on number of cores in APITCGEN2 for BEST

Legend explains the rest settings of simulation with the following format: Cycles of each application: Stall trigger levels: Number of iterations.

Figure 4.14 displays the APITCGEN for BEST. According to this figure, performance improvement compared to GEN which has the same number of cores as the comparison target, becomes the largest when we have 4 core systems, then our percentage of improvement keeps decreasing. Since the maximum amounts of total *inforuction* generated in a cycle are fixed, the utilization of the system compared to systems with general core only cannot be optimized if we have too many cores. The fixed amounts of *inforuction* generated per cycle, which is 64, come from the maximum processing power of an 8 core system. If we adjust the data to compare with the APITCGEN with 2 cores (GEN2), we observe a graph as Figure 4.15. The graph demonstrates the remarkable improvement when we increase the number of cores from 2 to 4. With this change, the maximum processing power of the system becomes half of the total amounts of *infoructions* generated per cycle. The result of this comparison clearly displays the diminishing performance improvement when we increase the number of cores in a system, especially if the system can only pre-fetch the fixed amounts of *inforuction* per cycle. When we increase the number of cores from 2 to 4, we efficiently reduce the cause of *stalls* and do not increase the probability of *empty-running* at the same time. After the 4 core system, chances of *empty running* are increased more than decrease in the chances of *stalls*. Therefore the performance is not improved as in the case of 4 cores.

If we plot the same graph compared to the best single cores instead of GEN, the figure looks slightly different from the figures we observe. Figure 4.16 has similar characteristic as the previous figures up to 10 core system. However, we have the improvement for 12 and 14 cores systems. The reason why we cannot observe this improvement in the previous two figures can be explained with Figure 4.11. For the comparison with the best of single design core only settings, we have more range of changes in the performance measurement for each individual setting and each individual *inforuction* pattern. Therefore, we observe the improvements for 12 and 14 cores.

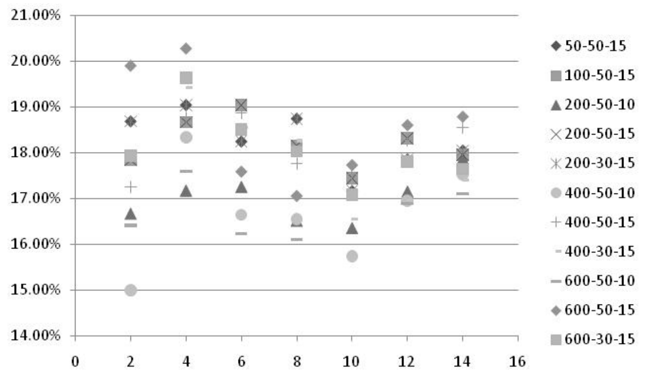


Figure4.16 Changes on number of cores in APITCALL for BEST

Legend explains the rest settings of simulation with the following format:
Cycles of each application: Stall trigger levels: Number of iterations.

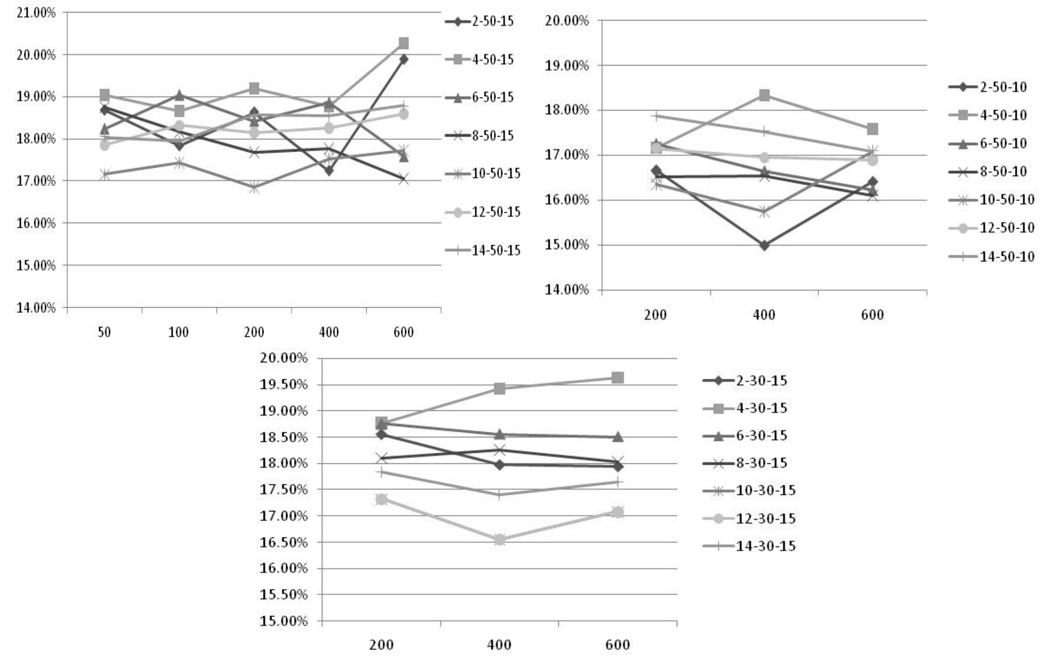


Figure4.17 Changes on APITCALL with different length of simulation for BEST

Legend explains the rest settings of simulation with the following format:
Number of cores: Stall trigger levels: Number of iterations.

Figure 4.17 displays the changes of APITCALL by changing the number of minimum length of each application. In Appendix B, we have larger size of these figures in Figure B.22 through Figure B.24. As we observe in the figures, our *static* system does not demonstrate any firm changes such as a monotonous decrease or increase when we change the cycles of each application. Instead of a monotonous increase or decrease, we observe the APITCALL stays in about .5 % of deviations. This comes from the differences for each length of simulation in the actual percentage of *inforuction* stream we simulate. Hence, we conclude this relationship as the following statement: the number of cycles for each application does not change the APITCALL significantly. Therefore our *static* operation seems to have steady performance improvement compared to the best configuration from single design core only system for any number of cycles.

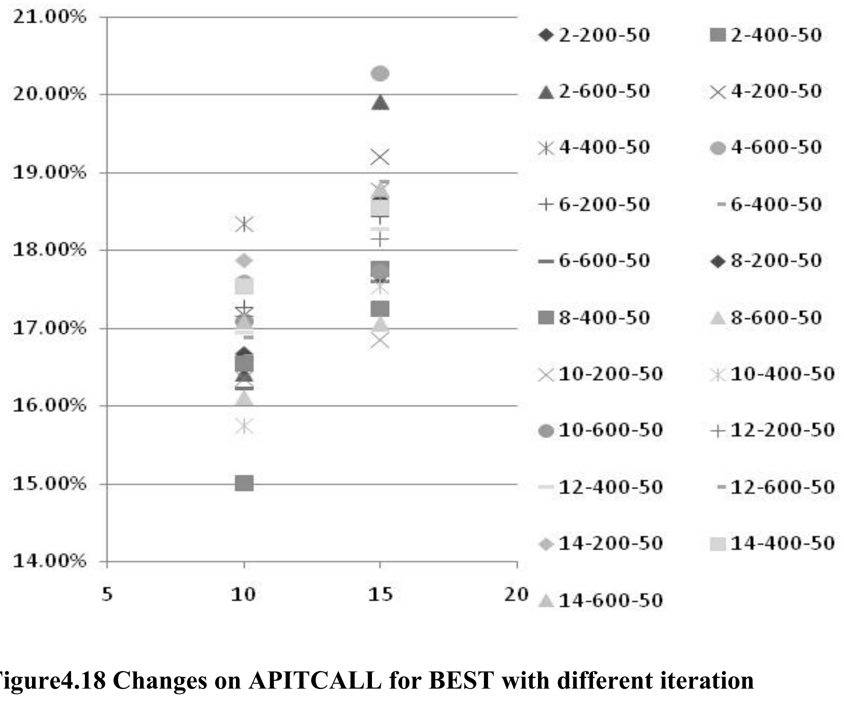


Figure4.18 Changes on APITCALL for BEST with different iteration

Legend explains the rest settings of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels.

When we study the graph in Figure 4.18, we can still verify the *local optimum* issues of the genetic algorithm, since the 10 iteration data has slightly smaller APITCALL compared to 15 iteration data. Therefore we should keep the number of iterations to a relatively large number to obtain the *global optimum* or better performance.

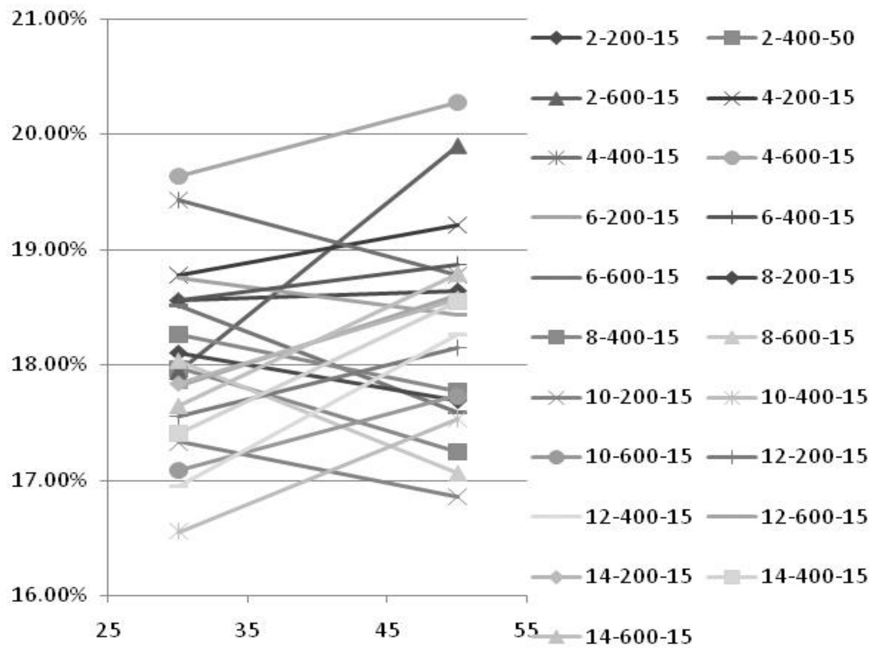


Figure4.19 Changes on APITCALL for BEST with different stall trigger

Legend explains the rest settings of simulation with the following format:
Number of cores: Cycles of each application: Number of iterations.

With Figure 4.19, we cannot observe the relationship between APITCALL and the changes of amounts of *inforuction* that causes the *stall,* which corresponds to size of the *inforuction buffer* that is related to the hardware hazards. Some of results display the performance improvement when we have tighter stall trigger condition and the others display a decrease in the performance with the tighter constraint. The conclusion of this observation is that the change in condition of stall trigger definitely changes the APITCALL, but we cannot predict the effect in APITCALL due to the changes in the *inforuction buffers* from the current observations. The reason we cannot observe the steady changes in the performance might comes from the following points: the *inforuction buffers* still have large enough size to avoid most stalls, the best configuration for single design core only systems already avoid the stalls as much as possible, the significant changes are averaged in the process of calculation of APITCALL, and/or the changes of best performance of single core only systems eliminate the proof of performance changes as in the observation of Figure 4.16.

In this chapter, we introduce the *static* genetic reconfigurable system which demonstrates the ability to find the optimized configuration. Even though our algorithm still remains the several rooms of improvement, we obtained the positive APITC with general cores only system for all settings. With the modified algorithm, we demonstrate the better performance in the optimization to each individual application sequence. This result stands for the following statement: the genetic algorithm and our *static* system can change configurations of the system to obtain better performance in PITCs. We also have to emphasize the following fact: the optimization process needs prior information of the stochastic data of *inforuction* generation or application sequences. Without any information, our *static* system and the static genetic algorithm would not produce the superior results and end up with meaningless results for reconfiguration.

CHAPTER 5

DYNAMIC RECONFIGURABLE SYSTEM

5.0 Specific details of the dynamic genetic algorithm

Since we discuss details of dynamic system in the previous sections, we will not repeat the same topic. Instead, we describe a topic which has not been discussed yet. That is how we determine the time to finish the previous reconfiguration process and start the new reconfiguration process. We have to use fixed time to estimate the fitness values, but we do not have to use the fixed time to terminate. We identify this time as *reconfiguration time*, which is the time we terminate the old reconfiguration process and then start the new process. If we do not choose this time interval carefully, our performance does not correspond to what we achieve in the real situation. For example, if we choose time interval shorter than the actual time for the reconfiguration of a FPGA (Field Programmable Gate Array) device, our simulation result is not realistic. Also, we have to take into consideration the calculation time for our reconfiguration process, since the timing of reconfiguration will change the performance we can get from our system.

In addition to the *fitness function* we define in the former section, we add one more parameter that memorizes the changes of processing performance within a time interval. After we find the best candidate by a genetic algorithm, we evaluate the predicted processing power of both the current configuration and the best candidate configuration. We compare these candidate performances for each time unit, beginning with the time frame we use for the genetic algorithm to the time of *reconfiguration penalties*. The time step we find through this process is set as the *reconfiguration time* that we previously defined. The flowchart of this process is shown in Figure 5.1, and the entire flowchart of the dynamic genetic algorithm is shown in Figure 5.2 and Figure 5.3.

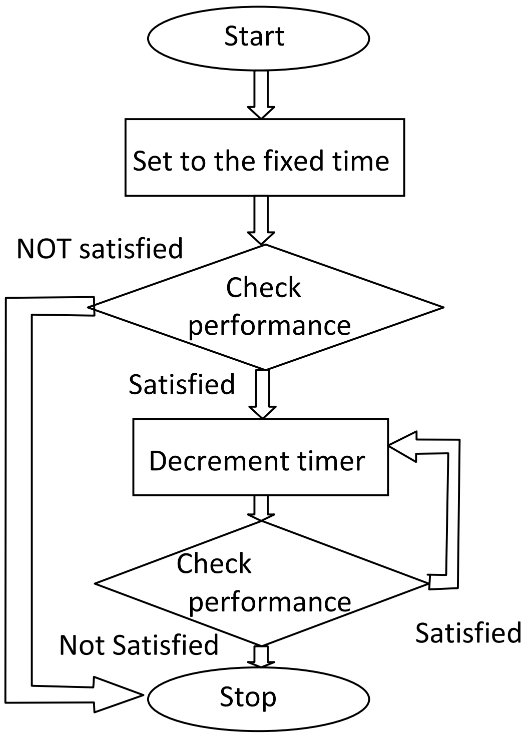


Figure5.1 Reconfiguration time determination process

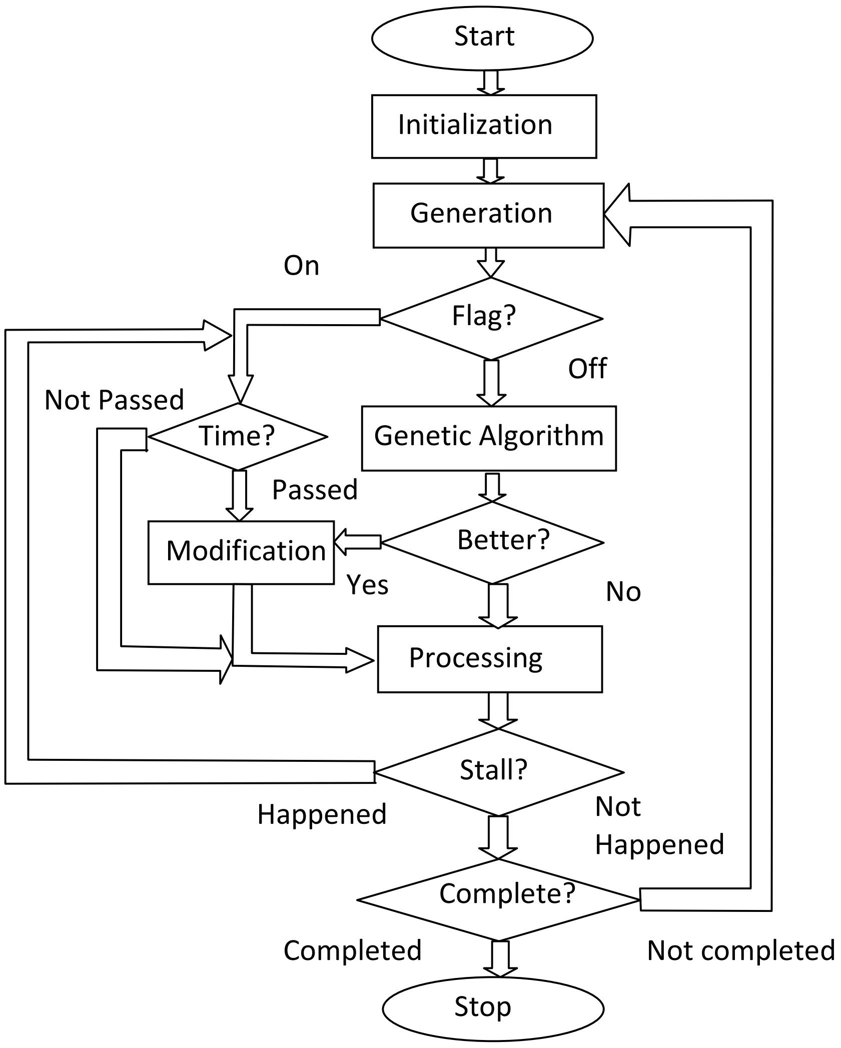


Figure5.2 Operation flowchart for the dynamic genetic algorithm

1.) Initialization Stage
    a. Set the number of application and number of iteration for each application
2.) Generation Stage
    a. Create the specific total amounts of information
3.) Reconfiguration Flag Check Stage
    a. Check whether reconfiguration is currently in the process or not
4.) Genetic algorithm Stage
    a. Implement genetic algorithm to find the better configuration
5.) Verification Stage
    a. Check whether the new configuration produce the better or not
6.) Process Stage
    a. Increment the time unit
    b. Process the # of information for the current processing power
7.) Stall Check
    a. Check whether the stall is happened or not
8.) Information check
    a. Check whether all information is completed or not
9.) Reconfiguration time Check
    a. Check whether the reconfiguration time is passed after the trigger of reconfiguration
10.) Modification Stage
    a. Change the performance of the system by adding the restriction or removing it

Figure5.3 Brief Description of flowchart of the dynamic genetic algorithm

5.1 Simulation settings

In the previous chapter, we verify that the optimization ability of the genetic algorithm with the *static reconfigurable system* on the average. We use several fixed settings as common factors between the previous chapter and the present chapter, such as: stochastic information of *inforuction* generation in each application, number of different application, total number of inforuction generated in a given cycle, the number of core in a system, types of core architecture, types of *inforuction*, performance of each type of core architectures that are listed in Table 4.2, .Table 4.3 and Table 4.4. We change the number of generation to smaller number since we want to run this algorithm faster and the generation is the only *stopping criteria* as we describe. We also do not limit the size of the *population* inside the algorithm, but we limit the size of the *population* which carries over to the next implementation of the genetic algorithm. These fixed settings, which are different from Table 4.4, are listed in Table 5.1. Also we use several variable settings as common to the setting listed in Table 4.5. The settings for variable parameter are listed in Table 5.2. All combinations of setting are listed in Table B.11 – Table B.14. As we describe in the previous chapter, we will observe the differences in the performance of *static reconfigurable* system and *dynamic reconfigurable* system. We could repeat the algorithm as we did in the *static* system, but we put more focus on the speed of the algorithm, since this is a time-critical operation which the environment changes as time goes and performance improvement is also dependent on the time we complete the algorithm. The fixed time we describe during the previous section is determined from the expression (5.1)

Table5.1 Common setting for all simulation

	Setting
# of total simulation	64 (= 4^3) [same as Table 4.4]
Total # of Inforuction at a time unit	64 [same as Table 4.4]
Size of Initial population	15 [same as Table 4.4]
# of generation	45
Maximum carry over factor (size)	4 (60 = 4*15)
Dependencies factor	.05
Prediction factor	2

Table5.2 Variable Setting for each simulation

	Candidates of setting we will setting
# of cores in the system	2, 4, 6, 8, 10, 12, 14 [same as Table 4.4]
Length of each application	50,100, 200, 400
Reconfiguration penalties	6,8,10
Stall trigger	30, 50 [same as Table 4.4]

$$Fixed\ time\ = Lenghth\ of\ RP * Prediction\ factor \tag{5.1}$$

where RP is reconfiguration penalties

The choice of reconfiguration penalty is based on [34], where they said that the time necessary to complete the full programmable device is typically done in several microseconds and is dependent on the type of devices used for the reconfiguration. Also the time is dependent on the area that needs to be reconfigured [34]. In addition to the actual reconfiguration time, which is the time necessary to change the architecture, we need to think about the time to calculate the best *dynamic* architecture corresponding to the *inforuction* streams as we describe previously. The process of calculation can be implemented along with processing of *inforuction* in our main system. So we choose the reconfiguration penalties, which are sum of time necessary to change the architecture and time needs to calculate the *optimized candidates*, as 6, 8, and 10 cycles. We run the MATLAB codes with computers which have the same specifications as listed in Table 4.1. The cumulated numbers of computers used for *dynamic* system simulation is 102 since we simulate each individual setting with a different computer.

5.2 Simulation Results and Observations

As we describe in the previous chapter and this chapter, we can use several results of observations from the previous chapter, since we use common settings. We set the model of the conventional system as general cores only and use the performance measurement of our system which is similar to what we derive in expression (4.1) and (4.2). We use the both SPITCGEN and APITCGEN, and also we use the SPITCBEST and APITCBEST, in which BEST is the result of *static* systems in the previous chapter. The expressions to calculate SPITCBEST and APITCBEST are in expression (5.2) and (5.3). With these methods, we can easily compare the results of performance improvements from the different types of systems.

$$SDPITCBEST = 1 - \frac{Result\ of\ our\ dynamic\ System}{Performance\ of BEST} \quad (5.2)$$

$$ADPISTCGEN = \left(\frac{\sum_{all\ application\ pattern} SDPITCBEST}{number\ of\ application\ pattern} \right) \quad (5.3)$$

where BEST performance we obtain in Chapter 4

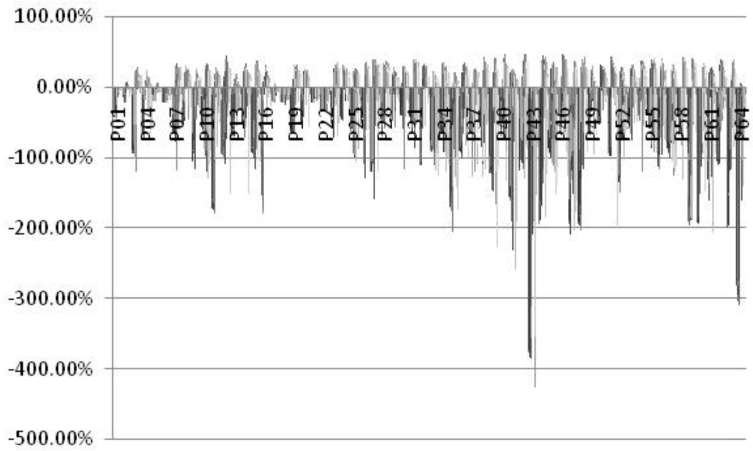


Figure5.4 SPITCBEST with dynamic systems

Each entry in the graph is a specific setting (D001 – S077) in Table B.11 through B.14.

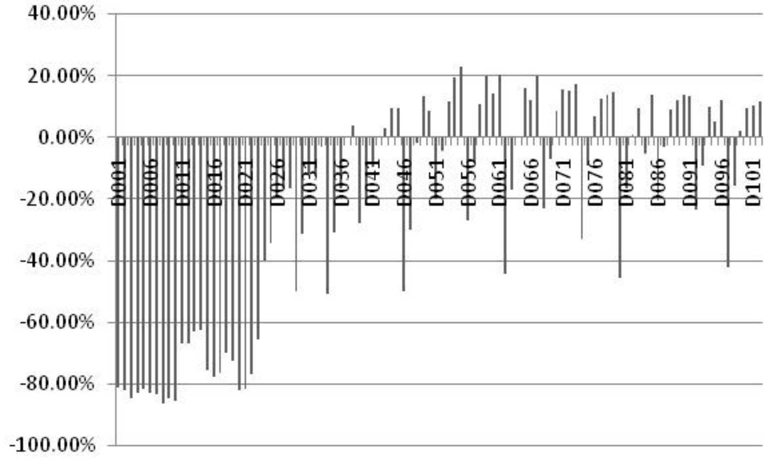


Figure5.5 APITCBEST with dynamic systems

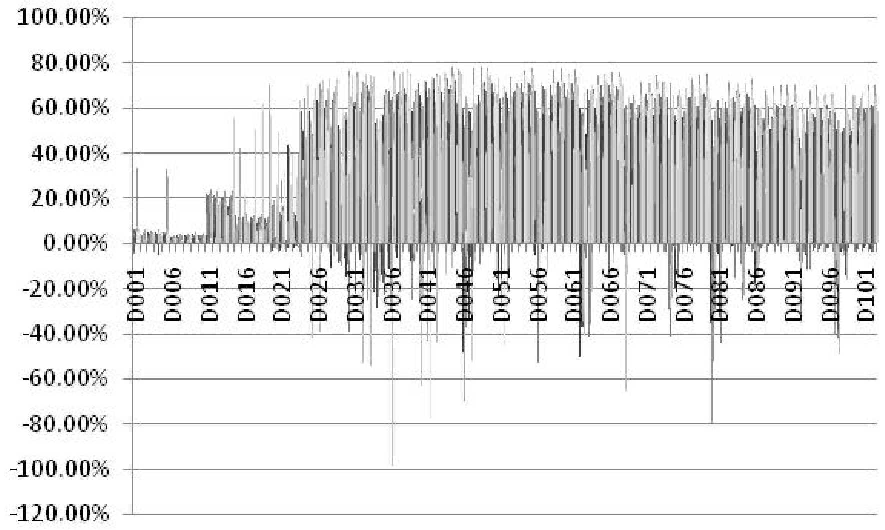


Figure5.6 SPITCGEN with dynamic systems

Each entry in the graph is a specific setting (D001 – S102) in Table B.11 through B.14.

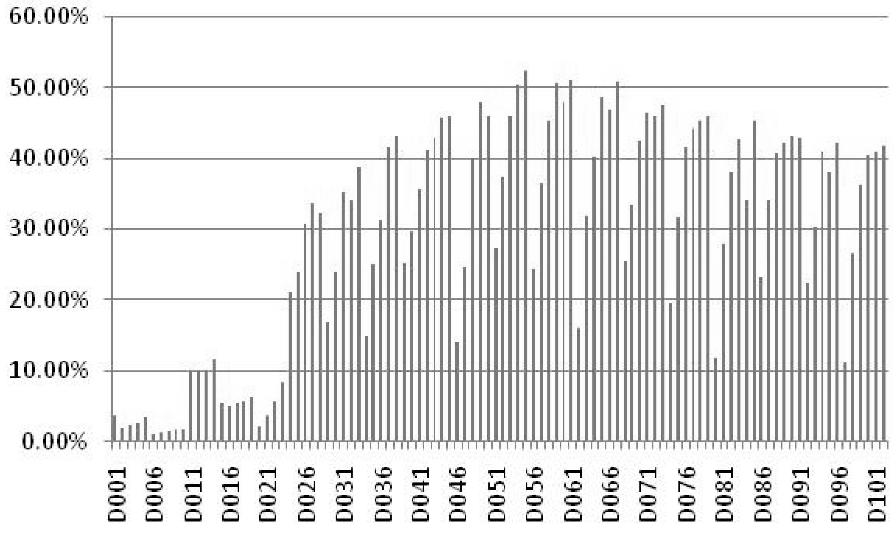


Figure5.7 APITCGEN with dynamic systems

Figure 5.4 and Figure 5.5 display the simulation results with SPITCBEST and APITCGEN. From these figures, we have to conclude that dynamic systems might not perform as well as the static systems, which are described in Chapter 4. APITCBESTs show that we have a performance reduction for most settings, but for several settings we have performance improvements. Since these improvements are observed in comparison to BEST, the dynamic systems have better performances compared to any other fixed architecture system under certain situations. The fact that we do not get a better performance improvement in most situations comes from two major reasons. One of them is inactive time of reconfigurable cores during reconfiguration. The *static* system does not have any performance reduction due to the reconfiguration. Even if we compare with a normal system, which has only general cores, the effect of performance reduction is clear. This statement is supported with Figure 5.6 and Figure 5.7. We have about 20 % of performance improvement on the average of APITCGEN, but we have several performance reductions in SPITCGENs. If we compare our result with highly optimized systems for specific *inforuction* streams, the performance reduction due to reconfiguration becomes more clear and significant than comparison between *dynamic* systems and GENs. Another reason is the issue of *local optimums*. As we describe in the previous chapter, our optimization process might be trapped with *local optimums.* To avoid *local optimum*s, we increase the number of iterations in the *static* systems. In our *dynamic* system, we cannot increase the number of iterations unless we have sufficient computational power for optimization processes, since the calculation cost would also increase as we increase the iterations.

We study the details of characteristics of the system with APITCGEN. We investigate the effect of changing the settings of our systems. Figure 5.8 demonstrates the performance changes due to increase in the numbers of cores in the system. Larger figures are shown in Figure B.25 through Figure B.27. From the graphs, we can observe that we have significant performance improvement between 4 core systems and 6 core systems, and we have very low improvement at 2 core systems. Since with 2 cores systems, we have fewer choices for configurations and performance reduction due to reconfiguration is significant, it is natural to have such low performances. Most results of the simulation have the higher performance improvement when we compare with GEN systems, which have the same number of cores. Even from these graphs, we can observe that the magnitude of performance improvement decreases as we increase the number of cores in a system. The reason of these behaviors is exactly the same as what we describe in the previous chapter: the fixed amounts of *inforuction* generation per cycle. From this point, we can conclude that the peak of performance improvement would be dependent on the number of *inforuction* generated per cycle. Figure 5.9 displays the relationship between the numbers of reconfiguration of the system and the number of cores in a system. Since we have performance reduction during the reconfiguration process, the performance improvement from the reconfiguration would become smaller when we have more cores with fixed size *inforuction* prefetching system.

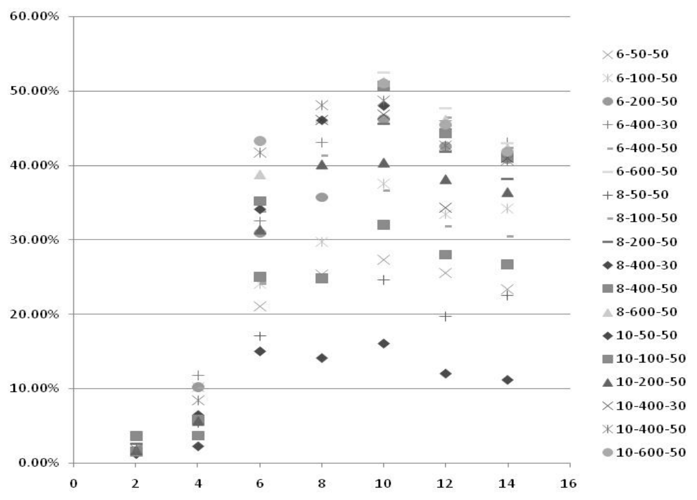


Figure5.8 APITCGEN with dynamic systems (Changing the number of core)

Legend explains the setting of each result of simulation with the following format: Number of delay cycles: Cycles of each application: Stall trigger levels.

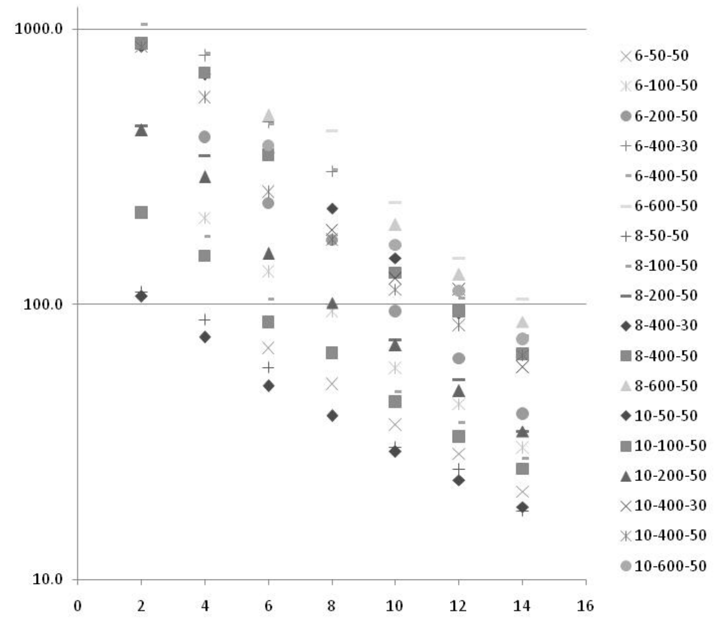


Figure5.9 Number of reconfigurations (Changing the number of cores)

Legend explains the setting of each result of simulation with the following format:
Number of delay cycles: Cycles of each application: Stall trigger levels.

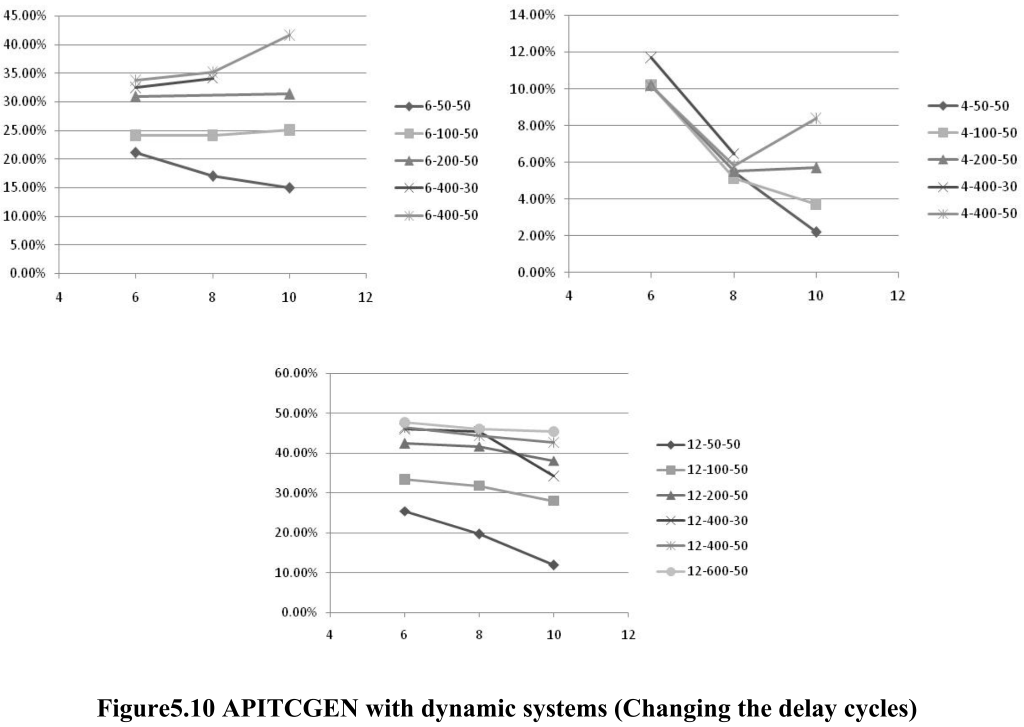


Figure5.10 APITCGEN with dynamic systems (Changing the delay cycles)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels.

With Figure 5.10, we explain the relationship between the performance improvement and the delay due to reconfiguration. All set of configurations is located in the Appendix B: Figure B.28 through Figure B.34. In these figures, we can observe two opposite trends in the change of performance improvement. The decrease of performance improvement can be easily explained with the performance reduction during the reconfiguration process. As the figures show, we have lower performance improvement when we have larger *reconfiguration penalties*, since larger reconfiguration penalties imply more performance reduction during the reconfiguration process. There are several cases that we have performance improvement due to larger *reconfiguration penalties*. This might come from the fact that we eliminate the unnecessary reconfigurations that make the performance lower. This can be supported with Figure that displays the average number of reconfiguration occurred to process specific information stream in Figure 5.11. Figure B.35 displays whole setting of Average number of reconfiguration.

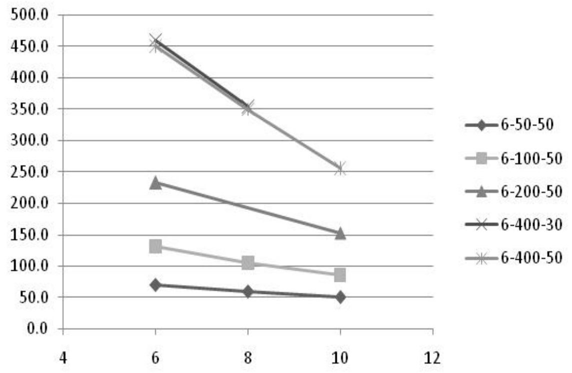


Figure5.11 Average number of reconfiguration (Changing the delay cycles for 6 cores case)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels.

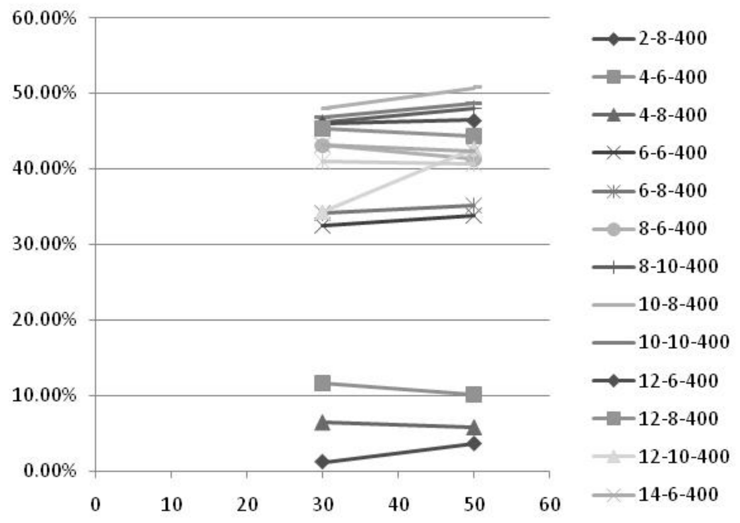


Figure5.12 APITCGEN Changes from the change of inforuction buffer

Legend explains the setting of each result of simulation with the following format:
Number of cores: Number of delay cycles: Cycles of each application.

Now we observe the relationship between the size of the *inforuction buffer* and the performance improvement. As Figure 5.12 displays, there is no strong relationship between the improvement and size of *inforuction buffers*. The reason of this behavior might come from two points: the percentage of performance change due to the changes in the buffer size is not large enough and/or the buffer size is still larger than the critical size which would dramatically change the performance. From the figure, we conclude that we do not have any strong relationship when we change the buffer size from 50 to 30.

Finally, we study the effect of length of simulation in the performance improvement. Figure 5.13 demonstrates that we have lower performance improvement for short time applications, but our performance improvement increases rapidly for medium length applications. Since we have more chances to reconfigure, the performances are increased. If we simulate longer cycles for each *inforuction* pattern, our performance improvement increases more slowly. The total cycles necessary to simulate all application increase, but the amounts of cycle we can improve without stalls and *empty running* do not increase as much as medium length applications. Therefore, the percentage of improvement seems to become saturated for longer cycles.

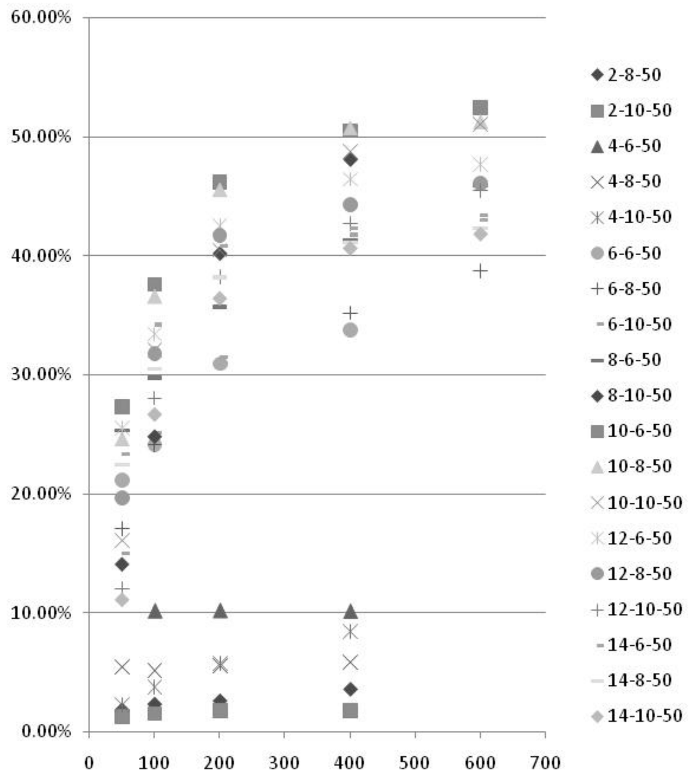


Figure5.13 APITCGEN Changes from the change in cycles of each application

Legend explains the setting of each result of simulation with the following format: Number of cores: Number of delay cycles: Cycles of each application.

We emphasize one of the facts we do not discuss yet. For the *static* applications, we need the prior knowledge of application sequences to efficiently improve the performance. However, for *dynamic* systems, we do not need such kinds of information, since our system has simple automatic learning mechanism, which learns the application patterns we processed in the several intervals. The major difference in the performance between the *static* and the *dynamic* computer comes from their optimization targets. In other words, the *static* computer optimizes their architecture to have the "best" performance using the result of simulation based on the prior knowledge of *inforuction* streams we will process, and the *dynamic* operation use the short time prediction to have better performance than the current configuration. Most part of performance differences in these systems come from the reconfiguration penalties which is only used in the *dynamic* system and mispredictions that might happen in the *dynamic* systems.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.0 Problems

Before concluding this study, we will go over some problems of our approach. Since the simulation results are based on several assumptions we made in this study, the results are not appropriate for a real world operation model. One of the most significant differences from real world operation is *reconfiguration penalties*. Even though we discuss the performance improvement from the *dynamic* reconfiguration in the previous chapter, the settings for *reconfiguration penalties* are not realistic. Since we have to simulate the short time behaviors to evaluate the *fitness functions* for *dynamic* systems, the calculation time for the reconfiguration process would be much greater than the time we assume in the previous chapter. Also the time to reconfigure the core will take much longer. From these defects, our simulation result is not accurate enough to demonstrate a genuine performance improvement from the *dynamic* systems. In addition, the core or computational unit used for the genetic algorithm was not included in the simulation. Therefore the comparison would not be fair because the number of cores used for GEN and our system is different. We will discuss how to avoid these issues in the future work section. Also, our approach of dependencies is significantly different from the real world operations of a computer. In the real world, we might have much more dependencies compared to our method and the dependencies would not occur in random locations. Also some operations of CPUs are not even described in this study. These functions, such as branch, are critical to determine the true performance of CPUs.

The stochastic data we use for each type of application during the generation of *inforuction* streams are not realistic. And the simulation itself is only a theoretical model for operations of a computer. It is not implemented in the lower level simulations such as SPEC and Simics, which have more detailed information of computer architectures and produce more accurate data. Our simulation is implemented in the higher models with assumptions we made in this study. Also the system does not calculate the delay of *inforuction* due to the length of paths from the buffer to each individual core.

6.1 Future work

We should improve the accuracy of our study to strongly justify our statement: "the human-brain-like-computer has superior power to process information compared to the conventional computer." To improve the accuracy of our study, we have to research specialized architectures. We should design more realistic core architectures and measure performances with benchmark programs. At the same time, we have to decide what kinds of specialized cores are appropriate to our list of configuration candidates. Also we have to improve accuracy of our computer models from two different viewpoints: improving the stochastic data of application and improving the model itself. To improve the stochastic data of applications, we need a long time observation of real computer operation with several famous benchmark programs. With observation, our long cycle simulations improve model reliability because our model corresponds to real systems. As we describe in the previous section, we should include the branch operation to make our model more realistic and improve the dependencies operation of our model to find the accurate performance benefits. We can modify our genetic algorithm to improve the search abilities in the optimization process. To improve functionality of our simulation, we should change the code. Currently, our simulation code uses several parameters as the constants of the program. Instead of using them as the constants, we should assign them as the inputs to improve usability of our code. Also the optimal cycle size of prediction and details of reconfiguration methods leave as a topic for future research since we do not have enough time to identify the best solution.

Table6.1 Previous Dynamic Approach and Major Problems

	Approach	Problem
Prediction Method	Each-cycle	Taking too long
Reconfiguration Penalties	6,8,10	Too short!
Target Time	Close future operation	Not sufficient time
Comparison	The same # of cores	Not Fair

As we promise in the previous section, we will discuss how we can improve the *dynamic* reconfigurable computer. We summarize the previous approach and issues related to it in Table 6.1. As we describe, the most significant issues of our system are related to time, such as the reconfiguration penalties and the calculation time for the genetic algorithm. One of the methods to solve these issues is to change the target time of optimization to further future operations, such that the reconfiguration and the calculation for the genetic algorithm will be completed beforehand. For example, if the calculation time of the genetic algorithm is 100 cycles and reconfiguration penalties are 150 cycles, we will predict the condition of our system after 500 cycles. Following that, we adjust the system to meet the demand in terms of processing power or to avoid bottlenecking of the system for 100 cycles from the prediction point. To accomplish this method, we need to reduce the calculation time of the genetic algorithm, since trials of prediction with each-cycle method in the genetic algorithm takes too long to get results on time. Instead of each-cycle prediction, we can use the multiple-cycle prediction. In the multiple-cycle prediction, we use the stochastic data of system to predict future operations, such as cycles that the system stalled, cycles that the system is in *empty-running* state, and cycles that the system encounters dependencies. Data we used with the previous method is also used to predict the generated *infoructions.* In this method, calculation of fitness values is similar to expression (3.1), but the method of evaluation is slightly different and more sophisticated. With the new method of prediction, there might be more chances of mispredictions. Hence, we also need to develop more sophisticated method of prediction. To make fair comparison, we will reduce one core from our system to assign the removed core as the control unit of the system. Another approach we can use to reduce the *reconfiguration penalties* is to set one core as the victim of reconfiguration. The selection of next victim is based on the lowest performance improvement among the activate cores in the system. We have one less core during the calculation time of genetic algorithm, but during the rest of the operation we have two cores less than GEN system. In other words, one core is in the process of reconfiguration during approximately half of the operation time. In this approach, we do not have to worry about the reduction of performance, since we already removed the core that would reconfigure for future operations. In this method, we should care about the performance changes due to reconfigurations, and due to the increase in the performance while waiting for the trigger of reconfiguration. Since we do not have enough time to implement the new idea of *dynamic reconfigurable* computer to simulate the performance, we leave this idea as future work.

We reemphasize the fact that our system is only theoretical right now. Therefore, we need research to implement our idea in the real system.

6.2 Applications

In this study, we prove the potential power of our multi-core *dynamic reconfigurable* genetic system compared to general homogeneous systems. In the process of our proof we also demonstrate the power of the genetic algorithm to find the better configuration in the *static reconfigurable* system. We will go over several different applications of our proposed systems in the real world. With the *static reconfigurable* system, we can determine the configurations and the number of pipelines required to obtain the maximum performance of processors within given resources for a specific purpose. The designers can use the results obtained in our algorithm for the starting point of their designs of superscalar processors for specialized applications that might be used in our *dynamic* systems. We can apply it to automatically find the optimum number of each type of pipelines; we do not have to do trial-and-error in the human brain. We can also create pseudo *dynamic* system from our *static* system. Instead of real run time reconfigurations implemented in our system, we keep switching the state of machines between inactive mode and active mode. During the active mode, we simply use the maximum processing power to execute *infoructions* and we obtain the data for next possible *inforuction* streams in some method such as the method used for the *dynamic* systems. Then after several time intervals we try to reconfigure the system with the stochastic information of *inforuction* streams. This method is similar to active mode and off mode of the wireless sensor network [35]. Since our *dynamic* system emulates the human brain, we can use our system as the brain for the general purpose robots. With our system, the robot can implement the several different tasks with optimized performance in the single system implementation. We can use our *dynamic* system where space is premium. Since our *dynamic* system can adjust the configurations to the purpose of the systems without any prior knowledge, we might reduce the space necessary for each type of specialized processors. If we change the *fitness functions* to find the minimum configuration within given resources, then we can minimize the number of cores necessary to complete the specific applications in the given cycles. In other words, we can create the system which dynamically turns off the unnecessary part of computer to reduce power consumption. We can apply our algorithm to create *dynamic* memory architectures. Also we can apply it to create much larger scale of *reconfigurable* systems.

6.3 Conclusion

In this study, we propose a heterogeneous *dynamic reconfigurable* computer. To demonstrate the optimization power of the genetic algorithm, we introduce a simpler type of system, which is called a *static reconfigurable* computer. Both of our proposed systems implement a genetic algorithm as part of the system. We introduce several ideas and background information to explain the operation of the systems and techniques to reduce the complexity of our systems and simulations. The algorithm is used for the optimization process to find the better system configurations with predefined architectures. We implement higher level simulations in MATLAB to observe the system performances. In the *static* system simulations, we evaluate each individual system with the *fitness function,* which calculates the time necessary to complete all *infoructions* in a specific stream. In this higher level simulation, we demonstrate approximately 35 % performance improvement compared to the homogeneous general multi-core system which has the same numbers of cores as our *static* system. Our system also demonstrates approximately 18 % of performance improvement compared to the best result from any homogeneous multi-core systems. These accomplishments are due to the prior knowledge of the streams we want to process. After we demonstrate the optimization power of the genetic algorithm, we implement our *dynamic* systems. Even though there are defects in our simulation, we verify potential superiority of our proposed system, which shows approximately 15 % average performance improvement compared to the homogeneous general multi-core system with the same number of cores. As we describe, there are several defects in our system, such as the problem of *local optimum*, problems of assumptions for the systems, and unfair comparison with homogeneous systems. We need further research on the proposed systems to determine the actual performance improvement in the real world. However, we are still motivated to continue with the research on heterogeneous *dynamic reconfigurable* computers because the flexibilities of human-brain-like-computers have potential to improve system performance and many applications which have potential to save space and power.

APPENDIXIES

APPENDIX A: REFERENCES

[1] G. Moore, “Progress in Digital Integrated Electronics,” in *Technical Digest—IEEE International Electron Devices Meeting,* IEEE, 1975, p.p. 11-13

[2] Intel, “Moore's Law, The Future - Technology & Research at Intel,” *Intel*. [Online]. Available: http://www.intel.com/cd/corporate/techtrends/emea/eng/209729.htm. [Accessed: April 22, 2008]

[3] Intel, “Intel® Core™2 Duo Processor Overview,” Intel [Online] Available: http://www.intel.com/products/processor/core2duo/ [Accessed: May. 20, 2008]

[4] Intel, “Intel® Core™2 Quad Processor Overview,” Intel [Online] Available: http://www.intel.com/products/processor/core2quad/ [Accessed: May. 20, 2008]

[5] IBM, “The Cell architecture,” IBM [Online] Available: http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html [Accessed: May. 20, 2008]

[6] J. Rice, K. C. Pace, M. D. Gates, G. R. Morris, and K. H. Abed, “Reconfigurable computer application design considerations” Southeastcon, 2008, IEEE, p.p. 236-243, 3-6 April 2008

[7] G.R. Morris, “Floating-Point Computations on Reconfigurable Computers,” *DoD High Performance Computing Modernization Program Users Group Conference*, 2007, p.p. 339-344, 18-21 June 2007

[8] I. Ouaiss, and R.Vemuri, “Hierarchical memory mapping during synthesis in FPGA-based reconfigurable computers,” *Design, Automation and Test in Europe, 2001. Proceedings of Conference and Exhibition 2001,* p.p. 650-657, 2001

[9] David A. Patterson, and John L. Hennessy, “Computer Organization & Design: The Hardware/Software Interface”, Morgan Kaufmann Publishers, San Francisco, California, 2005

[10] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in computer architecture evaluation," *Computer*, vol.36, no.8, p.p. 30-36, August 2003

[11] Standard Performance Evaluation Corporation, "SPEC Benchmark Suite," Standard Performance Evaluation Corporation [Online] http://www.spec.org [Accessed: May. 20, 2008]

[12] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the New Millennium," *Computer*, vol.33, no.7, p.p. 28-35, July 2000

[13] Virtutech, "Simics Benchmark Suite", Virtutech [Online] http://www.virtutech.com/whatissimics.html [Accessed: May. 20, 2008]

[14] A. E. Dunlop, and B. W. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," Transactions on *IEEE Computer-Aided Design of Integrated Circuits and Systems,* vol.4, no.1, p.p. 92-98, January 1985

[15] Bryan R. Buck and Jeffrey K. Hollingsworth, "Data Centric Cache Measurement on the Intel ltanium 2 Processor", in Proceeding on of the 2004 ACM/IEEE conference on Supercomputing, p.58, November 06-12, 2004

[16] A. Akram, J. Kewley, and R. Allan, "A Data Centric approach for Workflows," *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW '06. 10th IEEE International*, p.p. 10-10, October 2006

[17] Guo-Fang Nan, Min-Qiang Li, Dan Lin and Ji-Song Kou, "Application of evolutionary algorithm to three key problems in VLSI layout," Proceedings of 2005 *International Conference on Machine Learning and Cybernetics*. Volume 5, p.p. 2929 – 2933

[18] S. Coe, S. Areibi, and M. Moussa, "A hardware Memetic accelerator for VLSI circuit partitioning," *Computers and Electrical Engineering*, vol.33 no.4, p.p. 233-248, July, 2007

[19] S. Coe, S. Areibi, and M. Moussa, "A genetic local search hybrid architecture for VLSI circuit partitioning," in Proceeding on *16th International Conference on* Microelectronics, 2004. ICM 2004, p.p. 253-256, 6-8 Dec. 2004

[20] S. Areibi, M. Moussa, and G. Koonar, "A Genetic algorithm hardware accelerator for VLSI circuit partitioning," *International Journal of Computers and Their Applications*, 2005 vol. 12, no.3, p.p. 163-180.

[21] S. Areibi, and Z. Yang, "Effective memetic algorithms for VLSI design automation = genetic algorithms + local search + multi-level clustering," *Evolutionary Computation*, vol .12, no.3, p.p. 327-353, September 2004

[22] C. R. Reeves, and J. E. Rowe, “Genetic Algorithms: Principles and Perspectives: A Guide to GA Theory,” Kluwer Academic Publishers, Norwell, MA, 2002

[23] C. Darwin, “The Origin of Species,” Literature.org [Online] Available at: http://www.literature.org/authors/darwin-charles/the-origin-of-species/index.html [Accessed: May. 20, 2008]

[24] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht, “Genetic Algorithms for the Traveling Salesman Problem”, in Proceeding on *1st International Conference on Genetic Algorithms and Their Applications*, p.p.160 - 168, 1985.

[25] F. Zhuang, and F. D. Galiana, “Unit commitment by simulated annealing,” *IEEE Transactions on Power Systems*, vol. 5, no. 1, February 1990, p.p. 311-318.

[26] G. G. Yen and H. Lu, "Dynamic population size in multiobjective evolutionary algorithm,” in Proceeding on 9th IEEE Congress of Evolutionary Computation, p.p. 1648-1653, 2002.

[27] R. Takahashi, “Solving the traveling salesman problem through genetic algorithms with changing crossover operators,” *i*n Proceeding on *4th International Conference on Machine Learning and Applications, 2005*, p.p. 6, 15-17 December 2005
.
[28] A. Acan, H. Altincay, Y. Teko, and A. Unveren, “A genetic algorithm with multiple crossover operators for optimal frequency assignment problem,” *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.* , vol.1, p.p. 256-263, 8-12 December 2003.

[29] S. C.Esquivel, A. Leiva, and R. H. Gallard, “Multiple Crossover Per Couple in genetic algorithms,” *IEEE International Conference on Evolutionary Computation, 1997,* p.p. 103-106, 13-16 April 1997

[30] M.T. Hagan, H.B. Demuth, and M.H. Beale, “Neural network design”, PWS Publishing Company, Boston MA, USA, 1996.

[31] S. Karlin and H. Taylor, *A First Course in Stochastic Processes*, 2nd ed. San Diego, CA: Academic, 1975.

[32] H. Robbins and S. Monro, “A stochastic approximation method,” *Ann. Math. Statist.*, vol. 22, pp. 400–407, 1951.

[33] H. Kushner and G. Yin, “*Stochastic Approximation Algorithms and Applications,”* New York: Springer-Verlag, 1997.

[34] F. Mehdipour, M. S. Zamani, H. R. Ahmadifar, M. Sedighi, K. Murakami, "Reducing reconfiguration time of reconfigurable computing systems in integrated temporal partitioning and physical design framework," *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006,* p.p. 8, 25-29 April 2006

[35] S. Park, A. Savvides, and M.B. Srivastava. 2000. "SensorSim: a simulation framework for sensor networks".in Proceeding on the 3rd *ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems* (MSWIM 2000), p. p. 104-111. Boston, Massachusetts, United States: ACM Press.

APPENDIX B: TABLES AND FIGURES: SETTINGS AND RESULTS

Table B.1 Changes in stochastic information of inforuction generator – Part 1

where the top row represent pattern ID, left column represent order we use, G is general stochastic pattern, LM is logic /mathematic operation intensive pattern, FL is floating point operation intensive pattern, and M is memory operation intensive pattern

	P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12
1st	G	G	G	G	G	G	G	G	G	G	G	G
2nd	G	G	G	G	LM	LM	LM	LM	FL	FL	FL	FL
3rd	G	LM	FL	M	G	LM	FL	M	G	LM	FL	M
	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	P23	P24
1st	G	G	G	G	LM	LM	LM	LM	LM	LM	LM	LM
2nd	M	M	M	M	G	G	G	G	LM	LM	LM	LM
3rd	G	LM	FL	M	G	LM	FL	M	G	LM	FL	M
	P25	P26	P27	P28	P29	P30	P31	P32	P33	P34	P35	P36
1st	LM	LM	LM	LM	LM	LM	LM	LM	FL	FL	FL	FL
2nd	FL	FL	FL	FL	M	M	M	M	G	G	G	G
3rd	G	LM	FL	M	G	LM	FL	M	G	LM	FL	M
	P37	P38	P39	P40	P41	P42	P43	P44	P45	P46	P47	P48
1st	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL
2nd	LM	LM	LM	LM	FL	FL	FL	FL	M	M	M	M
3rd	G	LM	FL	M	G	LM	FL	M	G	LM	FL	M

Table B.2 Changes in stochastic information of inforuction generator – Part 2

where the top row represent pattern ID, left column represent order we use, G is general stochastic pattern, LM is logic /mathematic operation intensive pattern, FL is floating point operation intensive pattern, and ME is memory operation intensive pattern

	P49	P50	P51	P52	P53	P54	P55	P56	P57	P58	P59	P60
1st	M	M	M	M	M	M	M	M	M	M	M	M
2nd	G	G	G	G	LM	LM	LM	LM	FL	FL	FL	FL
3rd	G	LM	FL	M	G	LM	FL	M	G	LM	FL	M
	P61	P62	P63	P64								
1st	M	M	M	M								
2nd	M	M	M	M								
3rd	G	LM	FL	M								

Table B.3 Setting of Static systems (Only displays variable settings) – Part 1

where A is number of cores, B is length of each application in the time unit, C is numbers of iterations we implement, D is amounts of inforuction that cause stall and left column represent configuration ID for static system.

	A	B	C	D
S001	2	50	10	50
S002	2	100	10	50
S003	2	200	5	50
S004	2	200	10	50
S005	2	200	10	30
S006	2	400	5	50
S007	2	400	10	50
S008	2	400	10	30
S009	2	600	5	50
S010	2	600	10	50
S011	2	600	10	30
S012	4	50	10	50
S013	4	100	10	50

Table B.4 Setting of Static cores (Only displays variable settings) – Part 2

Where A is number of cores, B is length of each application in the time unit, C is numbers of iterations we implement, D is amounts of inforuction that cause stall and left index is configuration ID for static reconfigurable core.

	A	B	C	D
S014	4	200	5	50
S015	4	200	10	50
S016	4	200	10	30
S017	4	400	5	50
S018	4	400	10	50
S019	4	400	10	30
S020	4	600	5	50
S021	4	600	10	50
S022	4	600	10	30
S023	6	50	10	50
S024	6	100	10	50
S025	6	200	5	50
S026	6	200	10	50
S027	6	200	10	30
S028	6	400	5	50
S029	6	400	10	50
S030	6	400	10	30
S031	6	600	5	50
S032	6	600	10	50
S033	6	600	10	30
S034	8	50	10	50
S035	8	100	10	50
S036	8	200	5	50
S037	8	200	10	50
S038	8	200	10	30
S039	8	400	5	50

Table B.5 Setting of Static cores (Only displays variable settings) - Part 3

Where A is number of cores, B is length of each application in the time unit, C is numbers of iterations we implement, D is amounts of inforuction that cause stall and left index is configuration ID for static reconfigurable core.

	A	B	C	D
S040	8	400	10	50
S041	8	400	10	30
S042	8	600	5	50
S043	8	600	10	50
S044	8	600	10	30
S045	10	50	10	50
S046	10	100	10	50
S047	10	200	5	50
S048	10	200	10	50
S049	10	200	10	30
S050	10	400	5	50
S051	10	400	10	50
S052	10	400	10	30
S053	10	600	5	50
S054	10	600	10	50
S055	10	600	10	30
S056	12	50	10	50
S057	12	100	10	50
S058	12	200	5	50
S059	12	200	10	50
S060	12	200	10	30
S061	12	400	5	50
S062	12	400	10	50
S063	12	400	10	30
S064	12	600	5	50
S065	12	600	10	50

Table B.6 Setting of Static cores (Only displays variable settings) - Part 4

Where A is number of cores, B is length of each application in the time unit, C is numbers of iterations we implement, D is amounts of inforuction that cause stall and left index is configuration ID for static reconfigurable core.

	A	B	C	D
S066	12	600	10	30
S067	14	400	10	30
S068	14	600	5	50
S069	14	600	10	50
S070	14	600	10	30
S071	14	50	10	50
S072	14	100	10	50
S073	14	200	5	50
S074	14	200	10	50
S075	14	200	10	30
S076	14	200	10	50
S077	14	200	10	30

Table B.7 Simulation result of S001 in PITCGEN – Part 1

where PITC is Performance improvement in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, AVER is average of best performance we find during each iteration, LOG is the logic specialized cores system, MAT is the mathematical operation specialized cores system, FLO is the floating point operation specialized cores system, MEM is the memory operation specialized core system.
S001 is defined in Table B.3.

	P01	P02	P03	P04	P05	P06	P07
BEST	11.90%	9.20%	48.80%	40.45%	5.19%	10.07%	38.92%
AVER	-16.74%	-48.25%	11.50%	16.50%	-33.87%	-54.18%	-0.23%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	0.00%	9.20%	0.00%	0.00%	9.09%	17.57%	4.77%
MAT	0.00%	-18.71%	0.00%	0.00%	-18.38%	-35.14%	-10.53%
FLP	-76.09%	-130.88%	28.71%	-30.71%	-124.18%	-162.68%	-1.20%
MEM	-76.09%	-130.88%	-24.63%	12.50%	-124.18%	-162.68%	-54.43%

Table B.8 Simulation result of S001in PITCGEN – Part 2

where PITC is Performance improvement in terms of Time Compared to, GEN is the general cores only system, BEST is the “BEST” performance we find during optimization over all iteration, AVER is average of best performance we find during each iteration, LOG is the logic specialized cores system, MAT is the mathematical operation specialized cores system, FLO is the floating point operation specialized cores system, MEM is the memory operation specialized core system. S001 is defined in Table B.3.

	P08	P09	P10	P11	P12	P13	P14
BEST	34.92%	48.72%	40.02%	61.61%	49.92%	41.09%	33.90%
AVER	-5.16%	13.67%	-3.61%	26.74%	20.40%	10.26%	-3.00%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	5.56%	0.00%	4.71%	0.00%	0.00%	0.00%	5.90%
MAT	-11.05%	0.00%	-9.47%	0.00%	0.00%	0.00%	-11.68%
FLP	-60.62%	28.22%	2.99%	64.49%	31.00%	-32.08%	-63.53%
MEM	-17.09%	-24.26%	-48.99%	-6.15%	19.98%	12.08%	-21.41%
	P15	P16	P17	P18	P19	P20	P21
BEST	49.44%	57.07%	0.00%	17.78%	39.70%	28.16%	9.92%
AVER	22.68%	17.49%	-39.06%	-32.43%	6.58%	-0.44%	-35.70%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	0.00%	0.00%	9.00%	17.78%	4.74%	5.55%	17.28%
MAT	0.00%	0.00%	-18.10%	-35.75%	-9.47%	-11.03%	-34.66%
FLP	31.18%	-9.03%	-122.85%	-169.86%	-0.41%	-59.40%	-161.95%
MEM	18.58%	53.87%	-122.85%	-169.86%	-52.94%	-16.45%	-161.95%
	P22	P23	P24	P25	P26	P27	P28
BEST	14.30%	23.85%	29.54%	39.10%	23.62%	55.96%	29.17%
AVER	-76.35%	-7.95%	-7.99%	2.19%	-17.05%	11.29%	13.28%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	25.00%	9.19%	10.70%	4.65%	9.22%	3.11%	3.51%
MAT	-50.00%	-18.33%	-21.41%	-9.24%	-18.44%	-6.15%	-7.01%
FLP	-199.91%	-22.75%	-85.63%	2.35%	-23.27%	45.44%	10.67%
MEM	-199.91%	-73.37%	-43.27%	-49.31%	-73.69%	-24.67%	0.08%

Table B.9 Simulation result of S001in PITCGEN – Part 3

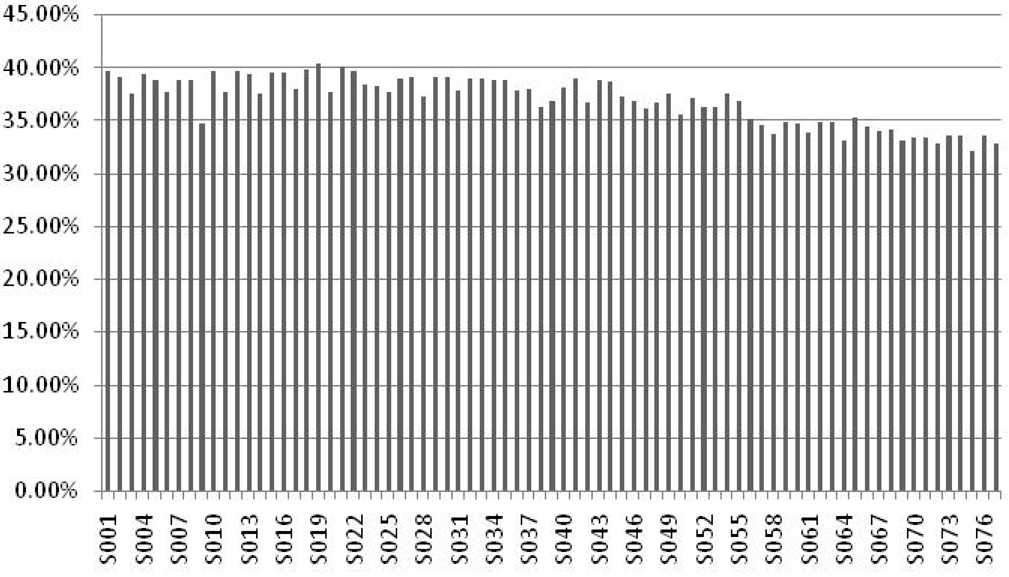
where PITC is Performance improvement in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, AVER is average of best performance we find during each iteration, LOG is the logic specialized cores system, MAT is the mathematical operation specialized cores system, FLO is the floating point operation specialized cores system, MEM is the memory operation specialized core system. S001 is defined in Table B.3.

	P29	P30	P31	P32	P33	P34	P35
BEST	33.25%	28.33%	30.38%	52.80%	48.59%	39.02%	61.78%
AVER	4.89%	-12.53%	14.76%	5.31%	17.08%	-6.47%	24.13%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	5.61%	11.03%	3.56%	3.97%	0.00%	4.86%	0.00%
MAT	-11.29%	-22.12%	-7.17%	-8.02%	0.00%	-9.77%	0.00%
FLP	-58.24%	-88.37%	11.28%	-32.01%	28.02%	-0.31%	64.29%
MEM	-17.21%	-48.40%	-2.82%	29.98%	-24.69%	-51.83%	-5.40%
	P36	P37	P38	P39	P40	P41	P42
BEST	50.37%	40.15%	34.63%	55.48%	29.09%	62.90%	45.39%
AVER	26.45%	-4.12%	-3.99%	16.60%	9.16%	26.62%	17.61%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	0.00%	4.69%	9.26%	3.21%	3.53%	0.00%	3.21%
MAT	0.00%	-9.48%	-18.47%	-6.41%	-7.03%	0.00%	-6.36%
FLP	31.22%	-0.26%	-23.47%	44.07%	10.60%	62.90%	44.28%
MEM	20.61%	-51.91%	-73.87%	-25.66%	-0.93%	-7.55%	-25.49%
	P43	P44	P45	P46	P47	P48	P49
BEST	66.65%	66.64%	49.12%	30.06%	48.67%	66.62%	39.44%
AVER	13.33%	23.50%	20.29%	7.81%	20.30%	30.30%	5.01%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	0.00%	0.00%	0.00%	3.59%	0.00%	0.00%	0.00%
MAT	0.00%	0.00%	0.00%	-7.19%	0.00%	0.00%	0.00%
FLP	79.99%	58.11%	31.15%	10.78%	58.43%	31.41%	-31.92%
MEM	0.00%	20.22%	18.48%	-2.32%	20.16%	44.43%	12.83%

Table B.10 Simulation result of S001in PITCGEN – Part 4

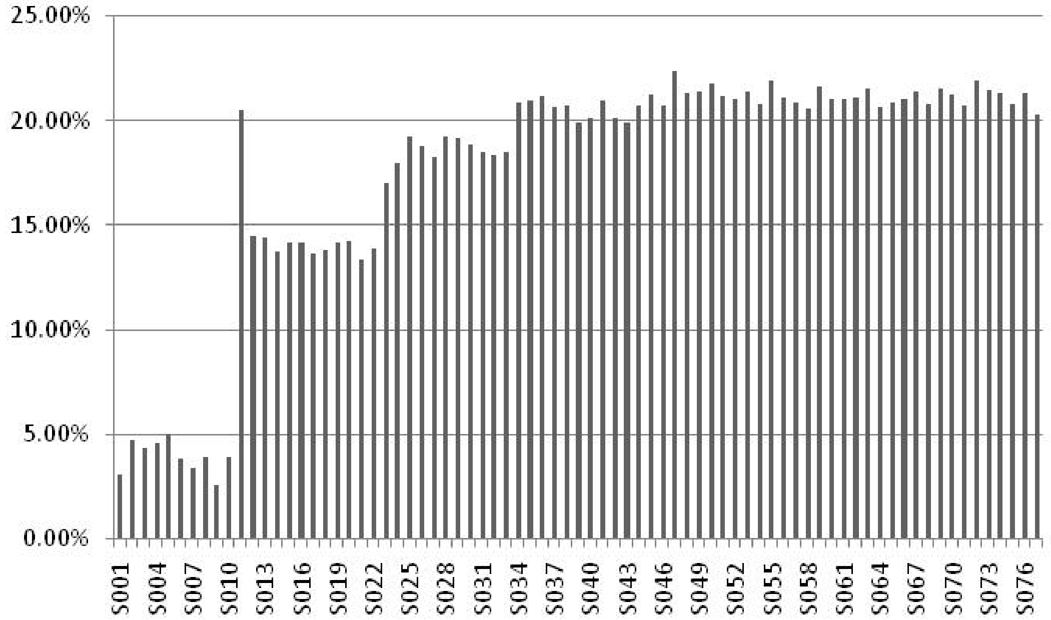
where PITC is Performance improvement in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, AVER is average of best performance we find during each iteration, LOG is the logic specialized cores system, MAT is the mathematical operation specialized cores system, FLO is the floating point operation specialized cores system, MEM is the memory operation specialized core system, and S001 is defined in Table B.3.

	P50	P51	P52	P53	P54	P55	P56
BEST	33.71%	49.37%	57.19%	32.63%	14.83%	25.96%	52.84%
AVER	-8.70%	22.67%	11.29%	-10.70%	-28.89%	8.44%	5.86%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	5.76%	0.00%	0.00%	5.63%	11.08%	3.52%	3.97%
MAT	-11.77%	0.00%	0.00%	-11.88%	-22.22%	-7.09%	-7.98%
FLP	-63.01%	31.23%	-9.94%	-63.56%	-88.80%	11.39%	-31.85%
MEM	-17.66%	18.97%	52.79%	-19.44%	-47.59%	-1.84%	31.24%
	P57	**P58**	**P59**	**P60**	**P61**	**P62**	**P63**
BEST	34.92%	27.58%	58.86%	40.08%	56.88%	47.50%	66.66%
AVER	16.51%	9.07%	23.02%	24.69%	21.21%	7.23%	23.80%
GEN	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
LOG	0.00%	3.66%	0.00%	0.00%	0.00%	4.07%	0.00%
MAT	0.00%	-7.32%	0.00%	0.00%	0.00%	-8.22%	0.00%
FLP	30.55%	10.07%	58.86%	31.86%	-10.75%	-32.79%	32.01%
MEM	20.41%	-2.28%	19.82%	45.79%	53.64%	28.94%	44.98%
	P64						
BEST	73.23%						
AVER	23.09%						
GEN	0.00%						
LOG	0.00%						
MAT	0.00%						
FLP	0.00%						
MEM	73.23%						



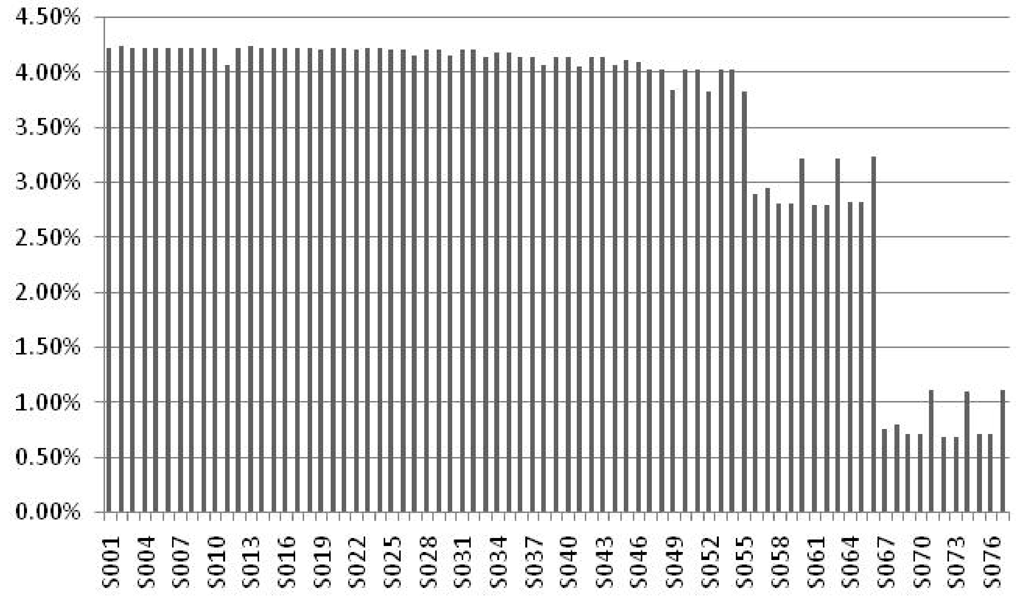
FigureB.1 APITCGEN for BEST for each setting

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only system, and BEST is the "BEST" performance we find during optimization over all iteration, system. S001 through S077 are defined in Table B.3 through Table B.6.



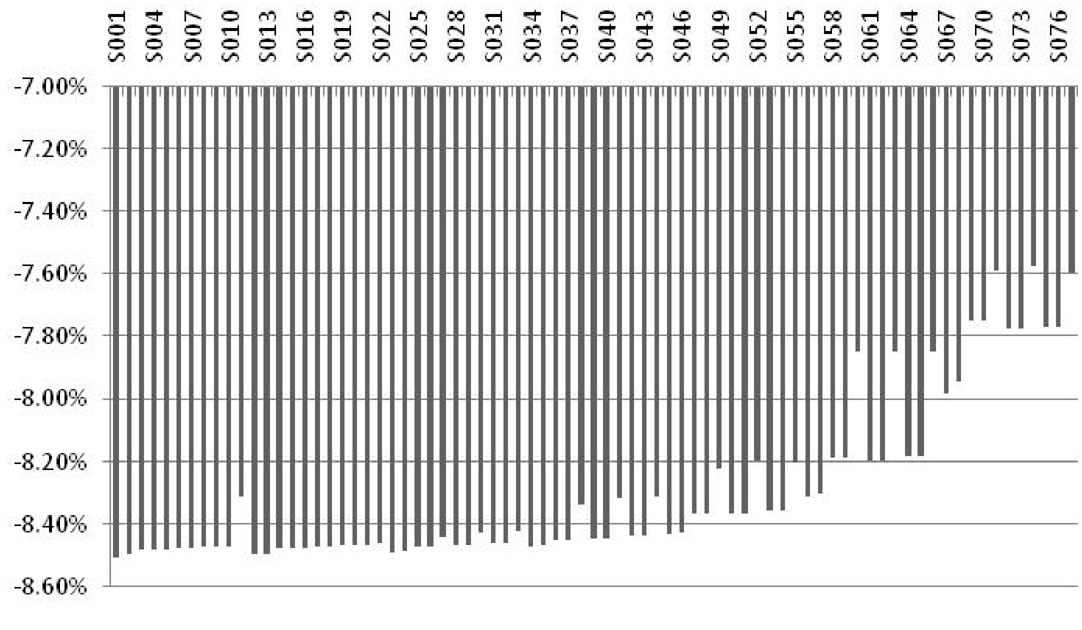
FigureB.2 APITCGEN for AVER for each setting

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only system, and AVER is average of best performance we find during each iteration. S001 through S077 are defined in Table B.3 through Table B.6.



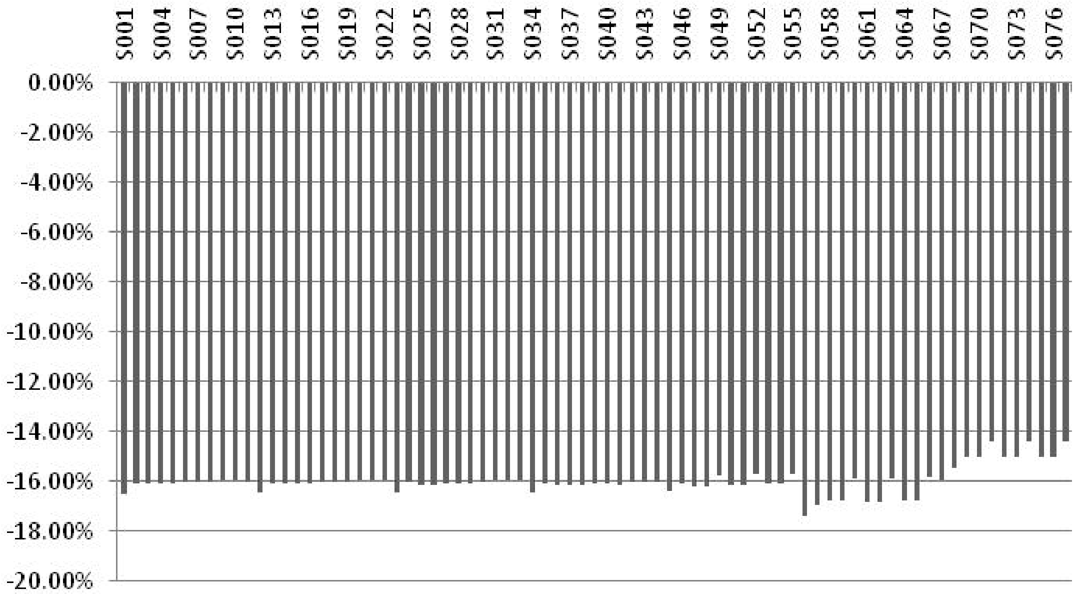
FigureB.3 APITCGEN for LOG for each setting

where APITC is average Performance improvement in terms of Time Compared to, GEN is the general cores only system, and LOG is the logic specialized cores system
S001 through S077 are defined in Table B.3 through Table B.6..



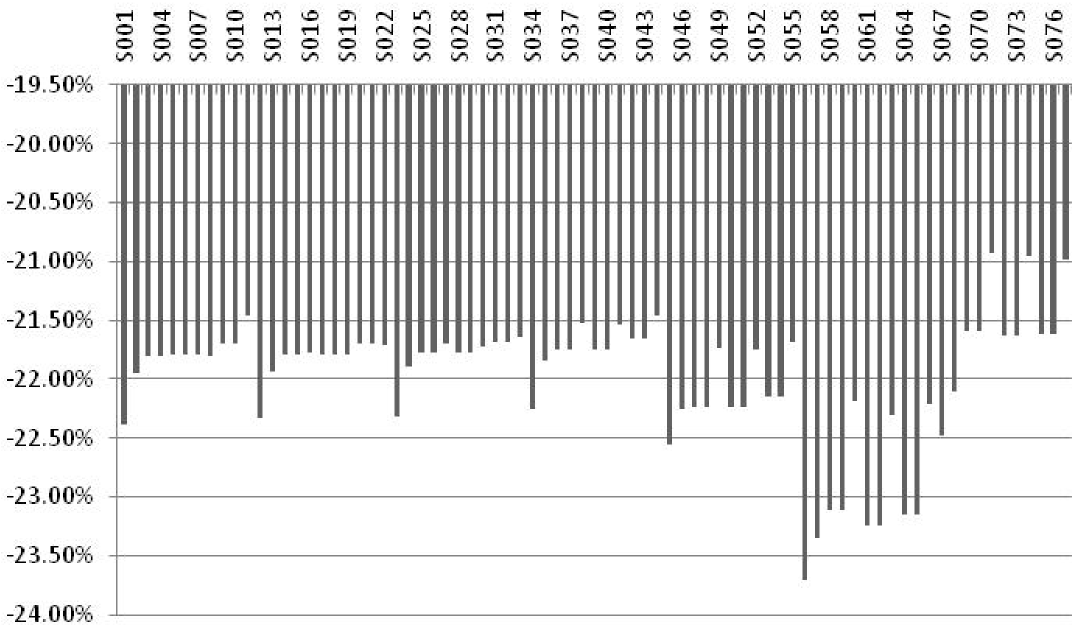
FigureB.4 APITCGEN for MAT for each setting

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only system, and MAT is the mathematical operation specialized cores system S001 through S077 are defined in
Table B.3 through Table B.6.



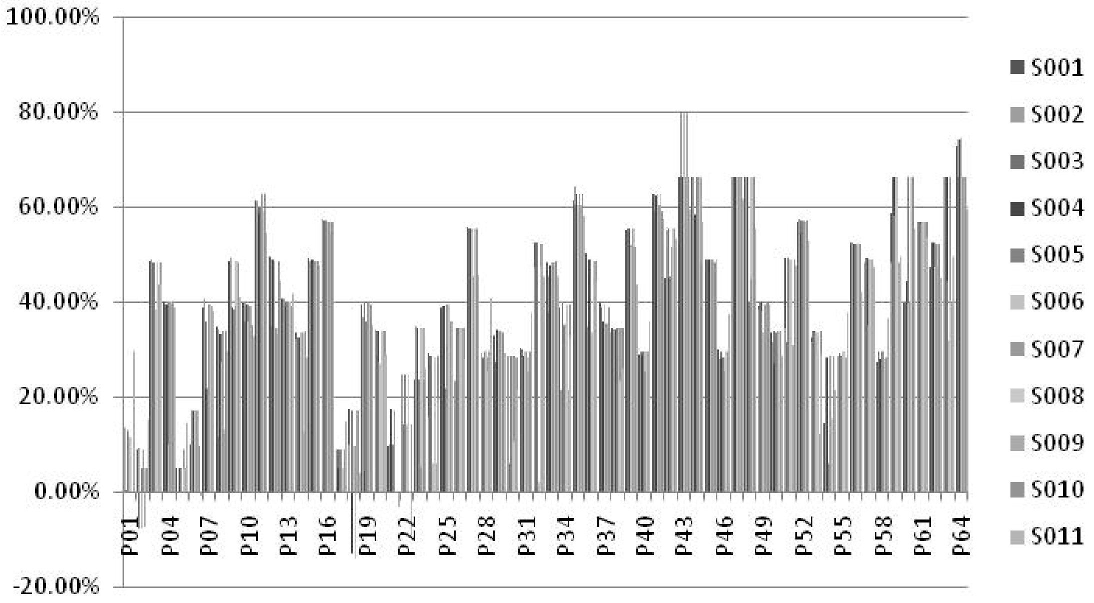
FigureB.5 APITCGEN for FLP for each setting

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only and FLP is the floating point operation specialized cores system, S001 through S077 are defined in Table B.3 through Table B.6.



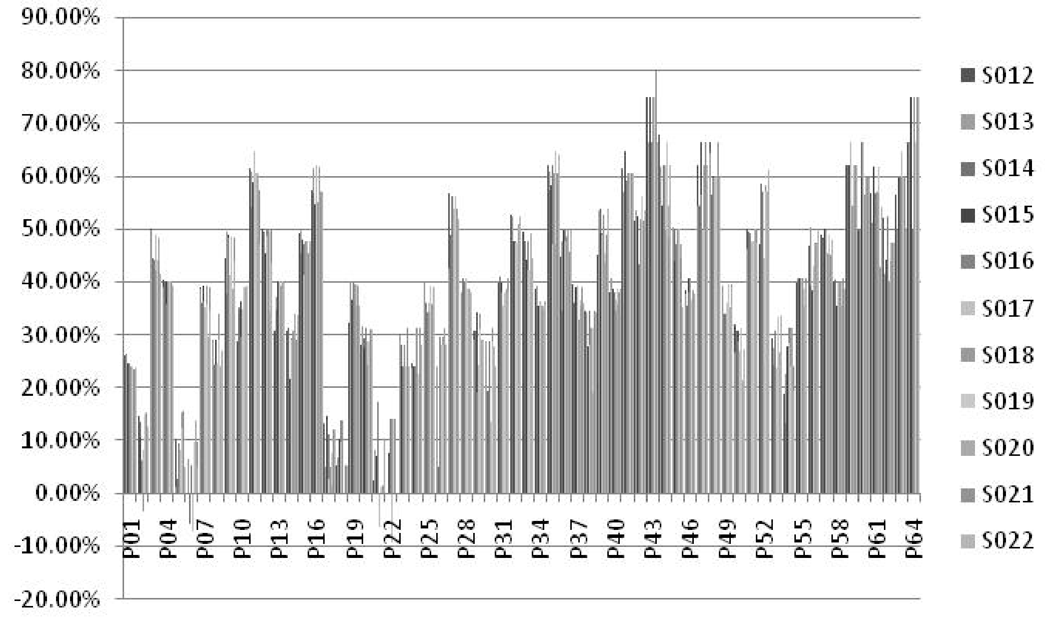
FigureB.6 APITCGEN for MEM for each setting

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only system, and MEM is the memory operation specialized core system. S001 through S077 are defined in Table B.3 through Table B.6.



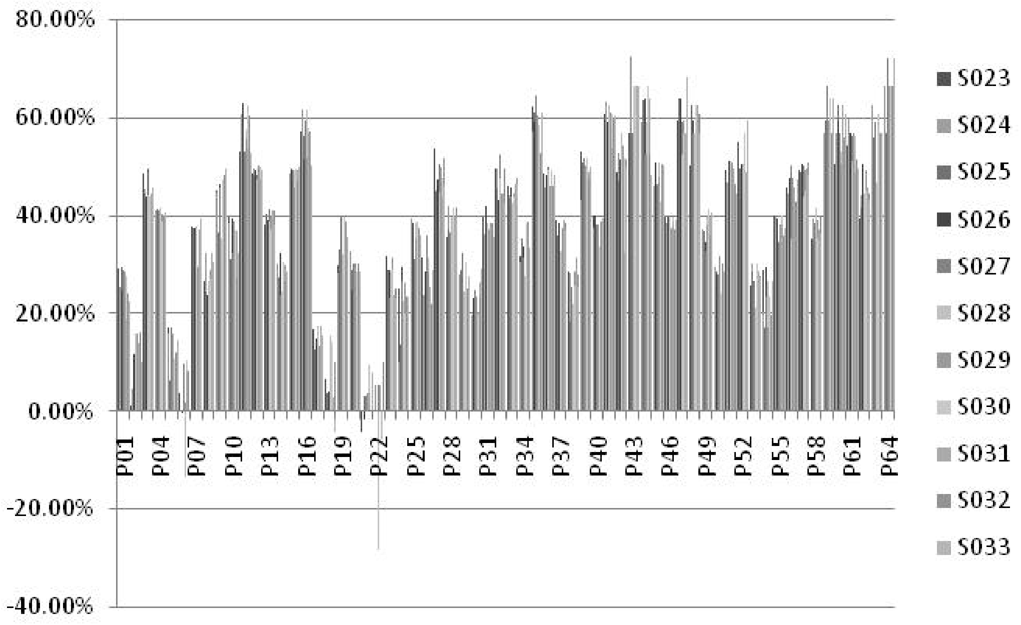
FigureB.7 PITCGEN for BEST with among 2 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



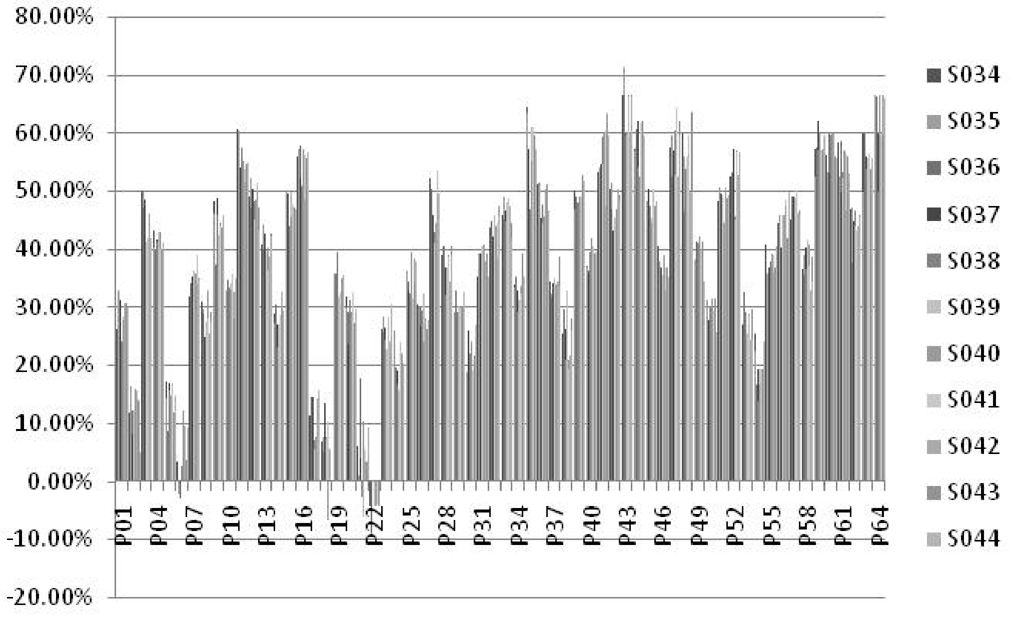
FigureB.8 PITCGEN for BEST among 4 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



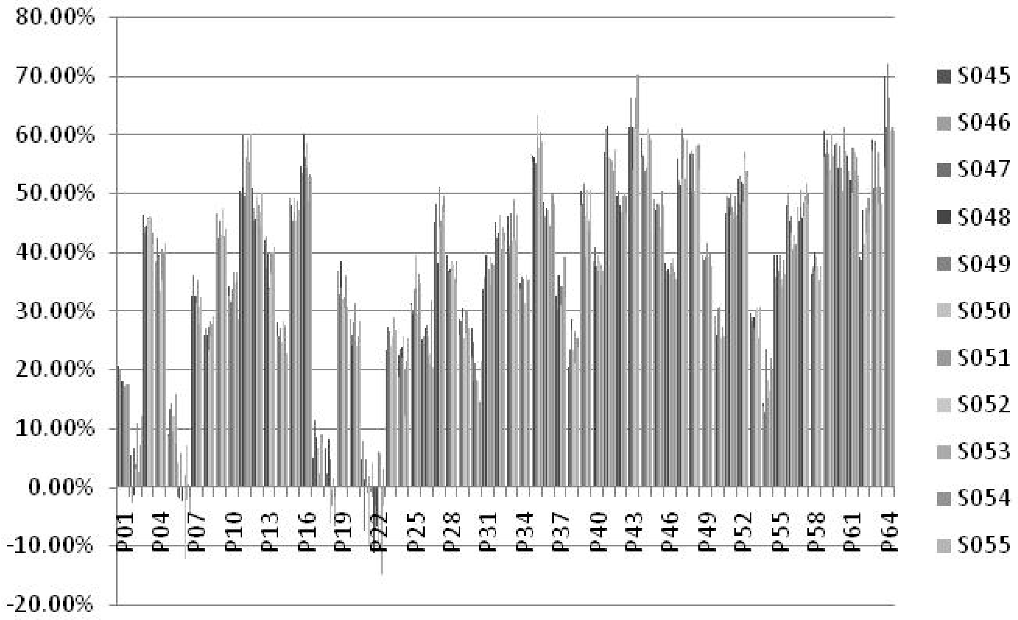
FigureB.9 PITCGEN for BEST among 6 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



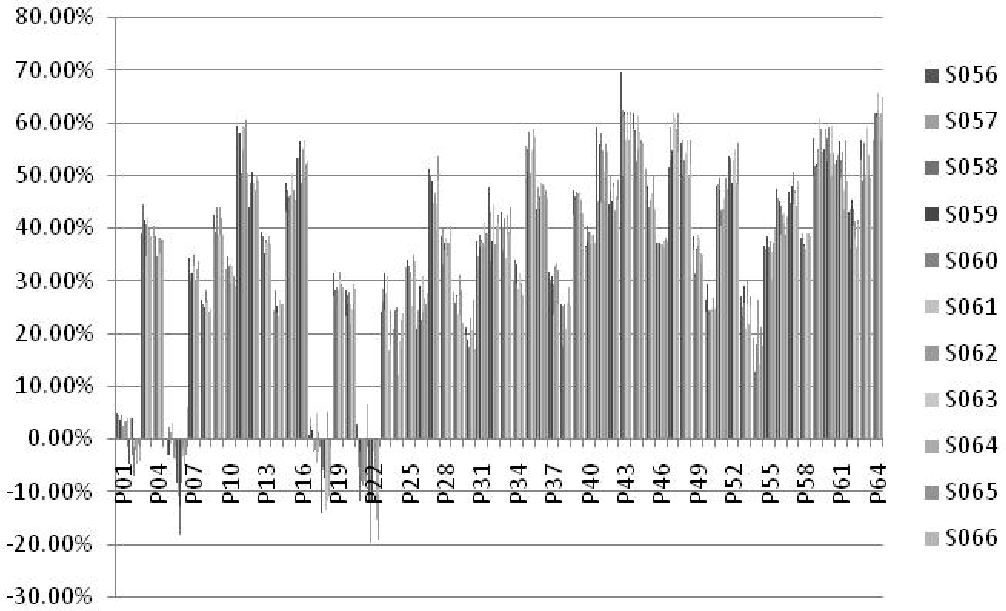
FigureB.10 PITCGEN for BEST among 8 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



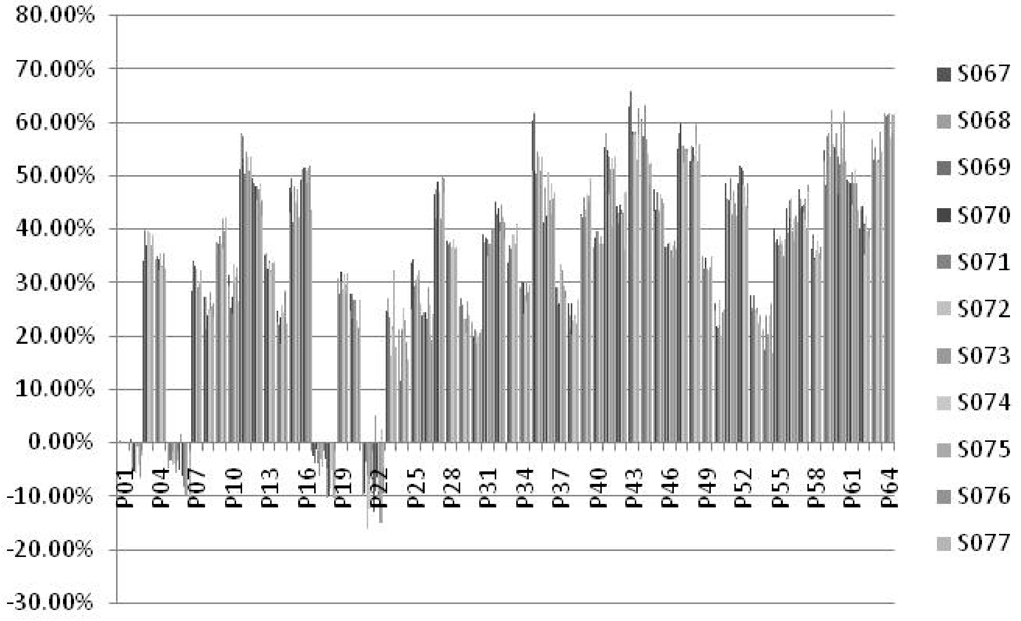
FigureB.11 PITCGEN for BEST among 10 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



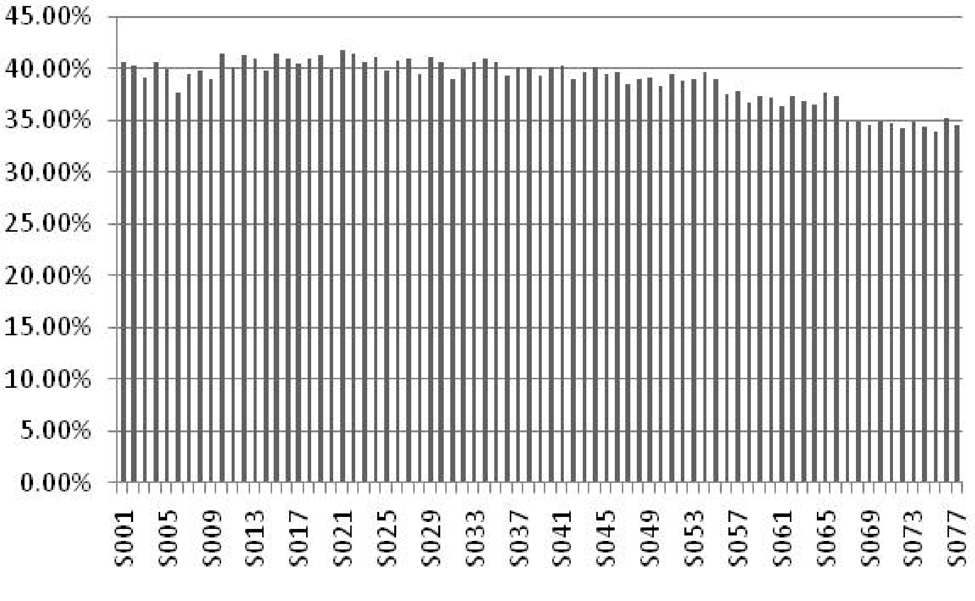
FigureB.12 PITCGEN for BEST among 12 cores systems

Where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



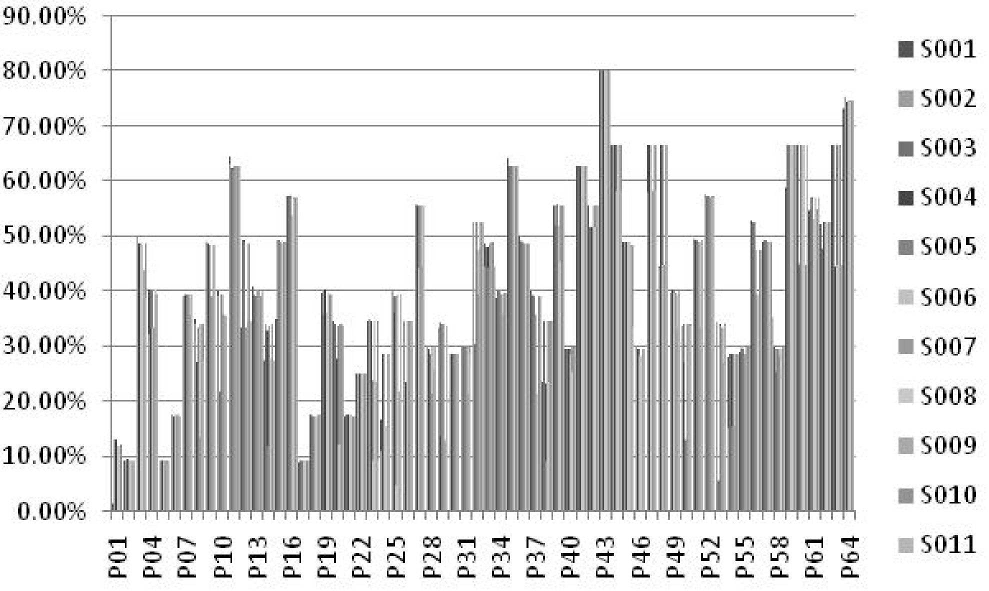
FigureB.13 PITCGEN for BEST among 14 cores systems

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



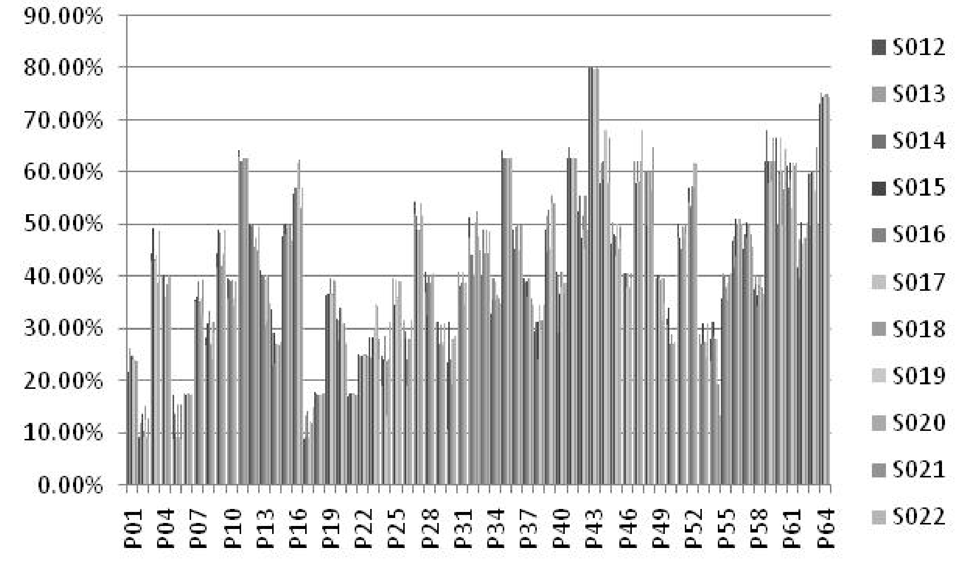
FigureB.14 APITCGEN for BEST for each setting with modified algorithm

where APITC is average Performance improvement over all application sequences in terms of Time Compared to, GEN is the general cores only system, and BEST is the "BEST" performance we find during optimization over all iteration, system. S001 through S077 are defined in Table B.3 through Table B.6.



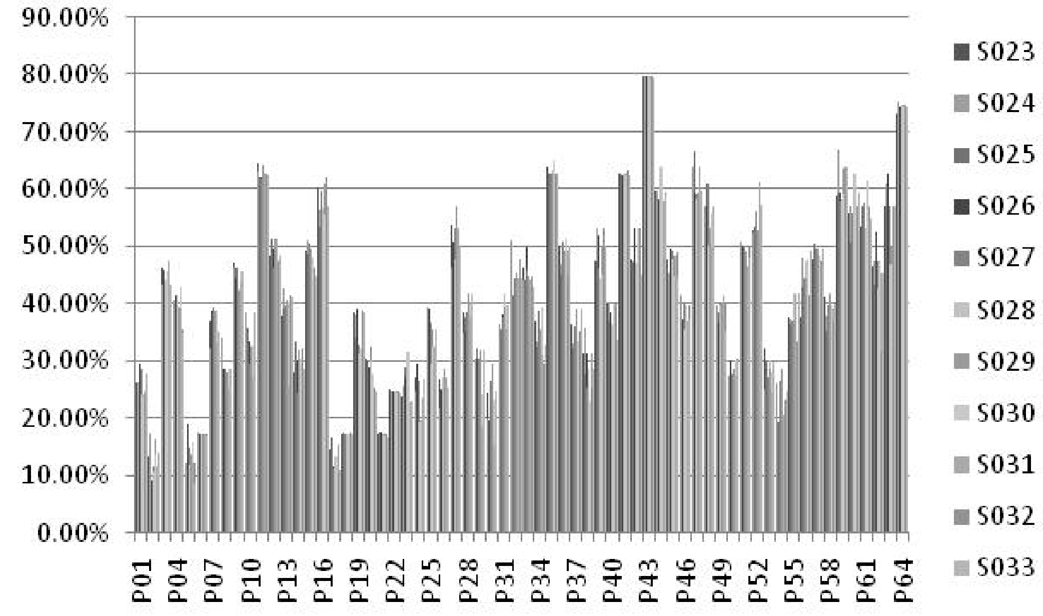
FigureB.15 PITCGEN for BEST among 2 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



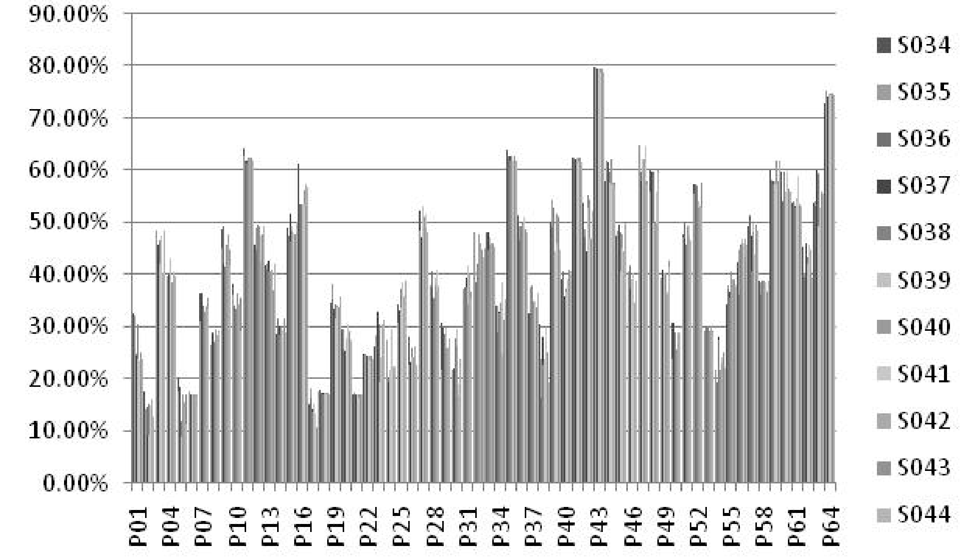
FigureB.16 PITCGEN for BEST among 4 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



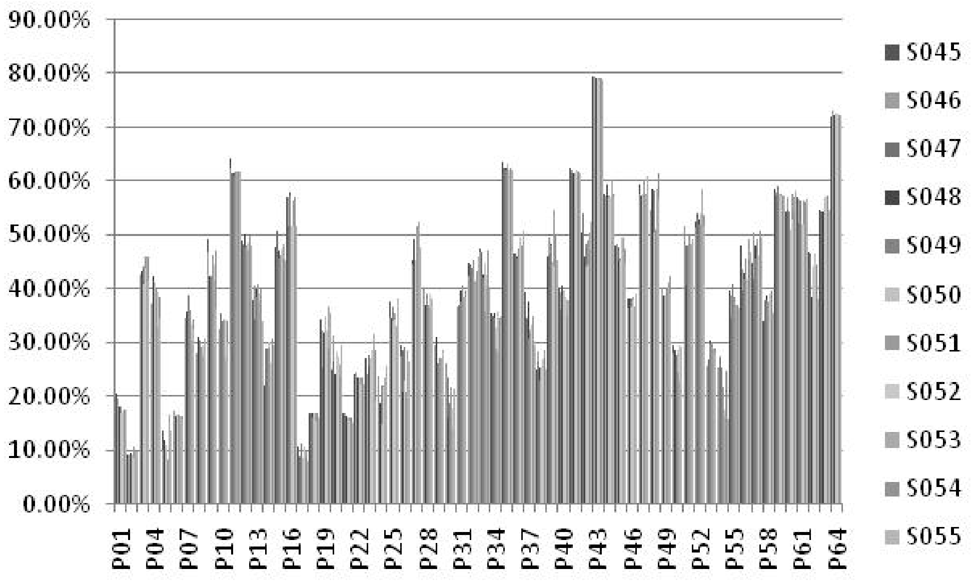
FigureB.17 PITCGEN for BEST among 6 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



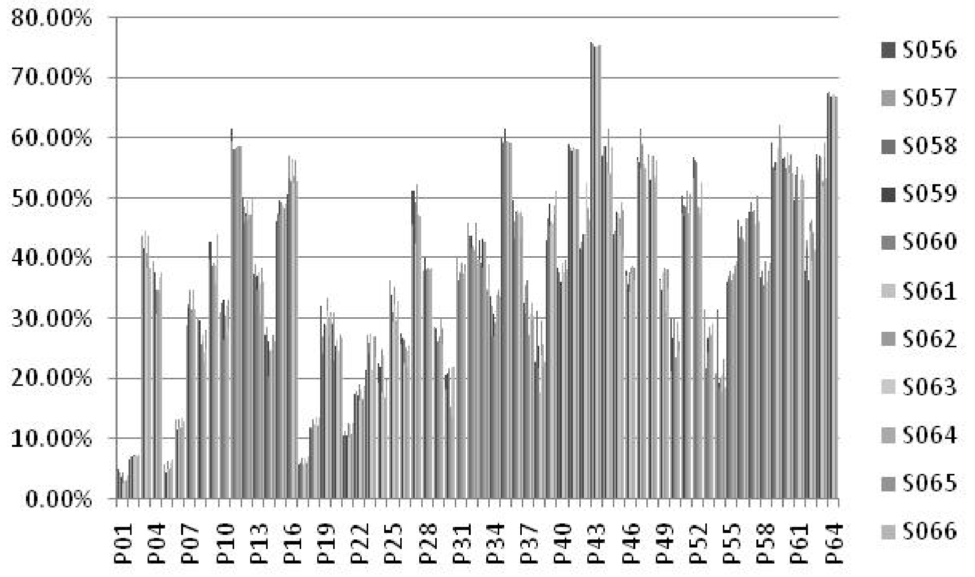
FigureB.18 PITCGEN for BEST among 8 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



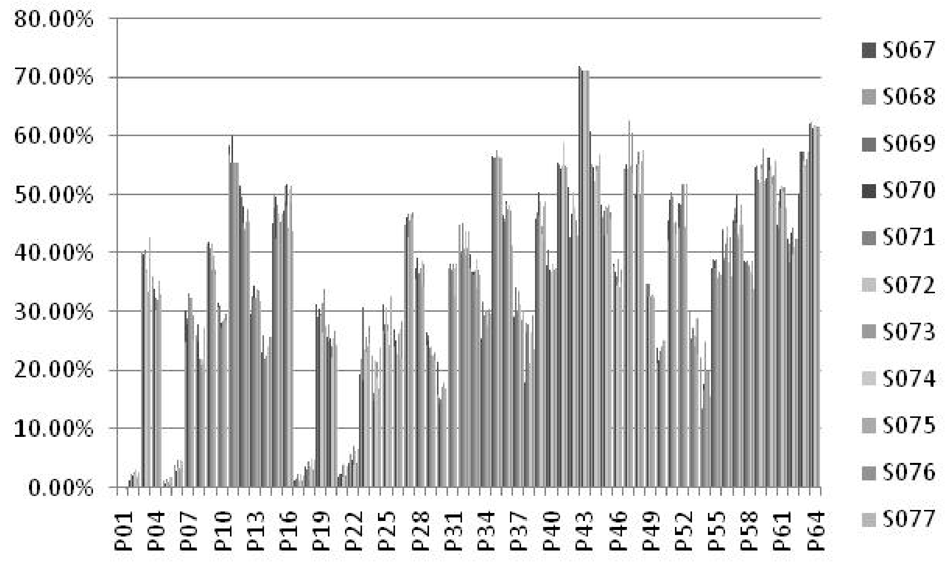
FigureB.19 PITCGEN for BEST among 10 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



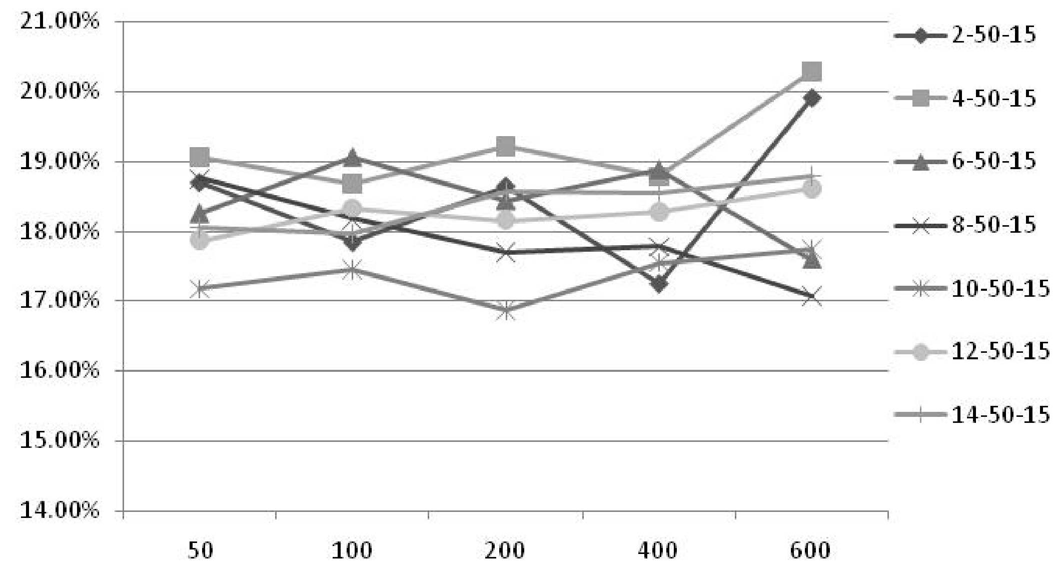
FigureB.20 PITCGEN for BEST among 12 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



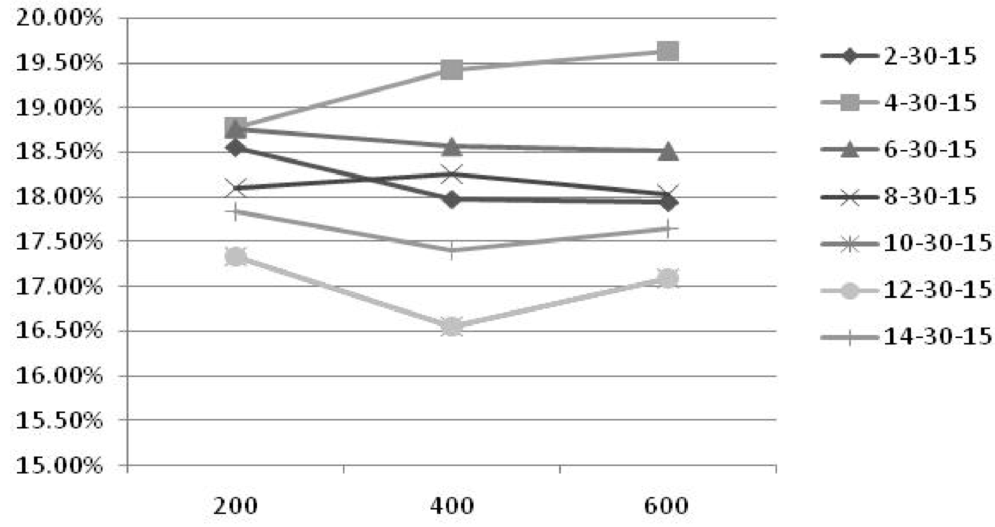
FigureB.21 PITCGEN for BEST among 14 cores systems with modified algorithm

where PITC is Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iteration, P01 through P64 is defined in Table B.1 and Table B.2. .
S001 through S077 is defined in Table B3 through B6.



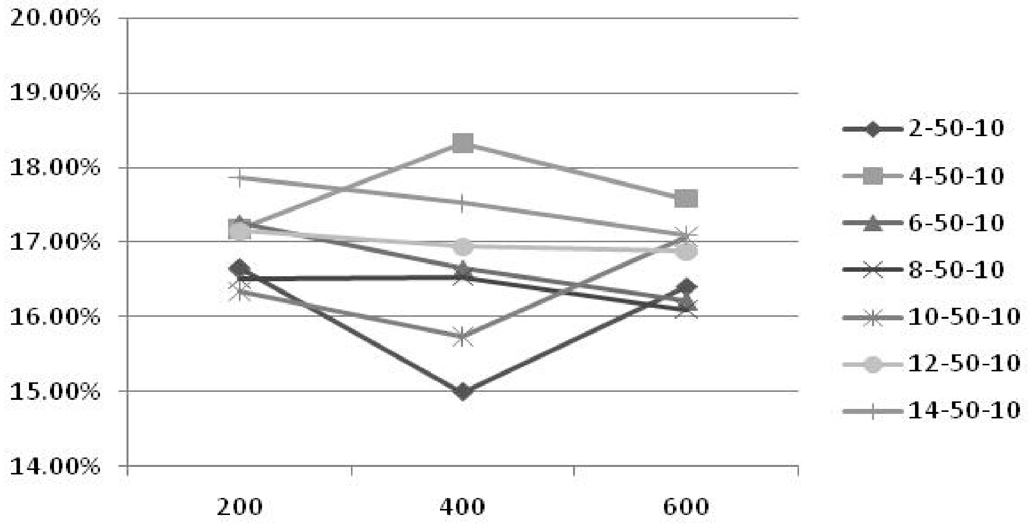
FigureB.22 APITCGEN for BEST with different simulation cycle

where APITC is Average Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iterations. Actual cycles of simulation is more than three times of numbers labeled on the axis. Legend displays numbers of core: amounts of inforuction for stall trigger, and iterations of the algorithm, respectively.



FigureB.23 APITCGEN for BEST with different simulation cycle

where APITC is Average Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iterations. Actual cycles of simulation is more than three times of numbers labeled on the axis. Legend displays numbers of core: amounts of inforuction for stall trigger, and iterations of the algorithm, respectively.



FigureB.24 APITCGEN for BEST with different simulation cycle

where APITC is Average Performance improvement for each application sequences in terms of Time Compared to, GEN is the general cores only system, BEST is the "BEST" performance we find during optimization over all iterations. Actual cycles of simulation is more than three times of numbers labeled on the axis. Legend displays numbers of core: amounts of inforuction for stall trigger, and iterations of the algorithm, respectively.

Table B.11 Setting of Dynamic systems (Only displays variable settings) – Part 1

where A is number of cores, B is length of delays penalty, C is length of each application in the time unit, D is amounts of inforuction that cause stall and left column represent configuration ID for dynamic system.

	A	B	D	E
D001	2	6	400	50
D002	2	8	50	50
D003	2	8	100	50
D004	2	8	200	50
D005	2	8	400	50
D006	2	8	400	30
D007	2	10	50	50
D008	2	10	100	50
D009	2	10	200	50
D010	2	10	400	50
D011	4	6	100	50
D012	4	6	200	50
D013	4	6	400	50
D014	4	6	400	30
D015	4	8	50	50
D016	4	8	100	50
D017	4	8	200	50
D018	4	8	400	50
D019	4	8	400	30
D020	4	10	50	50
D021	4	10	100	50
D022	4	10	200	50
D023	4	10	400	50
D024	6	6	50	50
D025	6	6	100	50
D026	6	6	200	50

Table B.12 Setting of Dynamic systems (Only displays variable settings) – Part 2

where A is number of cores, B is length of delays penalty, C is length of each application in the time unit, D is amounts of inforuction that cause stall and left column represent configuration ID for dynamic system.

	A	B	C	D
D027	6	6	400	50
D028	6	6	400	30
D029	6	8	50	50
D030	6	8	100	50
D031	6	8	400	50
D032	6	8	400	30
D033	6	8	600	50
D034	6	10	50	50
D035	6	10	100	50
D036	6	10	200	50
D037	6	10	400	50
D038	6	10	600	50
D039	8	6	50	50
D040	8	6	100	50
D041	8	6	200	50
D042	8	6	400	50
D043	8	6	400	30
D044	8	6	600	50
D045	8	8	400	30
D046	8	10	50	50
D047	8	10	100	50
D048	8	10	200	50
D049	8	10	400	50
D050	8	10	400	30
D051	10	6	50	50
D052	10	6	100	50

Table B.13 Setting of Dynamic systems (Only displays variable settings) – Part 3

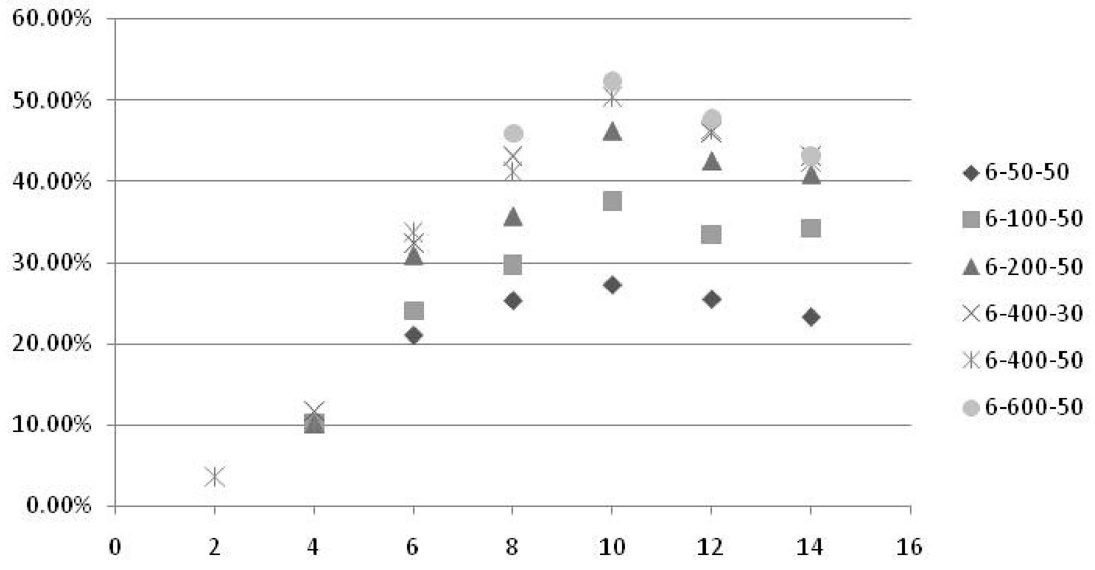
where A is number of cores, B is length of delays penalty, C is length of each application in the time unit, D is amounts of inforuction that cause stall and left column represent configuration ID for dynamic system.

	A	B	C	D
D053	10	6	200	50
D054	10	6	400	50
D055	10	6	600	50
D056	10	8	50	50
D057	10	8	100	50
D058	10	8	200	50
D059	10	8	400	50
D060	10	8	400	30
D061	10	8	600	50
D062	10	10	50	50
D063	10	10	100	50
D064	10	10	200	50
D065	10	10	400	50
D066	10	10	400	30
D067	10	10	600	50
D068	12	6	50	50
D069	12	6	100	50
D070	12	6	200	50
D071	12	6	400	50
D072	12	6	400	30
D073	12	6	600	50
D074	12	8	50	50
D075	12	8	100	50
D076	12	8	200	50
D077	12	8	400	50
D078	12	8	400	30

Table B.14 Setting of Dynamic systems (Only displays variable settings) – Part 4

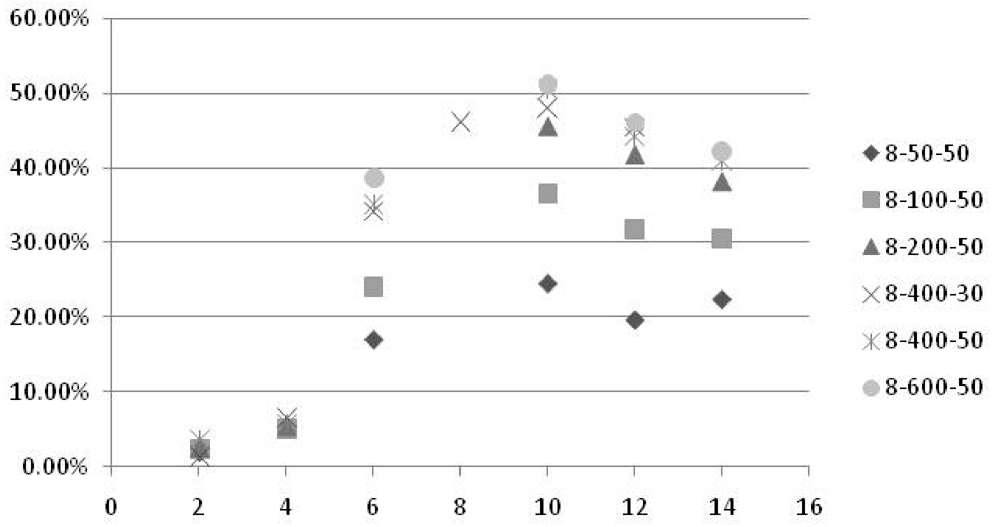
where A is number of cores, B is length of delays penalty, C is length of each application in the time unit, D is amounts of inforuction that cause stall and left column represent configuration ID for dynamic system.

	A	B	C	D
D079	12	8	600	50
D080	12	10	50	50
D081	12	10	100	50
D082	12	10	200	50
D083	12	10	400	50
D084	12	10	400	30
D085	12	10	600	50
D086	14	6	50	50
D087	14	6	100	50
D088	14	6	200	50
D089	14	6	400	50
D090	14	6	400	30
D091	14	6	600	50
D092	14	8	50	50
D093	14	8	100	50
D094	14	8	400	50
D095	14	8	200	50
D096	14	8	600	50
D097	14	10	50	50
D098	14	10	100	50
D099	14	10	200	50
D100	14	10	400	50
D101	14	10	400	30
D102	14	10	600	50



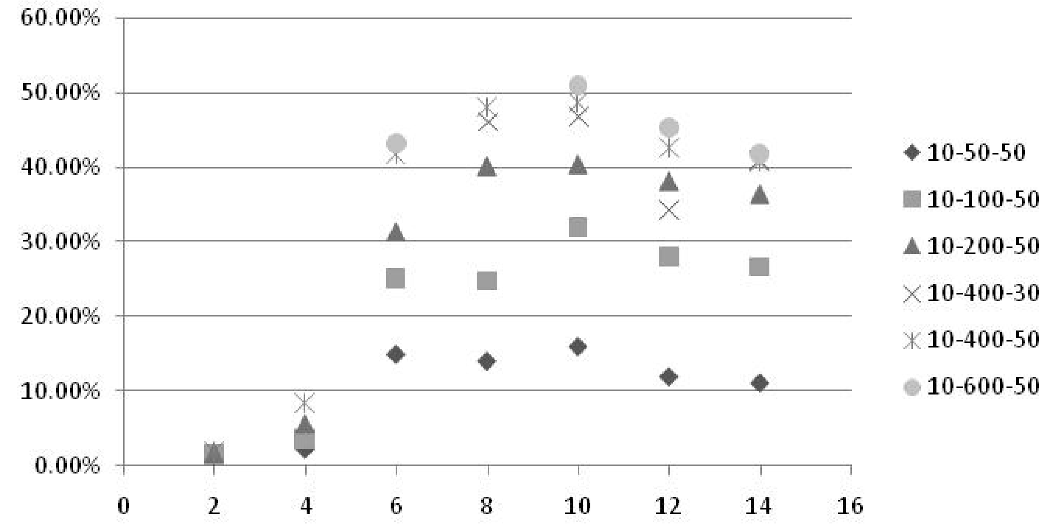
FigureB.25 APITCGEN for dynamic system with 6 cycle delay

Legend explains the setting of each result of simulation with the following format:
Number of delay cycles: Cycles of each application: Stall trigger levels



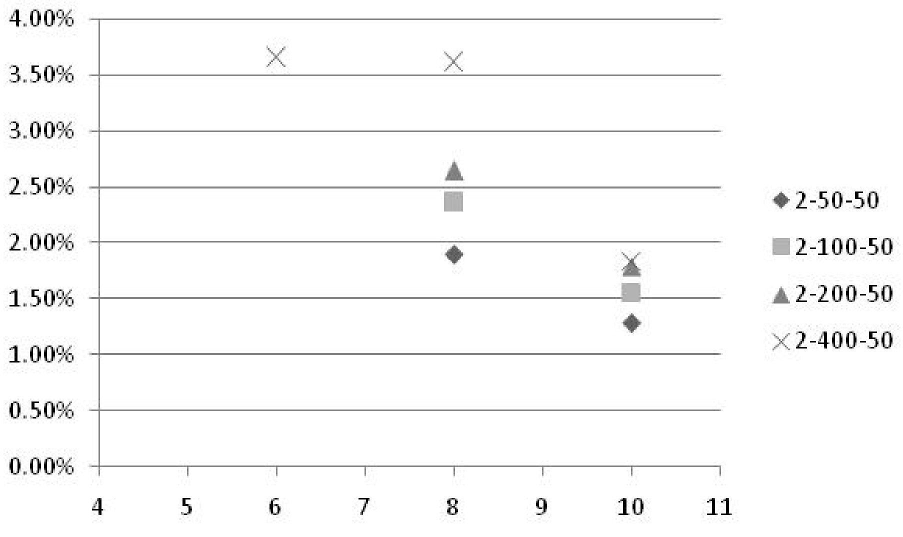
FigureB.26 APITCGEN for dynamic system with 8 cycle delay

Legend explains the setting of each result of simulation with the following format:
Number of delay cycles: Cycles of each application: Stall trigger levels



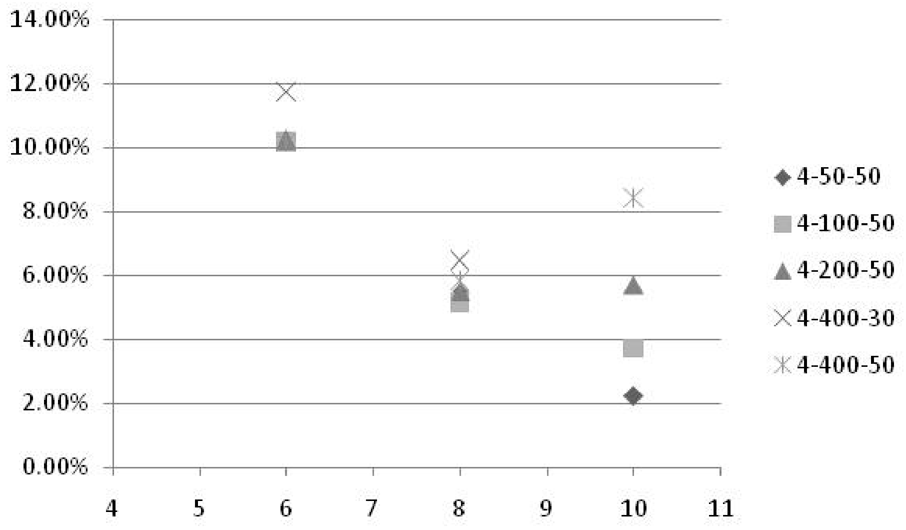
FigureB.27 APITCGEN for dynamic system with 10 cycle delay

Legend explains the setting of each result of simulation with the following format:
Number of delay cycles: Cycles of each application: Stall trigger levels



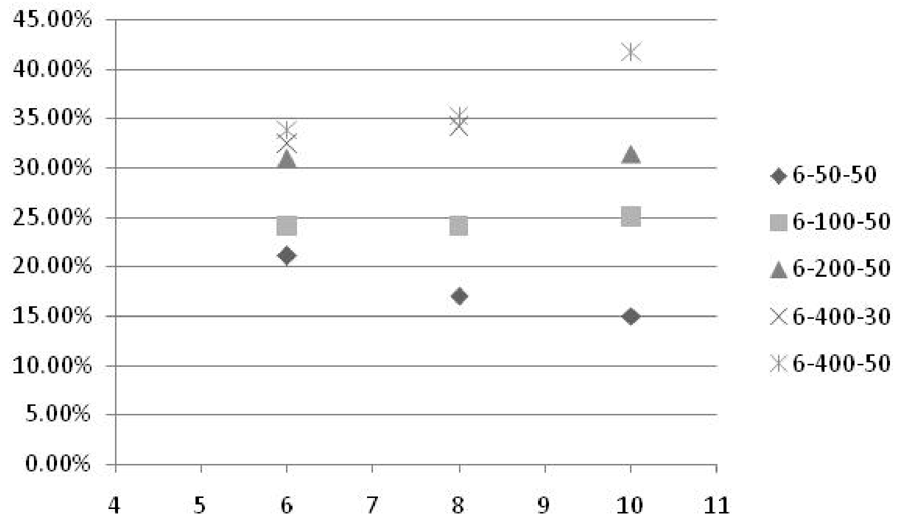
FigureB.28 APITCGEN with dynamic systems (Changing the delay cycles fixed for 2 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



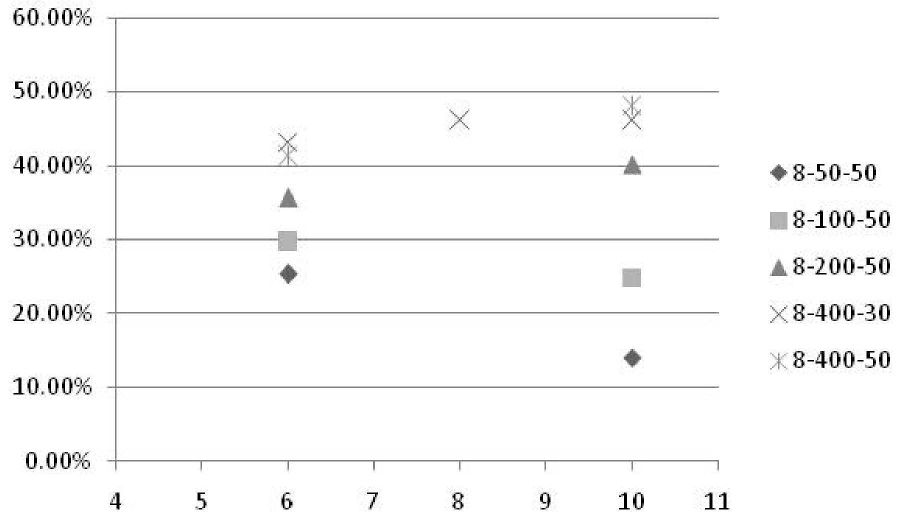
FigureB.29 APITCGEN with dynamic systems (Changing the delay cycles fixed for 4 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



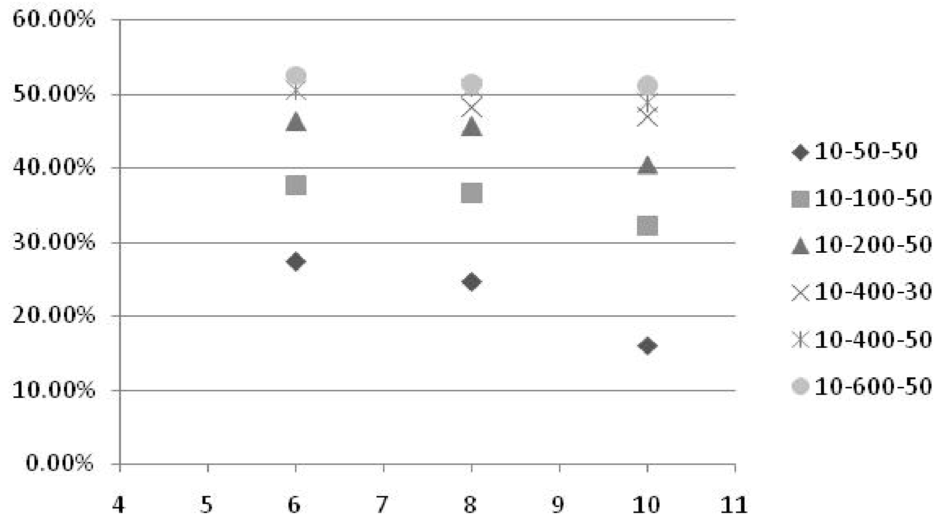
FigureB.30 APITCGEN with dynamic systems (Changing the delay cycles fixed for 6 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



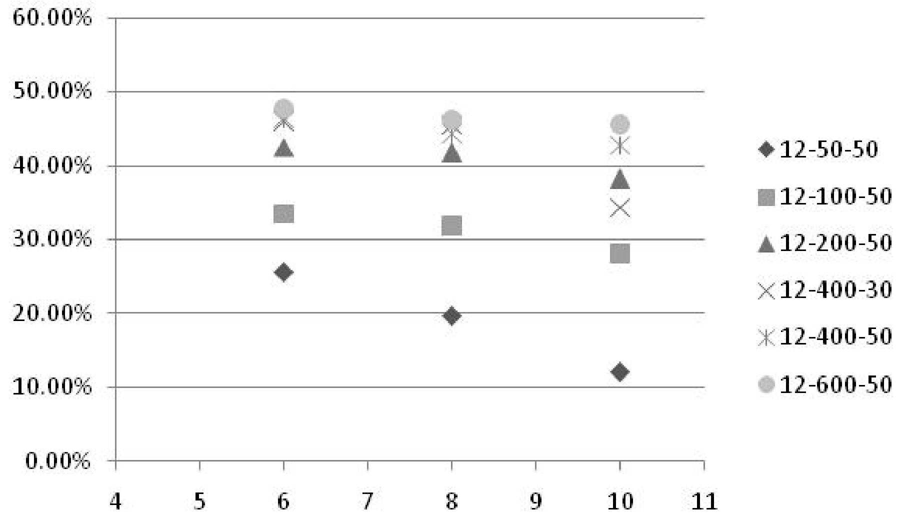
FigureB.31 APITCGEN with dynamic systems (Changing the delay cycles fixed for 8 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



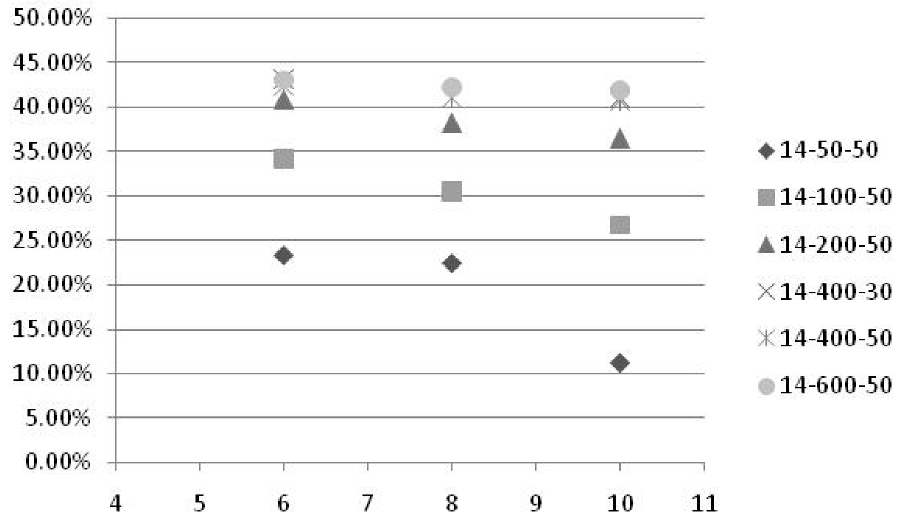
FigureB.32 APITCGEN with dynamic systems (Changing the delay cycles fixed for 10 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



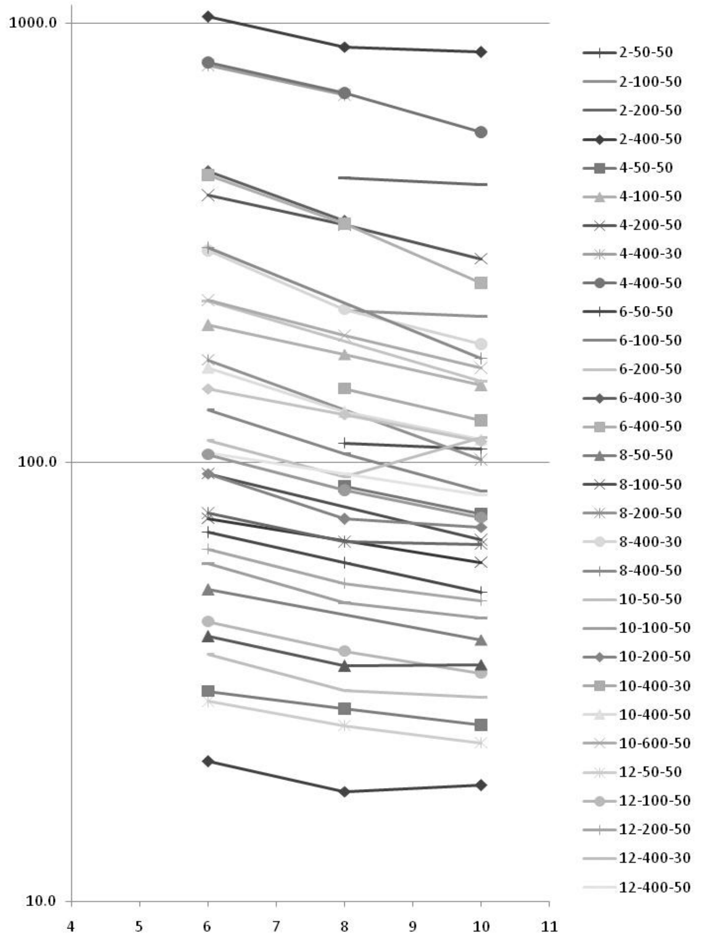
FigureB.33 APITCGEN with dynamic systems (Changing the delay cycles fixed for 12 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



FigureB.34 APITCGEN with dynamic systems (Changing the delay cycles fixed for 14 cores)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels



FigureB.35 Average number of reconfiguration (Changing the delay cycles for all cores case)

Legend explains the setting of each result of simulation with the following format:
Number of cores: Cycles of each application: Stall trigger levels

VITA

Kazunori Nishimura

Candidate for the Degree of

Master of Science

Thesis: A DYNAMIC RECONFIGURABLE COMPUTER WITH A DYNAMIC GENETIC ALGORITHM

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Fukuoka, Fukuoka, on September 30, 1980, the son of Kazuo and Reiko Nishimura.

Education: Graduated from Chikushigaoka High School, Fukuoka City, Fukuoka in March 1999; received Associate of Arts degree in General Study from Central Christian College of Kansas in May 2002, and received Bachelor of Science degree in Electrical Engineering from Oklahoma State University in May 2006. Completed the requirements for the Master of Science degree with a major in Electrical Engineering at Oklahoma State University in July, 2008.

Experience: Employed by Central Christian College of Kansas as a computer lab assistant from August 2001 to May 2002. Employed by Oklahoma State University, Department of Electrical and Computer Engineering as a teaching assistant, 2007 and as a graduate research assistant, 2007; Oklahoma State University, Department of Electrical and Computer Engineering, 2002 to present.

Professional Memberships:
IEEE member, Eta Kappa Nu Honor society, Phi Kappa Phi honor society

Name: Kazunori Nishimura Date of Degree: July, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A DYNAMIC RECONFIGURABLE COMPUTER WITH A DYNAMIC GENETIC ALGORITHM

Pages in Study: 122 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

The performance of the human brain is incredible and motivated us to create a system which behaves similarly. The flexibility of performance is the main function we emulate. We will create a system model of heterogeneous dynamic multi-core reconfigurable computers with a genetic algorithm. We introduce several concepts to find the next configuration with the genetic algorithm. Before we create our model, we introduce concepts of heterogeneous static multi-core reconfigurable computers to verify the optimization capability of the genetic algorithm. From the simulation results, we conclude that the genetic algorithm can optimize the configuration candidate of static systems. Then, the dynamic reconfigurable systems are generated, simulated, and observed. We conclude that there might exist potential performance improvement of our systems compared to homogeneous multi-core systems. Our evaluation model is fully parameterized, and will be available to the research community. We suggest future applications and improvements for static and dynamic systems.

ADVISOR'S APPROVAL: Dr. Sohum Sohoni