Studies on Automatic Parallelization and Power Reduction of a Multiplayer Game

July 2013

Yasir ALDOSARY
Studies on Automatic Parallelization and Power Reduction of a Multiplayer Game

July 2013

Waseda University
Graduate School of Fundamental Science and Engineering
Major in Computer Science and Engineering
Research on Advanced Computing Systems

Yasir ALDOSARY
Abstract

Video Games have been a very popular form of digital entertainment in recent years. They have been delivered in state of the art technologies that include multi-core processors that are known to be the leading contributor in enhancing the performance of computer applications. Since parallel programming is a difficult technology to implement, that field in Video Games is still rich with areas for advancements. This paper investigates performance enhancement and power reduction of Video Games when using parallelizing compilers and the difficulties involved in achieving that. This experiment is conducted in several stages to attempt parallelizing a well-renowned sequentially written Video Game called ioquake3. First, the Game is profiled for discovering bottlenecks, then examined by hand on how much parallelism could be extracted from those bottlenecks, and what sort of hazards exist in delivering a parallel-friendly version of ioquake3. Then, the Game code is rewritten into a hazard-free version while also modified to comply with the Parallelizable-C rules, which crucially aid parallelizing compilers in extracting parallelism. Next, the program is compiled using a parallelizing compiler called OSCAR (Optimally Scheduled Advanced Multiprocessor) to produce a parallel and then a power reduced version of ioquake3. Finally, the performance of the newly produced versions of ioquake3 on a Multi-core platform is analyzed.

The following is found: (1) the parallelized game by the compiler from the revised
sequential program of the game is found to achieve a better performance than the original one on various machines, (2) the low power version of ioquake3 consumes at 27% less power than the original, and (3) issues opposing parallelism are mainly hazards caused by thread contentions over globally shared data and as well as thread private data, and combinations of interconnected local and global variables that compose a singular module that is hazardous as a whole and must be treated as a whole, and function pointers, and (4) AI driven players are represented very similarly to human players inside ioquake3 engine, which gives an estimation of the costs for parallelizing human driven sessions, and (5) 70% of the costs of the experiment is spent in analyzing ioquake3 code, 30% in implementing the changes in the code.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE OF CONTENTS</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>1.1 Background and Purpose</td>
<td>15</td>
</tr>
<tr>
<td>1.1.1 Sequentialism</td>
<td>16</td>
</tr>
<tr>
<td>1.1.2 Parallelism</td>
<td>17</td>
</tr>
<tr>
<td>1.2 Video Games</td>
<td>18</td>
</tr>
<tr>
<td>1.2.1 What is a “Video Game”?</td>
<td>18</td>
</tr>
<tr>
<td>1.2.2 Compact and the <em>Waterfall</em></td>
<td>19</td>
</tr>
<tr>
<td>1.2.3 Why approach Video Games for Research?</td>
<td>20</td>
</tr>
<tr>
<td>1.2.4 Inherit Sequentialism</td>
<td>21</td>
</tr>
<tr>
<td>1.2.5 Why Choose Multiplayer Games over Singleplayer Games?</td>
<td>23</td>
</tr>
<tr>
<td>1.2.6 Multiplayer Games from a Parallel Point of View</td>
<td>24</td>
</tr>
<tr>
<td>1.2.7 Relationship with the Research</td>
<td>25</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>26</td>
</tr>
<tr>
<td>Chapter 2 OSCAR Compiler and API</td>
<td>32</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>32</td>
</tr>
<tr>
<td>2.2 Parallelizing Compilers</td>
<td>33</td>
</tr>
<tr>
<td>2.3 The OSCAR Compiler</td>
<td>34</td>
</tr>
<tr>
<td>2.4 How is parallelism extracted by OSCAR?</td>
<td>34</td>
</tr>
<tr>
<td>2.5 Human Intervention Necessity</td>
<td>36</td>
</tr>
<tr>
<td>2.6 Parallelizable C</td>
<td>38</td>
</tr>
<tr>
<td>2.7 Power Reduction</td>
<td>38</td>
</tr>
<tr>
<td>2.7.1 Minimum Time Execution</td>
<td>40</td>
</tr>
<tr>
<td>2.7.2 Satisfaction of Real-Time Deadline</td>
<td>40</td>
</tr>
<tr>
<td>2.8 The OSCAR API</td>
<td>40</td>
</tr>
<tr>
<td>2.9 Summary</td>
<td>42</td>
</tr>
<tr>
<td>Chapter 3 ioquake3 Structure Analysis</td>
<td>46</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>46</td>
</tr>
<tr>
<td>3.2 What are FPS Video Games?</td>
<td>47</td>
</tr>
</tbody>
</table>
3.3 Quake, the Benchmark ..................................................................................................49
3.4 The Bot Saga ...............................................................................................................51
3.5 What are Mods? ..........................................................................................................52
3.6 Analytical Steps .........................................................................................................54
3.7 Profiling ....................................................................................................................56
3.8 Critical Path General Overview ...............................................................................57
3.9 The Bottlenecks ...........................................................................................................59
  3.9.1 BotAI(): Overview .................................................................................................59
  3.9.2 BotAI(): Structure ...............................................................................................60
  3.9.3 BotAI(): Function Characteristics .......................................................................62
  3.9.4 ClientThink(): Overview .....................................................................................62
  3.9.5 ClientThink(): Structure .....................................................................................63
  3.9.6 ClientThink(): Function Characteristics ..............................................................64
  3.9.7 SendClientMessages(): Overview .....................................................................65
  3.9.8 SendClientMessages(): Structure .................................................................66
  3.9.9 SendClientMessages(): Characteristics .........................................................66
3.10 Summary .....................................................................................................................67

Chapter 4 A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore
Platform ..........................................................................................................................69
  4.1 Introduction .................................................................................................................69
  4.2 Related Works ............................................................................................................69
  4.3 OSCAR Compiler & the Programmer Relationship ...................................................71
    4.3.1 The Role of the OSCAR Compiler .......................................................................71
    4.3.2 The Role of the Programmer ..............................................................................72
  4.4 Parallelization Methodology .......................................................................................73
  4.5 Profile Program for Areas of Focus .........................................................................74
  4.6 Examining focused areas for potential .................................................................75
  4.7 Discovering the hazards & Reworking the code ......................................................76
    4.7.1 BotAI(): Hazards & Solutions ..........................................................................76
    4.7.2 ClientThink(): Hazards & Solutions ..............................................................82
    4.7.3 SendClientMessages(): Hazards & Solutions ..............................................87
4.8 Hazard Summary ...........................................................................................................91  
4.8.1 Reading/Writing to Global Variables.................................................................91  
4.8.2 Function Pointers ...............................................................................................92  
4.8.3 Thread-to-Thread Illegal Data Access.................................................................92  
4.8.4 Shared Memory Pools .......................................................................................93  
4.8.5 Interconnected Variables & functions in a Module .................................................93  
4.9 Performance Analysis on a Multicore Platform ........................................................94  
4.9.1 Sanity Check ......................................................................................................94  
4.9.2 Evaluation Environment: IBM POWER5+ .........................................................95  
4.9.3 Evaluation Environment: RPX .........................................................................100  
4.10 Summary ..............................................................................................................100  

Chapter 5  
A Power Reduction Approach for ioquake3 on a DFVS Platform ...........................................106  
5.1 Introduction ........................................................................................................106  
5.2 Power Reduction Methodology ........................................................................106  
5.3 OSCAR Power API Overview ........................................................................107  
5.4 Determining Power Reduction Potential ...............................................................107  
5.5 Choosing the Proper OSCAR Method .................................................................109  
5.6 Adding OSCAR Directives into ioquake3 ............................................................111  
5.7 Environment and Results ................................................................................111  
5.7.1 Modified Kernel ...............................................................................................112  
5.7.2 Rebuilding OSCAR ..........................................................................................113  
5.8 Performance Results ..........................................................................................113  
5.9 Summary ............................................................................................................114  

Chapter 6  
Conclusions & Discussions ...............................................................................................118  
6.1 Summary of Results ........................................................................................118  
6.1.1 Parallelizing ioquake3 for IBM POWER5+ & RPX .....................................120  
6.1.2 Power Reduction of ioquake3 for RPX .......................................................120  
6.1.3 Benefits ........................................................................................................120  
6.2 Potential Work ................................................................................................122
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Example of a sequence code</td>
<td>17</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The previous code executed in an up-down flow resembling a waterfall</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Core video game engine that operates in a waterfall manner</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4</td>
<td>A sample macro task graph</td>
<td>35</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Execution structure of minimum time execution method</td>
<td>41</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Execution structure of deadline method</td>
<td>42</td>
</tr>
<tr>
<td>Figure 7</td>
<td>A snapshot of Wolfenstein 3D</td>
<td>48</td>
</tr>
<tr>
<td>Figure 8</td>
<td>IoQuake3, railgun sniping a player</td>
<td>50</td>
</tr>
<tr>
<td>Figure 9</td>
<td>A snapshot of Counterstrike</td>
<td>53</td>
</tr>
<tr>
<td>Figure 10</td>
<td>A snapshot of the famous game Half-Life</td>
<td>55</td>
</tr>
<tr>
<td>Figure 11</td>
<td>The visual upgrade in remake of Half-Life using the new source engine</td>
<td>57</td>
</tr>
<tr>
<td>Figure 12</td>
<td>An abstract IoQuake3 session flow</td>
<td>59</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Bota() computation flow</td>
<td>61</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Clientthink() computation flow</td>
<td>63</td>
</tr>
<tr>
<td>Figure 15</td>
<td>The execution flow of FireWeapon()</td>
<td>65</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Critical areas’ general code structure</td>
<td>75</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Relocating read/write operations outside of the parallelized area</td>
<td>78</td>
</tr>
<tr>
<td>Figure 18</td>
<td>: reworking function pointer</td>
<td>81</td>
</tr>
<tr>
<td>Figure 19</td>
<td>The link &amp; unlink operation</td>
<td>84</td>
</tr>
<tr>
<td>Figure 20</td>
<td>The lock implemented into FireWeapon()</td>
<td>85</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Pseudo code hazard prevention in FireWeapon() by using locks</td>
<td>86</td>
</tr>
<tr>
<td>Figure 22</td>
<td>The redesign of snapshotentities</td>
<td>89</td>
</tr>
<tr>
<td>Figure 23</td>
<td>A snapshot of a single-layered map in IoQuake3</td>
<td>96</td>
</tr>
<tr>
<td>Figure 24</td>
<td>A snapshot of a multi-layered map in IoQuake3</td>
<td>97</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 1: OSCAR API's list of directives.................................................................46

TABLE 2: The spec table of IBM POWER5+ and RPX ...........................................93
Chapter 1 Introduction

1.1 Background and Purpose

Video Games have been a very popular form of digital entertainment, which are presented nowadays on many different platforms. Video Gaming platforms vary from fully dedicated systems such as large Arcade machines and home entertainment systems, personal computers, and to even handheld devices. As computer developers sought to achieve high performance by dramatically shifting to multi-core processors, so did Video Gaming companies. However, because of difficulties such as resource contentions and pointer analysis parallel programming is still a very challenging technology to implement.

To minimize the cost of implementing parallel programming while still achieving higher performance parallelizing compilers have been researched and developed. To the knowledge of the author, no research has yet been conducted that studies parallelism in Video Games by using parallelizing compilers. As Video Games are on a wide array of platforms that includes handheld machines, power consumption too becomes crucial in keeping the battery life favorably longer. No paper has evaluated power consumption of a Video Game using an automatic compiler either.

In this research the potential performance enhancement and power consumption reduction of a sequentially written Video Game by the use of a parallelizing compiler
Introduction

while investigating the difficulties in achieving that goal shall be examined. The target application shall be a well-renowned first-person shooter Video Game called ioquake3 which presents many of the important elements found in Video Games such as intelligent bots, physics, networking and others.

1.1.1 Sequentialism

Almost every task in this universe could be broken down to smaller independent or interdependent tasks. Furthermore, the execution of this main task could be carried out by applying one of two - or both - main ideas; sequential-ism or parallelism. As this title implies, sequentialism is when the tasks that compose the main job are executed in sequence.

This philosophy can be much easily seen in computers where applications are fundamentally a batch of operations that are executed by the processor. And, this batch of operations have certain a sequence that was laid down by the programmer for the processor to follow. This style is fairly easy to implement because the main focus during execution is the task at that executing moment, and no consideration needs to be given to the following or proceeding tasks; hence easier to manage and execute. Figure1 shows an abstract example of how the flow of execution is carried out in a sequential manner by the processor. The figure shows a piece of code that its operation will be executed in sequence. Therefore, based on the previous reasoning, when the processor executes line 20 “x=y+5;” it would not need to care about the values of x nor y, nor would have to care about the operations in line 10 nor 30. This sort of up-down flow of execution
Introduction

resembles a *waterfall*; hence, this is usually referred to a *waterfall execution*. Figure 2 displays how the previous piece code will resemble a waterfall when executed.

### 1.1.2 Parallelism

The other idea that could be implemented when executing a job is called *Parallelism*. This style is when the smaller composing tasks are executed concurrently, in *parallel*, at certain point(s) in during execution time. For this approach to be applicable multiple processing units must be present.

In the computing world, this philosophy has several major benefits; such as the ones that relate to the work presented in this paper are *enhancement of performance*, and the *reduction of overall power consumption*. Because this approach is mainly implemented by executing the smaller tasks concurrently based on the aspect of how interdependent and independent those tasks are, the amount of parallelism highly varies based on that aspect; consequently, the potential reducible power consumption will be effected as well.

```plaintext
10 y = abs( y );
20 x = y + 5;
30 x2 = power( x );
40 fprintf("The power of x = %d", x2);
```

Figure 1: Example of a sequence code
1.2 Video Games

1.2.1 What is a “Video Game”?

Video Games can be roughly defined as any digitally controlled form of entertainment from the ones that are packaged to target highly complex high-end computers- such as large servers, or designed to run on state-of-the-art Video Game consoles that will be enjoyed in the comfort of one’s own home- such as Playstation3 and Xbox360. Moreover, many Video Games are designed with deep Gameplay, network capabilities, complex storylines and intriguing graphics, and come now in

```
y = abs(y);

x = y + 5;

x2 = power(x);

fprint("The power of x = %d", x2);
```

Figure 2: The previous code executed in an up-down flow resembling a waterfall
formats that would run on compact systems that would fit on the palm of one's hand—such as PSP, Nintendo DS or even mobile phones. The mention of Video Games in this paper will be considered as such.

As Video Games vary in platform requirements, Video Games also vary in mechanics in which differ in program implementation. For example, racing Games have players drive racing cars through relatively simple courses, but require highly complex real-time computations to deal with the physics requirements of simulating actions such as drifting and over different surfaces—such as mud, sand and snow, object collisions and so forth. Another example is Role Playing Games (RPG). In a conventional RPG Game a player roams and adventures through massive and complex maps with hidden paths and areas, and fights AI driven monsters and interacts with AI driven characters. Even though computations in RPG Games are not computed as “real-time” as Racing Games are, RPG Games still require high capacities of storage to deal with the data of such massive maps and seemingly endless array of characters, and an efficient method to handle, retrieve and store such large amounts of data and still give the player a smooth adventurous experience.

Those were only two examples within the extensive Video Game genres that show varying requirements in the one same field of Video Games.

1.2.2 Compact and the Waterfall

To help make Gaming consoles affordable Gaming companies strived to produce Gaming consoles that offer intriguing features—such as beautiful graphics, touch panels,
gravity-sensitive peripherals and so forth— at cheap prices; thus, these consoles came in highly compact packages that had limited computing resources that started at 8-bits and cartridge for storage back in the Nintendo Entertainment System and SEGA Megadrive that drove developers to work tirelessly to extract the optimum performance from those compact platforms. Therefore, this resulted in Video Games as computer applications having inherently compact designs that rely on programming shortcuts.

Video Games are basically an interacting application that is computed based on user-inputs, and calculated using built-in logics, finally outputting the results on the screen. This *waterfall* structure— *event-handling, logic, and rendering*— has been tried out and tested the test-of-time, and has come to be the universal basic block of Video Game implementations, shown in figure 3. These target platforms and internal structures are the cornerstones of Video Game implementations and must be considered when approaching Video Game development.

### 1.2.3 Why approach Video Games for Research?

Video Games as applications have a significant characteristic of being utterly real-time where the computations are purely user based. A typical Video Game must contain a core module that computes the rules and logic of the Game itself. For example, a simple Game of chess must contain at least the following modules; a graphics module to render the chess board and pieces; a user-interface module to handle the user input moves; and the logic module that computes how the Game chess is played— how pieces move, how the Game is won and so forth. Additionally, Video Games usually also
contain other several critical modules such as AI module, physics simulator module, navigation module and such. Because Video Games are extremely user-based applications, the combination of access frequency and the interactions between the different objects within the Video Game makes for a delicate application to handle when it comes to parallelism. Therefore, in general, Video Games as applications present a very interesting problem to explore.

### 1.2.4 Inherit Sequentialism

As explained in the previous sector, Video Games conventionally follow the *waterfall* structure, and three basic blocks of input-handling, logic, and rendering. Furthermore, because the target compact systems, Video Games are implemented to be compact themselves; thus relying on many programming shortcuts for computational speed. Such aspects give Video Games as computer application an inherently sequential character.

Parallelism relies on decomposing a sequential program into smaller parts and finding the concurrently executable areas, then divides the program based on that, finally, implements a new structure to that program to be executed in a parallel fashion to minimize execution time, and/or reduce power consumption or gain other valuable benefits.
Therefore, sequentially inherited character of Video Games as applications must be deeply considered when engineering it towards an unconventional direction such as parallelism.

The fundamental approach in optimizing any task in this world is to find the area that consumes most effort, and try and focus on that. In parallelism, this mentality is very true. The hurdles facing the parallelization of Video Games will be explained in full detail in the following chapters.
1.2.5 Why Choose Multiplayer Games over Singleplayer Games?

Graphics is a naturally appealing feature in any sort of Video Game. This becomes very true in singleplayer Games, because the Gamer solely is immersed into the Universe of the Game he is playing; thus, visuals become critical in the overall Gaming experience. Graphics are mainly computed nowadays outside of the main CPU’s scope and majorly in the GPU. On the other hand, multiplayer Games’ main attraction is interactive Gameplay between the participants; thus, this imposes that multiplayer Games are less graphics based, and more mechanics based. Therefore, this focus incurs that the major Game activity resides inside the Video Game’s core modules, such as logic, AI, physics and so forth, that will be processed by the main CPU.

Another important difference that is found between multiplayer and singleplayer Games is the amount of computations that are incurred by user input increases with the increase in the number of players. For example, in a singleplayer Game there is only one player existing; thus, only one player’s computations are needed, while in a multiplayer Game with 4 participants, 4 separate computations are needed.

In singleplayer Video Games the form of interactions usually fall under two types that is player/world and player/object interactions. World is the unchanging entities such the map. Objects are the interactive entities such as AI driven enemies, items and so forth. On the other hand, in multiplayer Video Games one more crucial type of interaction comes into play, and that is player/player interactions. Consequently with this additional type of interactions, the required amount of computations mentioned in previous sections increases exponentially. For example, theoretically, in a Game with
only 2 players, 2 interactions are naturally resulted. Furthermore, in a Game with 3 players, the potential number of player/player interactions is 9 interactions, and with 4 players, 16 interactions. Even though this exponential growth is theoretical, in high paced Video Games such as FPS with relatively compact maps or mobile vehicles, the practical growth of interactions is not at all far from the mentioned theory.

Therefore, it can be clearly seen that multiplayer Games impose much higher computational requirements for the main CPU than singleplayer Games; thus, presenting a much challenging topic of research. Therefore, from a parallelism point of view, multiplayer Games hold much more computational intense areas that are of the responsibility of the main CPU than singleplayer Games, which will be explained in more detail in the following section.

1.2.6 Multiplayer Games from a Parallel Point of View

This short section is to complement the trail of logic behind the research choice where it will explain the reason for choosing multiplayer Games for research from a parallel point-of-view. The reasons are as follows; first is to explore how much concurrent tasking that parallelism could be extracted from the naturally inherit sequential Video Game as an application. Second is to examine the potential parallelism that exists in a benchmark multiplayer Game’s main modules- such as AI, and so forth- that are the responsibilities of the main CPU. Third is to discover what kind of parallelizing difficulties these modules impose. Forth is to explore the difficulties that are created by known-to-be coding customs found in Video Games that hinder the
analysis that will be made by the parallelizing compiler. Last is to investigate the amount of effort needed to prepare the Video Game code for an automatic Parallelizing compiler for the extraction of parallelism.

1.2.7 Relationship with the Research

One important reality the author would like to include in this paper is the condition of the Gaming industry in relation to the research field. Even a single Video Game composes many aspects together, such as graphics, sound, controls, networking, story-line, power consumption, and even social and psychological aspects amongst many others. Researchers have graciously touched on the more easily accessible areas such as graphics. This was aided by the fact that there are many graphics companies outside of the Gaming industry, such as nVidia [1] and ATI [2] that make research on graphics more easily accessible. As for the psychological impacts, since it studies people; they have made steps in advancing those areas [3] [4]. The author does not agree of research being carried out by researchers who have no true roots in that field of experiment.

Due to the Video Gaming industry being a highly competitive one, was at least, has contributed to Gaming companies being highly secretive about the technologies they develop into their Games. Because of this closed nature open research field has been scarce in themes touching the heart and core of Video Gaming technologies such as physics, AI and other important areas; thus creating a gap between researchers and Video Game developers. Furthermore, utterly complex technologies such as parallelism
that is very fresh in the Gaming industry, and only introduced in the latest generation, and that flows against the inherit sequential character of Video Games could become a challenging if not crippling hurdle for Video Gaming companies to tackle alone, namely the smaller companies who cannot afford investing into restructuring their magnificently architected Video Games to abide by the new rule of parallelism. This costly gap is mainly what this paper is trying to narrow. It is trying to help melt that gap by laying a primitive but important stepping stone for others to add to in order to make the great benefits of parallelism be affordable and accessible to all Gamers.

1.3 Thesis Outline

This thesis consists of 6 chapters.

Chapter 2, “OSCAR Compiler and API” describes how the OSCAR compiler [5, 6] automatically extracts parallelism from a computer application, and similarly how it reduces the power consumption requirements of a given application. First, the OSCAR compiler begins by analyzing a computer application to discover control and data dependencies amongst its macro-tasks; basic blocks; loops; and function calls. Then, it assigns those macro-asks to the designated number of processing units to reduce the overall execution time. And, depending on the target platform, OSCAR may take advantage of embedded accelerators and cache optimization and other beneficial to performance. Furthermore, to help reduce power consumption the OSCAR compiler may apply one of the following methods: (1) Minimum Execution Time; sleep when idle (2) Satisfaction of Real-time Deadline; lower the frequency to exploit idle time.
Furthermore, Parallelizable-C rules are integrated into the target application’s code to help OSCAR analyze the program thoroughly and exploit optimum amount of parallelism and power reduction potential as well.

Chapter 3, “ioquake3 Structure Analysis” explores the structural detailed analysis of a well renowned multiplayer Video Game called ioquake3. To understand which areas of ioquake3 should be the focus of the optimizations, the execution of ioquake3 was profiled using a performance profiler called Visual Studio Performance Profiler. The profiling results revealed that after the ioquake3 engine finished all Game initializations, there existed three bottlenecks that utilized over 90% of the CPU processing load. First, to understand the driving mentality behind ioquake3’s operational engine, the internal modules and mechanisms were broadly explored. The analysis showed that the three bottlenecks operated in a waterfall manner where the flow of operation is as follows; (1) BotAI was at the beginning and was responsible for replacing the human counterpart for an AI driven player, and acted as the brains for that player by doing the following: first, viewing the surroundings, then, making decisions based on personal and areal conditions, last, inputting the commands to carry out that chosen decision. (2) ClientThink was responsible for carrying out the actions that were chosen by the player- human and AI driven alike- into the virtual world, and calculating the logic of those interactions while maintaining the changes incurred by those actions by updating the data of all the interacting entities that are categorized into three groups; player, World, and items. (3) SendClientMessages was responsible for updating all the participating players- human and AI driven- with their latest surroundings that resulted from the
previous actions by using a camera-like mechanism and taking a snapshot of the surroundings and conveying it to the designated player.

Chapter 4, “A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore Platform” shows the parallelizing methodology that was implemented to ioquake3 so that it could be compiled on the OSCAR compiler, and the performance results of that new ioquake3 implementation in comparison with the original one. After establishing where the focus of the CPU should be and how the internal modules interconnected, the target for the research was achieved and the experiment was carried as follows. First, the analyzed structures of the three bottlenecks that were discovered were examined to determine the amount of parallelism that could potentially be extracted from them. The results showed that the three bottlenecks were mainly of a “for loop” structure that handles heavy amounts of computations; thus indicating feasible potential for parallelism. Second, as a preliminary experiment to see how much extractable parallelism could be made without any alteration to ioquake3, the original program was compiled by using the OSCAR compiler, then tested on a multicore machine. The results showed that no performance enhancement could be achieved. That was so due to the existence of many automatically insolvable hazards; thus the original code had to be rethought. Third, the bottlenecks were deeply analyzed by hand and by the aid of a debugger to discover all the potential hazards that may occur from attempting to parallelize ioquake3 while noting the non-compliant Parallelizable-C areas. Forth, all the discovered hazards were examined, rethought, solutions were planned out, and then implemented while revamping the code to become Parallelizable-C compliant. Fifth, the newly revised ioquake3 code was then compiled by the OSCAR
compiler to produce a newly parallelized version. Finally, the performance of the newly produced code was measured on two multicore machines- IBM POWER5+ and RPX- and the performances were analyzed. The results showed that the newly produced OSCAR code could achieve 5.1 times speedup on the IBM POWER5+ machine using 8-cores, and 2 times speedup on RPX using 4-cores.

Chapter 5, "A Power Reduction Approach for ioquake3 on a DFVS Platform” expresses the methodology that was implemented to reduce the power consumption requirements of ioquake3 when running on a platform that is equipped with DFVS capabilities, and also shows the environmental setup to produce the desired results. The approach in which ioquake3’s power consumption was reduced was by: first, the mechanics of ioquake3 as an application were examined to choose one of OSCAR’s two power reducing methods mentioned earlier. Video Games are designed with the known structure of time frames, where the logic of the inputted actions must be computed within a certain time frame. This structure perfectly fits OSCAR’s Deadline approach; thus it was chosen. Second, for OSCAR to appropriately schedule program tasks to enable power reduction, the time and clock-cycles for those tasks must be explicitly specified for OSCAR. Therefore, the time and clock-cycles for the internal modules of ioquake3 were measured, then, implemented into the code as directives for the OSCAR to input for its analysis. Third, ioquake3 was compiled on OSCAR using the power reduction options. Finally, the power reduced version of ioquake3 was executed on RPX that is equipped with DVFS features, then, the power consumption of RPX was measured and an analysis was made. On the one hand, the results showed that when running the original ioquake3 version, RPX consumed 1.6 watts on average. On the
other hand, the results also showed that when running the power reduced version of
ioquake3, RPX consumed 1.25 watts on average. Therefore, the OSCAR compiler was
able to reduce the power consumption of an ioquake3 down to 73% on average. It is
important to mention that originally RPX was running with a Linux environment that
had a 10000 micro seconds frequency transition time, which was too long for ioquake3’s
frame rate of 30 frames per second, 0.0333 seconds per frame. Therefore, a modified
version of Linux was installed that had 50+ microseconds frequency transition time,
which allowed the OSCAR to exploit idle times within ioquake3 successfully.

Chapter 6, “Conclusions & Discussions” concludes this thesis and explains future
works.
Chapter 2 OSCAR Compiler and API

2.1 Introduction

As mentioned in the previous chapter in sectors Sequentialism and Parallelism, parallelism allows for many benefits to be gained from multicore structures, such as enhanced performance and reduced power consumption. Enhanced performance through parallelism is mainly achieved by dividing a singular large task into smaller ones, and then distributing those decomposed smaller tasks onto several processing unit to be processed concurrently. This reworked processing structure allows for the same original task to be processed in a shorter amount of time; thus, producing an enhanced performance.

On the other hand, reduced power consumption is another benefit gained from parallelism. By exploiting inner-application idle times, and features found in certain processors such as Power Gating [7] and/or Dynamic Frequency and Voltage Scaling (DFVS) the power consumption of that computer structure may be reduced. For example, by shutting down the processor, or reducing its voltage, during idle times could help reduce the overall power consumption of a given computer application. Another example, exploiting the existing idle time found between one process and the consecutive one by reducing the processor’s frequency and extending the processing time of that process may help reducing the power consumption of that processor.
Further details of those two examples will come in the following sections. Important to mention is that parallelism helps in achieving power reduction by expanding and increasing the amount of those crucial idle times.

These attractive benefits achievable through the power of parallelism unfortunately do not come as easy as simply developing straightforward sequential software. Resource contentions and pointer analysis are just a few of the many hurdles that must be taken into consideration when attempting to parallelize a computer application [8]. Furthermore, such difficulties are highly difficult to detect pre-production, and very complex to debug. Therefore, parallel computing is still a difficult concept to implement and has become the focus of many researchers over the world.

2.2 Parallelizing Compilers

To overcome the difficulties opposing the achievement of parallelism, automatic parallelizing compilers have been researched and developed throughout the recent generations [5]. Automatic parallelizing compilers (or for sake of shortness, parallelizing compilers) were developed mainly achieve two objectives:

1. To automatically extract potential parallelism from a sequentially written computer application to achieve higher performance.

2. To mask the parallelizing complexities from the developers to help them focus more on the core engine of the developed computer application rather than be burdened by the extraction of parallelism itself.
2.3 The OSCAR Compiler

One famous automatic parallelizing compiler is the OSCAR compiler. The OSCAR compiler is developed at Waseda University, Japan, by Kasahara & Kimura Laboratories [9]. The OSCAR compiler has proven to extract parallelism from sequentially written computer applications and achieve higher performance and reduce power on multicore platforms. It has shown excellence in fields such as multimedia, simulators and others [10]; such as MPEG decoders, and earthquake simulators. The OSCAR compiler is able to achieve high amounts of speed up not just by extracting parallelism, but by using techniques such as optimizing cache usage, and others [11].

This research has the novelty of investigating the ability of the OSCAR compiler in extracting parallelism from Video Game applications.

2.4 How is parallelism extracted by OSCAR?

OSCAR excels at enhancing the performance of a sequentially written C Program by extracting parallelism at the multigrain level and exploiting data locality. Here, multigrain parallelism is the technique of extracting parallelism at different grains such as coarse grain task parallelism, loop iteration parallelism, and statement level near fine grain parallelism. In the following text, loops, function calls, and basic blocks are defined as coarse grain macro-tasks (MT).
Data dependencies and control flow amongst macro-tasks are hierarchically analyzed. Then, *Earliest Executable Condition* analysis that is based on those Data Dependencies and Control Flow is made to determine parallelism amongst those MTs. The analysis result is represented as a Macro Task Graph (MTG). If an MT is a subroutine call or a loop that has coarse grain task parallelism, the compiler generates inner MTs inside that MT hierarchically- figure 4 shows an example of an MTG.
Finally, the OSCAR compiler assigns MTs to the targeted processor groups or processor cores by using either static or dynamic scheduling. The OSCAR compiler determines the execution time for each MT to minimize the program’s overall energy consumption, and, chooses a critical path, the longest execution time needed for the MT.

If several MTs share the same piece of data that is larger than the cache size or the local memory, the OSCAR compiler will decompose the MTs into smaller ones so that it will be able to fit the data accessed by those sharing MTs into the cache or memory space by loop aligned decomposition. Then, these decomposed MTs are scheduled onto Processor elements, which access the same data successively as much as possible.

2.5 Human Intervention Necessity

OSCAR’s excellence in extracting parallelism and enhancing the performance of sequential computer applications on multicore platforms has been shown and proven in many papers. The beginning of the OSCAR compiler started with a version aiming the FORTRAN programming language [12], where it showed great success in speeding up the performances of many applications in fields such as multimedia and simulators. Then, a next generation of the OSCAR compiler was developed that targeted to serve programs written in the standard C-language. Then again, the OSCAR compiler was proven to have high capacity in extracting parallelism from sequentially written application of the same fields. However, with the strength of the OSCAR compiler in extracting parallelism, due to many hurdles that burden the analysis of C programs;
such as pointer analysis and others, the Parallelizable-C rules have been research and composed to help simplify the analysis for the OSCAR compiler, which in certain cases makes the difference having good speedup and NO speedup as it will be shown here in this paper. Furthermore, the OSCAR compiler has the strength to analyze programs for data and control dependencies, and rearrange the code to a certain extent to avoid hazardous conditions. However, it is highly critical to state that the OSCAR compiler still lacks the ability to handle extensive data dependencies that extend many scopes within a single program. Furthermore, issues such as pointer analysis, read-only global variables, local static variables and other issues can still become a crippling hurdle to OSCAR’s program analysis and extraction of parallelism. Furthermore, as the mass array types of computer application, the different architectures of each type of computer application differs the need. Therefore, human intervention is still a required step towards achieving higher performance [13]. The OSCAR compiler need for human intervention to help restructure a given computer application to be more parallel-friendly, and the programmers desire for higher performance through automatically extracted parallelism by the OSCAR compiler, this give and take relationship can be considered a successful and fulfilling one. This shows that the relationship between the programmer and the OSCAR compiler is still a cooperative one that should yield at the end very rewarding results.
2.6 Parallelizable-C

To overcome hurdles that hinder parallelizing sequential applications, such as pointer analysis, a composite of programming rules are advised to be used to guide compilers towards performing pointer analysis precisely, and extract the most possible amount of parallelism from a sequential program. Those rules are called the Parallelizable-C rules [13]. These set of rules govern the logical way of implementing a computer application by helping to avoid the implantations that are difficult to analyze, and make compiled code easier for the parallelizing compiler to extract potential parallelism from it.

2.7 Power Reduction

Some of the recently developed multicore platforms come equipped with Dynamic Voltage and Frequency Scaling (DFVS) and Power Gating features that can be controlled by the OS, but the control is normally limited to outside the inner power status of a running application. The OS implements power control by shutting down the CPU or lowering the frequency during idle times of an entire application; for example between runs. Therefore, it may result in reducing the power consumption of the system as a whole and not for a specific application in particular.

The OSCAR compiler on the other hand has achieved automatic power control schemes by using DVFS and Power Gating for multicores that allow power control of an application from within that application itself by exploiting idle times of application that
exists between its internal operations. It achieves this by implementing one of two approaches, which will be explained in the following section.

After the MT scheduling phase, the power reduction algorithm determines the suitable voltage and frequency for each MT. When determining the MT voltage and frequency phase is concluded, the OSCAR compiler applies the dynamic frequency scaling to reduce energy consumption while considering MTs idle times and their overheads. Overheads such as power and frequency transition times must be measured before using the OSCAR compiler for reducing power. They are measured and then fed into the OSCAR compiler’s code then the OSCAR compiler is rebuilt with that target platform specific transition times for optimum accuracy.

The two methods that OSCAR implements to reduce power consumption of an application is the following [14]
2.7.1 Minimum Time Execution

In this method, the OSCAR compiler analyzes the MTG to find out idle times that exist between processes. As shown in figure5, if CPU0 finishes executing its designated operation and there is enough idle time before needing to execute another operation at synchronization time, and before CPU1 finishes executing its operation, then OSCAR compiler may exploit that idle time, and instruct CPU0 to sleep or reduce its frequency. By executing this, the overall power consumption of the machine is reduced.

2.7.2 Satisfaction of Real-Time Deadline

In this method, the OSCAR compiler again analyzes the MTG to find out the idle times that lie between the ending of an operation and a given overall deadline. As shown in figure6, if CPU0 is able to finish the execution of an operation within the designated deadline while having some slack, idle time, then, the OSCAR compiler may exploit that idle time by expanding the execution of that operation to use that idle time. Furthermore, the OSCAR expands the execution of that operation by reducing the power level or the frequency of that executing CPU. This results in an overall reduction in power consumption of that application.

2.8 The OSCAR API

One of the main missions of the OSCAR compiler is to produce parallel code for a wide array of computer architectures. Therefore, the code that is produced by the
OSCAR compiler will be then compiled by a native compiler of the targeted architecture. To help make the code transparent to the underlying structure, the developers of the OSCAR compiler have also strived to develop an API that contains various directives can be implemented into the desired code before it is fed into the OSCAR compiler. Then, the new produced code by OSCAR would be recompiled again by the interpreter to produce a new code for the targeted platform, with all the directives translated into platform-specific functions that could be then recompiled again on a native compiler to produce the desired code. Table 1 shows a table containing the main OSCAR directives. The main directives that concern this paper are:

1. Parallelism:
   - **flush**: Specifies that all threads have the same view of memory for all shared objects.
   - **Parallel sections**: Identifies code sections to be divided among all threads.

2. Power reduction:
- **fvcontrol**: Instructs the processor unit to a different frequency.

3. Time:

- **get_time**: Retrieves the current time stamp used for synchronizing between processing units.

### 2.9 Summary

This chapter described the major motives of automatically extracting parallelism from a sequentially written computer application, and unburdening the programmers from the need to delve into analyzing those complexities were the main driving force behind the development and research of what are called automatic parallelizing compilers. The OSCAR compiler is a compiler of that sort that has proven its excellence in extracting parallelism from computer applications in many fronts such as multimedia and simulators. It does so by analyzing a sequential program that was written in C.
language to find the control and data dependencies amongst its macro-tasks, and then assigning those macro-tasks to the processing units to minimize the overall execution time of that computer application. Furthermore, the OSCAR compiler may exploit other features such as cache optimization to yield the optimum amount of speedup possible from an application.

The OSCAR compiler may also exploit features found in the latest processors; such as DFVS and power gating to reduce the power consumption of a computer application. The OSCAR is capable of achieving this energy saving mechanism by mainly using the idle times found within a given computer application by applying one of two main methods. Satisfaction of Real-Time Deadline is the method of exploiting the idle times within the execution of a single operation by expanding the execution of that computer application and reducing the power level of the frequency of that processing unit, which should result in an overall reduction in power consumption. Minimum Time Execution, on the other hand, is the method of exploiting the idle times within the

<table>
<thead>
<tr>
<th>Parallel Execution API</th>
<th>Data Transfer API</th>
<th>Power Control API</th>
</tr>
</thead>
<tbody>
<tr>
<td>* parallel sections</td>
<td>* dma_transfer</td>
<td>* fvcontrol</td>
</tr>
<tr>
<td>* flush</td>
<td>* dma_contiguous_parameter</td>
<td>* get_fvstatus</td>
</tr>
<tr>
<td>* critical</td>
<td>* dma_stride_parameter</td>
<td>* Synchronization API</td>
</tr>
<tr>
<td>* execution</td>
<td>* dma_flag_check</td>
<td>* groupbarrier</td>
</tr>
<tr>
<td>** Memory Mapping API **</td>
<td>* dma_flag_send</td>
<td>** Timer API **</td>
</tr>
<tr>
<td>* threadprivate</td>
<td>** Cache Control API **</td>
<td>* get_current_time</td>
</tr>
<tr>
<td>* distributedshared</td>
<td>* cache_writeback</td>
<td>** Accelerator API **</td>
</tr>
<tr>
<td>* onchipshared</td>
<td>* cache_selfinvalidate</td>
<td>* accelerator_task_entry</td>
</tr>
<tr>
<td></td>
<td>* complete_memop</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: OSCAR API’s list of directives
execution of a single operation by shutting off or lowering the frequency of a singular processing unit during the idle time it has before the next synchronization point to reduce the overall power consumption of that multicore system.

To help the automatic compiler analyze a sequentially written computer application in C language, the Parallelizable-C rules may be applied. Parallelizable-C rules are a composite of guidelines that govern the logical syntax of that written program.
Chapter 3 ioquake3 Structure Analysis

3.1 Introduction

In this chapter a Video Game benchmark called ioquake3 [15] that was used in this experiment will be explained in minute detail. First, the origins of the First-Person Shooter (FPS) Gaming genre will be shown with examples of its evolution and development throughout the years. Then, an important phenomenon called “Modding” [16] found generally in PC Games will be explained, and the consequential benefits will be introduced, benefits such as the production of the Video Game, ioquake3 that is being used in this research. Next, a generous introduction will be given to the father of the FPS genre, Quake III will be made, and everlasting attributes that allowed Quake III to reach its high status in the Gaming community will be explained. Afterwards, one of the important contributions that Quake III, the Bot that has swept rivaling Games aside will be introduced. Then, the research entry of performance profiling of ioquake3, the mod of Quake III, that includes introduction of the profiler, Video Game setup, and results will be explained. Finally, a thorough detailed explanation of the discovered ioquake3 bottlenecks will be shown.
3.2 What are FPS Video Games?

Back in 1992, a Game called *Wolfenstein 3D* [17] practically gave birth to a totally new genre of Video Games called *First-Person Shooters*, or what lately came to be known as FPS. The major characteristics that distinguish a Video Game of FPS genre are three:

- **First-Person**: The player who controls the in-Game character views the virtual world from a *first-person* perspective.

- **Shooter**: The controlled character must be able yield a *shooting* weapon; such as a handgun, shotgun, laser, or even a rocket launcher.

- **2D movement**: The player must be able to navigate the controlled character in at least a two dimensional plane, similar to walking on a flat plane.

As figure7 shows, Wolfenstein 3D introduced this style of Game mechanics into the Gaming field, and gave birth to the FPS genre that has evolved for many generations to come. However, Wolfenstein 3D amassed a great followership on its original PC platform, it. But, even though it was also released for the Super Nintendo Entertainment System [17], because of the limitations of home console Gaming machines, the FPS genre did not flourish much in that pre-3D generation.

With rapidly advancing dedicated graphics cards, and equally advancing computational specs, PC had higher potential in releasing graphical intense Video Games, which saw the FPS genre have humongous leaps. After Wolfenstein 3D laid the foundations for what an FPS may be, the developing company *id Software* again took the Gaming world by storm with their revolutionary FPS Game, titled *Doom* [18].
introduced a totally 3-D world with intense Gaming feel with enemies emerging from all
directions, including the vertical plane, and a wide array of weapons that included a
nuke weapon that could wipe out mass hordes of enemies. Doom gained so much
popularity that it generated a Doom “subculture” with many Doom clones trying to
mimic the Doom Game. Along with the newly created 3D feel, rich arsenal, and complex
map structure, one major contribution to this massive following was the Deathmatch
feature. A “Deathmatch” is when more than one player connects into one singularly
shared session, or match, and play together, and battle to the death. Doom spawned
many expansion packs that added to the original Game. Two major sequels followed;
Doom II and Doom III. However, what greatly contributed to the long life span of Doom
was that after id Software generated enough revenue, they fully disclosed the code to the
public. Now, not only do players simply can enjoy the Game Doom, but they can add
new ideas of their own into the Game, and make their own versions of it, or what came to be called *mods*; will be explained in the next section. Doom sparked what was only the beginning of was about to come in this beautiful genre.

### 3.3 Quake, the Benchmark

In 1996, the Gaming community witnessed the arrival of a Game that would change the world of Gaming world forever, and that Game was called *Quake* [19]. To an average observer, Quake did not bring much change from the Doom series other than an enhanced physics mechanics such as the jump, swim, run and others. However, from a Gaming perspective, these new additions created a totally new Gaming mechanics that were filled with intensity and required utter precision and fluid controls that engulfed the FPS genre and the network Gaming world at large, and marked the birth of a new era, the *post-Quake era*. Because Quake was so hardcore-Gamer inclined, pro-players could be distinguished easily from the armature ones. Local and world-scale tournaments became a celebrated custom in the FPS world with players competing to see who was the best at this worldly renowned platform called Quake. Servers were erect to support tens and hundreds of players from all over the world.

Again id Software did what made them a legend in this field. After generating sufficient revenue from the Game, they released the code to the public. And with this move, because Quake’s engine was so perfected and at the same time very flexible to expansions and additions, it spawned an endless array of Game series that still extend till this very day in 2013; such as Call of Duty, Half-Life and many others. Furthermore,
because home consoles have started to support 3D graphical rendering better, ports on the Playstation [20] were released giving a taste of the intensity of the FPS genre to the home console fans as well.

The Gaming world, companies, publishers, and most importantly Gamers will be forever grateful to Quake for paving the way for this great genre. A picture of the Gameplay from ioquake3 can be seen in figure8.
3.4 The Bot Saga

Three major sequels followed Quake, Quake II, Quake III Arena, and Quake4 [21]. ioquake3 is mod of Quake’s 3rd installment, Quake III Arena. Quake III Arena was less focused on single player mode, and was more, as the name implied, inclined towards a rich and intense arena multiplayer Gaming style. It was such a hit that it still hasn’t left the Gaming communities at all with countless of servers hosting matches for hundreds of players today.

One memorable and novel feature that Quake III Arena contributed was AI driven players, or Bots [22]. id Software developed a highly challenging and interactive AI to help add flavor to the whole experience of playing Quake II Arena. If a session needs additional players to balance the teams, or if simply to add intensity to the mix, AI driven players, bot players, could be called upon to fill in that role. Bots were designed with varying difficulty levels from one to five; with the increase in number is the increase in intelligence. They were also designed with varying skins and personalities. Each personality or character has different dueling styles that has varying weapon preferences and aggression levels; thus, allowing for a richer overall experience. The paper will show the significance of bot players to this research in the following chapters.

The bot in ioquake3 uses what is called Area Awareness System (AAS) to navigate the virtual world. The process of using the AAS system is as follows:

- **Loading the map**: When a Game session starts, a map is chosen and loaded. During this loading process, the map is divided into a tree of nodes.
- **Linking players:** After the map has been loaded, players are *spawned* into the virtual map. And, each player must be *linked*, assigned, to the node they existed in at that moment.

- **Updating the tree:** When a Game starts, naturally, players will move about inside the map. If a player leaves the area he was in, and moves to another, they must be *unlinked* from the original area to the *linked* the new area.

### 3.5 What are Mods?

A philosophy found mainly in the PC Gaming platforms is *modding*. *Modding* is basically the ability to modify a Game partially or majorly; hence the term *modding* from the verb to *modify*. After a developer of a certain Game generates enough revenue to cover expenses and fair profit, they release the Game code, and sometimes a *modding* kit. Armature and professional developers can create their own versions of the Game, while noting that the core engine that of the original’s if they chose to publish for commercial profit. Some of the major benefits of modding are:
1. **Lengthening the life span of the Game**: Because modding allows for a newer group of fans to delve into the Game from a different perspective, the development perspective; thus, creating a brand new life cycle for that Game that is seemingly endless.

2. **Expansion of ideas**: A single person is limited to his own imagination, and so are developers of the original Game. However, if you open the door for masses and masses of fans to contribute with their own individual ideas, the possibilities are limitless, which may lead to new hits purely born from the modding community. One Game that stands as a clear example to this phenomenon is a Game called *Counter-Strike* shown in figure 9. Even after its original release back
in 1999, it remains one of top played Game in the world [23]. Counter-Strike is a mod of the famous Game called Half-Life [24] figure10; it was created by two developers, and gained so much popularity that it was then re-released as commercial Game.

Half-Life was a Game that was built on the Quake engine, basically a mod. Half-Life was also one of the grand of all grands. Last, in 2012, a massive public production re-released the original Half-Life was again using the new engine Source-SDK [25] as a Game called Black Mesa [26] shown in figure11, featuring newer graphics and better physics.

Similarly, ioquake3 is a refined version of Quake III Arena that includes improvements such as bug fixes, voice-over-IP capabilities, ports to other machines such as smart phones and others. ioquake3 is a Quake III Arena mod.

3.6 Analytical Steps

To understand what areas should be the focus of this research, the mechanics and mish-mashes of the inner engine of ioquake3 must be examined and analyzed. This analysis is conducted in several steps:

1. **Finding a suitable profiler:** A suitable profiler that has the capacity of determining the CPU intense areas in a computer application must found.
2. **Setting up the target application:** The most challenging execution of the target application must be found; an execution that would challenge the CPU so that the critical path in the application would be more clearly visible.

3. **Profiling:** The target application is profiled using a profiler and with the optimum setup.

4. **Analyzing the results:** An analysis of the profiling results to determine which functions exist within the critical path.

5. **Choosing the area of focus:** Choosing the function(s) from the critical path that cause *bottlenecks* to the overall program; thus, needing to be focused for the

---

Figure 10: A snapshot of the famous Game Half-Life
3.7 Profiling

Now the details of profiling ioquake3 will be explained.

1. **Finding a suitable profiler:** Microsoft Visual Studio’s Performance Profiler (MVPP) was chosen because it has the desired attributes; such as thorough analysis even at operation level; easy to use interface; interconnectivity between the analysis and the debugger in Visual Studio to help ease of use, and others.

2. **Setting up the target application:** To better learn where the bottlenecks are, the Game computational demands better be increased. Therefore, because the engine’s maximum number of players of 32 does not challenge the IBM POWER 5+ CPU enough, the engine was modified, and the maximum number of players was increased to 112; achieved by increasing the size of the memory pool, and correcting any subsequent errors.

3. **Profiling:** The target application, ioquake3, was profiled using MVPP. Since using an all human session is beyond the University’s capabilities, an all bots session was used instead; thus, using 112 Bots. The computational demands became clear, and the critical path became visible.

4. **Analyzing the results:** The results showed a critical path that was consuming over 90% of the overall computational time.

5. **Choosing the area of focus:** This critical path relied on three main functions:
Figure 11: The visual upgrade in remake of Half-Life using the new Source Engine

- BotAI()
- ClientThink()
- SendClientMessages()

Note: For ease of reading all functions in this paper will denoted with a pair of closed brackets at the end of them.

3.8 Critical Path General Overview

A typical Game session in ioquake3 flows as in figure12:

1. **Read Game session options**: The Engine reads the given options, either manually inputted by the session leader or read from pre-written file. These options includes:
   - Chosen map
   - Number of connectable players
- Participating Bots and their difficulty level
- Frag limit
- Game type (Deathmatch, Capture The Flag and others)
- Time limit... and others.

2. **Load files:** Based on the chosen options, the required files are read. During this stage, caching of certain files such as map of the AAS system is done.

3. **Spawning the World Map:** After all the options are known and loaded, the engine *spawns* the World Map.

4. **Spawning items and players:** Now the engine populates the virtual world with the chosen entities, which includes items and players.

5. **Synch and countdown:** The engine sends a synchronization signal to all connected players that Game will be starting. And, a countdown commits.

6. **Start**

The critical path that was discovered exists after the Game *starts*. The Game will continuously loop until a *quit* condition is met, such as frag or time limit, or a player exits.

**Note:** *Spawning* is the act of manifesting an entity by the engine into the virtual world, which includes the World Map, players, and items.

**Note:** And *Entity* is any existing object that can be interacted with inside the Game.
3.9 The Bottlenecks

In this section the structure of the three bottlenecks will be described in detail.

3.9.1 BotAI(): Overview

The main responsibility of BotAI() is to act as the brains of an AI driven player, *bot*. It basically replaces what the human *does* in-Game. For example, a player views the world around him, discovers it. He then proceeds to find objectives such as other players,
and makes judgments such as is that player friend or foe, is he carrying a stronger weapon than mine, can I kill him, should I run, should I engage and so forth. Then, that human player makes a precise decision. Finally, that human player inputs the command as key strokes or mouse movements and clicks. This function presumes all of these tasks in the case of a bot player.

### 3.9.2 BotAI(): Structure

As figure 13 shows, the general operation of BotAI() is to iterate through all the connected bots. It does the following:

1. **First**: it reads the *messages* that belong to the currently iterating bot that were sent/stored for it by the `SendClientMessages()`, which contain a *snapshot* of that bot’s surroundings. The AAS system is consulted for path *find* costs to better evaluate a decision.

2. **Second**: From those messages, it learns the surroundings of that designated bot.

3. **Third**: It examines the private conditions of that iterating bot found inside its own personal data space.
4. **Forth:** It makes a decision for action; such as move, retreat, chase, fireweapon, jump, swim and so forth.

1. **Fifth:** It inputs commands as **key strokes** and/or **mouse clicks** and movements.
3.9.3 BotAI(): Function Characteristics

The amount of data writing into global variables here is relatively low; hence the possibility for hazards is anticipated to be relatively low. And, here is where the BotAI()'s processing ends. From here on, human and bot players become transparent to the engine where they will both be interpreted in exactly the same manner.

3.9.4 ClientThink(): Overview

ClientThink()'s main responsibility is to carry out client commands into the Virtual World while handling all the interactions that may occur between the designated client and everything else in the Virtual World. The interactions fall under three categories:

1. **client-client interactions** such as Fireweapon() at foe, and the consequently DoDamage() to that foe
2. **client-entity interactions** such as Pickup() weapons and items
3. **client-world interactions** such as Move() in the world

Then, most important aspect of ClientThink() is that it has the responsibility of updating both the data of an “Acting” client and the “Acted upon” object-client/entity/world - according to the consequent changes incurred from the previously carried out action.

**Note:** A client is basically a player that includes both humans and bots.
3.9.5 ClientThink: Structure

As figure 14 shows, the ClientThink() operates in the following flow:

1. ClientThink() reads the inputted key strokes made by the player, human and bot alike.

2. The key stroke is interpreted into its corresponding meaning. For example, the space key is set to jump; thus, if the space key was pressed, then, interpret it as the player is ordering for a jump action.
3. The engine executes the designated \textit{Think()} function that will carry out the action interpreted actions. For example, if \textbf{Fireweapon} key stroke was interpreted, and the player was carrying a rocket launcher, then, the \texttt{fireweapon()} of a rocket launcher will be executed. This is made possible by the use of function pointers that are interchanged based on the player’s conditions.

4. Within carrying out of the chosen action, such as fireweapon(), move() and so forth, the data of both the client and interacting object will be updated. figure15 shows the action of fireweapon().

\section*{3.9.6 ClientThink(): Function Characteristics}

In regards to parallelism, ClientThink() is characterized to be highly active in terms of data manipulation of globally shared and local data. Furthermore, because ClientThink() is responsible for calculating all the logic that happens consequent to entity interactions, it requires to read the required personal, and shared, data of those entities. Moreover, because it has to output the results of those interactions it has to update those data. Moreover, it has to perform all this at a high rate, which was made possible by the use of function pointers that are non-Parallelizable-C rules compliant. Therefore, hazardous situations are anticipated here.
3.9.7 SendClientMessages: Overview

The SendClientMessages() function is responsible for sending all the updates that happened to the surroundings of each client from the previous computations to that designated client.

5. "Why would a bot that exists solely inside the virtual world need Messages to learn of its surroundings when it could already access the
"world map directly?" The answer is so that bots would be subject to the same limitations—such as vision boundaries—as a human player would and respond accordingly; thus, not having any unfair advantages over human players, which creates a better Gaming experience.

3.9.8 SendClientMessages: Structure

The operation flow in which SendClientMessages() works is as follows:

1. A snapshot of the surroundings of the designated client in a 360 degree horizontal view is taken, which is subject to the same limitations a human view would, such as eyesight strength and obstacles hindering further eyesight ability.

2. The taken snapshot of the surroundings is built into a message.

3. The newly composed message is conveyed to the designated client; in the case that the client is a networked human player, that message is sent to him via the network; in the case the client is a bot (whose sole existence is inside the server), the messages are saved into a global variable where that bot will read the messages directly—during the BotAI() stage.

3.9.9 SendClientMessages: Characteristics

The general operation in SendClientMessages() is to first to view surroundings of the iterating client, then build a snapshot of surroundings into a message, last to convey
the message to the client - in case of the client being a human player send the message to client, and in case of a bot player copy the message into a global allocation for the bot to read from. This function seems to require container(s) to hold snapshots which may be hazardous if it is in a globally shared condition. Therefore, in regards to parallelism, SendClientMessages appears to have a mild potential for hazardous conditions.

3.10 Summary

This chapter analyzed the major structures of ioquake3 while forecasting the hazardous potential. It explained the history of ioquake3, and the important and attractive attributes that ioquake3 possess that made it a suitable benchmark for this research. This chapter also gave a deterministic decision on what areas of focus should be. Therefore, the situation was readied for following chapter that will cover the act of parallelizing ioquake3 on a multicore platform.
4.1 Introduction

In this chapter, the parallelizing of ioquake3 by using the OSCAR compiler for multicore platforms will be explained. As shown in chapter 2, the OSCAR compiler has been proven to be successful in extracting parallelism in numerous fields; such as multimedia, simulators and other applications [27]. This experiment is the first attempt at using the OSCAR compiler to parallelize a Video Game. Furthermore, as explained in chapter 1, parallel computing in the field of Video Games is relatively young, whereas only a few number of papers have been found that attempted to the research of parallel computing in Video Games. Similarly, no paper was found that examines parallelizing a Video Game using an automatic parallelizing compiler.

4.2 Related Works

A paper with the title “Behavior and Performance of Interactive Multiplayer Game Servers” has described the methodology and requirements in benchmarking Video Game servers were it thoroughly examined a Video Game called Quake [28]. The behavior and requirements of benchmarking Quake resembled benchmarking Online
A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore Platform

Transaction Processing Systems used for banking transactions. Furthermore, increasing the number of players from 16 to 100 without overloading the CPU was achievable. Consequently, the computational bottlenecks created by the increased number of users were both Game-related as well as network-related processing that was in about a one to one ratio. Since Game servers circulate only control information, network bandwidth was not an issue and all graphical operations were handled on the client-side. Furthermore, incoming and outgoing bandwidth per player were practically in the size of a few Kbytes. Thus, current network connections on game servers can support thousands of users. Similarly, memory requirements were not an issue for modern systems. The results of the experiment suggested that there is a lot of room for improving server performance at the microprocessor architecture level. Furthermore, scaling game servers beyond a few tens of players may require rethinking their internal architecture. The study is valuable as it provides insight on what parts of the server may have to be parallelized or are worth to look at.

A paper titled “Parallelization and Performance of Interactive Multiplayer Game Servers” investigated the parallelism and scalability of interactive and multiplayer game servers [29]. It did so by designing and implementing a parallel version of Quake by hand. This pioneering investigation of parallelism in Gaming engines, namely Quake, has found that scaling interactive multiplayer Games such as Quake to large number of players by using parallelism is a challenging task. The evaluation of the Quake parallel server’s behavior and scalability was made on a hyper-threaded quad SMP system. Moreover, the main bottlenecks were lock synchronization and high wait times that consisted of 35% of the total execution time. Finally, although server load increases
A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore Platform

super-linearly with the player count, the parallel server can support 25% more players than the sequential server. Scaling game servers to several hundreds or more remains a challenging task that may require rethinking many aspects of the internal architecture of this class of applications; where significant future improvements are possible by taking advantage of Game-specific knowledge.

Two other papers “Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server” focused on examining the difficulties in porting the parallel version of Quake from previous research to implement Transactional Memory and the eventual performance [30]. The results show that it is possible to run most transactions in that workload without falling back to catch-all schemes like irrevocability. Similarly, that despite that coarse granularity of the transactions used, an abort rate of less than 26% with 8 clients existed.

"QuakeTM: Parallelizing a Complex Sequential Application Using Transactional Memory" paper attempted to design a parallel version of Quake by hand that uses Transactional Memory completely from the original Quake [31]. The difficulties involved in achieving that and how much performance improvement could be achieved from this technology were investigated. Speedup of the fine grained lock version is 1.63 while the speedup of a TM implementation is 1.5 with eight threads.

4.3 OSCAR Compiler & the Programmer Relationship

OSCAR compiler excels at many parallelizing optimization aspects such as cache optimization and so forth. However, due to the vast range of computer application
structures, the OSCAR compiler is yet to have the ability to efficiently optimize all of such types. That is why human intervention is inevitably necessary in achieving higher performance. One example on this necessity is the impact of rewriting an application to abide by the Parallelizable-C rules has on the optimization capacity and parallelism the OSCAR compiler [13] will be able to extract.

Similarly, Video Games as applications may require special attention of the programmer because they rely heavily on unpredictable real-time user inputs. Furthermore, Video Games’ internal mechanisms are highly complex where a single job, such as firing a weapon, require a composite of many modules, such as ray-tracing, physics, and so forth- that are frequently accessed. Therefore, it can be anticipated that when it comes to automatically parallelizing a Video Game by using a parallelizing compiler, even if laying the responsibility of managing some of the global or local variables on the compiler, the highly interconnected relationships between many variables and functions that work together to create a complete module must be closely observed and managed accordingly. Therefore, not all variable management can be relayed to the parallelizing compiler; thus, the programmer must intervene to re-implement what is needed in the code to make the modules as a whole become more parallel-friendly for the OSCAR compiler to parallelize.

4.3.1 The Role of the OSCAR Compiler

When it comes to applications that are new and still untested on the OSCAR compiler- such as Video Games, the programmer has the fair amount of responsibility
towards the prevention of hazards. However, the extraction of Parallelism itself is the task of the OSCAR compiler completely; such as loop level parallelism, basic block level parallelism and so forth. Consequent to a successful parallelizing attempt, the OSCAR compiler has also the responsibility of power consumption reduction of the target application, which will be explained in the following chapter.

4.3.2 The Role of the Programmer

As mentioned in the previous section, the programmer basically has role of eliminating the hurdles that the OSCAR compiler cannot handle, and to produce a hazard-free & Parallelizable-C code. The major work of the programmer will be expressed in this paper with examples of some of the common difficulties found in Video Games, and how they were solved.

4.4 Parallelization Methodology

In this section the methodology that was carried out to parallelize ioquake3 will be explained. The steps for parallelizing ioquake3 were carried as follows:

1. **Profile program for areas of focus:** This step is shared by almost all optimization works similar to this one.

2. **Examine focused areas for optimization potential:** This analysis should be executed to reveal how much potential parallelism the target application contains.
3. **Discovering the hazards:** This and the following steps are the most intense areas of work for the researchers. It examines the code in utter detail to discover any potential hazards that may be caused by parallelism.

4. **Reworking the code:** This step is where the solutions for the discovered hazards are implemented while also rewriting the code to be Parallelizable-C compliant.

5. **Compile ioquake3 using OSCAR:** This step is where the revised and reworked ioquake3 program is compiled using the OSCAR compiler to extract parallelism, and consequently be power reduced.

6. **Analyze the performance newly compiled ioquake3 version(s) on a multicore platform:** In this step, the new version(s) of ioquake3 are executed on a multicore platform, the performance is examined.

   Step 6 is where conclusions of the experiment of the work are shown.

4.5 **Profile Program for Areas of Focus**

   As explained in the previous chapter, in the post-initialization area that ioquake3’s bottlenecks operate in an iterating fashion. In abstract, after the initialization of the Game session, the session loops continuously until a *quit* condition is met. During this continuous loop, the engine iterates through each connected client, and executes the three bottlenecks in concession for that iterating client.
while(!quit)
{
    foreach(client = clients)
    {
        BotAI(client);
        ClientThink(client);
        SendClientMessages(client);
    }
}

4.6 Examining focused areas for potential

In the parallel computing field, one of the areas that are known to be of high potential for parallelism is for loops that compute relatively high amounts of calculations. Therefore, if this would to be applied to ioquake3’s structure, it can be safely assumed that ioquake3 possess a high potential for parallel computing, figure 16 shows the major structure of the three bottlenecks.

- First Parallelizing Attempt

As a preliminary experiment, the program, ioquake3, in its original structure was compiled using the OSCAR compiler. This was conducted to examine how much performance enhancement could initially be achieved using the original code. Eventually, the Game performed at the same original speed.

After the OSCAR compiled code of ioquake3 was examined, it was discovered that the previous loop of the newly compiled code had the same sequential structure as the original code; thus, resulting in a sequential execution, which executes at the same speed as the original sequential code. Furthermore, the results of the examination showed that because of the existence of several hazards within the previous major loop,
the compiler was unable to salvage any extractable parallelism within it. Therefore, eliminating those hazards is highly essential for OSCAR compiler's ability in extracting parallelism from ioquake3.

4.7 Discovering the hazards & Reworking the code

Because “Discovering the hazards” and “Reworking the code” are strongly interconnected and complimentary to each other, for ease of reading they were grouped into one section.

In this step, ioquake3’s code will be thoroughly analyzed to discover any potential hazards producible by a parallelizing attempt, and then, suitable solutions for those hazards will be produced and implemented into the code while abiding by the Parallelizable-C rules. Unfortunately, due to the lack of proper debuggers equipped with parallel computing detailed recognition, the process done in this step was carried out in brute force style of old-school code reading, graphing by hand, executing, with the help of a debugger that can give at least a reading of variable values and calls stack held on concurrent threads. The debugger used here was of Microsoft Visual Studio’s 2008 Professional Edition [32], which supports OpenMP [33].

4.7.1 BotAI(): Hazards & Solutions

As shown in figure13, the general operation of BotAI() here is to:

1. Read Messages that belong to the iterating bot
A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore Platform

2. Know surroundings of that designated bot
3. Retrieve the private conditions of that bot
4. Make a decision for action
5. Input commands

Note: The amount of data writing into global variables here is relatively low; hence we anticipate low possibility of hazards.

- Relocating Read/Write Operations Outside of the Parallelized Area:

Read and write operations that are made from and to complex global data structures such as Linked-Lists can become highly corrupted when multiple accesses are executed concurrently, in parallel. An effective method to avoid data corruption that could be implemented in this situation is relocating those reads and writes operations to be outside of the parallelized area, and then execute them as a batch. This is most applicable when the costs of the read and write operations are cheap relative to the cost of that parallelizable area, where there is no feasible performance speedup gain from executing those read and write operations in parallel.

One notable example on this hazard and its fix is the chat feature that is very common in networked Gaming. In ioquake3, the bots are designed to have the ability to chat with other players, both humans and other bots. All chats are conveyed to everyone using chat messages. However, these messages should not to be mistaken with messages used for updating player surroundings. These chat messages are read/written from/to a global Linked-List that is shared by all players. To protect the consistency of that Linked-List when the encapsulating loop is parallelized, the read/write operations were
removed, re-implemented to be outside of that parallelized loop. Then, those read/write operation were structured to be executed as a batch. This technique avoids any potential corruptions.

As shown in figure 17, the read and write operations were relocated outside the parallelizable loop to avoid data race. The read/write operations were originally located after the execution of the critical path; therefore, they were relocated to be after the parallelizable loop.

- **Function Pointers:**

  Function pointers is an important concept in programming in general because it helps the programmer make the switching between many different types of executions.

  ```c
  //Parallelized For Loop
  foreach(client = clients)
  {
    BotAI(client);
  }

  //Batch write & read operations moved here.
  foreach(client = clients) //1
  {
    WriteChatMessages(client);
  }

  foreach(client = clients) //1
  {
    ReadChatMessages(client);
  }

  //float zero = 0.0/(3)
  BotAI(client_t *client)
  {
    float zero = 0.0;/(3)
    /*static*/int varX;/(2)
    ...
    //ReadChatMessages(client);/(1)
    //WriteChatMessages(client);/(1)
    ...
  }
  ```

  **Figure 17: Relocating Read/Write Operations Outside of the Parallelized Area**
much faster, which suits relatively light weight applications such as Video Games. Because of function pointers’ light weight aspect, they are heavily used within ioquake3. Therefore, because function pointers are against Parallelizable-C rules, they had to be resolved. Two approaches to this problem were implemented:

i. **Moving the Calling of the Function into the Calling Tree**

If a function pointer is calling to a function that is called throughout many areas and libraries within ioquake3, the better suited solution would be to transform the function pointer into a normal function call. This function call will be interpreted by OSCAR as a blackbox area called from a shared library. As shown in figure18, part 1 shows the original code where a function pointer called funcPtr0 will be used to call functions funcPtr1 and funcPtr2. To help OSCAR be able to analyze the code, funcPtr1() & funcPtr2() functions were added as normal library function calls into the original calling tree, area. OSCAR compiler will interpret them as blackboxes, as shown in part2.

ii. **Moving the Body of the Called Function into the Calling Tree**

If a function pointer is calling to a function that is called in a relatively manageable number of areas, the optimum solution here is to include the whole body of the code of the called functions by the function pointer into the calling tree; thus, they will be optimized by the OSCAR compiler as well. As also shown in figure18 part3, funcPtr1() & funcPtr2()’s bodies have been moved to the calling area of SomeFunc(); thus, the OSCAR compiler should now have full view of the calling scope. Implementing this gives the OSCAR compiler the ability to optimize the code of the functions originally called by the function pointers as needed.
- **Parallelizable-C: Local Static Variables**

Parallelizing compilers have difficulty analyzing static variables that are defined inside function scope. Therefore, such static variables were rewritten into automatic variables, while asserting the integrity of the program.

- **Parallelizable-C: Localize read-only global variables**

Similarly, read only global variables confused the compiler to be hazardous. Therefore, they were rewritten to become local read-only variables.
/Part 1: Original code
SomeFunction()
{
    int x = 0;
    //funcPtr0,1,2: function pointer
    //GameClientCommand() function will be called using gvm
    funcPtr0 = funcPtr1;
    funcPtr0( x );
    //do some work
    funcPtr0 = funcPtr2;
    funcPtr0( x );
}

//----

//Part 2: Blackbox approach
//funcPtr1() & funcPtr2() will be interpreted as a blackbox function by OSCAR
#include "funclib.h"
SomeFunction()
{
    funcPtr1( x );
    //do some work
    funcPtr2( x );
}

//----

//Part 3: Adding the function to the call tree approach
//funcPtr1,2() as a whole were placed into SomeFunction()'s compilation tree
funcPtr1( int x )
{
    //do calculations
}

funcPtr2( int x )
{
    //do calculations
}

SomeFunction()
{
    funcPtr1( x );
    //do some work
    funcPtr2( x );
}

Figure 18: Reworking Function Pointer
In summary, the BotAI() function exhibits a theoretically high level of parallelism. However, BotAI() contains a high amount of callback functions that rely on function pointers, which are anticipated to hinder the compiler from extracting optimum parallelism from this area.

### 4.7.2 ClientThink(): Hazards & Solutions

As mentioned in the previous chapter, the ClientThink() function is responsible for handling the logic computations for all the incurred interactions that happen between any two entities in the ioquake3’s virtual world. Furthermore, ClientThink() is also responsible for updating both the relative data pertaining both the “Acting” and the “Acted upon” entities- client/entity/world. Because all of those computations are carried out during the same iteration and unmonitored, ClientThink() is anticipated to be highly prone to race conditions.

- **Implementing Locks to Prevent Data Hazards**

  - **Locking the Access to Complex Data Structures**

    To prevent hazardous situations in contended, globally shared, complex data structures such as trees, an OpenMP[17] critical directives can be implemented to lock the read and write operations, and allow only one thread access at any time; thus avoiding any race conditions potentially caused by concurrent thread access to the same data piece. Those locks should be implemented in areas of low access frequency where they should not place any additional thread access wait times, and those areas are of rigid nature that disallows them from being reallocated smoothly.
Note: For ease of reading the implementation of an OpenMP critical directive to prevent data contentions amongst threads shall be referred to as a lock throughout the remainder of this paper.

An example of a contended, globally shared, complex data structure is the World Map Area Tree, which represents the map that the players populate. As previously explain, during the initialization stage, this World Map is loaded, then based on a specific division algorithm it is split into area nodes then are composed into the tree leaves. Next, the Game engine maps clients into this Area Tree representation based on their current locations within the map. When a client executes a move() operation, and leaves an area, a leaf they resided into another, the engine remaps the client into their new area. This remapping operation requires a dual set of Link() and Unlink() operations, as shown in figure19.

To prevent this Area Tree from becoming corrupted by multiple concurrent remaps, links and unlinks, the Link() and Unlink() functions were locked.

ii. Locking Illegal Private Data Access Amongst Threads

Another type of data hazards that can be remedied with locks is functions where the executing thread has unmonitored access to private data of another thread. Having the potential for concurrent read/write situations when parallelized, this engine structure may lead to race conditions for the same data area when parallelized.

An example of this engine structure is FireWeapon() that executes the action of firing weapons of the iterating client and applying the damage on the spot to the hit
target. Therefore, if more than one client Fires a weapon at the same target, both clients could be applying the damage to the same target concurrently, which could lead to a hazardous condition. Therefore, to avoid such hazards a lock access to FireWeapon() was implemented, as shown in figure20. Because a variable called player health must remain unchanged throughout the execution and only should be over written at the end of the function Fireweapon(). The code in figure21 shows the implementation of the lock that was required to be placed at the entry of the function.

- Preventing Hazards by Transforming Memory Allocation from Temporary to Permanent

Figure 19: The Link & Unlink operation
Temporary allocation and deallocation of memory resources are potential for hazardous conditions if multiple occurrences happen concurrently. Transforming temporary memory allocations to one-time permanent allocations eliminates the need for deallocations, and by locking the resulting one time allocation processes, such hazards could be avoided.

An example of this is the action of dropping weapons upon client death. A dying client requires temporary memory allocation to drop the weapon they were holding last into the world. The dropped weapon is temporarily assigned an allocation from a shared
A Parallel Approach, Hazards, and Solutions for ioquake3 on a Multicore Platform

memory pool, and then returned when the pool becomes empty and the allocation is no longer in-use by the client. Otherwise, the memory allocation remains with the client during the session duration.

To prevent any hazardous situations from occurring, first the memory pool size was increased to the size that eliminates the need for any deallocations; thus, all first time allocations become permanent ones. Then, to prevent concurrent allocations those first-time allocations were locked. This implemented technique prevents hazards with the small cost of additional memory that systems nowadays have abundance of.

In summary, the ClientThink() function is a complex area to Parallelize because it deals with processing the logic of interactions that occur between two (or more) entities (world, players, entities), and computes the changes and updating private and global data on the spot.

```c
void ClientEvents( client_t *client ) {
    int event;
    foreach ( event = client->events) {
        switch ( event ) {
            case EV_FIRE_WEAPON:
                //Restricts access to one instance at a time
                #pragma omp critical
                {
                    FireWeapon( client );
                }
                break;
        }
    }
}
```

Figure 21: Pseudo code hazard prevention in fireweapon() by using Locks
4.7.3 **SendClientMessages: Hazards & Solutions**

The general operation in SendClientMessages() is to:

1. View surroundings of the iterating client
2. Build a snapshot of surroundings into a message
3. Convey the message to the client - in case of the client being a human player send the message to client, and in case of a bot player copy the message into a global allocation for the bot to read from.

This function requires container(s) to hold snapshots which may be hazardous if it is in a shared condition.

- **Transforming Global Variables into Localized Variables**

One method implemented in this work to avoid race over globally shared variables is to transform the globally shared variable into a localized thread data.

For example, gSnapshotEntities is a globally shared variable that simulates a camera. gSnapshotEntities holds the IDs of entities that will be built into a snapshot of the surrounding entities' locations and movements. Therefore, if two or more clients need their snapshots to be built simultaneously, clients may race to use gSnapshotEntities, the engine’s single camera.

gSnapshotEntities was replaced with lSnapshotEntities, which was implemented as a variable into the client’s local data structure - a personal camera; thus, snapshots can be safely built into the new local variable belonging to the designated client; thus
avoiding any potential race conditions. To increase memory efficiency, the new structure of lSnapshotEntities was also implemented to be lighter weight.

- **Replicating The tasks of Global Counters by the Use of Local Variables**

  Global one-dimensional counters were originally implemented to regulate tasks, such as preventing duplicates in a list by acting as a unique tag for each newly created list. The tag is stamped on each item entering that list to indicate that the stamped item has been added to that list. This global counter is incremented with each new iteration where a new list is created. Therefore, this action may cause race hazards if more than one list is being created concurrently.

  As shown in the right-hand-side of the first line in the loop of figure, a case of this was gSnapshotC that acted as a unique id number for each built snapshot. gSnapshotC was copied, stamped, into a snapshotted client’s snapshotID variable space and then incremented during that iteration. This unique gSnapshotC ID was originally implemented to prevent the same client from being included into the same snapshot more than once, to prevent list duplicates.

  The method of hazard prevention implemented here was by replacing the global variable gSnapshotC with a local variable that is unique in value amongst all clients such as the client’s unique ID - clientID. Next, the now obsolete gSnapshotC was deleted. Then, the unique clientID number of the iterating client was copied into the snapshotID
in place of the deleted gSnapshotC; thus eliminating any possibility of race, as shown in the bottom line of that loop in figure 22.

- **Transforming Tag Containers from 1-Dimensional size into Per-Thread Size**

Complimentary to the task in the previous hazard, list duplicates prevention, a space for a unique listID, a stamp space, is required. When an item enters into more than a single list at a time, it must acquire a stamp per list; thus one stamp space is insufficient for such situations. Therefore, adding more stamp spaces equal to the size of the number of valid lists per-time is a proper solution.

Again as shown in the left-side of the top line in the loop of figure 21, a client that is integrated into a snapshot uses the local snapshotID to hold the unique snapshotID tag - previously gSnapshotC but was transformed, clientID- in the sequential state. If that client is to be built into more than one snapshot concurrently, it requires a snapshotID container per concurrent snapshot built.

Therefore, the client’s snapshotID was re-implemented from a 1-dimentional integer variable to an array of integers with size equal to the number of simultaneously

```cpp
// Parallelized For Loop
foreach(client = clients)
{
    // snapshotClient->snapshotID = gSnapshotC++;
    snapshotClient->snapshotIDArr[omp_get_thread_num()] = client->ID;
}
```

Figure 22: The redesign of SnapshotEntities
running threads, while making all the necessary modifications to preserve the integrity of the program as shown on the left-side of the bottom line of that loop in figure 22. 

omp_get_thread_num() is a function from the OpenMP library that returns the thread number that is currently executing.

- **Parallelizable-C: Array[Array[struct_t]]**

Based on the Parallelizable-C rules, Array[Array[struct_t]] is a data structure that is difficult to analyze when attempting to extract parallelism. Therefore, all such structures were re-implemented to be compliant to the Parallelizable-C rules, such as the format of Array[struct_t], while maintaining that the integrity of the program is not be broken.

*In summary*, the SendClientMessages() function maintains its operations, and monitors its sanity via the use of Global Variables. To transform SendClientMessages to be parallelizable those hazardous global variables and their relative operations were privatized and/or localized. Furthermore, those modifications were made possible without the aid of any parallel locks. Similarly, SendClientMessages() does not rely on function pointers as much as the previous bottlenecks do. Therefore, SendClientMessages() should now exhibit high level of parallelism.

This concludes the *hazards & solutions* part of this paper where the difficulties in parallelizing an originally sequential code were discussed and explained.
4.8 Hazard Summary

One main goal of this research is to discover the type of hazards found in multiplayer Video Games when considering parallelism through a parallelizing compiler. In this section, a compilation of the discovered hazards and solutions will be summarized. The results of the examination in the previous chapter, the hazard analysis, solution discovery, and implementation in this chapter revealed important information regarding the major programming structures found in ioquake3 that hinder optimization attempts by using the OSCAR compiler. The major of hazards fell under one of the following:

4.8.1 Reading/Writing to Global Variables

As the title suggests, the existence of a global variable that is being read/written to multiple threads unmonitored can lead to hazardous situations.

- Solution(s)
  1- As implemented in the hazard found in BotAI()'s WriteChatMessages() & ReadChatMessages(), the reallocation of the accessing functions that are light weight to be outside of the desired parallelizable area, then, those relocated functions can be executed as a batch can be a sufficient solution.
  2- As implemented in the hazard found in ClientThink()'s Link/Unlink() functions that accessed the WorldMap tree, a proper lock at the least costly locations should help prevent any hazardous conditions.
4.8.2 Function Pointers

Function pointers are not considered hazards in themselves, but they do hinder the analysis by the OSCAR compiler. Therefore, they had to be dealt with accordingly.

- Solution(s)
  1- Re-implementing function pointers to be regular function calls from an outside library should allow the OSCAR compiler the ability to analyze the calling area.
  2- Re-implementing the whole body of the function that is called by the function pointer into the calling area allows the OSCAR compiler the ability to analyze both the original calling area and the newly included functions.

4.8.3 Thread-to-Thread Illegal Data Access

In ioquake3, clients had unmonitored and complete access to other clients’ private data; both reading and writing were permissible. This direct access can be the cause for hazards when more than one client attempts to access (excluding concurrent reads) the same client’s data. This situation was seen in the FireWeapon() function.

- Solution(s)
  1- Locking the direct access to the private data of the clients by other clients prevents any potential hazards. This sort of solutions is must suitable for areas that if locked, do not place significant additional waiting time in the program’s overall execution.
4.8.4 Shared Memory Pools

In ioquake3 there was one hazard caused by the lack of sufficient memory allocations that were accessible by more than client. If more than one client, thread, would race at a remaining single memory allocation they would cause a race condition. This was found in the die() function that was responsible for executing player death sequences.

- Solution(s)
  1- Increasing the size of the memory pool to provide permanent allocations for all needing clients avoids the need to return any memory allocations back to the pool. This results in all first-time memory assignments becoming permanent assignments throughout the length of the Game; thus avoids any race conditions.

4.8.5 Interconnected Variables & functions in a Module

This is a situation where a combination of several global variables and functions interconnect to create a complete module. If multiple threads attempt to execute this singular module with its global variables, it may cause hazardous conditions. An example of such situations is visible in SendClientMessages() where a composite of many variables and functions work together to create the action of taking snapshots and sending them through the messaging system.

- Solution(s)
  - This unique situation is a combination of many different hazards; thus it is most favorable to address such situations as special cases. It is most preferable
4.9 Performance Analysis on a Multicore Platform

To measure the impact of the modifications made in this experiment, the performance of the parallelized ioquake3 was compared with the original, sequential version on two multi-core platforms. The measurements covered all three bottlenecks. Because the three bottlenecks comprise of over 90% of the total CPU load in a single Game session, the combined performance results of those three bottlenecks will be taken as an indicator as an overall performance change.

4.9.1 Sanity Check

To ensure program correctness post-OSCAR compilation, a sanity check was made. Every ioquake3 Game session produces a Game log dump. First, a log dump of an original ioquake3 Game session was taken. Next, another Game session log dump of ioquake3 that was revised and compiled by OSCAR was taken. Last, the two Game dumps were compared to discover any discrepancies. Since, both logs appeared identical, it was understood that OSCAR produced code without any oddities. Furthermore, as a further measurement for assurance, several sessions were played by the researchers to
investigate any existence of ill logic within the Game that was produced by OSCAR. And surely enough, the Game was responding soundly.

4.9.2 Evaluation Environment: IBM POWER5+

As shown in table2, because the IBM POWER5+ machine is of high specs, a slightly different approach was taken to extract further analytical details. All Game matches were executed using an increased engine maximum player number of 112. Bots and score limited where a bot earns a point for every kill it makes, and immediate respawn settings. Spawn, is the act of the engine placing a player into the virtual world in a session. Respawning is the act of spawning a player after death. With immediate respawning, the map should be occupied with 112 bots almost all of the session length, which should mean that the CPU computational load should always be at its highest. The measurements were made for five seconds, which should be enough to cover all

<table>
<thead>
<tr>
<th></th>
<th>IBM POWER5+</th>
<th>RPX</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Linux (modified version)</td>
</tr>
<tr>
<td>Frequency Transition Time in microseconds</td>
<td>Unmeasured</td>
<td>10,000 (unmodified) 50+ (modified)</td>
</tr>
<tr>
<td>CPU</td>
<td>8-cores, 1.5 GHz</td>
<td>4-cores, 648 MHz</td>
</tr>
<tr>
<td>Memory</td>
<td>16 GB</td>
<td>4 GB</td>
</tr>
<tr>
<td>Number of Bots</td>
<td>112</td>
<td>32</td>
</tr>
<tr>
<td>Time</td>
<td>5 seconds</td>
<td></td>
</tr>
<tr>
<td>Runs</td>
<td>The best out of 100 runs</td>
<td></td>
</tr>
<tr>
<td>Server State</td>
<td>Dedicated Server state</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The specification table of IBM POWER5+ and RPX machines
processing scenarios. To examine what influences performance, several session variations were created, which will be explained next.

The engine has total control over spawning locations. However, the order in which players are spawned can be controlled by the host user. Different spawning orders should yield different initial spawning locations, which should result in bots encountering a relatively different type of enemies in each order. To examine whether or not different enemies yield different bot computations two spawning orders were created, Spawning Order-A, and Spawning Order-B.

Another aspect that was taken into consideration was map structure. To examine

Figure 23: A snapshot of a single-layered map in ioquake3
if different map structures yield different bot computations two maps were included into the performance examination, map Q1dm3 that is a single layered map, and Q3dm3 that is a multi-layered map.

Three different setups were readied:

1. Spawning Order-A in Q1dm3, a single-layered map as in figure23, this shall be the performance baseline.
2. Spawning Order-B in Q1dm3 map, this setup is to investigate if different enemies influence bot computations.

Figure 24: A snapshot of a multi-layered map in ioquake3
3. Spawning Order-A in Q3dm3, a multi-layered map as in figure24, to investigate if map structure influences performance.

To avoid the OS influencing the measurements, each setup using the sequential and parallel implementations was executed 100 times, and then the fastest execution of each was chosen as the experimental result.

The IBM POWER5+ platform is equipped with 8-cores at clock-rate 1.5 GHz, 16 GB of RAM. Each processor core has access to 32+32 KB/core of L1, 1.9MB of L2 and 36MB of L3 dedicated cache. The gettimeofday() function from the time Linux C-library was implemented as the measuring instrument.

As shown in figure25, 26, and 27 the speedup measurements shows that the program scales fairly well with all three setups displaying an almost identical grades of speedup. The performance displayed a great amount of speedup at all number of cores, where the 1st, 2nd and 3rd setups at 8-cores achieved 4.3, 4.43 and 5.1 respectively.

Reasoning for the added performance in the third setup can be attributed to the change in map structure from single-layer, 1st & 2nd setup, to multi-layered, 3rd setup. In a multi-layered map the frequency of client interactions is less than the first two maps; thus, execution of clients interaction computing functions such as fireweapon() (locked area) becomes less than in the first two setups. Therefore, the 3rd setup has less lock induced waiting time in ClientThink(). This also can be seen in figure27 where ClientThink() in the 3rd setup outperforms the first two setups.
Furthermore, SendClientMessages() displayed a linear speedup, as shown in figure 25, 26 and 27. The lack for an access to a cache analyzer made it difficult to examine the definite reasoning for this behavior. However, it can be assumed that because SendClientMessages() abides by the Parallelizable-C rules more than ClientThink() and BotAI(), it exhibited a better performance. Furthermore, a frequently accessed global variable called level.gEntities that holds important entity data was called and accessed by all three bottlenecks. Therefore, there is a high possibility that level.gEntities was already in the cache when SendClientMessages() needed to access it; thus, no time was spent in retrieving it.

Further analysis of the results shows that the speedup does not step up from 6-cores to 7-cores in all three setups. This lack of added speedup at 7-cores can be associated with ClientThink() slightly underperforming at 7-cores, shown in the previous figure. Due to the lack of proper analytical tools it was difficult to identify the exact cause of this behavior. However, since different structures and different enemies did not influence this behavior, it might be related with a parallel aspect such as unfair load balancing. Details of this chapter about enhancing the performance of ioquake3 on IBM POWER 5+ by using the OSCAR compiler was published in the proceedings and presented at the 17th International Conference On Computer Games (CGAMES12) on July 30th 2012. Furthermore, a paper was published in the International Journal of Intelligent Games & Simulation (IJIGS) Volume 7 Number 1 on April 2013 with further extensive and detailed analysis of the issues in this chapter.
4.9.3 Evaluation Environment: RPX

The experimental results for RPX will be shown. The RPX machine is equipped with 4-cores at clock-rate 648 MHz, 2 GB RAM. Each processor core has access to 32+32 KB/core of dedicated cache. Therefore, because of the inherited difference between the two machine’s specifications a slightly more general approach was made into evaluating the speedup on RPX. On RPX, ioquake3’s overall performance of the base-line map was evaluated. Last, a Game session of 32 bots of the baseline setup using spawning Order-A in Q1dm3 map was the most suitable for RPX that gives the optimum amount of computing load. As shown in figure 28, the speedup measurements show that ioquake3 scaled relatively well with the base-line setup, displaying a 2 times speedup at 4-cores. The main reason for the different amounts of speed could be related to the following reason:

- Since the number of bots per session on IBM POWER5+ was 112 rather than 32 as in RPX, it can be understood that the load balancing between the two machines for four cores shall be different; 8 per core for RPX, and 28 per core on IBM POWER5+. This load difference in balancing shows that the amount of parallelism is visibly different, which in turn affected the degree of speedup between the two machines.

4.10 Summary

This chapter has described the experience of achieving enhanced performance in ioquake3 by the use of the OSCAR parallelizing compiler. The automatically parallelized
Figure 25: Performance results of Spawning Order-A in Q3dm1

Game by the compiler from the revised sequential ioquake3 was found to achieve a 5.1 faster performance at 8-threads than the original one on an 8-core IBM POWER 5+ platform, and 2 times speedup on 4-cores on RPX. According to the detailed results on IBM POWER5+, the areas of the program that were majorly modified to follow the Parallelizable-C rules and avoided lockage and SendClientMessages() exhibited the highest level of performance speedup.

Understanding Game code and the engine’s mechanics is a highly demanding process that takes hours and hours on end to accomplish. However, this acquired Game-specific knowledge of the code helped produce an impressive amount of speedup. Using this understanding of ioquake3 aided the researches to find ways to eliminate data contentions, and hazardous conditions, and also minimize the use of lockage. Therefore, these results are recommended to Game coders who have a fair understanding of parallel computing, but not necessarily experts in that field [29].
From this experiment, it has been understood that Video Games as applications are written to be highly resource efficient where implementing programming shortcuts is almost a “rule of thumb”. However, such programming techniques eventually resulted in contentions over global resources, which came to be the main cause for the hazards when parallelism is taken into consideration in ioquake3. Another cause of hazards was the result of illegal access to private data amongst threads.

Several effective methods for avoiding hazards that were caused by read/write operations from/to shared complex data structures that were hard to localize were found effective. For example, batch execution of the read/write operations outside the parallelized loop. Other hazardous areas required restructuring and re-implementing of the engine to avoid the hazardous contentions.
In ioquake3, the mechanisms of representing both bots and human players inside the engine highly resemble each other. Therefore, this work should be highly beneficial to understanding parallelism of human driven sessions as well. Experimenting with large numbers of human players is beyond the capabilities of this paper. However, since
SendClientMessages() scales well with human players [29], a high level of speedup should be expected in the view of the results from this experiment.

Finally, results from this paper should encourage more Gaming companies to open their Game code to the public domain. This should aid dedicated researchers and freelance programmers to investigate better ways in achieving higher performance from parallelism, and investigate other crucial Video Gaming aspects as well.
Chapter 5 A Power Reduction Approach for ioquake3 on a DFVS Platform

5.1 Introduction

In an industry where power becomes a decisive aspect in the success or failure of the product, power saving becomes a highly desirable feature to acquire. Handheld devices, solar power driven devices are some examples of such products that could highly benefit from this feature, and with the rise of green energy awareness the need for power saving has become ever more important.

5.2 Power Reduction Methodology

The methodology that was carried out to reduce the power of ioquake3 is as follows:

1. Examine the results of the parallelized ioquake3 to determine power reduction potential.
2. Analyze the ioquake3 engine’s structure to choose the most suitable OSCAR power reduction method to implement.
3. Adding OSCAR directives into ioquake3 code
4. Compile the new ioquake3 using OSCAR and measure performance
5.3 OSCAR Power API Overview

In this experiment, RPX CPU’s frequency control is done through the use of the OSCAR Power Control API, which mainly includes the fvcontrol directive. The fvcontrol directive has the ability to set the frequency of a CPU unit to a range of full, half, quarter, and an eighth, which corresponds to 648, 324, 162, and 81 MHz. The change in frequency in turn reduces the used voltages; thus reduces the amount of consumed power. Similarly, get_time() function from the Time API is used to retrieve the current time from the system for inter-core synchronization. When using the fvcontrol function, the power status notation used an integer value that ranges from 12.5 to 100. The values from 100 to 12.5 represent the percentages of clock frequency of the specified module where 100 is the maximum clock frequency of 648 MHz.

5.4 Determining Power Reduction Potential

In theory, enhancing the performance of that application by the use of parallel computer should open the door for reducing the power consumption of it. A simple example that could show proof of this is the following. As shown in example in figure29, the original and sequential ioquake3 runs at 30 fps on a 500 MHz CPU, while a parallelized version of ioquake3 can run at twice that speed when using 2-cores for that same speed CPU. A mathematical representation would be:

1. 30fps : 1CPU @500 MHz
Figure 29: How computational slack is generated when ioquake3 is parallelized

2. 60fps : 2CPUs@500MHz

Therefore,

3. \( \frac{60\text{fps} : 2\text{CPUs@500MHz}}{2} = 30\text{fps} : 2\text{CPUs@250MHz} \)

If we assume that the power consumption is as follows

- CPU@500 MHz > CPU@250 MHz

Then, theoretically, the power consumption of a parallelized version of ioquake3, or any application in that regard, can be reduced accordingly.

Applying this theory to ioquake3 that performed on RPX, the parallel version of ioquake3 for 4-cores on RPX was able to achieve 2 times speedup, which means that ioquake3’s performance could be represented in the following representation:

- Performance in time:
A Power Reduction Approach for ioquake3 on a DFVS Platform

Figure 30: An example of how frames compose the overall Game loop

\[ \text{ioquake3@1-core} = \text{ioquake3@2-cores} \times 2 \]

- Frame rate analysis showed that the above equation could be represented as:

\[ \text{ioquake3@1-cores:30fps} = (\text{ioquake3@2-cores:60fps}) \times 2 \]

- Hence, for RPX

\[ 30\text{fps} : 2\text{CPUs@324MHz} = 60\text{fps} : 2\text{CPUs@648MHz} / 2 \]

Therefore, it is possible to assume that the power consumption of ioquake3 could be achieved. Since RPX is the only machine with DVFS features available to the researchers, it shall be the target platform for the power reduction attempts.

### 5.5 Choosing the Proper OSCAR Method

As explained in chapter3, Video Games are designed using a mechanism that resembled pictures composing a film- shown in figure30, with a single Game frame representing a single picture. The actions that occur within a single Game frame must be computed completely before the rendering of it at the end of the time frame limit. For
example, if a Game is running at 30 frames per second (fps), then it means that there are 30 frames in a single second. Furthermore, a single Game frame has a time frame of:

- $1 \text{ second} / 30 = 0.0333 \text{ second per frame}$

Therefore, this means that the actions that occur are computed each 0.033 seconds, and rendered at that rate as well. If the computations are too much for the CPU to be completed before the end of that time frame, that Game frame will be dropped by the ioquake3 engine. This time frame is basically a natural deadline for the CPU to finish all of the relative computations. This deadline is highly critical in determining which OSCAR power reduction method must be chosen. Now since the deadline mechanic is the guideline of ioquake3’s engine has been established, OSCAR power reducing methods will be examined to understand which is most fitting.

As mentioned in chapter2, the OSCAR compiler offers two methods in in reducing the power consumption of a computer application:

1. **Minimum Time Execution**: Exploiting idle times that exist between processor’s synchronization points.

2. **Satisfaction of Real-Time Deadline**: Exploiting idle times that exist between the end of an executed operation and a given deadline.

From those two methods above, the second one, satisfaction of real-time deadline seems to be perfectly suitable for the Game frame mechanism; thus, deadline method was chosen for this reducing the power consumption of ioquake3.
5.6 Adding OSCAR Directives into ioquake3

To help OSCAR optimize the power consumption of ioquake3 properly it is highly advised to add power reduction directives into the ioquake3 code prior to OSCAR compilation. These power directives inform the OSCAR compiler on the requirements of each area so that OSCAR may schedule the frequency power switching most optimally. The OSCAR directives are inserted as shown in figure31.

OSCAR’s power directives require the designated area’s clock consumption during a sequential run. Therefore, first, ioquake3’s critical bottlenecks clock consumptions were measured accordingly. Second, the collected clock measurements were written with the newly inserted OSCAR power directives into ioquake3 similar to the example in figure31.

5.7 Environment and Results

It is highly important to note that there were several customizations made to the environment and the compiler before usage.
void gameFrame( int frameMsec )
{
    #pragma mc LOW_POWER_DEADLINE 30000, LAYER
    // to encapsulate all functions under the same OSCAR power directive
    for (hmdf = 0; hmdf < 1; hmdf++) {
        #pragma mc COST 5000 BLOCK
        BotAI(){frameMsec};
        #pragma mc COST 5000 BLOCK
        // to encapsulate two functions under the same OSCAR power directive
        if(1==1)
        {
            VM_Call(GAME_RUN_FRAME, frameMsec);
            ClientThink( i );
        }
        #pragma mc COST 5000 BLOCK
        SendClientMessages();
    }
}

Figure 31: Abstract ioquake3 code with power directives inserted into it

5.7.1 Modified Kernel

One important customization was that the RPX’s Linux was replaced with a modified version [34] that has far much quicker frequency transition times. The first attempt at reducing the power consumption of ioquake3 for the original Linux version produced a version of ioquake3 that operated at full power. The reason why OSCAR could not optimize ioquake3 and reduce its power consumption was because the original version of Linux that was installed on RPX had high frequency transition times of 10,000 microseconds. Therefore, a modified version of Linux was installed instead. The modified version had frequency transition times of 50+ microseconds. The comparison between the two versions can be easily seen in the following reasoning:

- ioquake3 runs at 30 fps = a single ioquake3 frame requires 0.03 seconds
- Switching frequency = 0.01 seconds
A Power Reduction Approach for ioquake3 on a DFVS Platform

- Minimum number of frequency switching per frame = post + pre computation = 2
- Minimum frequency switching time per frame = $0.01 \times 2 = 0.02$ seconds
- Computation time = frame time – frequency switching time
- Computation time = $0.03 - 0.02 = 0.01$ seconds

Based on the above deduction, the CPU has only 0.01 seconds to finish all ioquake3 computations for a single frame. Furthermore, the measurements showed ioquake3 on average requires over 0.018 seconds per frame, at most 0.027 seconds. Therefore, the original Linux is unsuitable for power reduction attempts for ioquake3.

5.7.2 Rebuilding OSCAR

After this change, replacing the operating system with a modified version of Linux, the OSCAR compiler itself had to be rebuilt with these new parameters so that it may be able to optimize properly for the modified target platform.

5.8 Performance Results

After the arrangement to the environment, ioquake3 was recompiled again, and OSCAR was able to produce code that was better optimized. Then, the newly produced code was tested on RPX machine, which is equipped with DVFS technology. On the one hand, figure32 shows the power consumption of RPX when it runs non-optimized ioquake3 version. It shows that RPX consumes 1.6 watts on average. On the other hand, figure33 shows the power consumption of RPX when it runs an optimized version of
ioquake3. It shows that that RPX consumes 1.25 watts on average. Therefore, OSCAR compiler was able to reduce the power consumption of ioquake3 to 73%. Details of this chapter about power reduction of ioquake3 on RPX by using the OSCAR compiler was published in a technical report that was presented at the IPSJ SIG on Computer Architecture on April 25th 2013.

5.9 Summary

This chapter described a power reducing scheme that was applied onto ioquake3 to be run on the target platform RPX that is equipped with DVFS capabilities. The power reduction was made possible through the successful parallelization attempt made in chapter 4 that reduced the computation time for ioquake3 on 4-cores to half; thus
allowing the frame rate of ioquake3 to be doubled from 30 to 60 fps. Doubling the frame rate opened the opportunity to run ioquake3 on half of the original RPX frequency of 648 MHz to 324. Therefore, running the parallelized ioquake3 for 4-cores on half speed of 324 MHz, ioquake3’s frame rate returns to the original frame rate of 30 fps, and the power consumption of RPX also becomes reduced to about 73%, from 1.6 watts to 1.25 watts, both on average.

This power reduction comes highly critical to handheld devices where the power resource is limited; such as smart phones; Nintendo DS; PSP and so forth. Furthermore, it was understood that in order to reduce the power consumption of RPX when it executes ioquake3 frequency transition time comes highly critical. This became clearly

![Figure 33: RPX's power consumption when executing ioquake3 with reduced power](image-url)
visible in the case of the performances of ioquake3 on the original Linux of transition time of 10,000 microseconds and the modified Linux with transition time of 50+ microseconds. On the one hand, the power consumption of RPX with the original Linux did not change regardless of the versions of ioquake3. On the other hand, the power consumption of RPX with the modified Linux was reduced to 73% with the optimized ioquake3 by the OSCAR compiler.
Chapter 6 Conclusions & Discussions

6.1 Summary of Results

This thesis proposes the automatic extraction of parallelism from a conventional Video Game by using an automatic parallelizing compiler called OSCAR.

The benchmark Video Game used in this experiment is called ioquake3 that is a sequentially written Video Game in C-language. The method that extracted parallelism and reduced power consumption of ioquake3 in the following steps:

Step1: Profile ioquake3 using a performance profiler called Visual Studio Performance Profiler to discover which areas are the most computational demanding. The profiler unveiled three main bottlenecks that consume over 90% of the CPU time.

Step2: Investigate the three discover bottlenecks’ overall structure to learn how much potential parallelism exists.

Step3: Deeply analyze the code of three bottlenecks to discover any potential hazards that may occur due to any parallelizing attempts. This step was conducted by reading the code thoroughly and with the help of a debugger that has the ability to recognize a fair amount of parallel computing, such as thread data and thread’s individual call stacks.
Conclusions & Discussions

**Step4:** Solutions for those hazards where engineered and implemented into ioquake3’s code. This step was implemented while also reworking ioquake3’s code to abide by the Parallelizable-C rules, which aid OSCAR compiler to extract parallelism further.

**Step5:** The newly modified ioquake3 was compiled on OSCAR compiler to extract parallelism. Then, the produced code was executed on two multicore platforms – IBM POWER5+ and RPX- to examine the impact of this attempt on the performance.

**Step6:** After a successful parallelizing result on RPX that is equipped with DVFS properties, ioquake3 was recompiled using OSCAR to produce a reduced power consuming version. To compile on OSCAR, the clock consumption of all three were measured and inserted as OSCAR directives into the code.

**Step7:** The newly produced code was executed on RPX equipped with a modified Linux kernel and the performance of ioquake3 was measured.

One of the core goals of this work is to investigate the types of hazards found in a benchmark Video Game such as ioquake3 that occur when a parallelizing is attempted using a parallelizing compiler such OSCAR compiler. Furthermore, this work also investigates how to solve the discovered hazards. A summary of the hazards and their solutions were explained in detail in chapter 4.

This thesis has evaluated the performance and the power consumption of ioquake3. Results of this thesis are summarized as follows.
6.1.1 Parallelizing ioquake3 for IBM POWER5+ & RPX

This thesis has evaluated the performance speedup of the proposed approach on 8-cores and 4-cores multicore RP-X using a Video Game ioquake3. This approach produced a 5.1 times faster version of ioquake3 on IBM POWER5+ on 8-cores. It also produced a 2 times faster version on RPX on 2-cores.

6.1.2 Power Reduction of ioquake3 for RPX

This thesis has also evaluated the power reduction performance of the proposed approach on RPX using 4-cores. The power consumption of RPX when executing the optimized ioquake3 was reduced to 73% from the original.

6.1.3 Benefits

Enhancing the performance of a Video Game such as ioquake3 opens the opportunity for many new choices for the programmers to improve the experience of the Gamers. **Gamer-side** benefits as like:

- Adding complexity to the areas that the main CPU is responsible for such as more advanced AI engines that would give more challenging Game play for the human players.
- Improving the physics engine that is responsible for replicating the physical reactions from our own world into the virtual world of the Game to create a more intriguing experience.
Conclusions & Discussions

- Allowing more computational space for creating larger maps that would give more strategic Gameplay for the players.

- Enhancing the performance of ioquake3 reduces the processing requirements. Therefore, this should help ioquake3 to be ported to smaller platforms such as handheld devices.

Furthermore, enhancing the performance of a Game like ioquake3 that has server-side functions may give many benefits as well. **Server-side** benefits are like:

- With enhanced performance of ioquake3, the number of connectable players to a single session can be increased; thus, resulting in many more Gameplay options for larger scale Games with new objectives and so forth. Example of varying Gameplay can be seen in Games like Battlefield3 [35] and Halo3 [36] that allow for 64 players joining a single Gaming session, and producing a much more intriguing Gameplay.

- Enhancing the performance of ioquake3 can also allow for reducing the server requirements while still producing the same original performance. Reducing the server requirements helps in allowing for cheaper costs; thus opening the opportunity for more servers as well, even personal ones. This may have even an indirect impact of helping players to find their own local servers thus playing with much lower internet latency connections.

As for benefits of reducing the power consumption of a Game such ioquake3 allows for benefits such as:
Reducing the power consumption of ioquake3 would help to give a longer battery consumption time for handheld devices, and cheaper electric bills for home console machines.

### 6.2 Potential Work

This thesis has realized an automatic parallelization for a multiplier Video Game on a multicore platform.

Potential work is as follows:

- Test performance enhancement of this parallelization attempt for human-bot, and all human session.
- Invest alternate solutions for the extensive locks that were implemented in ClientThink() to produce a more parallel-friendly version of ioquake3.
- Rethink and root out the use of function pointers to create ioquake3 code that is more abiding by the Parallelizable-C rules.
- Examine the performance of the parallelized ioquake3 on actual Gaming platforms such as the Playstation.
- Implement a CPU *sleep* mechanism that drops the CPU power or frequency at a lowest into all idle areas of ioquake3 rather than just the idle that is in between frames; thus, reducing the power consumption even further.
- Explore the power reduction potential on other platforms than RPX that are equipped with DVFS or Power Gating capabilities.
Bibliography


Acknowledgements

First and foremost, the author would like to express his deep and sincere appreciation to Professor Hironori Kasahara of Waseda University for his continuous guidance and support during the whole span of this work. The author would like to express his generous thanks and gratitude towards Professor Keiji Kimura of Waseda University for his thoughtful advices and critical insight through the course of this work. The author would like to also give special thanks to Professor Kobayashi Tetsunori for his thorough and insightful guidance. The author would like to also express deep appreciation to Professor Narita, formally of Waseda University, for his constant support and encouragement that continued even beyond his retirement. The author would like to also give generous thanks to the Saudi Arabia Ministry of Higher Education for without their support this research could not have been possible. The author would like to also give special thanks to Mr. Akihiro Hayashi for his continuous support on many fronts during the length of this work. The author would like to also thank Mr. Keiichi Tabata for his help in understanding the parallelism of the OSCAR compiler, Yuki Furuyama and Dominic Hillenbrand for their support during the power saving part of this work. The author would like to also
deeply thank id Software for releasing the source code for Quake III: Arena to the public domain that made this research possible. The author would also like to thank the developers of ioquake3 for making their mod to the public domain, and for their support in answering many questions regarding the ioquake3’s engine and mechanisms. The author would also like to give thanks the anonymous reviewers on their insightful comments and coding explanations. Finally, the author would like to give his never ending thanks to his parents, Ibrahim Al-Dosary and Norah Al-Johar for their endless, boundless, and selfless support throughout the author’s work, and life time.
Publications

○ Enhancing the Performance of a Multiplayer Game by Using a Parallelizing Compiler, 17th International Conference On Computer Games (CGAMES12), July 30th 2012, Yasir ALDOSARY, Keiji KIMURA, Hironori KASAHARA, Seinosuke NARITA

○ Enhancing the Performance of a Multiplayer Game by Using a Parallelizing Compiler, International Journal of Intelligent Games & Simulation (IJIGS) Volume 7 Number 1, April 2013, Yasir ALDOSARY, Keiji KIMURA, Hironori KASAHARA, Seinosuke NARITA

Technical Report

○ Enhancing the Performance of a Multiplayer Game by Using a Parallelizing Compiler, IPSJ SIG on Computer Architecture, April
25th 2013, Yasir ALDOSARY, Dominic Hillenbrand, Yuuki Furuyama, Keiji KIMURA, Hironori KASAHARA, Seinosuke NARITA