Vol.2013-ARC-205 No.4
Vol.2013-OS-125 No.4
2013/4/25

Enhancing the Performance of a Multiplayer Game by Using a Parallelizing Compiler

Yasir I. M. Al-Dosary[†1] Yuki Furuyama[†1] Dominic Hillenbrand[†1] Keiji Kimura[†1] Hironori Kasahara[†1] Seinosuke Narita[†1]

Abstract—this paper investigates performance enhancement and the reduction of power consumption in Video Games when using parallelizing compilers and the difficulties involved in achieving that. This experiment conducts several stages in attempting to parallelize 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 version and low power version of ioquake3. Finally, the performance of the newly produced parallel and lower power versions of ioquake3 on a Multi-core platform are 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, (3) hazards are caused by thread contentions over globally shared data, and as well as thread private data, and (4) AI driven players are represented very similarly to Human players inside ioquake3 engine, (5) 70% of the costs of the experiment is spent in analyzing ioquake3 code, 30% in implementing the changes in the code.

Keywords: Video Games; Quake; ioquake3; parallel Computing; parallelizing compilers, OSCAR

1. INTRODUCTION

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 mobile phones. 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 [7].

To minimize the cost of implementing parallel programming while still achieving higher performance parallelizing compilers have been researched and developed. The main objective for parallelizing compilers [2] is to mask the complexities of parallelism from the programmer and produce high performance from an originally sequential program.

To our knowledge, no research has yet been conducted that studies parallelism in Video Games by using parallelizing compilers. As Video Games are available 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.

An important feature in Video Games is the AI, which is an integral part in the total Gaming experience Offline and Online. For example, in highly popular Games such as the *Halo* [8], *Call Of Duty* [9] FPS series, players can join forces together in Gaming sessions and complete missions against AI driven players. For ease of reading Game sessions shall be referred to as *sessions*, and AI driven players as *bots*.

Enhancing the performance of Game servers could allow for many benefits for developers, host and users alike. With enhanced performance, programmers could have more computing freedom to develop more advanced AI driven players, more intriguing Game mechanisms, larger and different Game styles with far more participants and complex objectives. Moreover, these enhancements should also lower server requirements which should lead to cheaper hardware costs on the hosts.

In this paper the potential performance enhancement and power consumption reduction of a sequentially written Video Game by the use of a parallelizing compiler 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 [3][4] which presents many of the important elements found in Video Games such as intelligent bots. First Person Shooters are Video Games that simulate human-like movement in a 3D world where players combat each other using artillery weapons, *Shooters*, while viewing the virtual world from the eyes of the controlled character, *First Person*.

The main contributions of this paper are (1) examining the source code of a popular multiplayer Game, ioquake3, from a view of a parallelizing compiler, then showing the modifications of several code fragments so that the compiler can exploit parallelism from the source code, (2) showing that the performance of the Game enhances with the increasing numbers of processor cores exploited by the compiler, and (3) investigating the difficulties in parallelizing sessions that are populated particularly by large numbers of bots, and (4) showing that the power consumption of the machine when executing the Game could be reduced by the compiler.

Finally, a hazard-free version of ioquake3 was successfully implemented, then, was compiled using the OSCAR [2,6] compiler to produce a parallel version of ioquake3. Then, the performance on a multi-core platform was analyzed, IBM

[†1] Department of Computer Science and Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan
{yasir, kimura}@kasahara.cs.waseda.ac.jp, {kasahara, narita}@waseda.jp
http://www.kasahara.cs.waseda.ac.jp/

POWER5+ [18], and RPX. The parallelized Game by the compiler from the revised sequential program of the Game was found to achieve a 5.1 and 2 faster performance at 8-threads and 4-threads on two different machines than the original ioquake3. Moreover, the power consumption was reduced to %73 of the original. Finally, the experience during this work is summarized. The rest of the paper is as follows. Section 2 mentions some of the main researches in this field that relate to this work. Section 3 presents a brief overview of the OSCAR compiler and Parallelizable-C [5]. Section 4 presents the methodology that was taken to achieve a parallelized ioquake3. Section 5 presents the performance results and analysis of this experiment. Finally, in Section 6 the conclusions are drawn.

2. Related Works

The methodology and requirements in benchmarking Video Game servers were thoroughly examined using a Video Game called Quake [11]. The behavior and requirements resembled benchmarking Online Transaction Processing Systems. Furthermore, increasing the number of players from 16 to 100 without overloading the CPU was possible. Consequently, the bottlenecks created by the additional users were both Game-related as well as network-related processing in about a 1:1 ratio.

The parallelism and scalability of interactive, multiplayer game servers was investigated by designing and implementing a parallel version of Quake by hand.[12] The pioneering investigation of parallelism in Gaming engines found that scaling interactive multiplayer Games such as Quake to large number of players by using parallelism is a challenging task. Moreover, the main bottlenecks were lock synchronization and high wait times where significant future improvements are possible by taking advantage of Game-specific knowledge.

The difficulties in porting a parallel version of Quake to implement Transactional Memory and the eventual performance were examined [13]. Another parallel version of Quake was designed by hand that uses Transactional Memory completely from the original Quake. The difficulties involved in achieving that and how much performance improvement could be achieved from this technology were investigated. [14]

3. OSCAR compiler And Parallelizable-C

OSCAR compiler [2,6] is a parallelizing compiler developed in Waseda University; it excels at enhancing the performance of a sequentially written C Program by extracting parallelism at the multigrain level and exploiting data locality. In this section, the process in which OSCAR is able to achieve performance enhancement with those techniques will be explained.

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 tasks**.

The OSCAR compiler begins by analyzing the sequential program and decomposes it into three types of Macro-Tasks (MTs); Basic Block (BB); Repetition Block (RB); Subroutine Block (SB). If there are parallelizable Loops they are decomposed into loops of smaller iterations as MTs- the number of iterations are determined by the original number of iterations and the number of Processor Cluster and Processor Elements.

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 macro-tasks. 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- figure1 shows an example on an MT G.

Finally, the OSCAR compiler assigns MTs to the targeted processor groups or processor cores by using either static or dynamic scheduling.

If several MTs share the same piece of data that is larger than the available 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.

One of the main difficulties in determining potential parallelism in a program is pointer analysis [7]. Parallelizable-C is a programming guideline to help automatic compilers perform pointer analysis precisely, and extract the most possible amount of parallelism from a sequential program. Parallelizable-C [5] is an accumulation of rules that guide the programmer while sheltering the programmer from the complexities of parallel tuning. Further details on OSCAR program optimizations. [6, 19, 10]

Some of the recently developed multicore platforms equipped with DFVS (Dynamic Voltage and Frequency Scaling) and power gating that can be controlled by the OS that is limited beyond the inner power status of a running application. However, the OSCAR compiler has achieved automatic power control schemes using DVFS and Power Gating for multicores that allow power control of an application from within by implementing two approaches; minimum time execution; satisfaction of real-time deadline.

After the MT scheduling phase, the power reduction algorithm determines the suitable voltage and frequency for each MT [reference]. The OSCAR compiler determines the execution time for each MT to minimize the program's overall energy consumption. Next, it chooses a critical path, the longest execution time needed for the MTG. The newly parallelized code must be produced while satisfying the designated deadline. 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.

In this experiment, the Power Control API is developed to control power through the modified version of the Linux kernel [1] that includes the fvcontrol directive. The fvcontrol directive sets the power status of a module to the specified value-get_time function from the Time API is used to retrieve the current time from the system for inter-core synchronization. The power status notation used in the Power Control API is an integer value ranging 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.

4. Methodology

In this section, the techniques implemented in creating a parallelized version of ioquake3 shall be explained.

4.1. Profiling

The first step in enhancing the performance of a computer application, such as ioquake3 is knowing which area of the program is critical to the overall performance. Those critical areas shall be referred to as *bottlenecks* throughout this paper. To learn what bottlenecks are created in bot sessions, a free-for-all (no teams) bots ioquake3 match in a medium to small sized map was profiled using Visual Studio Performance Profiler [17]. Smaller map should force more bot interactions, which should yield to a more intense processing situation. Furthermore, larger scale session was targeted in anticipation for its growing popularity in the industry; hence, ioquake3's engine limit was increased from the original Engine limit of 32 to 112 bots, which is the limit of our targeted machine. The Bots were a mixture of 8 types [15] that are of different personalities- different aggression levels, different weapon preferences and so forth. The bots were set to be at the highest level of difficulty that required more complex decision making computations; thus, more CPU intense. Therefore, this varying setup should cover many different computation patterns, which should yield a richer profiling result.

4.2. Profiling Results

The profiler showed the existence of 3 post-initialization bottlenecks inside the main Game loop that comprise of over 90% of the total CPU time with an almost equal distribution amongst them. The bottlenecks are BotAI(), ClientThink(), and SendClientMessages() that shall be explained later on in this section. In this paper, for ease of reading all functions() shall be written with brackets as such.

4.3. Program Code Analysis

a) Bottlenecks

In this section an overview of the general role each bottleneck has within the engine will be shown.

BotAI()

The BotAI() function takes on the role of the *brains* in bots where decisions are made. First, the BotAI() function, the *brains*, **views** other players and entities (such as items and weapons) around it using the *Messages* that were *built* for it by the SendClientMessages(). Then, the bot **makes** a logical **decision** of what **action** to take (pursue enemy and such) based on a combination of both; bot's surroundings (such as enemies), and personal conditions (such as health, ammo).

For a bot to recognize which *path* [15] to *move* and carry out its chosen action, it relies on the Area Awareness System (AAS)[15]. The AAS system contains the *World Map*, and all the routing costs for *moving* from one *area* in a map to another.

Finally, it **inputs** the desired **commands** exactly like a Human player (such as **left_key, aim_nozzle)** into its local command input data area. For ease of reading, a *Human* player shall be referred to with a first upper case letter.

From this point on, Humans and bots become transparent to the engine. They are simply *client* will be interpreted in exactly the same manner.

ClientThink()

ClientThink()'s main responsibility is to carry out client commands into the Virtual World while handling all the interactions that may occur between it and everything else in the Virtual World. The interactions fall under three categories: client-client; client-entity; client-world. Then, most importantly is that ClientThink() has the responsiblity of updating both the data of "Acting" client and the "Acted upon" object-client/entity/world.

SendClientMessages()

The *SendClientMessages()* function is responsible for sending all the updates that happened to the surroundings of each client from the previous computations to it. The process flow is as follows: First, a *snapshot* of the surroundings of the designated client in a 360 degree horizontal view is taken. Second, the taken *snapshot* of the surroundings is *built* into a *message*. Third, the newly composed *message* is conveyed to the designated client.

Bots communicate through messages so that they would be subject to the same limitations as a Human, and respond accordingly.

4.4. Could these Bottlenecks achieve reasonable parallelism?

As shown in listing1, the engine was implemented with major *For Loops* that iterate through each connected client and execute those three bottlenecks; *AI*, *Thinking*, and *Message Sending*. It is a common understanding that *for loops* which require relatively large CPU computations are potential for performance enhancing parallelism; thus potential parallelism.

4.5. First Parallelizing Attempt

As a preliminary experiment, the program in its original structure was compiled using the OSCAR compiler. 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 (listing1) of the newly compiled code has 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. The results of the examination showed that because of the existence of data dependencies within the previous major loop, the compiler was unable to salvage any extractable parallelism within it. Therefore, eliminating the hazards is highly essential for OSCAR compiler's ability in extracting parallelism from ioquake3.

4.6. Implementing Parallelism

This section is the core of this research where the main difficulties faced towards transforming a sequentially written ioquake3 into a parallelized state will be explained. Furthermore, how those difficulties were resolved to achieve parallelism shall be explained as well.

BotAI()

- ◆ **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 that parallelizable area, where there is no feasible performance speedup gain from those *read* and *write* operations.

```
while(!quit)
{
        foreach(client = clients)
        {
                BotAI(client);
                ClientThink(client);
                SendClientMessages(client);
        }
}
```

Listing1: An abstract view of bottleneck execution

For example, one common feature in networked Games is the *chat* feature. In ioquake3, the bots are designed to have the ability to chat with other players, Humans and other bots. All chats are conveyed using chat messages (not to be mistaken with *Messages* used for updating player surroundings). These chat messages are read/written from/to a global Linked-List. To protect the consistency of that Linked-List when the encapsulating loop is parallelized, the read/write operations were moved, re-implemented, outside of that parallelized loop, and structured to be executed as a batch. This technique avoids any potential corruptions.

The example in listing2 shows the write operation which was **relocated** outside the parallel loop to avoid data race. The *write* was originally located after the execution of the *critical path*; therefore it was **relocated** to be **after** the parallelizable loop -for the sake of space only the *write* operation was displayed.

◆ **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 were rewritten to become local since they confuse the compiler as being a race condition.

ClientThink()

◆ **Implementing Locks to Prevent Data Hazards**

i. *Locking the Access to Complex Data Structures*

To prevent hazardous situations in contended, globally shared, complex data structures such as Trees, an OpenMP[16] *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. For ease of reading the implementation of an OpenMP critical directive to prevent data contentions amongst threads shall be refered 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. 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 leavs. 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 Figure2.

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* are functions where the executing thread has unmonitored access to private data of another thread. Having the potential for concurrent *read/write* situations 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 target. Therefore, if more than one client *Fires a weapon* at the same target, both

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

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

BotAI(client_t *client)
{
        ...
        //WriteChatMessages(client);
        ...
}
```

Listing2: Hazard prevention in pseudo code

clients could be applying the damage concurrently, which could lead to a hazardous condition; thus a *lock* access to *FireWeapon()* was implemented, as shown in figure3.

Because a variable called *playerhealth* must remain unchaged through out the execution and only should be over written at the end of the function Fireweapon(). Therefore, the *lock* was required to be placed at the entry of the function, as shown in listing3.

◆ **Preventing Hazards by Transforming Memory Allocation from Temporary to Permenant**

Temporary allocation and dealloction of memory resources are potential for hazardous conditions if mutliple occurrances happen concurrently. Transforming temporary memory allocations to one-time permenant 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 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 occuring, first the memory pool size was increased to the size that eliminates the need for any deallocations; thus, all first time allocations

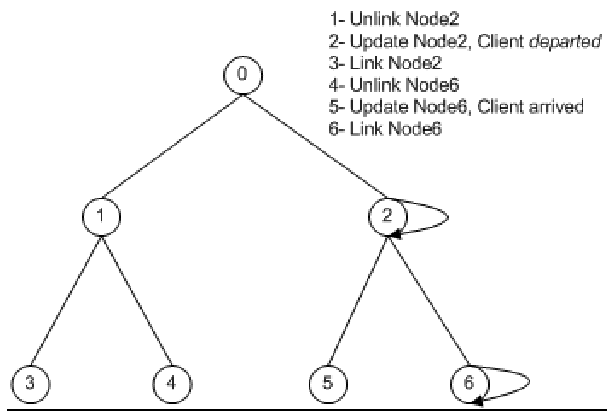
1- Unlink Node2
2- Update Node2, Client *departed*
3- Link Node2
4- Unlink Node6
5- Update Node6, Client arrived
6- Link Node6
0
1
2
3
4
5
6

Figure2: The Move()-Area Tree relationship

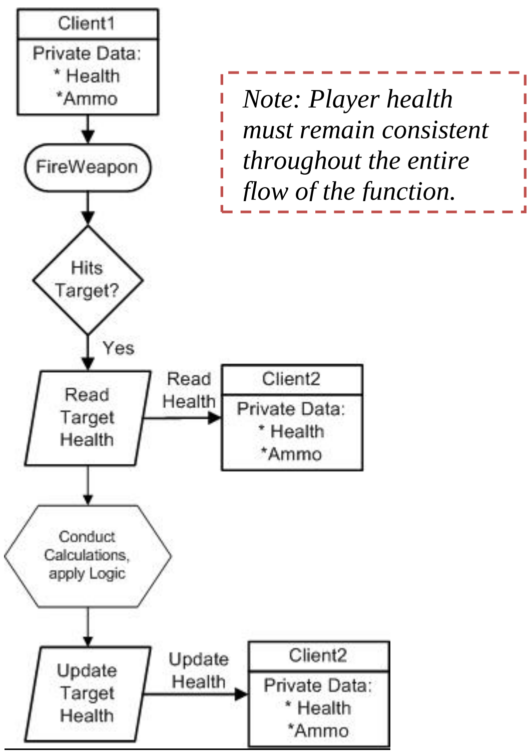

Figure3: A view of the task flow of FireWeapon()

become permenant. Then, to prevent concurrent allocations those first-time allocations were *locked*. This technique prevents hazards with the small cost of additional memory that systems nowadays have abundance of.

SendClientMessages()

◆ **Transforming Global Variables into Localized Variables**

One method implemented in this work to avoid race over globally shared variables is to transform the shared global 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, 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. The new structure of lSnapshotEntities was also implemented to be lighter weight, to increase memory efficiency.

◆ **Replicating Global Counters Jobs 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, and **stamped** on each item entering that list to indicate that *this* item have been **added** to *that* list. This global counter is incremented

```
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;
    }
  }
}
```

Listing3: Pseudo code hazard prevention by using Locks

with each iteration where a new list is created. Therefore, this action may cause race hazards when more than one list is being created concurrently.

As shown in the right-hand-side of the first line in the loop of listing4, 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 *snapshotID* variable space and then incremented. 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 clientID. Next, the now obsolete gSnapshotC was deleted. Then, the unique clientID number of the iterating client was copied into the snapshotID instead of the deleted gSnapshotC; thus eliminating any possibility of race, as shown in the bottom line of that loop in listing4.

◆ **Transforming Tag Containers from 1-Dimensional Into Per-Thread Size**

Complimentary to the task in the previous topic, 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. 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 listing4, a client that is built into a snapshot uses the local snapshotID to hold the unique snapshotID tag that was gSnapshotC in the sequential state, and later clientID. 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 running threads, while making the proper modifications to preserve the integrity of the program as shown on the left-side of the bottom line of that loop in listing4. In listing4, *omp_get_thread_num()* is a function from the OpenMP[16] library that returns the thread number that is currently executing.

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

Listing4: Hazard prevention in pseudo code

◆ **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.

4.7. Power Reduction

Multicore parallelism opens the opportunity for power reduction. In the case of reducing power by using the OSCAR compiler the target application must be examined to see which of the two power reduction schemes is most suitable, and measure the computational cost in CPU clocks. Video Games are implemented using a fundamental mechanism of "Game Frames", *Game Loop*, as a rule of thumb where the basic concept is similar to animating frames such as in movies. To give the appearance of motion for still pictures, frames are displayed at a certain rate per second; in ioquake3 it is 30 frames per second. Furthermore, the logic that happens within a single frame of 33 milliseconds must be calculated before the next frame is displayed. Therefore, this 33 millisecond is in other terms a deadline for the CPU to finish the processing load; thus, deadline power consumption scheme is most fitting for this experiment. Furthermore, the accumulative computational cost for the bottlenecks in ioquake3 is approximately 15000 CPU clocks per frame.

5. Performance Results

5.1 Speedup

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 a multi-core platform. 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 results will be taken as an indicator of the overall performance change.

- **IBM POWER5+**

All Game matches were executed using 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 load will always be at its highest. The measurements were made for 5 seconds, which should be enough to cover all 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.

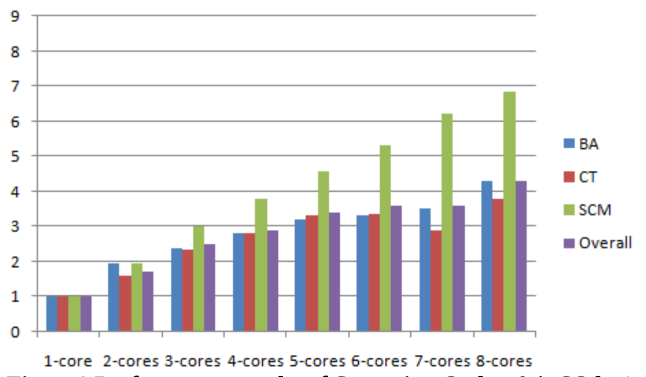

Figure4:Performance results of Spawning Order-A inQ3dm1

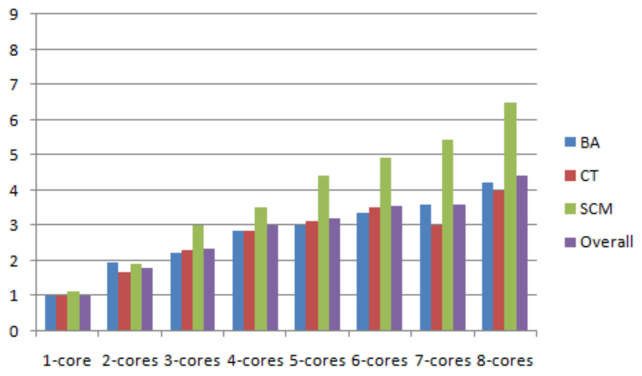

Figure5:Performance results of Spawning Order-B inQ3dm1

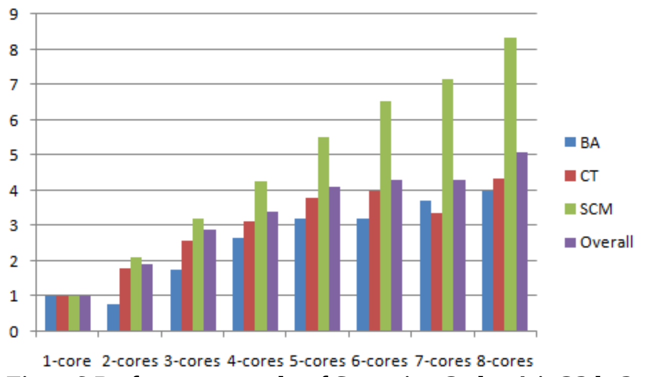

Figure6:Performance results of Spawning Order-A inQ3dm3

Another aspect that was taken into consideration was map structure. To examine 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, this shall be the performance baseline. 2) Spawning Order-B in Q1dm3 map, this setup is to investigate if different enemies influence bot computations. 3) Spawning Order-A in Q3dm3, a multi-layered map, to investigate if map structure influences performance.

Similarly, to avoid the OS influencing the measurements, each setup using the sequential and parallel implementations was executed 100 times, and then the fastest execution was chosen as the first experimental result. The experiment was conducted on an IBM POWER5+ platform, which 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 figure 4, 5 and 6, 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 figure6 where ClientThink() in the 3rd setup outperforms the first two setups.

Furthermore, SendClientMessages() displayed a linear speedup, as shown in Fig 4, 5 and 6. 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.

- **RPX**

Now the second experimental results will be shown, which were conducted on an RPX. 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 and the power consumption optimization of the base-line map were evaluated. Last, a Game session of 32 bots of the baseline setup was the most suitable for RPX that gives the optimum amount of computing load.

As shown in figure 7, the speedup measurements show that

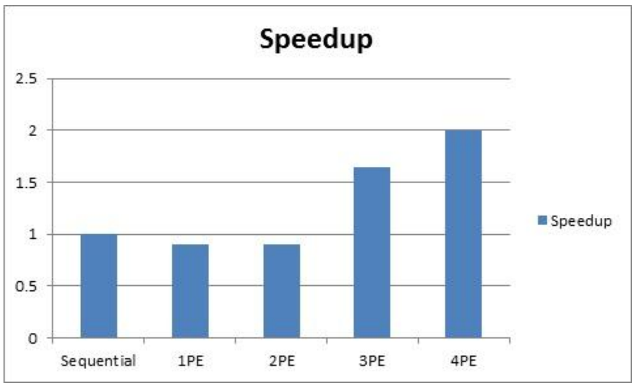

Figure7: Speedup results obtained on RPX

ioquake3 scaled relatively well with the base-line setup, displaying a 2 times speedup at 4-cores.

5.2 Power Reduction on RPX

On the one hand, figure 8 shows the power consumption of a 32 bot Game session of the sequential version on RPX. As the figure shows, RPX consumed in between 1.6 and 1.8 watts. On the other hand, figure 9 shows the power consumption of a 32 bot Game session of the compiled version of ioquake3 that is parallelized for 4-core and optimized for low power consumption. As the figure shows, RPX consumed in between 1.1 to 1.4 watts.

The results of the optimized low power consumption by OSCAR compiler for ioquake3 show a %73 reduction in power consumption on average.

6. CONCLUSIONS

This paper has described the experience of achieving enhanced performance and power reduction in ioquake3 by the use of the OSCAR parallelizing compiler. The autmatically parallelized Game by the compiler from the revised sequential program of the Game was found to achieve a 5.1 faster performance at 8-threads on an 8-core IBM POWER 5+ platform, and 2 times speedup using 4-threads on an 4-core RPX machine than the original. And consequently, It was also found that the OSCAR compiler could help reduce the power consumption by %27. 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. Moreover, this speedup in performance proves that taking advantage of Game-specific knowledge can greatly help reduce data contentions, and hazardous conditions, and with reduced *lockage* higher performance could be produced[13].

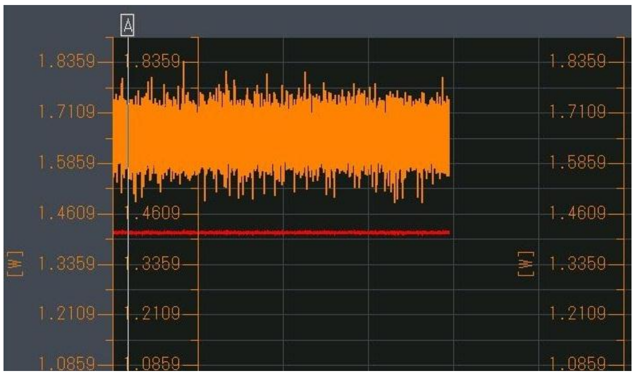

Figure8: Power consumption on RPX without power optimization

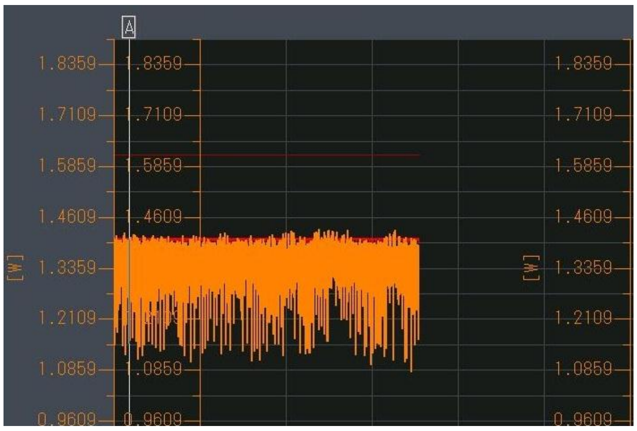

Figure9: Power consumption on RPX using power optimization

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 a shared complex data structures that were hard to localize were found effective. For example, batch excution 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 repereseting both the bot, and the Human player inside the engine highly resemble each other. Therefore, this work should be highly beneficial to understanding parallelism of Human driven sessions as well. Expirementing with large numbers of Human players is beyond the capabilities of this paper. However, since SendClientMessages() should scale well with Human players[12], a high level of speedup should be expected in the view of the 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 researchers to investigate better ways in achieving higher performance from parallelism and further reduce power consumption, and investigate other crucial Video Gaming aspects as well.

Acknowledgment

The authors would like to thank *id Software* for releasing the source code for *QuakeIII* to the public domain. The authors would also like to thank to the developers of *ioquake3* for making their Mod available to the public domain, and for their support in answering many questions. The authors are also thankful to the anonymous reviewers on their insightful comments. Special thanks to Mr. Akihiro Hayashi his support in this work.

References

[1] Opportunities and Challenges of Application-Power Control in the Age of Dark Silicon: IPSJ SIG Technical Report
[2] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, S. Narita: *A multi-grain parallelizing compilation scheme on oscar (Optimally Scheduled Advanced Multiprocessor).* : Proc. 4th Workshop on Language and Compilers for Parallel Computing, 1991
[3] id software: ioquake3. http://ioquake3.org, April 2012
[4] *QuakeIII:*http://www.idsoftware.com, April 2012
[5] Masayoshi Mase, Yuto Onozaki, Keiji Kimura, Hironori Kasahara: *parallelizable C and Its Performance on Low Power High Performance Multicore Processors.*: In: Proc. of 15th Workshop on Compilers for Parallel Computing, July 2010
[6] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, H. Kasahara: Hierarchical Parallelism Control for Multigrain Parallel Processing.: Prof. 15th Workshop on Language and Compilers for Parallel Computing, 2002
[7] Hind M.: *Pointer Analysis: Haven't We Solved This Problem Yet?* In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 54-61
[8] Microsoft: Halo; http://halo.xbox.com:(Apr 2012)
[9] ACTIVISION: Call Of Duty http://www.callofduty.com, Apr 2012
[10] Yasutaka Wada, Akihiro Hayashi, Takeshi Masuura, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, and Hironori Kasahara: *A Parallelizing Compiler Cooperative Heterogeneous Multicore Processor Architecture*: Transactions on High-Performance Embedded Architectures and Compilers IV,Lecture Note in Computer Science, Springer, Vol. 6760, pp.215-233, Nov. 2011.
[11] Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos: *Behavior and Performance of Interactive Multi-player Game Servers.*: ACM Cluster Computing Journal Volume 6 Issue 4, October 2003
[12] Ahmed Abdelkhalek, Angelos Bilas: *Parallelization and Performance of Interactive Multiplayer Game Servers.*: Parallel and Distributed Processing Symposium 18th International Proceedings, April 2004
[13] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adri´an Cristal: *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server.*: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, February 2009.
[14] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal: *QuakeTM: parallelizing a Complex Sequential Application Using Transactional Memory.*: Proceedings of the 23rd international conference on Supercomputing, June 2009
[15] Waveren, J.M.P. van.: *The Quake III Arena Bot,.* 2001.
[16] *The OpenMP® API specification for parallel programming.* http://openmp.org/wp, April 2012
[17] *Analyzing Application Performance by Using Profiling Tools* http://www.microsoft.com, April 2012
[18] *IBM eServer p5 550:* http://www.ibm.com, Apr 2012
[19] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, S. Narita: *OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicores and SMP Servers* : Lecture Notes in Computer Science, Springer, Vol. 5898, pp. 188-202, 2010