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

Yasir I. M. Al-Dosary, Keiji Kimura, Hironori Kasahara and Seinosuke Narita
Department of Computer Science and Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo, 169-8555, Japan

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 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 of ioquake3. Finally, the performance of the newly produced parallel version 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 5.1 faster performance at 8-threads than original one on an IBM Power 5+ machine that is equipped with 8-cores, and (2) hazards are caused by thread contentions over globally shared data, and as well as thread private data, and (3) 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 (4) 70% of the costs of the experiment is spent in analyzing ioquake3 code, 30% in implementing the changes in the code.

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

I. INTRODUCTION

Video Games have been a very popular form of digital entertainment that is presented on many different platforms. To deliver the most entertaining experience to the audience Video Gaming companies have always pursued the cutting edge in innovation and technology. [1][2]

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 [8].

To minimize the cost of implementing parallel programming while still achieving higher performance parallelizing compilers have been researched and developed. The concept of parallelizing compilers [3] is to mask the complexities of parallelism from the programmer and produce high performance from an originally sequential program.

Parallel computing in Video Games as a research field is relatively young, and is still rich for advancement. To our knowledge, no research has yet been conducted that studies parallelism in Video Games by using parallelizing compilers.

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* FPS series [9] and *Call Of Duty* [10] FPS series, players can join forces together in Gaming sessions and complete missions against AI driven players; Online and Offline. 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 and Game mechanisms, which should lead to a richer Gaming experience for the users. Furthermore, enhanced performance could also allow farther boarders in creating larger and different Game styles with far more participants and complex objective. Moreover, these enhancements should also lower server requirements which should lead to cheaper hardware costs on the hosts. Even with added players, bigger scale and more exciting Game styles, this enhancement should help maintain Game smoothness for the users.

In this paper the potential performance enhancement of sequentially written Video Games 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 Video Game called ioquake3 **[4][5]** which presents many of the important elements found in Video Game such as intelligent Bots. ioquake3 is an enhancement of the QuakeIII [18] Game,

which is an installment in the Quake *First Person Shooter* Game series; 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 fragment 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.

Finally, a hazard-free version of ioquake3 was successfully implemented, then, was compiled using the OSCAR [3,7] compiler to produce a parallel version of ioquake3. Then, the performance on a multi-core platform was analyzed, IBM POWER5+ [20]. The 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 than original one. Finally, the experience during this work is summarized. The rest of the paper is as follows. Section 2 mentions some of the main researches of this field that relate to this work. Section 3 presents a brief overview of the OSCAR compiler and Parallelizable-C [6]. 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.

II. RELATED WORKS

The methodology and requirements in benchmarking Video Game servers were thoroughly examined using a Video Game called Quake [12]. 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 50:50 ratio.

The parallelism and scalability of interactive, multiplayer game servers was investigated by designing and implementing a parallel version of Quake by hand.[13] 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 [14]. 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 was investigated.[15]

III. OSCAR COMPILER AND PARALLELIZABLE-C

OSCAR compiler [3,7] 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; Basic Block (BB); Repetition Block (RB); Subroutine Block (SB). If there are parallelizable Loops they are decomposed into loops of smaller iterations as macro tasks- 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 a MT is a subroutine call or a loop that has coarse grain task parallelism, the compiler generates inner MTs inside that MT hierarchically.

Finally, the OSCAR compiler assigns macro-tasks 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 [8]. 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 [6] 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 [7][21]

IV. METHODOLOGY

In this section the techniques implemented in creating a parallelized version of ioquake3 shall be explained. In Video Games, Client-side processes such as graphics rendering have already been deeply researched. In the ioquake3 engine AI computations and Game logic are executed completely on the server-side; thus, this research shall focus completely on server-side operations.

A. Profiling

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 [19]. The reason for choosing a relatively smaller map was to force more Bot interactions. Furthermore, larger scale sessions have recently been gaining popularity, such as *BattleField3* [11]; thus, to examine future potential growth ioquake3's engine limit was increased from 32 to 112 Bots. The Bots were a mixture of 8 types [16] 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; hence more complex decision making computations are required; thus, more CPU intense. Therefore, this setup and variation should allow for different computing outcomes, which should give a richer profiling result.

B. Profiling Results

A typical Game session begins by conducting all Game initializations, such as match time limit and so forth; afterwards, the main Game loop begins. The profiler showed the existence of 3 post-initializations bottlenecks 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.

C. Program Code Analysis

a) Session Flow Overview

First, the server starts up the session by retrieving the necessary Game options and map configurations. Then, the server manifests the virtual world with the proper information.

Now, the Match begins. The program spins in a continuous *loop* until an **End Game** condition is met; such as a player reaching the score limit. As mentioned earlier, over 90% of the total computational time is consumed by the three main bottlenecks found in this loop; BotAI(), ClientThink() and SendClientMessages().

b) Bottlenecks

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

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()- this function will be explained in the following sector.

Then, the Bot **makes a** logical **decision** of what **action** to take (pursue enemy, retreat, jump, fire weapon, pickup item and such) based on the combination of those surroundings and personal conditions (how much health it has; ammo amount; weapon type and so forth).

Furthermore, to carry out certain actions a Bot must *move* inside the *World*. For a Bot to recognize which *path* [16] to take, it relies on what is called Area Awareness System (AAS)[16]. The AAS system contains the *World Map*, and the routing costs.

Finally, when the Bot finishes making its decision, and chooses an action, it **inputs** the desired **commands** exactly like a Human player, such as **left_key, fire_weapon, jump_key, reload_weapon, change_weapon, aim_nozzle,** into its local command input data area. For ease of reading, a *Human* player shall be written with a first upper case letter.

And here is where the BotAI()'s task ends, and Human and Bots players become transparent to the engine; thus, they will both be handled in exactly the same manner by the engine.

ClientThink()

A Client is the *player*, which includes Humans and Bots. ClientThink()'s main responsibility is applying Client commands into the Virtual World while handling all the interactions that occur between the designated Client and everything else in the Virtual World; Client(player)-and-Client (Fireweapon() at foe, and consequently DoDamage()); Clients-and-Entities(such as weapons and items); Client-and-World(Move()). Most importantly is that it is also responsible for updating both the "Acting" Client and the "Acted upon" Client/Entity/World- detailed explanations will be presented in the following sector.

SendClientMessages()

This function is responsible for sending all the updates of the surroundings to each Client. First, a *snapshot* of the surroundings of a designated Client is *built* into a *message*. A *snapshot* is similar to a camera *snapshot* of the surroundings from the iterating Client's 360 degree horizontal view. Then, in the case of the networked Human players, that *message* is *sent* to the designated Client over the network. Furthermore, in the case of Bots (who can only exist inside the server), the *messages* are saved into a global variable. The Bots then *read* the *messages* directly- during BotAI().

- "Why would a Bot that exists *inside* the virtual world needs *Messages to learn of its surroundings*?":The answer is so that Bots would simulate the existence of a Human player and view the world with the same limitations, thus not having any unfair advantages over Human players.

4. Parallelism

A) Can these Bottlenecks achieve reasonable parallelism?

As shown in listing1, the engine was written to accommodate sessions with varying player numbers by implementing *For Loops* that iterate through each connected player and execute the given job; *AI*, *Thinking* and *Message Sending*. *For loops* that require relatively large CPU computations are potential for parallelism that may enhance the performance. Therefore, ioquake3 with these three CPU hungry bottlenecks may be assumed to have a fair amount of extractable parallelism.

B) First Parallelizing Attempt

As an initial experiment, the program 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 (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. This can be explained by that because of the existence of data dependencies within the previous loop, the compiler could not salvage any extractable parallelism amongst it. Therefore, eliminating the hazards is essential for the compiler to extract parallelism from ioquake3.

C) Implementing Parallelism

This is the core of this research where the difficulties that were found in parallelizing this sequential Game, ioquake3, and how they have been resolved to achieve parallelism shall be explained. However, for the sake of space only the major issues shall be mentioned.

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 corruptive when multiple accesses are made in parallel. An effective method to avoid corruption here is by relocating the reads and writes operations to be outside of the parallelized area, and then execute them as a batch. This is applicable when the costs of the read and write operations are cheap.

For example, the Bots chat with other players using chat messages (not to be mistaken with *Messages* used for updating player surroundings) that 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 outside of that parallelized loop, and executed as a batch. This technique avoids any potential corruptions. In listing2, the write operation is relocated outside the parallel loop to avoid data race. The *write* was originally located after the *critical path*; therefore it was relocated to be **after** the parallelized loop- for the sake of space only the write operation is displayed.

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

Listing 1: An abstract view of bottleneck execution

```
//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

◆ **Parallelizable-C: Local Static Variables**

Parallelizing compilers have difficulty in analyzing static variables that are defined inside function scope. Therefore, such static variables were rewritten into automatic variables.

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

Similarly, read only global variables were localized since they confuse the compiler as being a racy condition.

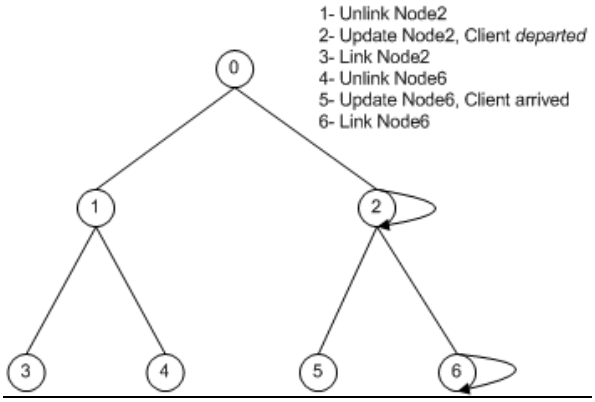

Figure 2: The Move()-Area Tree relationship

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[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. These *locks* were implemented in areas of low access frequency where they may not create any additional bottlenecks. For ease of reading the implementation of an OpenMP critical directive to prevent data contentions amongst threads shall be refered to as *lock* throughout the remainder of this paper.

An example of a globally contended shared and complex data structure is the *World Map Area Tree*, which represents the map that the players populate. The map is divided into area nodes that compose the tree leavs. The Game engine maps clients into this *Area Tree* based on their current locations within the map. When a Client executes a *Move()*, and leaves an area, the leaf, they resided into another, the engine remaps the Client into their new area that requires dual *Link()* and *Unlink()* operations, as shown in Fig.2.

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

ii. Locking Illegal Private Data Access Amongst Threads

Another type of data hazards that was remedied with *locks* were functions where the executing thread has unmonitored access to the private data of another thread. This engine structure may lead to racy 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 Clients will be applying the damage concurrently, which could lead to a hazardous condition; thus access to *FireWeapon() lock* was implemented, as shown in Fig.3.

Because a variable called *player health* must remain consistant through out the execution and only be over written at the end of the function, the *lock* was required 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 may become hazardous if mutliple occurrances happen concurrently. Transforming temporary memory allocations to one-time permenant allocations eliminates the need for deallocations, and by *locking* the one time allocation processes such hazards can be avoided.

An example of this is the action of *dropping weapons* upon Client death. *Dying* Clients require temporary memory allocation to drop their weapon into the *World*. The *dropped weapon* is temporarily assigned an allocation from a shared

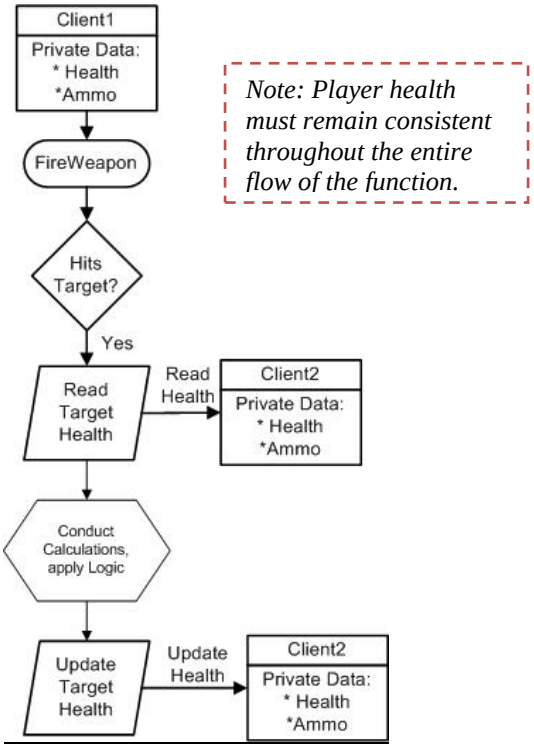

Figure 3: A view of the task flow of FireWeapon()

memory pool, and returned when the pool is empty, 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 become permenant. Then, 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 global variable- somewhat shared *camera*- that holds the IDs of Entities that will be **built** into a **snapshot** of the surrounding Entities' locations and movements; thus, if two or more Clients need their snapshots to be **built** simultaneously, Clients may *race*.

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 *race*. The new structure of lSnapshotEntities was also implemented to be lighter weight to increase memory use efficiency.

- **Replicating Global Counters Jobs by the Use of Local Variables**

```
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

Global one-dimensional counters were 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. This global counter is incremented with each iteration where a new list is created, which may cause race hazards when more than one list is being created in parallel in that iterating loop.

As shown in the right-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 where gSnapshotC was copied, **stamped**, into a **snapshotted** Client *snapshotID* variable space and then incremented. This unique *gSnapshotC* ID was 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 to replace the global variable gSnapshotC with a local variable that is unique in value. The 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**

A compliment to the task in the previous topic, when an item that will enter a list that prohibits duplicates requires a space for a unique listID, a **stamp** space. When this item enters int more than a single list at a time, it must acquire a stamp per list; thus one space is insufficient. Adding more 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, gSnapshotC 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, a **stamp** space. Therefore, the Client's variable space, *snapshot* 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 adjustment to preserve the integrity of program, as shown on the left-side of the bottom line of that loop in listing4. In

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

Listing4: Hazard prevention in pseudo code

listing4, *omp_get_thread_num()* is a function from the OpenMP[16] 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 for extracting parallelism. Therefore, all such structures have been 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 to not be broken.

V. PERFORMANCE RESULTS

To measure the impact of the experiment, the performances of the parallelized ioquake3 were compared with the sequential version on a multi-core platform. The measurements covered all three bottlenecks. Because they comprise of over 90% of the total Game session CPU load, the results can be taken as an indicator of the overall impact.

All Game matches were with 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 CPU load should 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 server administrator. Different spawning orders should yield different initial spawning locations, which should result in Bots encountering a relatively different type of enemies in each order. Therefore, two spawning orders were created, Spawning Order-A, and Spawning Order-B.

Three different setups were implemented: 1) Spawning Order-A in Q1dm3, a single-layered map, this shall be the performance baseline. 2) Spawning Order-B in Q1dm3 map, to investigate if different enemies influence Bot computations, hence performance. 3) Spawning Order-A in Q3dm3, a multi-layered map, to investigate if map structure influences performance.

Similarly, to avoid OS influencing the measurements, the best out of 100 runs of each session was chosen. The experiment was conducted on an IBM POWER5+ platform, which is equipped with 8-cores at clock-rate 1.5 GHz, 16

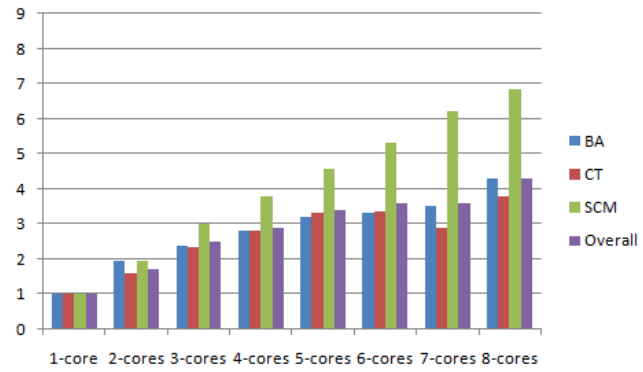

Figure4:Performance results of Spawning Order-A inQ3dm1

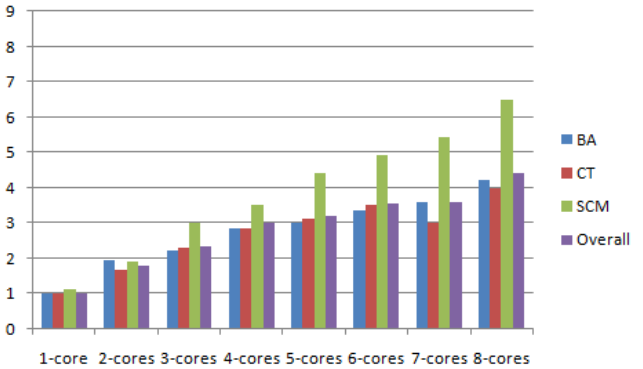

Figure5:Performance results of Spawning Order-B inQ3dm1

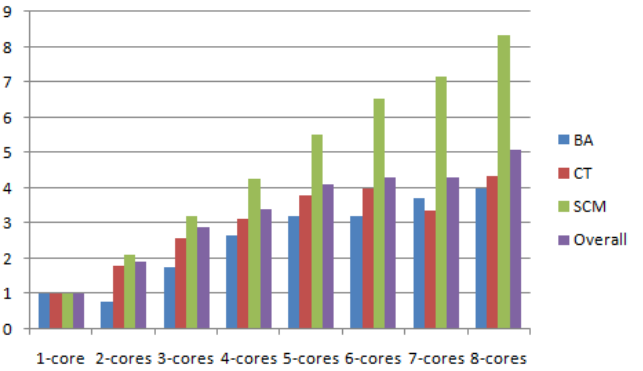

Figure6:Performance results of Spawning Order-A inQ3dm3

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 show in Fig. 4, 5 and 6, the speedup measured well where all three setups displayed an almost identical grade of speedup behavior. The performance displayed a great amount of speedup at all number of cores, where the 1st, 2nd and 3rd setup 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, which influences the frequency of Clients encounters to become lower than the first two maps; thus, Clients execute *fireweapon()* (Locked area) less than in the first two setups. Therefore, 3rd has less waiting time in ClientThink(). This also can be seen in Fig.6 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 reasoning for this behavior. However, it can be assumed that because SendClientMessages() abides by the Parallelizable-C rules more than ClientThink() and BotAI(), it displayed a better performance. Furthermore, a frequently accessed global variable **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 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 Fig. Due to the lack of proper analytical tools it was difficult to pin point 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. In every session a match starts as soon as the World Map is spawned, then, Clients are spawned continuously in a sequence, one Bot per-frame. Therefore, at the start of the session, because the number of Bots is low, not all threads will be occupied with Bot computations. The load amongst threads balances out as soon as the number of populating Bots grows and reaches the limit of 112. Proving this with the Linux version was unsuccessful, but was successful and clearly seen on the Windows7 version.

VI. CONCLUSIONS

This paper has described the experience of achieving enhanced performance 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 than original one on an 8-core IBM POWER 5+ platform. The areas of the program that was majorly modified into Parallelizable-C 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].

From this experiment, it has been understood that Video Games as applications are written to be highly resource efficient that implement many programming shortcuts that result in contentions over global resources, which come to be the main cause for the hazards. Another cause of hazards were the result of illegal access to private data amongst threads.

Several effective methods for avoiding hazards that are caused by read/write operations from/to a shared complex data structure that are hard to localize were found; batch excution outside the parallelized loop; *lockage* and so forth. Other hazardous areas required restructuring of the engine to avoid the hazardous contentions.

In ioquake3, the mechanisms of reperesenting both the Bot, and the Human player inside the engine highly resemble each other. Therefore, this work should be highly beneficial in understanding parallelism of Human driven sessions as well. Expirementing with large numbers of Human players is outside the capabilities of this paper. However, since SendClientMessages() should scale well with Human players[13], 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 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 *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 Mr. Akihiro Hayashi and Mr. Keiichi Tabata for their support in this work.

REFERENCES

[1] Sony: PlayStation®3 160GB system Tech Specs. http://www.playstation.com :(Apr 2012)

[2] IBM: Cell Broadband Engine Architecture and its first implementation. http://www.ibm.com:(Apr 2012)

[3] 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

[4] id software: ioquake3. http://ioquake3.org, April 2012

[5] ioquake3: http://en.wikipedia.org/wiki/Id_Tech_3, Apr 2012

[6] 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

[7] 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

[8] 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

[9] Microsoft: Halo; http://halo.xbox.com:(Apr 2012)

[10] ACTIVISION: Call Of Duty http://www.callofduty.com, Apr 2012

[11] EA.: BattleField: http://www.battlefield.com, April 2012

[12] 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

[13] Ahmed Abdelkhalek, Angelos Bilas: *Parallelization and Performance of Interactive Multiplayer Game Servers.*: Parallel and Distributed Processing Symposium 18th International Proceedings, April 2004

[14] 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.

[15] 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

[16] Waveren, J.M.P. van.: *The Quake III Arena Bot*,. 2001.

[17] *The OpenMP® API specification for parallel programming.* http://openmp.org/wp, April 2012

[18] *QuakeIII:*http://www.idsoftware.com, April 2012

[19] *Analyzing Application Performance by Using Profiling Tools* http://www.microsoft.com, April 2012

[20] *IBM eServer p5 550:* http://www.ibm.com, Apr 2012

[21] 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