Parallelization and Performance of Interactive Multiplayer Game Servers

Ahmed Abdelkhalek
Dept. of Elec. and Computer Engineering
University of Toronto
10 King's College Road
Toronto, Ontario M5S 3G4, Canada
Email: abdel@eecg.toronto.edu

Angelos Bilas[1]
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
P.O. Box 1385
Heraklion, GR 71110, Greece
Email: bilas@ics.forth.gr

Abstract

An important application domain for online services is interactive, multiplayer games. An essential component for realizing these services is game servers that can support large numbers of simultaneous users in a single game world. In this work, we use a popular, 3D, interactive, multiplayer game server, Quake, *to study this important class of applications. We present the design and implementation of a multithreaded version of the server. We examine the challenges in scaling this class of applications to large numbers of users, mainly task decomposition and synchronization. We present preliminary performance results for a server with up to eight processors. We find that: (i) Scaling interactive, multiplayer games that exhibit fine-grain interactions in a detailed 3D world to large numbers of players is a challenging task. (ii) The main bottlenecks are lock synchronization during request processing and high wait times due to fine grain workload imbalances at global synchronization points. (iii) Significant future improvements are possible using techniques that take advantage of game-specific knowledge.*

1 Introduction

Recent improvements in end-user connectivity with the Internet have renewed the interest in providing online services to large numbers of users. The increasing availability of broadband wired and wireless connections and powerful thin clients, such as cell phones, game consoles, and PDAs, allows us to consider offering online services to large numbers of non-expert users [12]. One application that is lately attracting particular attention in the entertainment industry is interactive, multiplayer games. This class of applications has gained a lot of attention due to the potential for providing customizable entertainment and interaction amongst real players [10]. As a result, many modern game titles today include an online, multiplayer mode.

An important component in offering such services is cost-effective scalable servers that can support large numbers of users. In virtually all cases, multiplayer games are enabled today by a central server. Clients connect to this server which is responsible for interpreting their actions, maintaining consistency, and passing information among them. A variety of multiplayer games exist with different characteristics and demands varying from simple card games, up to role–playing environments with many users. Being able to support hundreds and eventually thousands of users opens up additional opportunities for interaction and may enable new games or online multiperson experiences (e.g., a virtual world where hundreds of players interact simulating real–life–scale experiences) [10].

Understanding the scalability requirements of such services and how their demands change with the number of users is essential for designing cost-effective service and server architectures. In contrast to the scalability of scientific workloads, little is known today about the behavior and requirements of interactive multiplayer game servers. Most of the work related to game services has focused on the client side and in particular in graphics–related issues [18, 14].

In this work we study the scalability of a specific class of games; first–person action games. This game architecture is one of the prime candidates for evolving into highly interactive *real–life* simulators. First–person action games support fine grain, close to instantaneous control of player actions, and a high degree of interaction among the players in a detailed, 3D virtual world. While other types of multiplayer games exist, such as third–person action games, the level of interaction is typically coarser. From now on, we will use the term *server* to refer to game servers for first–person action games.

Our previous work [1] has examined basic issues in benchmarking this class of game servers and understanding the behavior and requirements of a sequential game server. This work showed that current servers support only a few tens of users and that server CPU processing is the bottleneck for scaling to larger player counts. Server network bandwidth and memory requirements are currently not important issues.

In this work we study how this class of applications can benefit from additional CPU resources in modern, multiprocessor systems. We design and implement a parallel version of a popular multiplayer game server, *Quake* [9]. *Quake* is a

[1]Also with the **Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece, Email: bilas@csd.uoc.gr.**

commercial, publicly available [8] game server that has been used extensively and exhibits many of the required characteristics. We design and implement a parallel version of the server and present the main challenges. We discuss how the workload is distributed among server threads to exploit parallelism in the server and we describe what synchronization is necessary for correct execution. In this early stage of our work we divide server execution in phases separated by global synchronization and deal with each phase separately. We use static task assignment techniques to distribute the load among threads. We use region–based lock synchronization in the request processing phase to guarantee correct user request processing. Finally, we evaluate the parallel version of the server on a modern, hyper–threaded, x86-based quad system and present preliminary results and detailed analysis of system bottlenecks.

We find that: (i) Scaling this class of applications to large numbers of users is a challenging task. Although the server workload increases superlinearly with the number of players, our parallel version, using all eight hardware threads, increases the number of players that can be supported by about 25% over the sequential version. (ii) The most significant bottlenecks in the parallel version are high wait and lock synchronization times. Although the load is distributed evenly at a macro scale, we find that there are significant fine grain imbalances due to global synchronization that lead to high wait times, up to 40% of the total execution time. Synchronization due to interacting players amounts for up to 20% of total execution time. (iii) Our analysis shows that there is significant room for improvements by using dynamic task assignment and request batching techniques, based on game–specific knowledge.

The rest of this paper is organized as follows. In Section 2 we provide the necessary background for typical multiplayer setups and the basic operations that take place on both the client and the server side. Section 3 presents at a high–level the parallel version of the server and discusses issues related to task assignment and synchronization. Section 4 presents our preliminary scalability results. Section 5 presents the remaining system bottlenecks and discusses methods for future improvements. Section 6 discusses related work. Finally, in Section 7 we draw our conclusions.

2 Background

In this section we present an overview of the multiplayer mode of *Quake*, *QuakeWorld* (version 2.40), henceforth referred to as *Quake*. We discuss in more detail the aspects relevant to our work. *Quake* [9] is a sequential (single–threaded) application, advertised to support up to a few tens of simultaneous users.

In a typical client–server setup [1], servers usually maintain consistency of the game world and handle coordination among clients, whereas clients perform all graphics and user–interface operations. More specifically: a set of players (clients) connect to a centralized game server, they join a game session, and participate in the game plot until they leave the session, their connection is terminated, or the session is ended. Clients communicate only with the server. The server executes client actions and notifies other clients accordingly. Players can locate available servers via well–known *directory servers* where servers publicize their network address and other game related information. In this work, we are interested in server behavior after a game session has been initiated. Game servers are usually stand–alone PCs or workstations. Client systems are, today, either home–level desktop systems or game consoles. For all practical purposes clients need to render at least 24–30 frames per second (fps). This frequency defines the duration for each iteration or client frame, resulting in 30 to 40ms frames.

Next we discuss in more detail the structure of the game server and the data structures that are important for our work.

2.1 Server structure

Figure 1 shows the three main stages in server execution. The main server task, computing a new frame, starts when a request from any client arrives, which causes the server thread to exit the *select* system call (S in figure). The server frame execution can then be broken down into three distinct stages: (i) updating the world physics (P), (ii) receiving and processing requests until no more requests are available (Rx/E), and (iii) forming and sending replies to all players that sent a request during this server frame (T/Tx). At this point the server frame ends and the process is repeated.

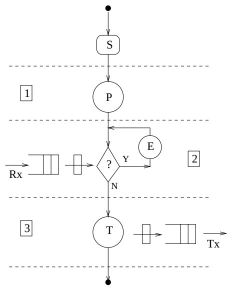


Figure 1. Sequential server stages.

The server spins in a tight loop waiting for client requests that carry player intentions. Upon receiving client input, the server determines how this interacts with the rest of the virtual world. The server replies only to *explicit* client requests, assuming that clients are always active sending frequent requests with user actions (i.e. if the client does not send a request, it will generally not receive an update from the server). All replies are sent after all requests in the server request queue have been processed. Ideally, the server replies with updates to a client's request before the client sends the next request (in the next client frame). However, in practice, as the number of players increases the server may take more than a few client frames to respond with updates, which leads to the perceived *lag* at the client side and unpleasant lack of smoothness to the real–time experience.

A game world or map consists of a polygonal representation of the 3D maze in which the players move about. In the maze there are several entities that clients interact with (e.g., pickup or activate). Each entity has its own characteristics and actions that it can perform. Moreover, to minimize bandwidth requirements and to support low bandwidth connections (e.g. modems), the server determines which entities are of interest to each client and sends out information only for those, i.e. it will notify a client only of entities that are visible to it or that may soon become visible and sounds that are audible.

Thus, server processing is a complex, compute intensive task that increases superlinearly with the number of players [1]. Finally, network bandwidth and latency are not important issues for the server, since a single 100 MBit Ethernet, commodity network interface can support large numbers of players [1].

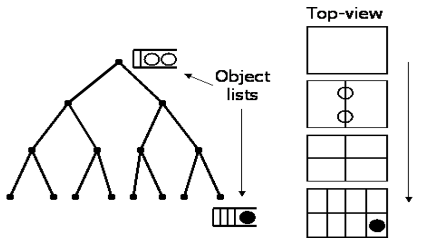


Figure 2. Left: object lists. Right: areanode tree building.

2.2 Data structures

The *Quake* virtual world that the server maintains consists of the game session 3D map [9] and all the players. A *Quake* map is a 3D volume in a 3D coordinate space. The map is represented as a BSP file, implementing a *binary space partition* [17] representation of a 3D world. The origin and size of space the map occupies in the coordinate space is defined by the BSP file for the given world. Each coordinate in the space is a pixel. The *BSP tree* is a structure maintained by the server to contain detailed information about the whereabouts of different objects in the game world and is used, for example, when simulating interactions of objects. However, this structure is fine grain and therefore inefficient to use if the server wants to generate a quick list of objects that an object at a particular location may interact with, for example, during move command execution. For this purpose, the server maintains a second binary–tree structure, called the *areanode tree*.

To form the areanode tree, the entire volume of the map is represented by areanode 1 at depth 0. This volume is then divided into two equally sized volumes by splitting it in half along a flat vertical plane in either the x or y–axis dimension, that define the base (ground) of the world. The resulting two 3D rectangles, areanodes 2 and 3, are said to occupy depth 1 of the areanode tree. Division of the 3D volume into smaller areanodes continues recursively. At each depth a different division *axis* is used (alternating). Figure 2(right) demonstrates the areanode tree building process.

Currently, the maximum depth is defined to be *four* in the server code, leading to a total of 31 areanodes, 16 of which are *leafs*. This can be modified with no effect on the client–side. Note that this division occurs only along the x- or y–axis and, therefore, this is a 2D structure, with all areanodes having the same height (z coordinate), which is the height of the entire world. Thus, the areanodes form a balanced binary tree. Areanodes in the same level represent distinct volumes and may only share the division planes. Each level of the tree represents the full 3D volume of the map, at progressively finer grain divisions.

The *creation* of these areanodes is not related to the structure of the map; the volume each areanode occupies has no correlation with rooms and each areanode's side planes have no correlation with walls or ceilings. However, each 3D point in the map must either be in an areanode that is a leaf or in a division plane. Moreover, a 3D *object* in the map, such as a player figure, may lie completely inside a leaf or may intersect more than one leaf. As the depth of the areanode tree increases an object is more and more likely to cross the division planes.

Areanodes maintain within their structures a list of game objects, such as player objects, that are associated with them. Objects not crossing any division planes in the map are associated with the leaf they reside in (solid circle in Figure 2). Otherwise, they are associated with a unique parent of the leafs they cross. For example, objects crossing the plane that was used to divide areanode 1 (the root of the tree) into its children are associated with areanode 1 (empty circles in Figure 2). When an object is moved during gameplay, it is necessary to update the areanode tree so that it reflects the possibly different areanode the object is associated with.

2.3 Move execution

The server receives a number of different types of request messages from clients. The most important type of request which causes the player to interact with the game world is the *move* request. All other requests are associated with the connection or disconnection protocols used when the client wants to join or leave the server game session, or other facilities that do not affect gameplay, so we do not discuss them any further. The move command is what directly affects gameplay. We now focus our discussion on move command processing.

The move command specifies: (i) angles of the player's view, (ii) forward, sideways, and upwards motion indicators, (iii) flags for other actions the player may be able to initiate (e.g. jumping), and (iv) the amount of time the command is to be applied in milliseconds. For 30 fps clients, this is typically around 30 ms for each move command. The move command execution can be broken down into two general components: (i) player figure motion, and (ii) other interactions the player may initiate as indicated by the flags in the move command. Note that this is a functional breakdown of move execution as opposed to a temporal one. While, processing interactions that a player initiates with the world, player figure motion processing may also occur, and vice versa.

In the player motion component, the server uses the motion indicators in the move command to simulate the motion of the player's figure in the game world in the direction and for the duration specified. This includes determining what game objects are close to the player in the game world that the player may interact with during its motion. After this simulation is complete, the server removes the player's object from its old position in the game world and links it to the new one. In *Quake*, this is typically in the vicinity of the old position but may sometimes be in far locations in the game world, e.g. if the player interacts with certain world objects (such as a teleporter).

The other general component of the move execution involves actions the client might initiate in the move command. In *Quake* this may be causing the player to switch items from its backpack for example, or it might be other actions that may affect objects relatively distant from the player in the game world, such as throwing an item at a target far from the player. In this component, these actions also cause the server to simulate their effect on the state of the world.

For each move command, the server creates a list of *all* objects the player *may* interact with during its motion. The server does this in two steps:

1. The server uses the starting location of the player and the *maximum* possible distance a player can travel in a single move to determine the *bounding box* of the player's motion in this move. Thus, the bounding box for a move is the region of the world that it may affect.
2. The server traverses the areanode tree and for every areanode it intersects it performs the following operations: (i) The server checks the list of objects associated with the areanode to see if any object intersects the motion's bounding box. (ii) Intersecting objects are added to the list of objects associated with the move. (iii) The server repeats this process for the areanode's children if the areanode is not a leaf and if the areanode intersects the move trajectory.

Thus, for all move executions, the server traverses the list of objects associated with areanode 1, since all moves intersect with the entire world. Objects associated with areanode 1 intersect the plane that divides the entire world into two equally sized volumes. Thus, the association of objects with areanodes is not the same as associating objects with different *regions* of the world, where objects that are next to each other are always associated with the same region. This is revisited in the next section when we discuss the synchronization necessary for correct execution in the parallel server.

3 Parallel server design

In this section we describe the basic issues in building a parallel version of the *Quake* server. As mentioned, server execution can be divided into three phases: (i) world processing phase, (ii) request processing phase, and (iii) reply processing phase. Each of the three phases in server execution contributes to frame processing time. Each phase uses global data structures that are updated during the previous phase.

To simplify parallel server design, we impose two invariants: (i) Each server phase is distinct and should not overlap with other phases; and, (ii) each phase should execute in the original order: world processing, request processing, and finally reply processing. These two invariants allow us to avoid semantic and correctness complications that could arise from reordering the frame computation stages or overlapping them. Future optimization efforts may relax these constraints and change the general server structure to achieve greater scalability after concurrency issues in game servers are better understood.

Since we would like to start from the sequential version and not rewrite the server, we choose to use the shared memory model [5] which is closer to writing sequential programs. Furthermore, we use `pthreads` [13] as the programming interface.

3.1 Task decomposition

First, we notice that world processing takes less than 5% of the total execution time of the sequential server, regardless of the player count [1]. Thus, we exclude this section from parallelization during this first attempt. The remaining two stages contribute significantly to the sequential execution time breakdown and it is important to target both phases.

Since we desire to scale the server to several hundred or thousands of players, we choose to multiplex threads to players and use each thread to handle multiple players. This scheme allows us to adapt the number of threads to various system characteristics, e.g. number of processors. In our implementation, each thread uses a different UDP port for communication with its clients. Thus, a server appears to clients as one IP address and a range of UDP ports.

Currently, we use throughout execution a fixed number of threads that matches the number of processors in the server and we create all threads at initialization time. We assign players to threads in a block fashion. Moreover, all work related to a specific player is performed by one thread. Other, dynamic policies are possible as well, especially policies that take into account locality information, however, these may require more intrusive changes to the server and are left for future work.

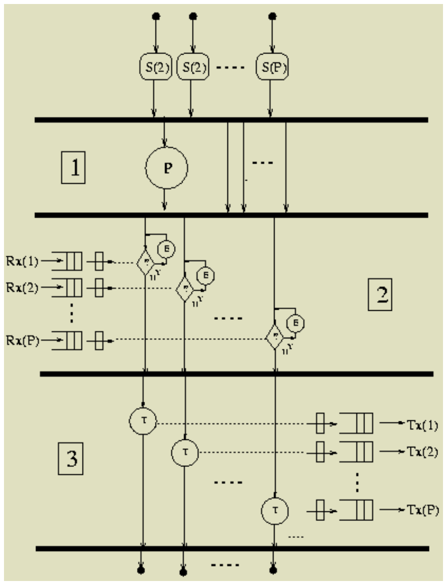


Figure 3. Parallel server stages.

3.2 Frame orchestration

Figure 3 shows a flow chart displaying the high level structure of the parallel server. In this figure, S=select, P=world physics update, Rx=receive, E=execute requests, T/Tx=form and send replies. The three phases of server execution are separated by global synchronization. Global synchronization is implemented with `wait/signal` primitives [13]. Lock synchronization is implemented with `pthread` library calls.

The first stage of server frame execution, which mainly involves updating the world, is performed sequentially by a single thread. The first thread that detects an arriving request from a client, through the `select` system call, is designated as the *master* thread for the new frame and is responsible for performing the world update. While it performs the update, other threads may exit `select`. These threads wait at the global synchronization point between the world and request processing phases.

After the master thread finishes updating the world, all threads that have requests to process will proceed to the next phase. Other threads that exit `select` after this point will have to wait until the next server frame to process their requests. However, they are guaranteed to be part of the execution of the next server frame. During the request processing stage, each thread enters a loop receiving and processing its client requests until its request queue becomes empty.

In the reply stage, each thread forms and sends replies to every player they received a request from during this frame and then re–enters the *select* call. The master thread uses a *frame end* signal to wake up any threads that missed the last frame. We note that not all threads will necessarily participate in each server frame.

3.3 Synchronization

Given that server execution phases are separated by global synchronization we only need to worry about intra-phase synchronization when accessing shared data structures. Although during world processing the server updates global state, this phase is executed by the master thread only. Thus, there is no need for intra–phase synchronization in the first stage.

Reply processing does not exhibit any read–write conflicts to shared data. Replying to a client request at the end of a frame, involves reading global state but writing only private (per–client) reply messages. Thus, sending replies is a *read–only* process with respect to global server state.

During request processing the two main tasks of each server thread are receiving client requests and processing each request. In our design, each thread receives requests in a private UDP port. This is possible because of static player assignment to threads. Thus, there is no need for any synchronization when receiving requests. However, similarly to the world update phase, during request processing each thread updates global server state. There are three types of global data structures that may be updated. Per–player reply message buffers, a common global state buffer, and game objects. Per–player reply message buffer updates can be synchronized with locks (one per buffer) in a straight forward manner.

The global state buffer is updated during the world update and request processing phases of the server frame. This buffer is cleared at the end of the server frame to make room for new updates during the next frame. This buffer is used to update *all* clients, regardless of whether or not the server received a request from a client during the current frame: each thread participating in the current frame uses this buffer to update the message buffers of its *complete* set of clients and the master thread performs this operation for clients belonging to threads not participating in the current frame. The master thread clears this global state buffer before signaling the end of the current frame. All accesses to the global state buffer are synchronized with a single lock.

Synchronizing access to game objects is more involved. Processing of client requests involves accessing and updating global objects in the world of the specific game session. To achieve game object synchronization in our work we use a region–based locking scheme, based on the areanode tree. Our approach to concurrently executing multiple move commands is to lock the regions of the world that are involved in command execution. The region of each move is defined by the bounding box of the move. This translates into two types of locking:

First, we lock the appropriate areanode *leaves* in the areanode tree that include all objects that overlap with the bounding box of the move. This takes care of objects that are linked to leaf areanodes. To avoid deadlock in locking areanode leaves, locking is always performed in the same order (no cycles).

Second, objects may also be linked to parent areanodes. Threads locking *different* regions may affect objects linked to the same parent areanode. To synchronize accesses to these objects we use locks at parent areanodes when accessing their object lists. For example, if two objects cross the vertical plane that is used to divide the root areanode both objects are associated with the root areanode. Thus, two concurrently executing threads must lock the root areanode when accessing its object list to ensure atomic access to this list. However, each thread will only need to interact with the object in its own bounding box. Since different threads access different objects in the parent areanode, parent areanodes remain locked only for the duration of the list read or write operation, whereas areanode leaves remain locked for the entire move execution. Essentially, parent areanode locking is an artifact of the server design and does not correspond to contention for shared game objects. Since only one parent areanode is locked at a time, there are no deadlock issues when locking parent areanodes.

Finally, each set of events can be executed in any one of a set of valid orders. The parallel version of the server may perform the various events in a different order compared to the sequential server. For instance, if two players attempt to pick up a world object at about the same time, then the parallel server may execute the events in a different order compared to the sequential server. Similar issues arise in the sequential server when considering, e.g. network latency.

4 Results and optimization

We measure the performance of the parallel server on a quad Pentium III 1.4 GHz system with hyper–threading [11]. Hyper–threading provides two independent hardware threads on each CPU allowing us to use up to eight hardware threads in the server. Table 1 shows the configuration of the server. We dedicate this system as the *Quake* server and use a number of dual-processor systems as clients, using the setup described in [1]. We perform most of our measurements by instrumenting the code, and using the Pentium counters directly. We find that in all cases, a few minutes of execution time is sufficient to capture the game server behavior. In our runs we exclude setup and initialization time and we run each experiment for two minutes and usually multiple times to verify consistency of our results. To automate the benchmarking procedure we replace human with automatic players, as specified in [1].

CPUs	4 x Intel Xeon 1.4 GHz, 2–way HT
1LC, 2LC	12 KB L1 execution trace 8 KB L1 data, 256 KB L2 unified
Memory, bus	2 GBytes, 400 MHz system bus
OS	Linux Redhat 7.3
C library	GNU C Lib. stable release ver. 2.2.5
NIC	100 MBit Ethernet

Table 1. Configuration of the game server system.

In *Quake* input maps specify the setting where the game will evolve. The main factors that determine our choice of maps are the complexity of the map and the induced level of interaction among players. The complexity of the map is defined by its layout and the number of objects it includes. These two aspects usually conflict since player interactions increase in small maps, whereas only large maps can contain many objects and employ elaborate layouts. Although we examine multiple maps we present results for one, large map that is more appropriate for scaling the number of players to a larger count.

We use one of the largest maps we could find [3], which was designed to support 16-32 players. Even with a large map, the observed level of interaction among players is very high. We expect that real life situations will exhibit less interaction. Hence, our experiments represent extreme situations and stress the server aggressively. A more comprehensive study of different maps is performed in [1] for the sequential server. Although we have performed experiments with different maps, due to space limitations we only present results for a single, representative case. Unless stated otherwise, in all our experiments we use the default total number of areanodes of 31 (16 leaves) in our experiments.

To compare server configurations we use two high–level metrics, server response rate and response time [1]. Response rate is the rate at which the server can service client requests. The response rate we report is average response rate throughout the run (in replies/sec). Response time is the time between the client sending a request and receiving a reply from the server. We report the average response time across all clients for the duration of the full experiment. We also present execution time breakdowns and various other statistics to better understand the differences among server configurations and to clarify specific issues.

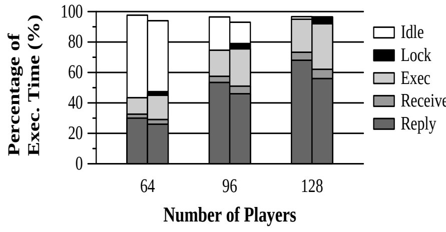
Our execution time breakdowns present a division of the total execution time in the following components: *Exec* is the time the server spends processing requests. In the parallel server we present the *lock* overhead as a separate component. In these cases, the actual request processing time is the sum of the *exec* and *lock* portions. *Receive* is the time spent in receiving and parsing requests during the request processing phase. *Reply* is the time spent in forming and sending replies to clients. This portion of execution time is the full reply processing phase. *Intra–frame wait* is the time server threads spend waiting at a global synchronization point for other threads to complete processing their request queues, before all threads proceed with the reply processing phase of the current frame. *Inter–frame wait* is the time server threads spend waiting at a global synchronization point for the world update phase to complete or for the current frame to end if they missed a frame. *Idle* is the time server threads spend in the `select` system call waiting for requests to arrive.

In our results we find that the cost of the actual system calls used (`sendto`, `recvfrom`, and `pthreads signal`) is less than 5% of total execution time, so we do not examine this any further. However, we do not measure the contribution of system calls that may be present elsewhere in the pthreads library. Moreover, the only significant lock overheads appear in the request processing phase for protecting global game objects. All other locking overheads are less than 2% of the total execution time. For this reason we consider separately only areanode locking during request processing.

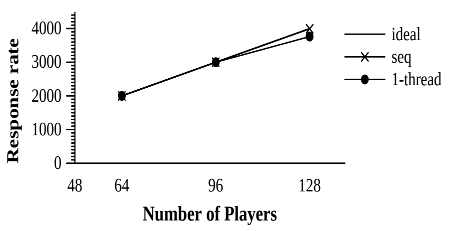
4.1 Single thread

First, we examine the overhead incurred by our benchmarking instrumentation and parallelization, henceforth referred to as overheads, compared to the sequential version of the server. Figure 4 shows our results for the sequential and the single–thread parallel server for 64, 96, and 128 players. We notice in Figure 4(a) that the parallel version incurs overheads that increase with the number of players. Second, although the overhead is very small at small player counts (less than 5%), it increases at 128 players (up to 15%). The reason is that locking is performed in recursive procedures that traverse the areanode tree and that the server needs to determine which regions to lock.

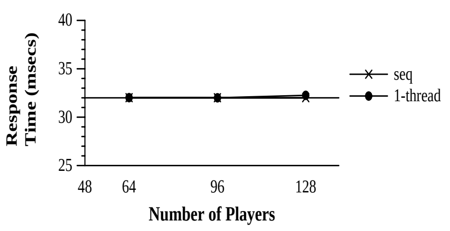
The request (*receive* + *exec* + *lock*) and reply (*reply*) processing phases are the two bottlenecks targeted by our parallel version of the server. We note that the reply processing phase is over twice as significant as the request processing phase. In general, in our experiments with several maps, we notice that the request processing time does not vary considerably, whereas, the reply processing time may vary between maps by as much as 15% of total execution time at server saturation. We believe that this is due to different levels of visibility in different maps, with maps exhibiting higher visibility incurring higher reply processing times as well. Finally, we notice that the impact on sequential server response rate and



(a) Execution time breakdown (left: sequential, right: single–thread parallel server).

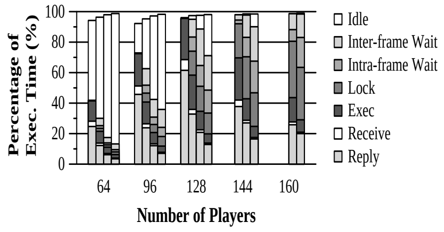


(b) Total server response rate.

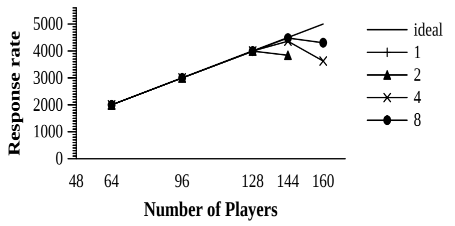


(c) Average server response time.

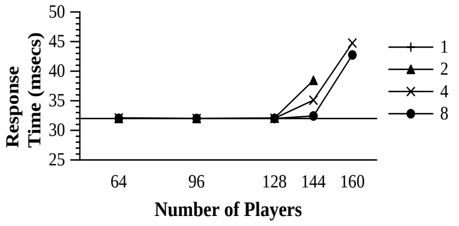
Figure 4. Overhead of parallel server.



(a) Average execution time breakdown for each thread count (left–to–right: 1,2,4,8).



(b) Total server response rate.



(c) Average server response time.

Figure 5. Parallel server performance.

time (Figure 4(b),(c)) is negligible.

4.2 Multiple threads

Figure 5 shows the parallel server performance for each thread count. With two threads, the server is overloaded at 144 players as seen from the reduction in total response rate and the increase in the response time. Using four threads increases the number of players to 160, whereas using eight threads does not improve performance any further.

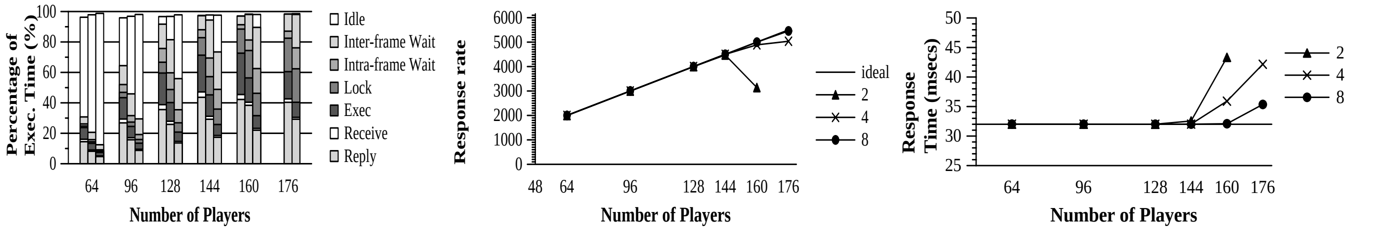
We note that the receive and reply components scale well with the number of threads. The reply phase experiences a slowdown as the server saturates at higher player counts. The request processing time (exec+lock) only shows speedup for lower player counts. As the number of players increases the *exec* component scales well. However, lock synchronization time grows from 2% to 35% for all thread counts, as we increase the number of players from 64 to 160. Lock time is due to the high inherent contention for regions (areanode leaves) when many players operate in the same part of the virtual world, as well as induced locking of parent areanodes as discussed in Section 3.3.

The multi–threaded server exhibits high inter– and intra–frame wait time for all thread counts. Total wait times increase from 5% to over 40%. The inter–frame wait component is in most cases more significant. Finally, from per–thread execution time breakdowns that are omitted, for space reasons, we observe that the workload (including all components of execution time except for idle and wait times) is quite balanced among different threads (under 10% variation). This indicates that the high wait times are due to micro–imbalances among threads. In our design we do not ensure that threads will need to process a similar number of requests in each frame. Since clients send requests in an asynchronous manner, different threads experience peak work bursts at different times, which leads to high intra– and inter–frame wait times.

Finally, Figure 5 shows that the server starts to saturate at 128, 144, and 160 players with 2, 4, and 8 server threads respectively. Overall, the parallel server with eight threads can support about 15% more players than the sequential server. We note that as the number of players increases, the server load increases superlinearly due to player interactions [1]. Given the dynamic nature of player interactions, characterizing exactly how the server workload evolves is beyond the scope of this work and we leave it for future extensions. Finally, at eight threads, lock and wait times dominate and they account for up to 70% of total time. Thus, it is important to understand and improve both lock synchronization as well as intra- and inter–frame wait times.

4.3 Optimized locking

The request processing phase generally consists of two components that simulate short–range and long–range interactions, respectively. The synchronization for each phase is performed separately. So far we have performed somewhat conservative synchronization for short–range interactions and highly conservative for long–range interactions. This means that we lock a slightly larger region than necessary for short–range interactions and the entire map (all areanode leaves) for long–range interactions. To improve locking we use game–specific knowledge for various objects with which players perform long–range interactions. There are two main types of objects in the game.



(a) Average execution time breakdowns for each thread count (left–to–right: 2,4,8).

(b) Total server response rate.

(c) Average server response time.

Figure 6. Performance with optimized locking.

The first type includes objects that are partly simulated during request processing and then their trajectory or actions are completed during the world physics processing phase. For these objects it is possible to perform *expanded* bounding–box locking, similar to the short–range phase. This expanded locking accommodates the maximum possible interaction range during request processing. With expanded locking we increase the extent of the region to lock outwards in every direction by an amount that depends on the object.

The second type includes objects that are fully simulated during request processing. For these objects we perform *directional* bounding–box locking. With directional locking we extend a bounding–box from the player to the end of the world in the direction the object is being simulated. This guarantees the executing thread will have exclusive access to any object/region that may be affected by this move. Directional locking is particularly useful when the player is near and facing one of the side planes or corners of the world. In such situations the region size can be comparable to that of short–range motion. However, if the player is at one corner of the world facing another, then directional locking may not be as effective.

Figure 6(a) shows the average server execution time breakdowns for each thread count using optimized locking. Lock time is reduced by more than half in all cases, compared to the non–optimized version (Figure 5(a)). All server configurations benefit from this enhancement. As a result idle time increases from 1% to 7% with 8 threads and 160 players. Figure 6 also shows the improved server response rate and time with optimized locking. Optimized locking allows us to increase the number of supported players by about 25% compared to the sequential server. However, lock time is still significant and remains between 1% and 20% of the total time.

For short–range interaction, one can potentially reduce the locked region by reducing the maximum speed of motion. However, this may change the game behavior, and thus, we do not explore this direction any further.

5 Analysis of bottlenecks

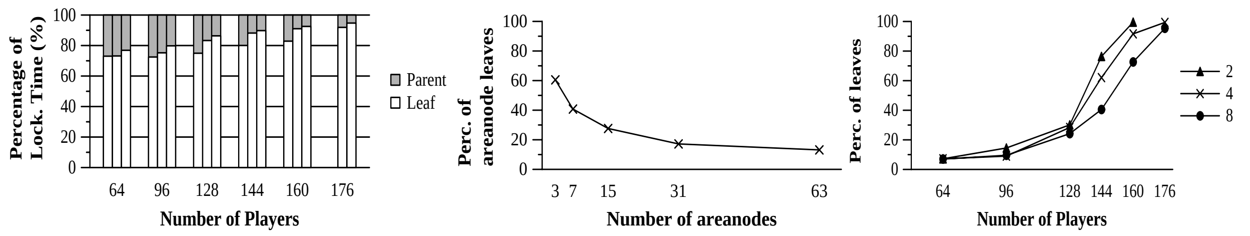
In this section we provide detailed analysis about lock and wait time overheads.

5.1 Lock time

As mentioned above, the most significant source of locking overhead is object-related locking that happens through the areanode tree and can be of two types: (i) Leaf locking that corresponds to getting exclusive access to a region of the map and the contained objects. (ii) Parent locking that is used to temporarily lock objects that cross dividing planes and, thus, are linked to parent nodes as opposed to leaves.

Figure 7(a) presents the relative portion attributed to each of leaf and parent locking. We see that locking leafs accounts for most of the overall lock time. This is expected since locked leafs are not released until the end of the move execution phase, whereas parent areanodes are locked only during areanode tree traversal. The relative importance of leaf to parent areanode locking increases with the number of threads and the number of players. As the number of threads increases, contention for regions of the map increases. As the number of players increases, threads are more likely to contend for intersecting regions. Thus, techniques for further reducing lock overheads related to areanode tree leaves, such as reducing the size of the required region can have a significant impact on server performance.

Next, we examine how the size of the areanode tree impacts lock overhead. We vary the total number of areanodes in the tree from 3 to 63. Increasing the size of the areanode tree may result in finer grain locking, reduced contention, and thus, improved performance. Figure 7(b) shows the percentage of *distinct* leaf areanodes locked per request as the size of the areanode tree increases. The number of leaves locked is a function of the size of the needed region and the number of the areanodes, and not the number of players or threads. Thus, the curve shows the percentage of the world a request locks on average. As we increase the number of areanodes, the portion of the world locked per request decreases rapidly. However, there is little or no change between 31 and 63 areanodes. Some leaves may be locked more than once during request processing. At 31 and 63 areanodes, 40% and 30% of leaves are relocked, respectively. However, in our experiments we observe that increasing the size of the tree does not seem to have an impact on the lock overhead. It is foreseeable, however, that changing game parameters, such as maximum player speed, that affect the distance objects can cover in a single move, may correlate with the number of areanodes



(a) Average percentage of lock time due to parent and leaf areanode locking for each thread count (left–to–right: 2,4,8).

(b) Average percentage of *distinct* leaf areanodes locked per request, as the total number of areanodes increases.

(c) Average percentage of leaf areanodes locked by at least two threads per frame.

Figure 7. Locking overhead and contention.

used.

To investigate the amount of activity in each server frame, we measure the average percentage of leaves locked in each frame and the average number of lock operations in each frame. We omit these figures for space reasons. As the number of players increases, the region of the map accessed per frame increases as well. At 64 players, around 20-25% of the map is accessed in all server configurations. At 128 players, the range is 30-50%, the 2–thread server being highest, and the 8–thread server lowest. At 160 players, the range is 40-100% (again 2-threads highest, 8-threads lowest). The number of lock operations per frame shows that on average each leaf is locked between zero and 20 times depending on server saturation, indicating high contention. As Figure 7(c) shows, the overlap in leaves locked by threads executing in the same frame increases significantly with the number of players with a knee between 128 and 144 players for each thread count. In particular, near saturation, almost 100% of the leaves of the world are shared by at least two server threads. These results show high contention between threads for map regions in the server.

Our analysis suggests that dynamically assigning threads to players taking into account the region they are located may reduce contention. Alternatively, restructuring move execution and areanode partitioning to allow threads to lock regions once per request could further reduce lock overheads. However, such techniques are left for future work.

5.2 Wait time

Inter- and intra–frame wait times consume up to 40% of the server non–idle time at high player counts (Figure 6). Intra–frame wait time occurs when threads finish processing their request queues and are waiting for other threads to finish their queues before they all enter the reply phase. At 128 players, results we omit show each thread of the server on average processes the same number of requests per frame (4, 2.5, and 1.5 requests per thread, for 2-,4-, and 8-thread configurations). For the 2-thread, 128 player configuration, we also measure the dynamic difference in the number of requests per thread per frame. We perform the measurement for the first fifty consecutive multi-threaded frames. We find that on average one thread services 3.3 more requests than the other. The standard deviation is 2.5. This high variation in workload for each thread per frame results in high intra-frame wait times.

Inter–frame wait time is considerably larger than intra–frame wait time (Figure 6). Figure 6 shows that threads still have idle time even at higher player counts. Only at saturation is idle time negligible. This means at saturation threads have work to do all the time. Note that this may not be the case in real situations since not all players will be active or send requests at the same rate, as is the case with our experiments. Thus, our experiments show worse–case server performance.

Inter–frame wait time accumulates when threads spend time waiting between frames for (i) the master thread to finish processing world information or (ii) other threads to complete the current frame. Results we omit for space reasons show that although a thread is waiting due to world processing up to 50% of frames at saturation, on average, only 25% of inter-frame wait time is due to world processing and 75% is due to waiting for the previous frame to complete.

Thus, our analysis shows that the most significant problem seems to be that requests are arriving or being noticed at the server out of sync and in many cases threads need to wait for the next frame before they can start processing requests. One possible approach to reduce wait times is to batch incoming requests. For instance, the frame master thread can wait for a period of time before starting the frame. However, such techniques are beyond the scope of this work and are left for future extensions.

6 Related work

Our previous work [1] investigates the behavior and performance of the sequential game server used in this work. The main goal is to develop a benchmarking methodology for this type of application. We use this methodology in this work.

The International Game Developers Association (IGDA) [10] examines the opportunity presented by online games in terms of market overview, business models, and existing game technologies.

There are a number of efforts, with significant participation from the industry, in trying to support thousands of users on a single server for massively multiplayer online games (MMOG) [6]. However, these games involve coarse grain interactions between users. In contrast, first–person action games such as *Quake* have a much faster pace that captures player reflexes [19]. MMOG servers are typically owned by the game manufacturer and support thousands of users in the same session. Our goal is to scale first–person action game servers to this level of scalability.

Recently, there has been a lot of effort in the computer architecture and in particular the parallel computer architecture community to enrich the pool of applications used for system evaluations with novel, commercially–oriented classes of applications [2]. This work has mostly examined the behavior of multimedia applications [15], database systems [16], and web servers [4]. However, we are not aware of any work that has studied scalability issues of applications in the area of interactive entertainment and in particular game server applications.

Finally, a lot of work is currently being conducted in both industry and academia in game client–side issues, such as improving 3D graphics realism and rendering speed, at both the hardware and software levels [7]. Such work directly affects the amount of detail 3D game developers can incorporate into their worlds and many aspects of game design which must take into account the available hardware/software performance to ensure smooth view transitions.

7 Conclusions

In this work we investigate the parallelization and scalability of interactive, multiplayer game servers. We use *Quake*, a popular member of this class of applications. We design and implement a parallel version of the *Quake* server for shared memory architectures. The main challenges are task decomposition and synchronization for correct game processing. We evaluate the parallel server behavior and scalability on a hyper–threaded quad SMP system.

We find that the reply processing phase and the receive component of the request processing phase scale well with the number of threads for all players counts. However, the game processing component of the request processing phase incurs high synchronization overheads, mainly due to lock contention for shared game objects. Locking overhead is up to 35% of total execution time. Using optimized locking, based on application specific knowledge, can reduce the lock time significantly to about 20% of total execution time. Moreover, although the overall load assigned to threads is similar, per–frame workload distribution is imbalanced, which creates micro–imbalances and leads to significant wait times at global synchronization points. The wait time is up to 40% of total server execution time.

Overall, although server load increases superlinearly with the player count, the parallel server can support 25% more players than the sequential server. Finally, scaling game servers to several hundreds or thousands of players remains a challenging task that may require rethinking many aspects of the internal architecture of this class of applications.

8 Acknowledgments

The authors would like to thank *id Software* [8] for releasing the source code for *Quake* in the public domain. We also thank Reza Azimi for useful discussions on various aspects of this work, Eugenia Distefano, Peter Jamieson, and Carlos Cavanna for helpful hints on setting up and using the experimental testbed. Finally, we thankfully acknowledge the support of the University of Toronto Fellowship, Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, and Communications and Information Technology Ontario.

References

[1] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. In *Proc. of The 2001 International IEEE Symposium on Performance Analysis of Systems and Software (ISPASS01)*, Nov. 2001.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *ISCA*, pages 3–14, 1998.

[3] R. Boucher. Filename: dmpak.zip, Map name: gmdm10.bsp. http://www.fileaholic.com/idgames.d/quake/levels/deathmatch/compilations/.

[4] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-level communication in cluster-based servers. In *Proc. of High-Performance Computer Architecture (HPCA8)*, 2002.

[5] D. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, 1998.

[6] Developer.com Staff. 32,000 Game Players on a Single Web Game Server. http://www.developer.com/lang/other/article.php/861381, 2001.

[7] J. Foley. Getting There: The Ten Top Problems Left. http://www.computer.org/cga/articles/topten.htm, 2002.

[8] id Software. id Software Home Page. http://www.idsoftware.com.

[9] id Software. Quake Home Page. http://www.idsoftware.com/quake.

[10] IGDA Online Games Committee. IGDA Online Games White Paper. http://www.igda.org/online, 2003.

[11] Intel Corp. Hyper–Threading Technology. http://www.intel.com/technology/hyperthread/.

[12] N. Leavitt. Will wireless gaming be a winner? *IEEE Computer*, 36(1):24–27, Jan. 2003.

[13] X. Leroy. LinuxThreads: POSIX 1003.1c thread package for Linux. http://pauillac.inria.fr/ xleroy/linuxthreads/.

[14] Microsoft Corp. DirectX. http://www.microsoft.com/windows/directx/default.asp.

[15] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, pages 124–135, 1999.

[16] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proc. of The 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS8)*, pages 307–318, 1998.

[17] C. Shimmer. Binary Space Partition Trees. Course presentation: http://www.cs.wpi.edu/ matt/courses/cs563/talks/bsp/bsp.html.

[18] Silicon Graphics Inc. OpenGL. http://www.opengl.org.

[19] Sony Online Entertainment. Everquest Online FAQ. http://eqlive.station.sony.com/library/faqs/faq_eqlive.jsp, 2002.