Behavior and Performance of Interactive Multi–player Game Servers

Ahmed Abdelkhalek, Angelos Bilas, and Andreas Moshovos
Department of Electrical and Computer Engineering
10 King's College Road,
University of Toronto
Toronto, ON M5S 3G4, Canada
{*abdel, bilas, moshovos*}*@eecg.toronto.edu*

Abstract

With the recent explosion in deployment of services to large numbers of customers over the Internet and in global services in general, issues related to the architecture of scalable servers are becoming increasingly important. However, our understanding of these types of applications is currently limited, especially on how well they scale to support large numbers of users. One such, novel, commercial class of applications, are interactive, multi–player game servers.

Multi–player games are both an important class of commercial applications (in the entertainment industry) and they can be valuable in understanding the architectural requirements of scalable services. They impose requirements on system performance, scalability, and availability, stressing multiple aspects of the system architecture (e.g., compute cycles and network I/O). Recently there has been a lot of interest on client side issues with respect to games. However, there has been little or no work on the server side. In this paper we use a commercial game server to gain insight in this class of applications and the requirements they impose on modern architectures.

We find that: (1) In terms of the benchmarking methodology, interactive game servers are very different from scientific workloads. We proposse a methodology that deals with the related issues in benchmarking this class of applications. Our methodology bears many similarities with methodologies used in benchmarking online transaction processing (OLTP) systems. (2) Current, sequential game servers can support at most up to a few tens of users (60–100) on existing processors. (3) The bottleneck in the server is both game–related as well as network–related processing (about 50–50). (4) Network bandwidth requirments are not an important issue for the numbers of players we are interested in. (5) The processor achieves a surprisingly low IPC of 0.416.

1 Introduction

The amount of bandwidth and associated latencies available to Internet's end-users has been improving rapidly. This offers the prospect of delivering traditional as well as novel services to large numbers of clients. Improved networks are only one of the conditions necessary for this prospect to materialize. In addition, powerful servers are needed to provide the required compute resources. The higher the number of clients and the quality of the provided service, the higher computing resource demands placed on the server side.

Understanding the scalability of such services (i.e., how their demands change with the number of clients) is essential for designing server architectures to successfuly support them. While the scalability of scientific workloads has been studied extensively, little is currently known about the scalability of commercial applications. Interactive multi–player games is a class of commercial applications that has not been studied thus far and that is interesting especially for the entertainment industry. This class of applications has gained a lot of attention lately due to the potential for providing customizable entertainment. Practically all modern titles today include an online, multi–player mode.

In virtually all cases, multi–player games are enabled by a central server. Clients connect to this server who is responsible for interpreting their actions, maintaining consistency, and passing information among them. A variety of multi–player games exist with different characteristics and demands varying from simple card games, up to role–playing environments with hundreds of users. There is no single application that is representative of all such servers. In this work we focus on first–person action games. We believe that this game architecture is the prime candidate for evolving into highly interactive "real–life–like" simulators. First–person action games support fine grain, close to instantaneous control of player actions and a high degree of interaction amongst the players and a detailed 3D virtual world. We believe these properties are fundamental for simulating a believable environment with a degree of fredom and response times that approximate real life experience. While other multi–player games players exist, the level of interaction is typically coarse. In the interest of space, we will use the term *server* to refer to servers for first–person action games in the rest of this paper.

Current servers are limited to few tens of users. This type of multi–player games can benefit dramatically from scalability. Being able to support hundreds (and eventually thousands) of users opens up additional opportunities for interaction and may enable new games or online multi–person experiences (e.g., a virtual world where hundreds of players interact simulating real–life–scale experiences). Accordingly, it is important to understand how game servers behave and what bottlenecks may exist. Recent work has focused on client side issues related to the CPU and graphics subsystems.

In this work we study game server scalability by looking at a commercial server application that has been used extensively and exhibits many of the required characteristics. We study *Quake* [5, 6], a sequential, publicly available, multi–player, game–server. Developing a benchmarking methodology for *Quake* poses a number of challenges: (i)

There is no well defined input to use for system benchmarking. (ii) The input stimulus is external to the application server (triggered by client systems). (iii) Typical setups require interaction of human users. (iv) The levels of scalability to be studied exceed the size of most university–level laboratories requiring hundreds or thousands of clients. These issues are similar to online transaction processing (OLTP) systems and the related benchmarking methodologies [13]. The methodology we develop here allows us to (i) perform large scale experiments on a relatively small setup, (ii) automate the benchmarking process, and (iii) compare results across runs.

We use a variety of metrics to understand application behavior at two levels: We first take a high–level look at the computation and communication processing breakdown in the server by using an experimental setup with 32, dual–processor PCs, interconnected with a private 100Mbit/s Ethernet network. This study provides us with insight on the relative importance of network processing versus computation and where scalability bottlenecks exist if any. Then, we take a closer look at various factors degrading performance at the processor level. This study reveals additional information of how the game server stresses the various architectural features (e.g., branch prediction and caches) of a modern high–performance processor.

We find that: (1) Current systems can support in the order of few tens of players (our systems support between 60 and 100). (2) Processor cycles are the main bottleneck. At large player counts the processor is fully utilized. The compute time is divided almost equally between game and network protocol stack processing. (3) Network bandwidth is less of a concern as players exchange only little information with the server (in the order of a few KBytes/s per client). (4) Processor performance improvements can help increase the numbers of players by a small multiplicative factor (e.g., doubling processor clock speed increases the number of supported players from 80 or so to about 100). (5) Parallelization offers a potentially viable path for supporting an order–of–magnitude increase in the number of players.

The rest of the paper is organized as follows. In section 2 we review a typical multi–player setup and the basic operations that take place on both the client and the server side. In section 3 we describe our methodology and experimental setup. In section 4 we present our server analysis for a small number of players. In section 5 we extend our methodology to large numbers of players and in Section 6 we present our results on system scalability. Finally, we comment on related work in section 7 and summarize our conclusions in section 8.

2 Overview and Background

Before we present our experimental analysis of server scalability and behavior, in this section we review a typical multi–player game setup. We emphasize the actions taken by the server and provide a high–level description of the structure of the server and client applications. *Quake* [6] is a popular representative of a subset of applications in this class. It is a sequential application, advertised to support up to 32 or so simultaneous users and is used extensively world–wide. Accordingly, studying *Quake* provides us with insight that could be applicable to other game server applications as well, as many of them have a similar software architecture. In our work we use version 2.40 of the multi–player mode of *Quake*, *QuakeWorld*.

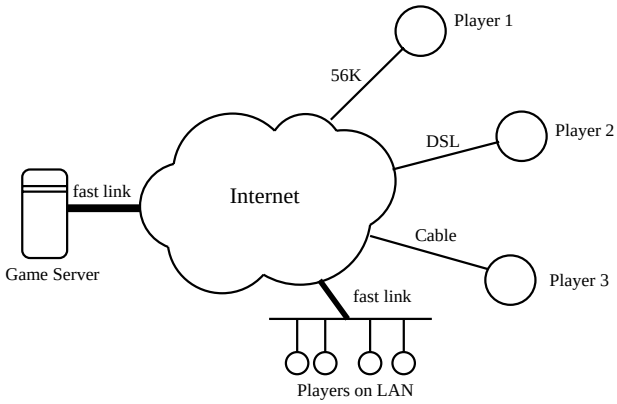

Figure 1: Typical Game session geographical setup

Figure 1 shows a generic setup for interactive, multi–player game sessions. In this client–server setup, servers usually maintain consistency of the plot and handle coordination among clients, whereas clients perform all graphics and user–interface operations. More specifically: A set of players, or *clients* connect to a centralized game server, or simply *server*, they join a game session, and participate until they leave the session, they are terminated, or the session is ended. Clients communicate only with the server. The server parses 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.

```
connect(server_ip,server_port);
while (connected)
{
  if (!time_for_frame()) continue;
  //
  // end previous time-slot
  // start new time-slot
  //
  if (get_input())         // from keyboard, mouse, etc.
    send_intentions();     // to server
  if (recv_reply())        // from server about request sent
                           // during previous time-slot
    update_state();
  perform_prediction();    // on all game entities
  update_screen();         // render video for this frame
  update_audio();          // play audio for this frame
}
```

Figure 2: Pseudo–code for client execution.

Figure 2 shows the pseudo code for the client execution. When a client joins a game session it enters a loop performing the following actions in well defined iterations (time–slots): (i) get the player input, if any, and send it to the server, (ii) receive server updates for all relevant entities for the request the client sent in the previous time–slot, (iii) perform prediction on all entities (e.g., players and objects with associated actions), and (iv) render the next frame. Prediction is necessary since network and server latencies introduce time slack amongst the clients and the server. Even with high–speed connections, this time slack

can be perceivable by humans. Accordingly, when servers do not respond on time, to provide the illusion of a single, time consistent world, clients guess where an entity will be based on its current speed and direction. This prediction is not always correct and deteriorates as network and server latencies increase. If there is no server response for previously sent requests the client performs only actions (iii) and (iv). In all practical cases the client needs to render at least 24–30 frames per second (fps). This frequency defines the time–slot for each iteration, resulting in 30 to 40ms time–slots for each iteration.

```
while (1) {
  //
  // wait until request arrives or timeout occurs
  //
  if (select(server_port,time_out)==ERROR)
    continue;

  //
  // start a server frame
  //
  update_world();         // move autonomous items, etc.
  while (recv_request())  // from any client
  {
    execute_request();    // update world using request
  }
  for (c=0; c<total_clients; c++)
  {
    if (received_req(c))  // if client sent a request
      send_reply(c);      // reply with update
  }
  //
  // end this server frame
  //
}
```

Figure 3: Pseudo–code for server execution.

Figure 3 shows the pseudo code for the server execution. 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. This is a non–trivial, compute intensive task and we describe it in more detail later on. The server replies only to *explicit* client requests, assuming that clients are always active sending frequent requests with user actions. All replies are sent after all requests in the request queue have been processed. Ideally, the server replies with updates to client requests within the same client time–slot that the client sent the request. However, in practice, as the number of players increases the server may take more than a few time–slots to respond with updates leading to the perceived *lag* at the client side and unpleasant lack of smoothness to the real–time experience.

The virtual world that the server maintains consists of the game session 3D map (a BSP file, implementing a *binary space partition* representation of a 3D world) and all the players. The 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. For example, two such entities are a lift and a flag. If a player steps on a lift, then the server needs to activate the lift, simulate its motion and also model the player's motion as induced by the lift. When a player drops a flag, the server has to simulate its motion, and determine when and where it collides with other objects in the world. If these objects are solid, then the flag's motion is terminated, otherwise the motion characteristics are changed (e.g., when it drops in water). All this state needs to be kept consistent amongst all players by server processing. Also, there are various types of matter that items or players can move through. For example, there is air, water, lava, etc. all of which have different properties that need to be obeyed when the server is updating the game state and informing players of these updates. 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). Thus, the server processing can become very involved and complicated especially when the number of players and level of world detail increase.

Finally, in our study we ignore the network path between the server and the client (both throughput and latency). Our goal is to understand server behavior and study its scalability. Providing sufficient network bandwidth and latency is an orthogonal issue, beyond the scope of this work. However, since network stack processing occurs today on host CPUs, we do examine protocol (TCP/IP) overheads in our experiments. Thus, there are three potential bottlenecks on the server side as the number of clients increases: (i) the number of incoming requests and the corresponding replies, (ii) the size of request and reply messages, and (iii) the aggregate processing time for each request due to game–plot updates.

3 Methodology

One of the most challenging aspects of our work is addressing methodological issues in benchmarking this class of applications. The ultimate goal is to be able to evaluate changes to the game server that result in improved scalability and/or perceived client quality at a particular scale. However, realistic setups for these applications have characteristics that do not match traditional benchmarking methodologies, but are similar to online transaction processing (OLTP) systems [13]: The input is external to the application triggered by client systems. The input is generated by human users and is based on global system state (interactions of all users in the system). Studying these applications at large scale involves hundreds or even thousands of clients, which would exceed most university–level laboratory setups. These characteristics make it difficult to both automate the benchmarking procedure and to also compare results across runs. In the next paragraphs we describe our experimental setup and we develop our methodology to deal with these issues.

Experimental platform: For the most part, we conduct our work on a cluster of 32, Pentium II–400 MHz, dual–processor workstations connected by a private, 100 Mbit/s Ethernet network. Each processor has two separate 16KB non–blocking L1 caches (4–way) for instructions and data, a 512KB unified, half–speed L2 cache (4–way), and a 100MHz system bus. We also provide some sample runs on an Pentium III–800MHz incarnation of this cluster, with each processor having two separate 16KB non–blocking L1 caches (4–way) for instructions and data, a 256KB unified, full–speed L2 cache (8–way), and a 100MHz system bus. Each node in the system has 512 MBytes of main memory and has an off–the–shelf, low–end graphics card. The operat-

ing system is Windows NT. We dedicate one node as the *Quake* server and use the rest as clients. We perform most of our measurements by instrumenting the code, and using the Pentium counters either directly or through EMON, a third–party tool [7]. EMON is an interface to using the Pentium counters for measuring a large number of CPU events, such as branch predictions, cache misses, resource conflicts, types and numbers of executed instructions. Some of the higher–level measurements are performed with the Performance Monitor tool available in Windows NT. 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 2 minutes (and usually multiple times to verify consistency of our results).

Performance metrics: We monitor a number of statistics. Although we are interested mainly on the game server behavior it is essential to monitor both the server and the client. On the server side we measure: (i) the number and size of incoming and outgoing messages that help us determine bandwidth requirements, (ii) the request processing time that shows how computationally intensive the game plot is, (iii) the execution time breakdown in compute time, send and receive processing time[1], and idle time, (iv) the number of client requests received and processed in each server time–slot, and (v) internal processor statistics about branch predictions, cache misses, instruction fetches, and resource conflicts. On the client side we measure: (i) incoming and outgoing bandwidth requirements (mainly for verifying our server measurements), (ii) request rate to the server, (iii) response rate from the server, and (iv) response time. We also use processor utilization and graphics rendering time as intermediate metrics to develop our methodology.

Capturing user intentions in action files: To automate the benchmarking procedure we propose replacing human with automatic players. *Quake* includes a demo–recording facility that can be used to record the keyboard and mouse events of a player to a file for later playback. The recorded events make sense only if the client is always spawned at the same starting location of the map as was used at the beginning of the recorded clip. Otherwise, the player might be, for example, walking into walls or falling into pits, or performing other actions a human player would obviously not perform in the same situation. We modify the client code to obtain user input from a previously recorded file rather than from the keyboard or mouse. Thus, there is no need for human players to interact with the server. From our experience, the game–play is still very interactive and *logical* with the automated players.

Providing standard maps: 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. To capture these aspects we use mainly two maps in the presentation of our results: A small and simple map that forces a high level of interaction among players even at small player counts. This map contains a set of small, interconnected rooms and due to its size it cannot be made very complex. The second map we use is a larger and much more complex map that includes many objects and an elaborate layout, which, however, may not force as many player interactions as the smaller map.

[1]Receive interrupt handler processing is currently accounted for in the compute time part.

4 Understanding System Behavior

We perform our basic experiments with the small input map using 1, 2, 4, 8, 12, and 16 players. This study allows us to identify computation and network processing scaling trends for a player load for which the server is not overloaded. In order to ensure that the perceived player behavior is similar to setups with human players we always use one human player during the experiments and compare results on the client side across all clients. This is a sanity check procedure we follow at this early stage of understanding this class of applications, to make sure that the values of the metrics we measure are similar for automated players and human players. We expect that future work will use only automatic players. The server always runs on a dedicated system. In this set of experiments, each player runs on a different, dedicated client as well. In all cases we show the average value for each metric. Where meaningful, we include error bars that show the minimum and maximum values and an explicit measurement for the average value of the same metric for a human player. Also, we include measurements for one human player (1R) and for one automatic player (1V) to show that they behave similarly with respect to each metric.

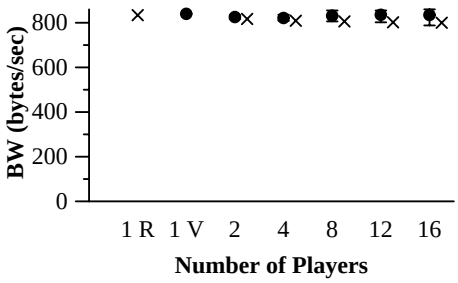

Figure 4: Server incoming network throughput per client. The 'x' plot represents the human avg for each run.

Server Network Throughput: Figure 4 shows the per player incoming network throughput in the server. We see that on average each player sends about 830 bytes/sec. Moreover this throughput is independent of the number of players. We also see that the incoming network throughput for the human player is practically identical to that of the automatic players. The difference between the minimum and maximum per client incoming bandwidth can be translated to approximately a difference of 1 request/sec. The (incoming) request size does not vary with the number of players and therefore we do not plot it.

Figure 5 shows the per client outgoing network throughput at the server for different numbers of total clients. We see that the average throughput to each client increases with the total number of players and doubles when going from one to about twelve players. Figure 6 shows that

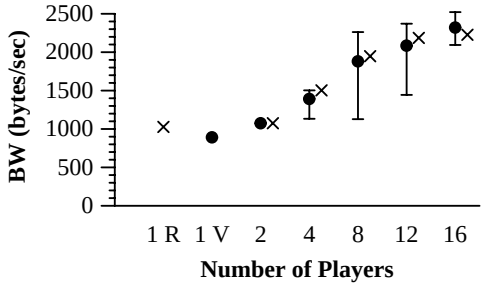

Figure 5: Server outgoing network throughput per client. The 'x' plot represents the human avg for each run.

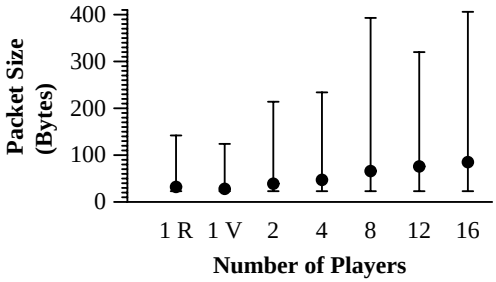

Figure 6: Server response size.

the average (outgoing) reply size is increasing almost linearly with the number of players, indicating an increase in the amount of information the server is sending out to clients rather than an increase in the number of messages per client. The reason for this increase is that as more players interact, the server needs to communicate more information to each player involved. This measure may be highly influenced by the geography of the input map. Our experiments indicate, however, that in practice, server network throughput requirements do not vary significantly across input maps (Section 6). Finally, we note that the outgoing network throughput for the human player is practically identical to that of the automatic players.

These results indicate that network throughput is not a problem and that a single, 100 Mbit/s Ethernet connection could support hundreds or even thousands of users. However, when trying to scale beyond that level, a single network connection may not suffice.

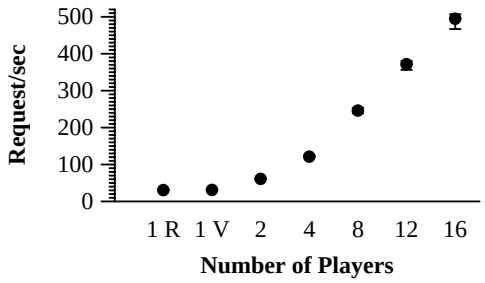

Figure 7: Server received request rate (requests/s).

Server Received and Processed Requests Throughput: Since the server first receives all client requests and then processes them, we look at the number of received and processed requests separately. Figure 7 shows the number of

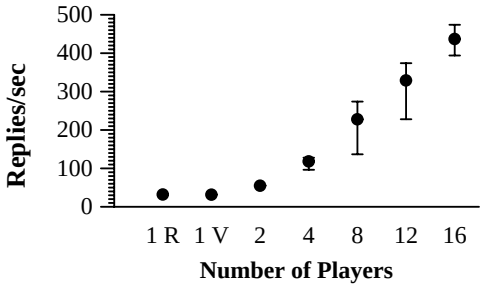

Figure 8: Server response rate (replies/s).

requests received by the server. We see that, as expected, the number of requests/sec increases linearly with the number of clients. Figure 8 shows the number of requests that are processed at the server per second. The average number of requests/sec processed increases linearly with the number of clients indicating that the server is not saturated. Moreover, if we compute the minimum and average request processing times, we observe that they stay about the same across different player counts, indicating that request processing does not depend significantly on the total number of players. However, the maximum request processing time varies noticeably across different player counts.

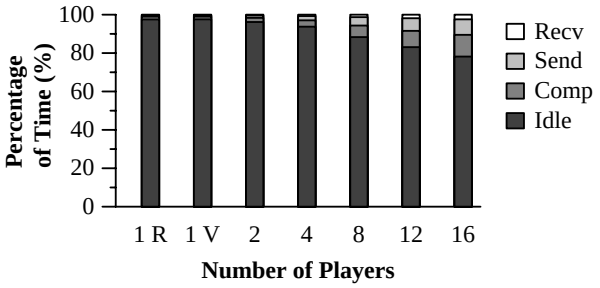

Figure 9: Server execution time breakdown.

Server Execution Time Breakdown: Figure 9 shows the execution time breakdown at the server. This measure is a high level metric that provides an overall picture of the activity in the server. However, it provides cumulative information across the full experiment and does not show potential bursty behavior that may affect perceived quality at the clients. We see that network stack processing time and game–related processing time (indicated as compute time) increase linearly with the number of players. At 16 players about 75% of the time the server is idle indicating that it is not saturated. Finally, we observe that the server spends about an equal fraction of its time in network and in game–related processing. These results suggest that both game–related processing as well as network stack processing should be distributed across multiple processors to achieve high–levels of scalability. Thus, multiple network connections may be necessary, depending on the underlying system architecture, not to increase the required network bandwidth, but to facilitate the reduction of network processing overheads by using multiple host CPUs for network protocol processing over separate network interface controllers (NICs).

Client Rendering Overhead and Frame Rate: Figure 10 measures the overhead associated with rendering a single

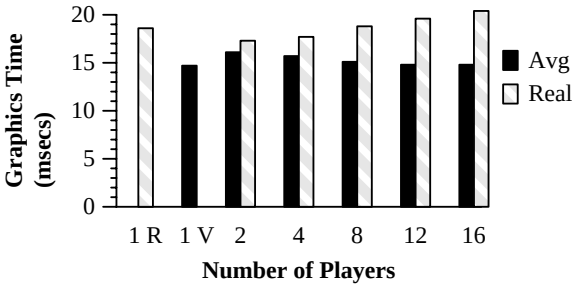

Figure 10: Graphics Rendering Time for Clients.

frame in our clients. This measure also includes any context switching overhead during frame rendering time. Overall, the single–frame rendering overhead is low and does not limit frame rate. The average overhead in each of our clients is about 20ms which allows for a frame rate greater than 45fps. Note that clients render frames at their own rate regardless of whether the server responds or not. When the server does not respond the client renders the next frame by using extrapolation from previous actions. Thus, this metric shows only if the client system is able to support a specific frame rate. As expected, the average frame rate for clients does not change as the number of players increases indicating our client's capability of sustaining adequate processing rates.

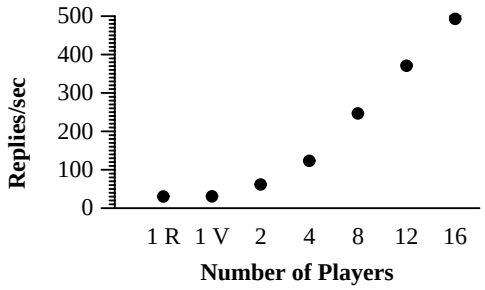

Figure 11: Server response rate as measured at the clients.

Client Received Response Rate: Figure 11 shows the server response rate as measured at the clients. Note here that the response time is measured at the client side when the client checks for server replies[2]. If we calculate the average response time (from Figure 11) we see that it is always within a 30ms–range time–slot, indicating that clients receive server replies on time to update their state before rendering the frame. However, our measurements show that response times exhibit a high variation with maximum values in the 100ms range and occasionally higher. Finally, as expected, we note the matching between response rate measured at the server (Figure 8) and the same measurement at the clients (Figure 11).

[2]Thus, it is possible that server replies arrive at the client prior to the time the client code checks the incoming port for server messages. However, the client need not process any replies prior to the time it checks for them. If a reply is found when the client checks the port, then there is no perceived delay according to the client. However, if a reply is not found when the client checks the port, then the client is forced to render a frame without an update from the server. This leads to unnatural movements if it occurs often. Thus, it is safe to monitor the response time as explained above.

Summary: We see that existing systems can comfortably support a few tens of players (server idle time for 16 players is about 75%). All of the presented statistics are useful in understanding server behavior. However, we propose using the total server response rate and the average response time as measured in all clients for comparing overall server performance across different experiments. This approach is similar to online transaction processing systems [13], where the number of completed transactions per second is the primary metric for server performance.

5 Scaling the Number of Players

Our main goal in studying server performance is the ability to support large numbers of players. However, currently, this requires an equal number of client systems. To eliminate this constraint we propose running multiple players on each client. The main issue with this approach is to not introduce new bottlenecks, either in the CPU, network, or any other resource on the client side. This approach does not introduce bottlenecks in terms of memory and network bandwidth and latency requirements. The client code uses at most 32 MBytes of memory (per player) which will comfortably allow for up to 16 players on each of our client nodes. Also, as discussed previously in Section 4, the network bandwidth available in our setup is sufficient for hundreds or thousands of players. In the next paragraphs we describe what system modifications are necessary to multiplex players on the same client, under what circumstances this is equivalent to using one player per client, and what is the maximum number of players that can be supported per client.

Modifications: In order to enable more automatic players to share a client node we modify the client code to reduce its compute cycle requirements. Our analysis shows (Figure 10) that from the three steps performed in each time–slot in the client (Figure 2), on systems without special graphics support, most of the time is spent rendering new frames, one in each time–slot. Given the desired frame rate of about 25–30 frames per second, each time–slot is about 30 ms. As explained about 20ms are spent rendering each frame. The rest of the time (about 10ms) is spent in other useful computation or spinning until it is time to start the next time–slot. To free compute cycles on the client side, we replace both the frame rendering procedure and any spin–waits in each time–slot with `sleep()` system calls that drop in the kernel. The client sleeps for some amount of time and then wakes up to continue with the next iteration. Our `sleep()` timer is such that it corresponds to rendering 30 frames per second.

Results: We now increase gradually the number of players from 2 to 16 on the client node and we compare our results with the single–player measurements in Section 4. Figure 12 shows that each client can support at least up to 24 players per node with overall processor utilization under 15%. Furthermore, the task queue length (not shown) is always very close to zero indicating almost no contention among the players that use the processors at the same time.

Moreover, all our results on the server side show that server behavior remains identical[3]. In particular, comparing server response rate in the multiplexed case with the case where each player runs on a single client (Figure 11)

[3]We omit these results for space reasons.

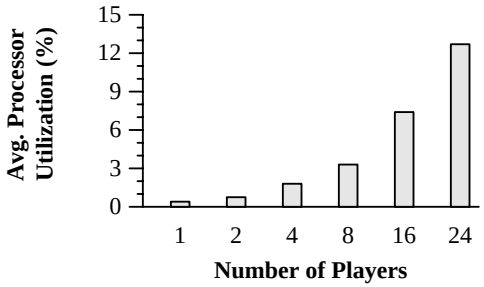

Figure 12: Client average processor utilization for processor 1 without the rendering and spin–wait overheads.

we find that the server response time for player requests is very similar on average for all player counts up to 16 players. Thus, we can multiplex at least up to 16 players on a single client node without introducing new bottlenecks. This number can be raised if necessary in future work, depending on the specific parameters of the client systems.

6 System Scalability

6.1 Increasing the Number of Players

In this section we present results for larger numbers of players. Figures 13–17 present our results for the small map with up to 96 players (16 players per client). The incoming network throughput and request rate continue to increase linearly with the number of players (the clients pay no attention to the lagging server, and continue sending their requests at the normal rate). We see (Figure 13) that after 64 players the server is saturated. The server is not able to keep up with player requests and the outgoing network throughput and server replies start dropping (Figures 13, 14). Specifically, we notice that upto 32 players the response rate seems to be doubling as we double the number of players. With 64 or more players the server does not sustain this trend and is therefore saturated. The execution time breakdown in Figure 15 shows that idle time drops to almost 0% at about 80 players. Processing time is still divided almost equally between network stack processing and game–related processing. Looking at request processing time (Figure 16) we see that it increases very little with the number of players (about 10% between 1 and 80 players). This indicates that the reason for the reduction in server response rate is the fact that the request queue increases linearly with the number of players and the server spends more time receiving requests as opposed to processing them. Finally, the maximum response time increases linearly with the number of players (Figure 16) as does the average and maximum server reply sizes (Figure 17).

6.2 Using Larger Maps

We now examine the system behavior with a larger and more complex map (Figures 18– 20). Figure 18 shows that the server response rate drops after 64 players, similarly to the small map results. The execution time breakdown (Figure 19) shows that at 64 players the idle time drops to about 3% as opposed to about 10% with the small map. This increase in compute time compared to the small map is due to an increase in the average request processing time at the server as demonstrated in Figure 20. However, similarly to the small map, the average request processing time

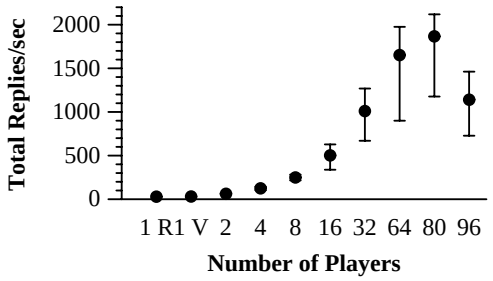

Figure 13: Server response rate (small map).

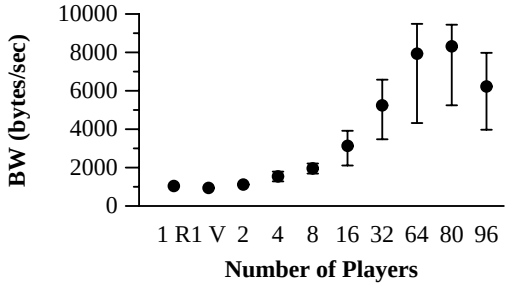

Figure 14: Server outgoing network throughput per client (small map).

does not vary significantly with the number of players and thus the server response rate drop is due to the size of the incoming request queue. As a consequence of the increased server processing time, outgoing server network throughput drops by about 1KByte/s with the larger map. Server incoming network throughput, server incoming requests per second, server outgoing reply size, and client rendering time are all virtually identical to the results obtained with the smaller map and are not shown here. In summary, between maps, the main source of differences is the average request processing time at the server and the induced increase in server–CPU utilization.

6.3 Increasing CPU Speed

To investigate the impact of faster processors on server performance, we now provide data for a server with 800MHz processors (please refer to Section 3 for more detail about the system). Figure 21 shows that the server response rate does not drop after 64 players but increases from 64 to 96 players by about 25%. Also, the average request processing time drops from the 170μs–range (Figure 20) to the

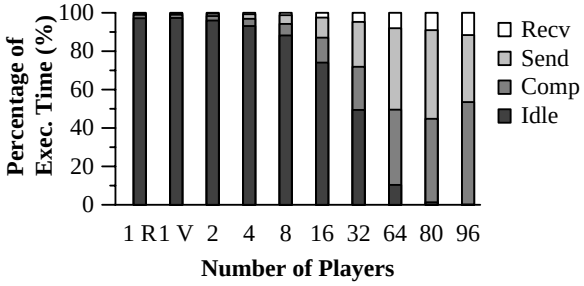

Figure 15: Server execution time breakdown (small map).

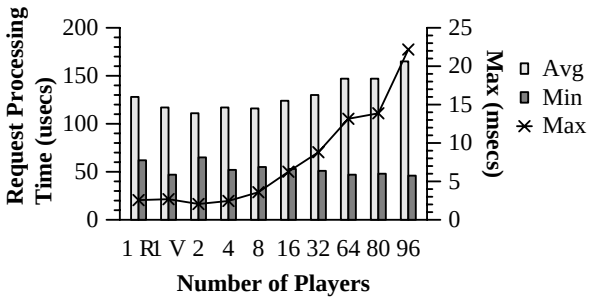

Figure 16: Request processing time at the server (small map).

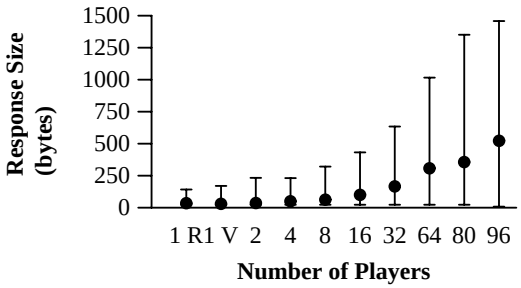

Figure 17: Server response size (small map).

90μs–range (Figure 23). The execution time breakdown (Figure 22) shows that the idle time at 64 processors is about 35%, compared to about 3% with 400MHz processors. However, at 96 players idle time drops to about 10% indicating that increasing the number of players further will quickly saturate the server.

6.4 Server CPU Behavior

In this section we present the results of our low-level processor event analysis of server behavior for the base 400MHz system. We used EMON [7] a hardware-counter manipulation tool provided by Intel to access the hardware performance counters of the Pentium II processor. Since during a single run of the server we can only measure two hardware events, we repeated each experiment multiple times measuring each time a different pair of events. There was only a negligible variation in measurements across different runs. Consequently, we were able to combine results from different runs as if they came from a single experiment. In the interest of space we restrict our attention to experiments with 64 players and that use the small map.

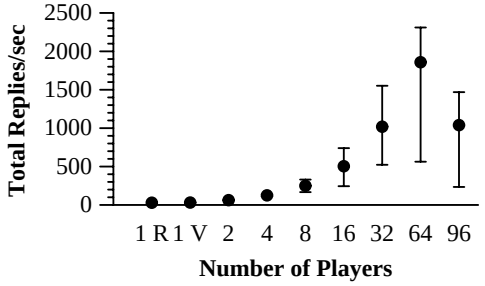

Figure 18: Server response rate (large map). In this subsection we omit the 80 player point in graphs since our main goal is to investigate the similarities between using small and large maps for experiments.

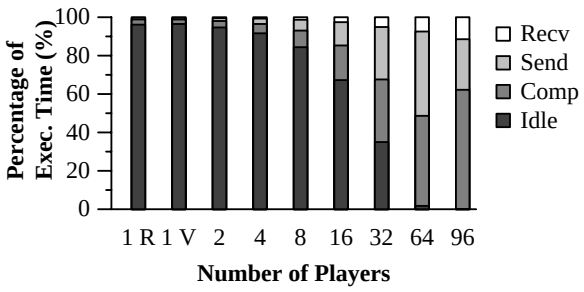

Figure 19: Server execution time breakdown (large map).

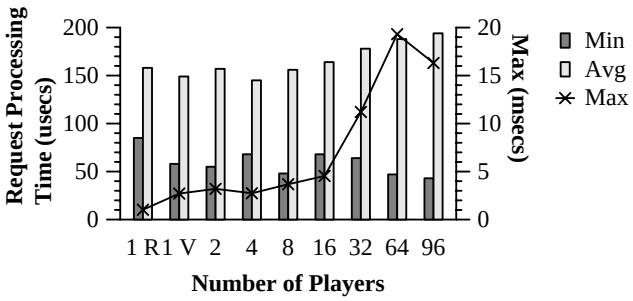

Figure 20: Request processing time at the server (large map).

The server achieves an IPC of 0.416 even though Pentium II processors can potentially issue multiple instructions per cycle. This suggests that the processor is frequently stalled. Ideally, we would want to measure the relative importance of each of the many possible stall causes. This is a challenging task as many of these stalls overlap (e.g., a data miss could happen at the same time as a branch misprediction, or two data misses can be in-progress simultaneously). Had we used simulation, it would have been possible to provide additional insight by selectively eliminating each stall source (e.g., by simulating perfect memory we can gauge the importance of memory stalls). However, given that we use an actual system we resort to indirect metrics.

To gain insight on the low IPC we first report statistics on the mix of instructions executed by the server. As shown in Table 1, 15.7% of the instructions are branches, 63.5% access memory and only 4.8% are floating point operations.

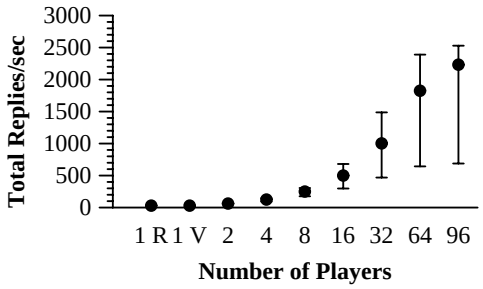

Figure 21: Server response rate (800MHz processors, large map).

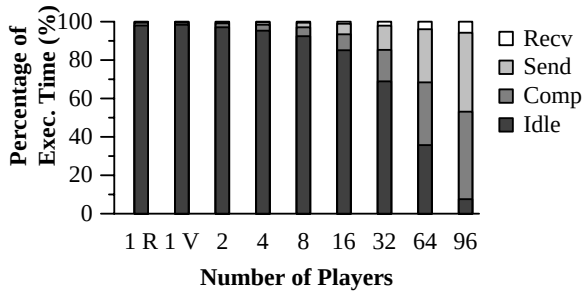

Figure 22: Server execution time breakdown (800MHz processors, large map).

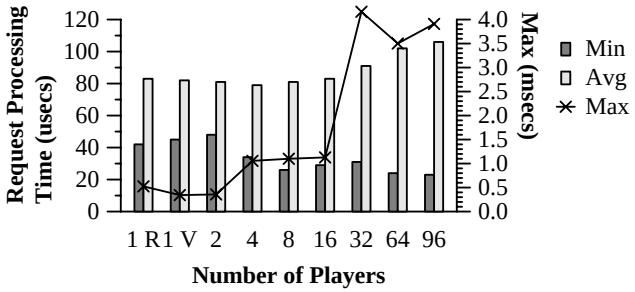

Figure 23: Request processing time at the server (800MHz processors, large map).

These fractions are quite different than the widely used rule of thumb where memory referencing instructions acount for 1/3 of all instructions and branches for 1/5. A relatively high fraction of memory references is common in IA-32 machines due to the relatively low number of registers. This is further exaggerated by the nature of the processing that takes place where relatively large datastructures (e.g., those representing the 3D world) are continuously traversed.

Table 2 reports the resulting IPC, branch missprediction and the instruction and data miss rates for the L1 and L2 caches. The branch misprediction rate is very high, about 14%. Given that more than 15% of the instructions are branches, this means that mispredictions occur every 45 instructions on the average. Taking into account that in Pentium II branch mispredictions induce a penalty of at least 10–15 cycles, this suggest that the instruction window is heavily underutilized.

Focusing on the memory system, we see that all caches observe very low miss rates for instructions and data, suggesting that on systems with reasonably fast memory, the memory subsystem should not be a problem.

Finally, to provide additional insight on memory performance, we report in Table 3 the percentage of memory references that are serviced at each level of memory. We see that for instructions only 0.07% ends up in main memory whereas the same number for data is 0.44%. That is, only 1 every about 1400 instruction references reaches main memory and 1 every about 230 data references does so too.

Branch	Mem	FP	Other
15.7%	63.5%	4.8%	16.0%

Table 1: Breakdown of committed instruction types (64 players, small map).

IPC	Branches	L1I	L2I	L1D	L2D
0.416	14.0%	0.6%	10.5%	7.2%	6.1%

Table 2: Server statistics: IPC, branch misprediction percentage, and cache miss rates for L1 and L2 caches, divided in instruction and data misses (64 players, small map).

	L1	L2	Main Memory
Instr.	99.4%	0.53%	0.07%
Data	92.8%	6.76%	0.44%
Total (I+D)	97.6%	2.26%	0.14%

Table 3: Percentage of Instr. fetches and Data references serviced by the various levels of memory hierarchy (64 players, small map).

These provide an additional indication that memory performance seems to be only a secondary issue.

Finally, we measure the resource stalls and we find that the processor spends about 25% of its time waiting for resources to become available. If we combine all the above observations we see that the very low IPC is due to branch mispredictions, resource stalls, whereas memory subsystem stalls seem to be secondary. From our measurements it is not possible to draw any conclusions about the available instruction level parallelism. Further looking into these issues, although very interesting, is beyond the scope of this work and is left for future research.

7 Related Work

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 (e.g., [3, 9, 14]). Work on processor and memory subsystem performance has mostly examined the behavior of multimedia applications (e.g., [4, 10]) and database systems (e.g., [1, 2, 8, 11, 12]). However, we are not aware of any work that has studied the behavior of applications in the area of interactive entertainment and in particular game server applications. Finally, the methodology we develop in this work bears similarities with previous work on the methodology used for benchmarking online transaction processing systems [13].

8 Conclusions

In this paper we investigate the behavior and scalability of a commercially–oriented application, an interactive, multi–player game server, *Quake*. *Quake* is a popular representative of interactive entertainment applications. Our goal is to understand the behavior of this class of applications and to provide a novel benchmark for scalable servers.

We first analyze the behavior of the game server and provide insight on the characteristics of this class of applications. We develop the required methodology to deal with idiosyngracies of interactive applications, namely the fact that human users generate the input and that comparisons across runs may not be consistent. We find that network bandwidth is not an issue since the game server circulates only control information and all graphics operations are handled by clients. The incoming required bandwidth per player is practically constant and in the order of a few KBytes/s whereas the outgoing bandwidth per user

depends on the total number of players (and the induced interactions) but does not exceed a few KBytes/s. Thus, current network connections on game servers can support thousands of users. Similarly, memory requirements are not an issue for modern systems. We propose the use of the server response rate and the average response time for evaluating overall server performance. These metrics should be reported as measured on the client–side.

We extend our methodology to enable experiments with large numbers of players on small experimental setups and we verify that our modifications do not introduce additional bottlenecks or other erratic behavior. We use this extended methodology to examine system scalability. We find that current architectures can support a few tens of players (our systems can support between 60 and 100 players) and that server utilization increases linearly with the number of players. Future microprocessor technologies may be able to provide increases in the number of supported players by a small multiplicative factor (doubling the speed of the CPU results in a less than double increase in the number of players) but only parallelization can extend this to the order(s)–of–magnitude required improvements. Also, multiple network interface controllers may be necessary, depending on the underlying server architecture, not to increase the required network bandwidth, but to facilitate the reduction of network processing overheads.

We analyze CPU behavior in terms of micro–events and we find that: (i) the server CPU achives a very low IPC of 0.416 out of the theoretical maximum of 4. (ii) The branch misprediction rate reaches a high of 14%. (iii) The memory subsystem exhibits very good locality and should result only in secondary effects. (iv) The CPU spends about 25% of the cycles stalled on resources. These results suggest that there is a lot of room for improving server performance at the microprocessor architecture level.

Similarly to all benchmarking efforts, our work does not capture every aspect of server behavior, given the limited number of inputs we use. However, we are able to make a decisive first step towards understanding and quantifying system behavior. Our next step is to use commodity–based parallel architectures, such as small–scale symmetric multiprocessors (SMPs) and larger–scale shared memory clusters that, cost–wise, scale linearly with system size, to examine if they can support hundreds or thousands of simultaneous players.

Finally, scaling game servers beyond a few tens of players may require rethinking their internal architecture. Nevertheless, our study is still valuable as it provides insight on what parts of the server may have to be parallelized or are worth to look at.

9 Acknowledgments

The authors would like to thank *id Software* [5] for releasing the source code for *Quake* in the public domain. We would also like to thank Reza Azimi for useful discussions on various aspects of this work, Peter Jamieson, and Eugenia Distefano for helpful hints on setting up and using the experimental testbed. We are thankful to the reviewers for their insightful comments. Finally, we thankfully acknowledge the support of 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] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of the 25th Intl. Conference on Very Large Databases*, pages 266–277, September 1999.

[2] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *ISCA*, pages 3–14, 1998.

[3] A. Bilas, J. Fritts, and J. P. Singh. Real time parallel MPEG-2 decoding in software. In *Proceedings of the 1996 International Parallel Processing Symposium*, April 1996.

[4] J. Fritts, W. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. In *SPIE Photonics West, Media Processors '99, San Jose, CA*, pages 2–13, January 1999.

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

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

[7] Intel Corp. EMON. http://www.intel.com.

[8] Kimberly Keeton and David Patterson. Towards a simplified database workload for computer architecture evaluations. In *Workload Characterization for Computer System Design, edited byh L. K. John and A. A. Maynard, Kluwer Academic Publishers*, 2000.

[9] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multiuser commercial workloads. In *The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, October 1994.

[10] Parthasarathy Ranganathan, Sarita V. Adve, and Norman 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.

[11] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz Andre Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Architectural Support for Programming Languages and Operating Systems*, pages 307–318, 1998.

[12] P. Trancoso, J-L. Larriba-Pey, Z. Zhang, and J. Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, 1997.

[13] Transaction Processing Performance Council. Transaction Processing Performance Council Home Page. http://www.tpc.org.

[14] Z. Zhang and S. Sarukkai. Commercial applications on shared-memory multiprocessors. In *In Proceedings of the First Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 1998.