Investigating Informative Performance Metrics for a Multicore Game World Server

James Munro, Kofi Appiah, Patrick Dickinson
School of Computer Science, University of Lincoln, Lincoln, UK
Corresponding author:pdickinson@lincoln.ac.uk

Abstract

Many real-time game world servers run on stand-alone PCs, such that user performance is bound to fairly modest hardware configurations. Studies of multicore architectures to optimize such servers are sparse, and evaluations typically involve the use of one or two arbitrary performance metrics. However, the behavior of game servers is complex and the interpretation of metrics, particularly in the case of parallel implementations, is not straightforward.

Our initial interest is in efficient load-balancing of multicore game engines. However, the focus of this paper is on performance metrics: starting with proposed metrics from other works, we investigate their effectiveness and inter-relationships, propose new variants, and discuss how they can be used in combination to gain a better understanding of actual performance.

The use of metrics to inform the design and optimization of game software has gained recent interest from academics and practitioners alike: we conclude to show, by example, how server metrics can be directly connected with game semantics, and used to predict the impact of game design changes on server performance.

Keywords:
Multi-core games server; performance evaluation; game server metrics

1. Introduction

Multiplayer games range from the technologically simple, to sophisticated endeavors such as *Massively Multiplayer Online Role-playing Games* (MMORPGs). The concept of a *client-server* architecture is ubiquitous: in

the case of MMORPGs, expansive environments are hosted on bespoke server configurations which facilitate huge numbers of users. For example, by 2007 the game *EVE Online* had recorded over one million unique players since its launch in 2003 [1]. Whilst a significant amount of research has investigated the use of distributed architectures to support large-scale game servers (e.g. [2, 3, 4, 5, 6, 7, 8]), this type of server setup is exceptional.

Many games allow players to create their own stand-alone *ad-hoc* servers which service smaller game worlds with tens rather than thousands of players. These servers run on standard consumer equipment, and performance is (unsurprisingly) closely bound to processing power [9]. Single machine servers represent a major part of the currently available multiplayer online gaming service, and are common for *first person shooters* games which involve fastpaced interactive gameplay and real-time simulation. Player experience for this game type is particularly sensitive to degradation in performance, in the order of milliseconds [10, 11], and so server optimization represents an ongoing challenge for developers. It is therefore surprising that relatively little work has been directed at optimizing stand-alone servers to utilize the parallel processing architecture of multicore CPUs. Game metrics have attracted recent academic interest (e.g. [12, 13]), and also interest from industry where they are perceived as a valuable tool for design, balancing, and optimization. As Abdelkhalek *et al.* note [9], benchmarking methods for interactive game servers are driven by somewhat different considerations from scientific processing: useful performance evaluation should reflect user experience in some way. Again, little work has yet considered suitable server-side metrics for the analysis of real-time multicore game engines.

1.1. Motivation

The starting point for our work is an existing server design proposed by Cordeiro *et al.* [14], implemented using *id software's QuakeWorld* game server. Cordeiro's work uses spatial partitioning to divide entity processing into discrete non-intersecting work packages which can executed in parallel (details of the architecture are given in section 2.3). Our initial interest is in load balancing, and optimizing the distribution of work packages across hardware threads; however, a survey of current work in this area reveals that the use of performance metrics is not standardized, making it difficult to compare algorithms. Moreover, a single metric is not in itself entirely

informative, and often leaves questions remaining about the the underlying processes. The measurement of performance of a multicore server thus becomes our primary interest, such that the motivations for our study are:

1. To investigate the relationship (if any) between currently used serverside performance metrics.
2. To determine which metric, or set of metrics, provide the most informative analysis of performance.
3. As a secondary motivation, we are interested in the impact that thechoice of load balancing algorithm has on performance in Cordeiro *et al.*'s architecture: this provides a context for points 1 and 2.

As mentioned, useful performance evaluation should reflect player experience in some way. In terms of perceived responsiveness, experience is a function of several factors of which server performance is just one. Others include data transmission latency, client-side performance, and also game play context: for example, the affects of latency on player experience have been well-studied (e.g. [11, 15, 16]). A proper analysis of perceived responsiveness encompasses all these factors, is context dependent, and lies outside the scope of the work presented here. Our focus is specifically on identifying meaningful comparators for multicore server architectures, which may be used to quantify performance and independently optimize design. Nevertheless, our metrics do relate directly to player experience. For example, we will use *server throughput*, which is a direct measure of the number of connected clients that can be processed concurrently, and so has a direct effect on experience.

1.2. Contributions

Our study takes the form of a set of empirical investigations into the performance of different simple load balancing strategies used in conjunction with Cordeiro *et al.*'s *QuakeWorld* server [14]. These experiments are primarily constructed to investigate the response of different metrics. Building on our preliminary results, presented in [xx], the contributions of this paper are:

1. We evaluate the effectiveness of a range of server-side metrics including frames per second, server throughput, thread wait time, and accumulated thread work load. We present conclusions concerning their inter-relationships and effectiveness, and which are most useful

in analyzing performance. A study of performance metrics in the context of multicore game servers has not previously been conducted, and is
of immediate use to developers working on stand-alone game server applications.

2. In relationship to Cordeiro *et al.*'s architecture [14], we show by example how metrics can be used to estimate the effect of game design changes on server performance.
3. We investigate the effects of different load balancing algorithms onserver performance. We use only simple balancing techniques, but these are still able to characterize the importance of effective thread balancing in Cordeiro *et al.*'s system. We further investigate how these results scale across varying numbers of CPU cores, ranging from one (serial) to six concurrent hardware threads, using our metrics.

Whilst we use a specific architecture and game engine to conduct our experiments, our results are easily generalized. The proposed metrics are low-level statistics which describe the performance of workgroups processed on hardware threads: these are thus independent of the workgroup allocation strategy, and equally applicable to any multicore game server design. Furthermore, the lockless server design which we employ [14] is based on the semantic constraints of objects moving in a physical simulation. This design may therefore be transposed to any functionally comparable game engine (e.g. *first person shooter*, or game which simulates a physical world).

1.3. The Structure of this Paper

The rest of this paper is presented as follows. Section 2 reviews the current literature regarding parallel and concurrent processing architectures and metrics in game engines, specifically server-side, and concludes with a description of the *QuakeWorld* server, and a detailed description of the parallel implementation presented by Cordeiro *et al.* Section 3 proceeds to describe our experimental setup, and is followed by sections 4 to 7 which present our experimental work and discussions of performance metrics. We conclude with a discussion of our results, and motivate some conclusions regarding the use of server-side metrics, and load-balancing strategies for stand-alone multicore game servers.

2. Background and Related Work

Whilst relatively little work has addressed the evaluation of multicore game servers, there has been considerable wider interest in the use of concurrent architectures to optimize game software. Aspects of client-side processing have been addressed by Gildea [17], who attempted to adapt the *Quake 3* client to support parallel execution (with limited success). He identified the difficulty in reconstructing concurrent processing threads which access shared memory. The use of GPUs to implement concurrent graphics processing is well established. Their potential for use in non-graphical processing in game engines has also been investigated: [18, 19, 20, 21, 22].

Our interest lies specifically in the optimization of game servers. A number of studies have considered distributed architectures: Bharambe *et al.* [23] succeeded in scaling the *Quake II* engine over many server nodes, supporting hundreds of players. A study by Ploss *et al.* [24] parallelized the *Quake III* server using a purpose-built scalable grid framework. A number of other studies ([3, 4, 6, 7, 8]) have dealt with distributing game state across multiple nodes.

2.1. Optimizing a Stand Alone Server

Practical considerations dictate that *ad hoc* servers are implemented on stand-alone machines; however, relatively little work has investigated the implementation, optimization, and benchmarking of appropriate parallel architectures. As mentioned, Abdelkhalek *et al.* [9] analyzed the performance of the standard sequential *QuakeWorld* server, empirically determining an approximately linear relationship between processing overhead and the number of players. They discussed the difficulty of meaningful benchmarking: noting the functional similarity with online transaction processing, they propose the use of server throughput and CPU idle time, as performance metrics.

In further work, Abdelkhalek and Bilas [25] implemented a parallel version of the *QuakeWorld* server. The response processing and reply phases were processed by concurrent threads running on separate cores of a quadcore CPU. Parallel execution was achieved by assigning each player permanently to a specific thread; however, memory synchronization was a limiting factor, and the resolution of lock contentions represents up to 35% of total execution time. An analysis showed that peak response occurs with around 25% more players attached than the serial version, which is a

significant improvement. In this work, Abdelkhalek and Bilas use only response rate and aggregated thread workload to analyze performance: we will show in our experiments that these alone are not sufficient to fully understand the behavior of a parallel server. Very recent and interesting work by Raaen *et al.* [26] proposes a complementary lockless processing architecture, implemented using a simple bespoke game. In this case, each entity is considered an atomic process and restrictions are placed on interactions. Server response time and CPU load are used to compare single and multi-threaded implementations, but as with Abdelkhalek and Bilas, these are insufficent to fully understand behaviour, or to compare with other architectures.

A number of studies have explored the use of software transactional memory (STM), as an alternative to lock-based shared memory. Results are thus far inconclusive. Zyulkyarov *et al.* [27] built *Atomic Quake* upon the parallel *QuakeWorld* server developed by Abdelkhalek *et al*. However, their system was not as effective as the original lock-based system. Gajinov *et al.* [28] developed another STM-based modification of the *QuakeWorld* server. They were able to achieve better performance; however, the the overheads incurred by STM were again high. A subsequent study by Lupei *et al.* [29, 30] introduced *SynQuake*, an STM-based server derived from *Quake III*. These results were more promising, reporting better performance and scalability than lockbased strategies. However, their experimental work does not use real clients connected on a network, which are essential to accurate evaluation of game servers [31].

In most cases, one quantifiable metric such as server throughput ([27]), or frame execution time ([28] [29, 30]) is used in an *ad hoc* fashion, without thorough consideration of how that metric relates to the underlying performance.

2.2. The QuakeWorld Server

Several existing studies have made use of *id software's Quake* series of game engine servers (e.g. [9, 23, 10, 24]). *QuakeWorld* is particularly suitable for academic study: the full client/server source code was released under the GNU General Public License in 1999, and is therefore fully accessible. Whilst the *QuakeWorld engine* is relatively old, it is directly related to more modern game engines. For example, it uses local prediction to compensate for highlatency networks, and was the first *Quake* engine designed for internet-

based play. *Quakeworld* was used as the basis for later game engines such as *Valve*
Software's GoldSrc engine.

The standard serial *QuakeWorld* server frame update comprises the following processes:

1. Processing the world state.
2. Receiving client input.
3. Entity/client processing.
4. Outputting responses to clients.

World state processing involves entities not associated with a player, and consumes a small portion (5%) of the overall processing time [9]. Step 2 collates inbound network packets: only clients which have sent input to the server will be processed in the current frame, and inputs are validated against game world semantics. Analysis by Abdelkhaleh *et al.* [9] shows that steps 3 and 4 (Entity/client processing and Outputting responses) together account for typically 90% of the frame processing time in the serial version. Moreover, execution time (in serial) scales approximately linearly with the number of clients. Client processing involves applying player inputs, and then executing the game world simulation associated with that player, such as movement, creation of new objects, and so on. Once all of the requests have been processed, the results of the frame are transmitted via the network connection to all clients.

2.3. A Lockless Server Architecture

Subsequent work by Cordeiro *et al.* [14] leverages game semantics to avoid the critical problem of synchronizing shared memory. Like Abdelkhalek and Bilas, concurrent processing is implemented using multiple hardware threads running in parallel on a multicore CPU: this is applied to the response processing and reply phases, which account for the majority of the frame processing in *QuakeWorld*. However, they have designed a strategy which predivides response and reply processing into workgroups which are guaranteed not to access the same memory resources.

Each object in the game world has its own distinct memory resources, and response processing in *QuakeWorld* potentially involves processing each of these objects, every frame. Two objects which do not interact with each other may be processed concurrently without synchronization problems.

However, interacting objects can result in lock contentions: this is the source of the wait time reported by Abdelkhalek and Bilas.

Cordeiro *et al.* introduce a pre-processing step which groups game objects into subsets which are independent and cannot interact during the current update. That is, objects within the same subset *may potentially* interact with each other, but not with any object in any other subset. These subsets are inferred from the spatial distribution of objects. Each object forms the node of a graph $G(V,E)$, where an edge $e_{i,j} \in E$ represents the Euclidean distance between objects i and j in the game world coordinate system. Each object has an associated maximum range of movement in the current frame d_i, such that if $e_{i,j} \in E > (d_i + d_j)$ then it is impossible for the two objects to interact. A connected components algorithm is used to identify subsets such every node $v_i \in V$ is assigned to a subset S_α such that:

$$v_i \in S_\alpha \longrightarrow \exists v_{j6=i} \in S_\alpha : e_{i,j} \leq (d_i + d_j) \wedge \neg \exists v_k \in S_{\beta 6=\alpha} : e_{i,j} \leq (d_i + d_k) \quad (1)$$

In Cordeiro's implementation, $d_i = d_j = d_k$ a constant equal to the *action distance* of entities in the game world: that is, the maximum spatial interaction of an entity, as defined within the engine (which is independent of frame rate). The subsets then form distinct workgroups which may be safely executed concurrently on separate threads. Each thread is managed separately: a copy of the relevant parts of main memory is created for each using a feature of the Linux-kernel known as *copy-on-write* [32]. These copies are nonintersecting, and so trivially resynchronized at the end of the reply phase. Cordeiro *et al.* use a dynamic load-balancing strategy to distribute workgroups on a per-frame basis across the available threads (including the main thread). This is achieved by weighting each package according to the number of objects it represents, then distributing packages based on a Longest Processing Time First (LPT) algorithm. They were able to increase the the server time spent in parallel execution to 55%, from the 40% achieved by Abdelkhalek and Bilas. The final frame process structure is shown in Figure
1.

2.4. Motivation for Our Work

The work presented by Cordeiro *et al.* implements a lockless game server, dynamic load balancing, and is clearly generalizable: these game semantics

are common to comparable real-time simulations, and could support other types of games. However, there are a number of issues:

1. It is difficult to critically compare performance with other studies. Thismainly due to the inconsistent use of *ad-hoc* metrics, without consideration of how informative they are, or how they respond to operating parameters. Cordeiro *et al.* use aggregated thread workload, and frames per second, to examine performance: we will later show that other metrics give a better analysis of performance.

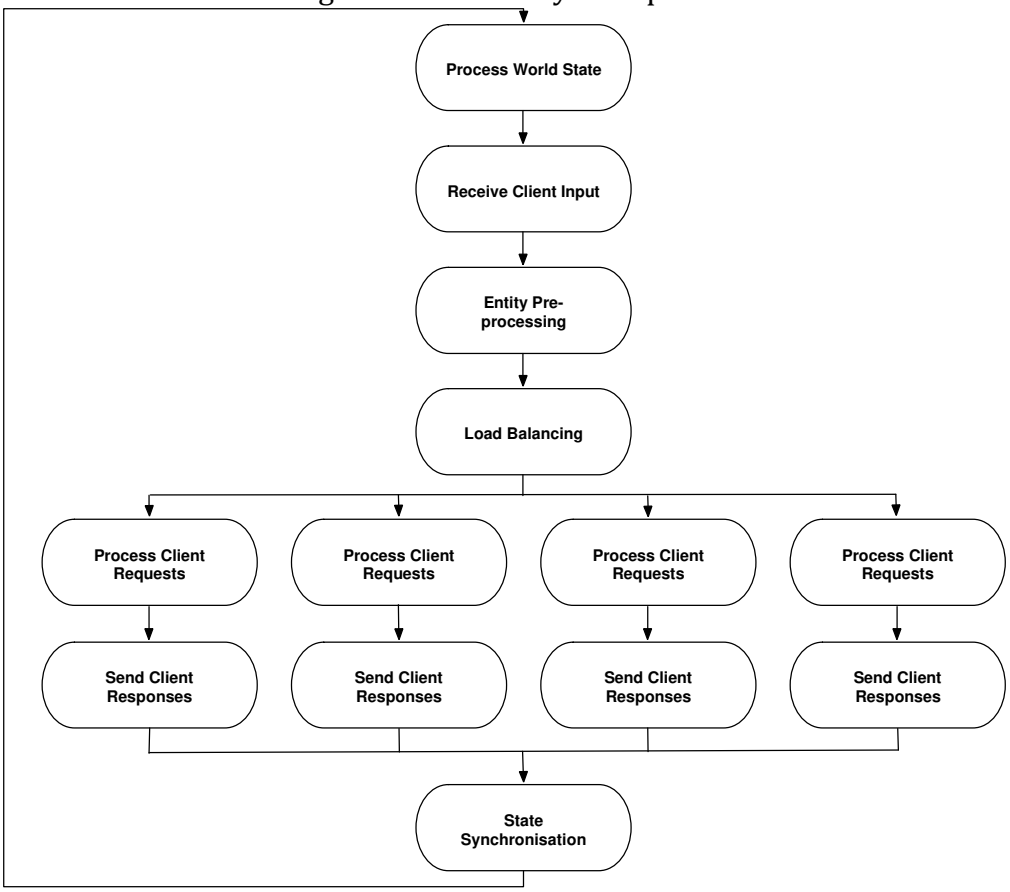

Figure 1: A concurrent QuakeWorld frame, as implemented by Cordeiro *et al.*

2. The authors report a better performance using three parallel threadsrather than four. This appears spurious, and warrants further investigation.
3. The interaction range used to construct workgroups (d_i in Equation 1) is an operational parameter which reflects specific game semantics, but

which also directly affects server performance. The effect of varying this parameter warrants investigation.

3. Experimental Setup

Our study is empirical in nature, and comprises a set of four experimental investigations. The experiments were conducted using eight PCs, connected on a LAN, one of which ran the game server code. The configuration of these machines is shown in Table 1. The last two experiments make use of an additional PC with a six-core CPU, specified in Table 2, to run the game server. The server threads we create in our experiments run on individual CPU cores (up to six).

Processor (CPU)	Intel Core 2 Quad Q8200
Graphics Card (GPU)	N/A
System Memory (RAM)	4GB 800MHz DDR2 SDRAM
Operating System (Client)	Microsoft Windows 7 32-bit
Operating System (Server)	Ubuntu Linux 9.10 32-bit

Table 1: Hardware specification 1

Processor (CPU)	AMD Phenom II X6 1035T (2.60GHz)
Graphics Card (GPU)	ATI Radeon HD 5670
System Memory (RAM)	4GB 1333MHz DDR3 SDRAM
Operating System (Server)	Ubuntu Linux 10.04 64-bit

Table 2: Hardware specification 2

One machine was used to run the server, while the remaining machines were used to run game clients. The number of clients required to test performance made the use of human players impractical: instead, each ran an automated client-side agent, or *bot*. The code to control the agent was originally developed by Cordeiro *et al.*, and we modified it to operate on a Windows 7 (32-bit) platform. The bot control code simply issues a walk command to the server every 10 client frames, and another action command (e.g. jump or shoot) to the server, every 5 frames (client-side). The number of requests per second received by the server is therefore independent of the server's frame-rate, consistent for a fixed number of clients, and scales linearly with the number of connected clients. Cordeiro *et al.* indicated that

this gave a per-client message rate comparable to a human player (see [14]). Using this simple client-side bot we were able to run up to 32 such clients per machine, so that we could easily operate two hundred attached clients using a manageable number of physical machines.

For the purpose of comparative evaluation it was also necessary to scale the performance of the game server to our LAN infrastructure. Initial experiments showed that several hundred connected clients were required to stress the server; however, the LAN (10Mb/s) did not carry sufficient bandwidth to support such large numbers of clients: packet congestion frequently caused client connections to fail. We were able to scale the server performance such that it could be stressed by fewer (approximately 200) clients, by adding an additional artificial processing overhead to the server's player entity processing function. We found experimentally that a fixed processing overhead of 300μs per entity was suitable. Whilst this scaling is artificial, any game engine's entity processing is of arbitrary complexity. Our evaluation is comparative, and intended to represent a generic game server process and hardware setup: scaling the server performance in this way is representative of a game engine with more complex entity processing, and allowed us to perform a full investigative analysis within our hardware parameters.

3.1. Our Experiments

Our work comprises four distinct experiments. Each examines both an aspect of thread workload balancing, and together they represent an investigation of performance metrics for a multicore server:

1. In our first investigation we set a baseline by reproducing Cordeiro *et al.*'s experiments on our platform. We add server throughput (previously used for a serial server [9]) as a metric, and investigate the relationship between throughput and frames per second (used by Cordeiro *et al.*).
2. We investigate performance using different (standard) dynamic loadbalancing strategies, and examine scaling from one to four threads. We investigate the effectiveness of aggregated thread workload, standard deviation of workload, and thread wait time, as comparative metrics.
3. Extending to six threads, we further analyze thread wait time.

4. We conclude by considering the effect of reducing workgroup size byvarying the parameter d_i in Equation 1, and discussing how this impacts on game mechanics.

We present the results for each experiment as a graph (or set of graphs) in which each data point is a measurement of server performance in some configuration: using a particular load balancing algorithm, with specific numbers of threads and attached clients. For example, LPT with 4 threads and 96 connected clients. We vary these parameters to derive performance graphs, which form the basis of our analyses. For each data point, we run the sever for a predefined amount of time (approx. 180 seconds). Moreover, we repeat each run 5 times, giving around 15 minutes of game play from which that data point is measured. We use multiple runs to aggregate the possible effect of any external processes or initial conditions. Our choice of 5 specifically was limited by practical considerations.

Each experiment comprises several 10s of data points gathered from hundreds of individual runs, each of which may involve hundreds of clients. A high level of automation was therefore critical to running these experiments, which we achieved using a bespoke client control program capable of automatically creating, connecting and managing multiple clients across multiple machines.

3.2. Metrics

In our experiments we use a range of performance metrics. Some of these are collated from other studies (Cordeiro *et al.*, Abdelkhalek *et al.* [9]); others are new metrics which we introduce ourselves. The full set used in our experiments are:

1. **Frames Per Second (FPS)**: Commonly used as a measure of game *engine* execution speed. This was calculated by dividing the total execution time for an experimental run by the number of executed frames. Timing was effected using the *QuakeWorld* engine's timer function, which uses the POSIX gettimeofday() microsecond system timer.
2. **Server Throughput**: A measure of the server's response to incoming client request packets. Abdelkhalek *et al.* demonstrated a linear relationship with the number of clients on a linear server, reaching a maximum when all computational resources are fully utilized (saturation). This value is calculated using a response counter which is

incremented directly before each thread sends a processed response packet to a recipient client. As the server completes a frame, the individual thread counters are summed to calculate the total number of responses sent, and this value is divided by the total elapsed execution time.

3. **Accumulated Workload Distribution**: Used by Cordeiro *et al.*, shows the total workgroup weight assigned to each thread, aggregated over some period of time (in our case all runs for each data point, so approx. 15 minutes). Weight in this case is defined as the number of game entities in the workgroup, and is an estimate of average processing load.
4. **Workload Standard Deviation**: We introduce standard deviation of thread workload which indicates the frame-by-frame variation of the distribution. This is useful in characterizing time-variation (note that average workload is not used, as it is no more informative than the accumulated value).
5. **Intra-Frame Wait Time (IFWT)**: Used by Abdelkhalek, this is the average time per-frame that the *main thread* spends waiting for the supplementary threads (noting that workpackages are also processed by the main thread). A high IFWT indicates that the main thread is being under-utilized, though a low IFWT may not necessarily show an optimal balance. This metric is also calculated using the POSIX gettimeofday() system timer.
6. **Total Wait Percentage** (TWP): We introduce IFWT measured as a percentage of total server frame execution time.

3.3. Load Balancing Strategies

Our experiments include performance comparisons using different load balancing strategies. Workgroups, constructed using the algorithm proposed by Cordeiro *et al.*, and expressed in Equation 1, are distributed between the number of active threads: the objective is to evenly distribute processing with the minimum overhead. The *Longest processing Time First* strategy used by Cordeiro *et al.* is well motivated, but this comparison provides a useful context for investigating the use of different metrics, and also for assessing the importance of load-balancing on performance in this context. The load balancing strategies we compare are as follows:

1. **Longest processing Time First (LPT)**: workgroups are sorted into order of descending size. Taking each in order, they are assigned sequentially to the thread with the least current weight. Like Cordeiro *et al.*, in our implementation the first (main) thread starts with an initial weight penalty of one, to ensure that the second thread always receives the first and largest workgroup.
2. **Shortest processing Time First (SPT)**: The opposite of the LPT described above, workgroups are sorted them in order of ascending, rather than descending, size. Our expectation is that performance will be less optimal than LPT if there is a consistent differential in workgroup weight size.
3. **Round-Robin (RR)**: The simplest algorithm; it does not perform any sorting and simply iterates through the list of workgroups and assigns each to alternating threads. This provides a baseline performance, in that it gives the most naive possible distribution, with the lowest overhead, comparable to that used by Abdelkhalek and Bilas.
4. **Sorted Round-Robin (SRR)**: An adaptation of the RR algorithm. It sorts workgroups in order of descending size, as LPT, then iterates through the list and assigns each in turn to an alternating thread.

4. Experiment One: Frames Per Second vs Throughput

The purpose of our first experiment is three-fold: firstly to validate and expand the results obtained by Cordeiro *et al.* [14] using our own platform, and to compare server throughput (used by Abdelkhalek and Bilas [9]) with frames per second (Cordeiro *et al.*) as a performance metric. Using Cordeiro *et al.*'s original code base (LPT), we took measurements for one and four threads. Our measurements included recorded FPS and server throughput rate for between 32 and 192 connected clients. The results are shown in Figures 2 and 3 respectively.

Our measurements for FPS are more extensive than those presented by Cordeiro *et al.*, and show a clear trade-off between the computational cost of managing multiple threads, with the number of clients which can be supported: with lower numbers of clients (less than 112), the overhead of maintaining multiple threads is dominant, and results in a better performance

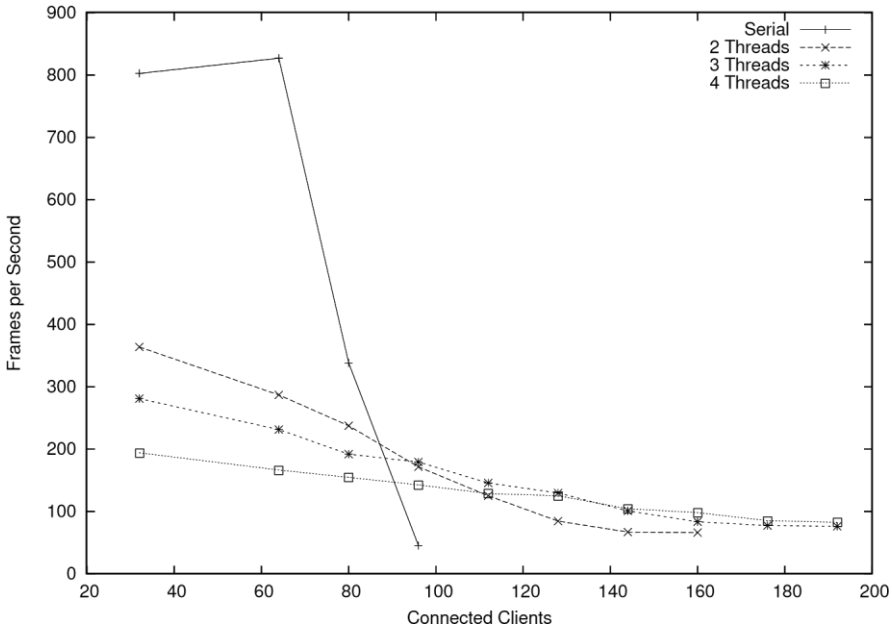

Figure 2: FPS of optimized server (LPT with 1-4 Threads)

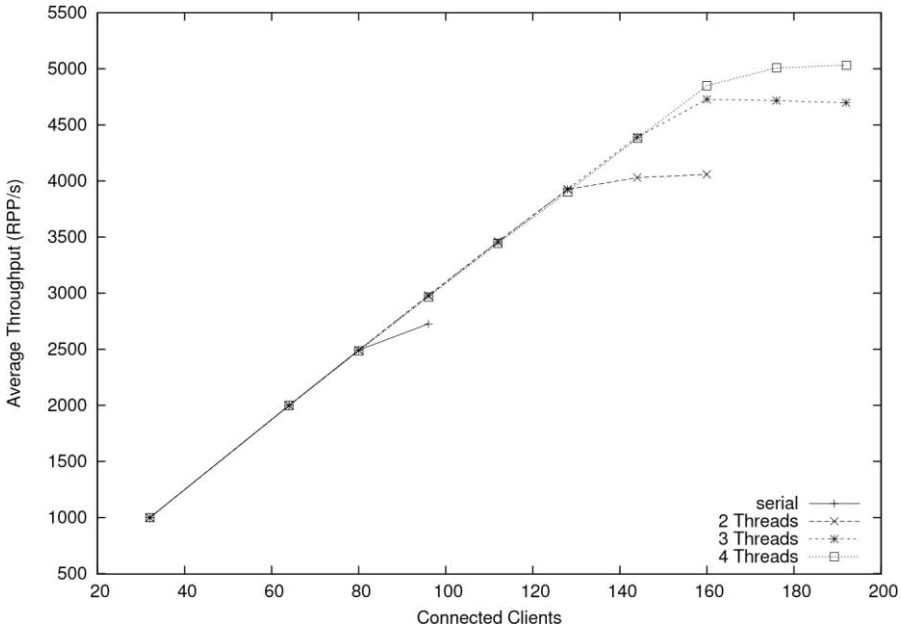

Figure 3: Server throughput of optimized server (LPT with 1-4 Threads)

with just one or two threads. However, with more than 112 threads connected, performance degrades quickly and the three and four thread

servers are superior. Figure 2 does show that FPS as a metric gives poor differentiation with higher numbers of attached clients: the recorded FPS for three and four threads with 192 clients is difficult to separate.

Better differentiation is evident using server throughput (Figure 3), however. Abdelkhalek and Bilas reported, for serial execution, a linear relationship between throughput and number of connected clients, until peak throughout is reached at the point of saturation. In our case, we observe the same relationship for multiple threads: saturation occurs between ≈ 100 clients (1 thread) and ≈ 176 clients (4 threads). The performance difference between 3 and 4 threads is also much more clearly distinguishable than it is using FPS.

We identify a relationship between throughput and FPS. For example, comparing the performances for 2 threads, shown in Figures 2 and 3, the linear increase in throughput from 1000 to 4000 RPPs between 32 and 128 clients is matched with an approximately linear decrease in FPS from around 400 to just under 100. Peak performance appears as a maximum in throughout, and also as a corresponding minimum in FPS. A similar pattern emerges for performance with 3 and 4 threads.

We conclude that there is a coupling between FPS and throughout; however, throughput displays greater differentiation under high computational load, and is therefore more able to identify the point of server saturation across different numbers of threads. It is also clear that there is a demonstrable computational overhead to maintaining multiple parallel threads (with a significant payoff with more than 112 clients in our case). This overhead is expended primarily on partitioning the clients into workgroups using equation 1, copying and resynchronizing entity memory resources, and scheduling the workgroups across the available threads (using LPT). However, there appears to also be a diminishing return: 4 threads only marginally outperforms 3, saturating at around 176 clients compared to 160. We examine the effect of using larger numbers of threads in Experiment Three.

5. Experiment Two: Comparison of Load Balancing Strategies

In this set of experiments we investigate the relative performance of the load balancing algorithms described in section 3.3. We start by considering server throughput as a base metric, and introduce accumulated workload

distribution (previously used by Cordeiro *et al.* [14]), workload deviation, and IFWT to further examine server behaviour.

Figure 4 shows server throughput for each load balancing algorithm. There is a clear divergence between the algorithms starting at around 144 clients: by 176 clients, SPT and RR appear to have reached saturation, whilst LPT and SRR are close to peak, and producing similar performances.

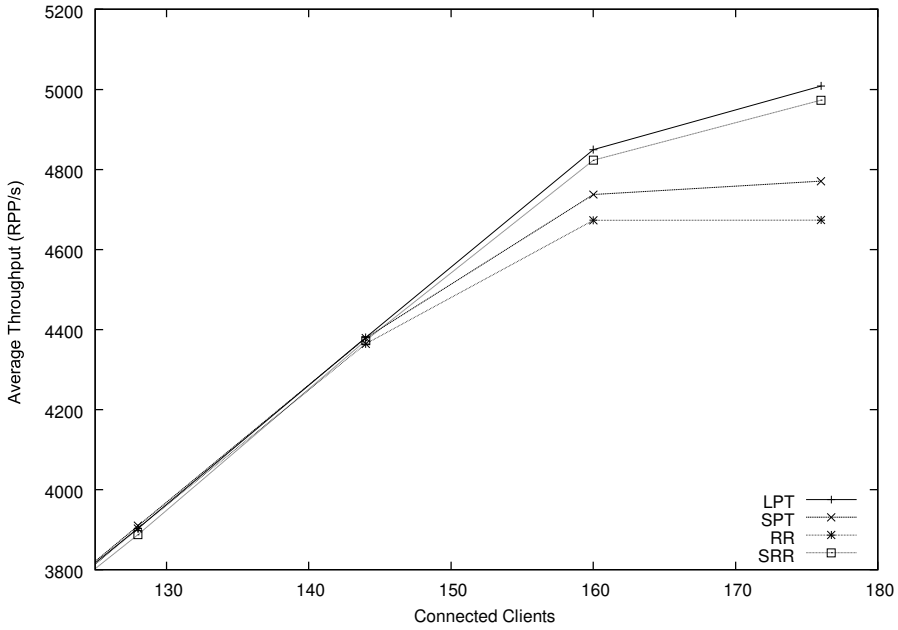

Figure 4: Server Throughput for all algorithms using 4 threads.

RR appears as the least effective distribution method, which may be expected given that it is essentially arbitrary. However SPT offers little improvement, which suggests that the variation in workgroup weight is large enough to limit the performance gain of a this distribution strategy (at least to the extent that it is negated by the overhead of sorting the groups). Whilst LPT and SRR appear more effective, this result raises further questions as to why this is the case, and how they could be further improved. Whilst server throughput appears to be an effective comparative metric, it does not provide much insight into why LPT and SRR are more effective, nor how we may further optimize performance. In need of further analysis, we proceed to consider whether accumulated work load distribution [14] can offer more insight.

5.1. Accumulated Workload Distribution

Figures 5 to 8 show the accumulated workload distribution for the four algorithms.

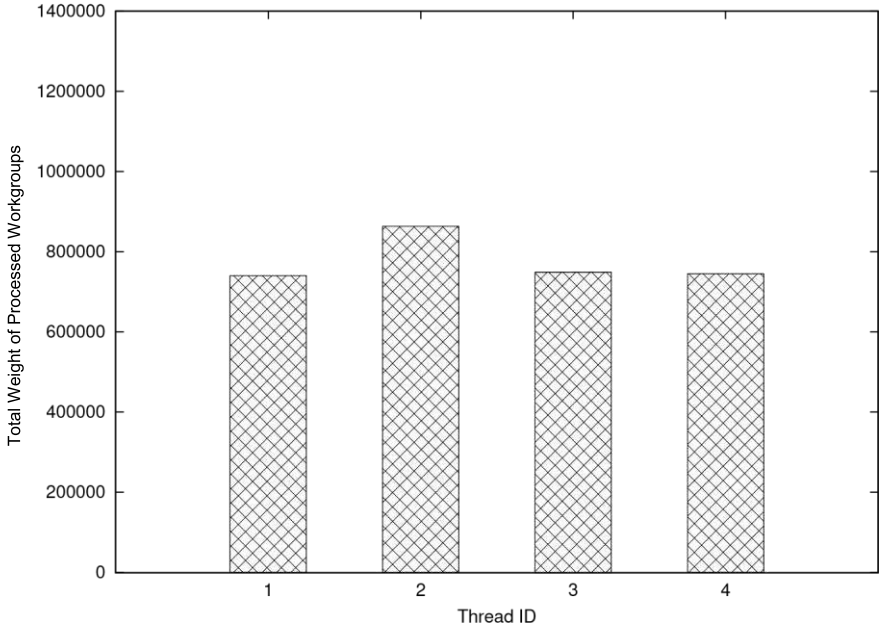

Figure 5: Workload Distribution for LPT using 4 threads.

The accumulated distribution for LPT (Figure 5) shows a similar profile to that recorded by Cordeiro *et al.* The penalty applied to the main thread results in an unevenly high processing load on the second thread. SPT appears to create a more even distribution than LPT, which is somewhat surprising given that its throughput is measured to be lower: we might reasonably anticipate that algorithms exhibiting an even accumulated distribution would produce more optimal performance (which is assumed by by Cordeiro *et al.*). This assumption is further discredited by figure 8 which shows that SRR produces a distinctly uneven accumulated distribution.

Our results suggest that accumulated workload is not a good indication of comparative performance. Further inspection of the recorded data shows that the frame-by-frame distributions are not as stable as appears in the accumulated statistics: this variation impacts on performance. Figure 9 shows the frame-by-frame standard deviation of workload for each thread: SPT

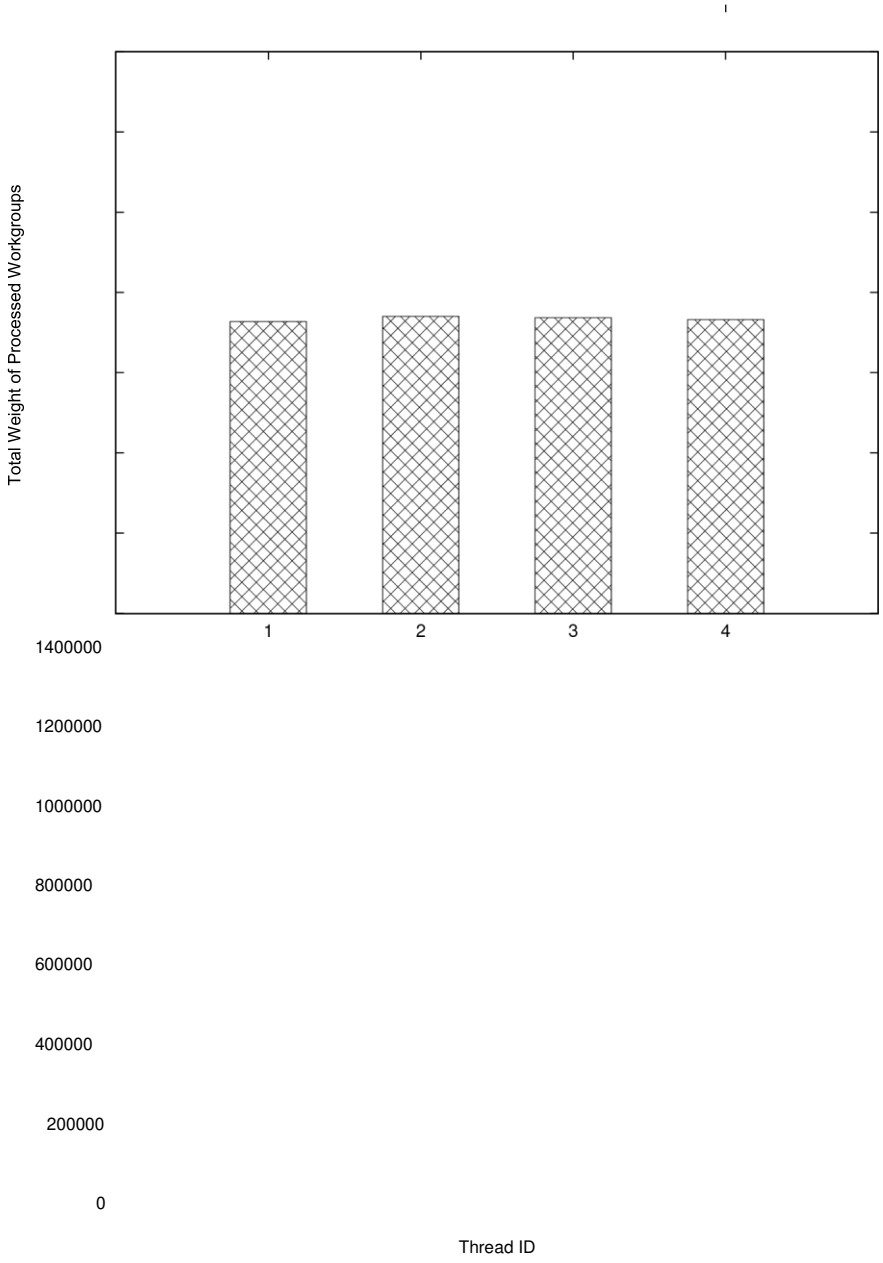

Figure 6: Workload Distribution for SPT using 4 threads.

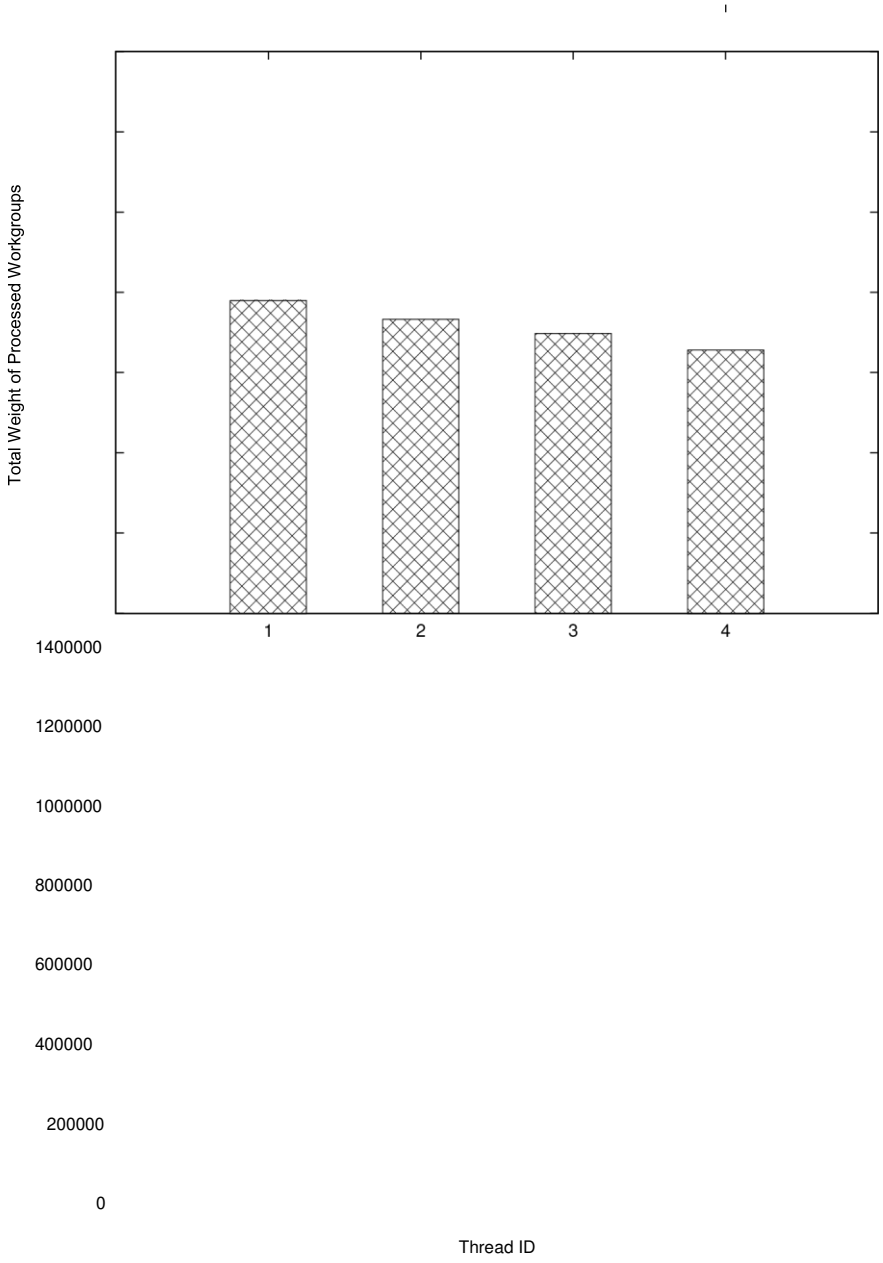

Figure 7: Workload Distribution for RR using 4 threads.

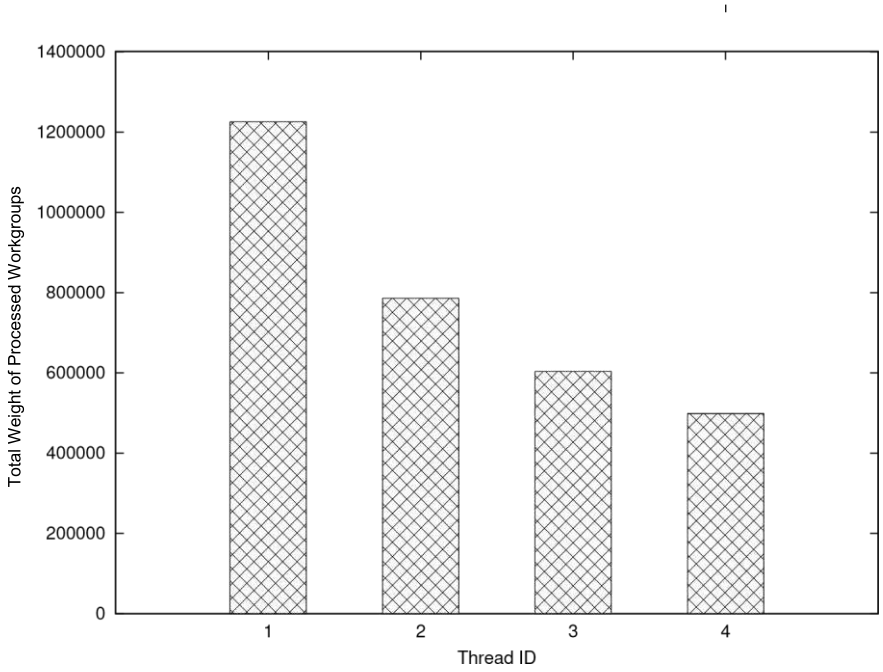

Figure 8: Workload Distribution for SRR using 4 threads.

and RR both demonstrate comparatively high standard deviation, indicating that this is a considerably more informative metric that the accumulated workload used by *Cordeiro et al.* This observation applies equally to any parallelization/load balancing strategy.

5.2. IFWT

We conclude Experiment Two by considering an alternative metric, interframe wait time (IFWT), used by Abdelkhalek and Bilas to analyze threads in their parallel implementation [25]. Whereas workload is an estimated measure of processing weight, IFWT is a direct measure of the *actual* distribution of processing time. In this case, we simplify matters by restricting our consideration of IFWT to the main execution thread, and results are shown in Figure 10.

The two worst performing algorithms (SPT, RR) show a relatively large increase in IFWT with increasing numbers of players. This is consistent with

the high standard deviation in workload distribution reported in Figure 9. LPT shows a much slower increase, suggesting that there is some systematic under-utilization of the main thread (consistent with the penalty used by

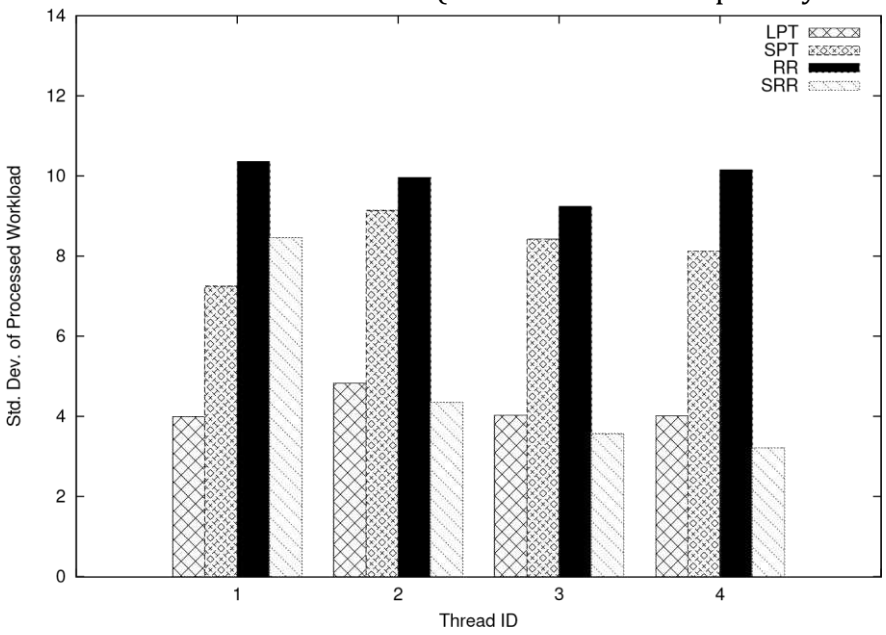

Figure 9: Standard deviation of processed entity workgroups for LPT, SPT, SRR and RR.

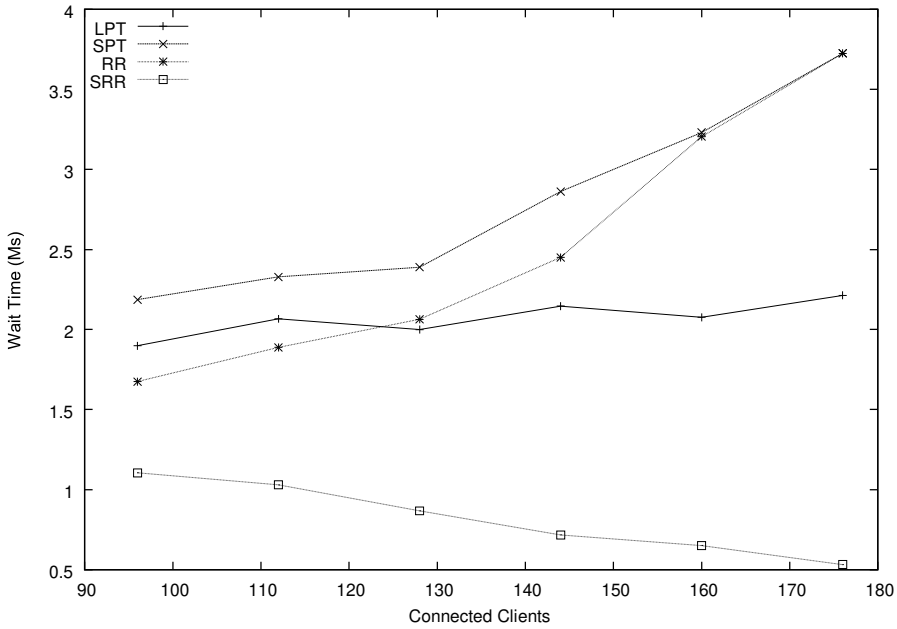

Figure 10: Intra-frame wait time for all algorithms using 4 threads.

Cordeiro *et al.* to locate more weight on the second thread). SRR, however, shows a decreasing IFWT, which indicates an increasing load on the main thread. The IFWT does not reach zero, however: referring back to Figure 9, frame-by-frame variation means that even with a main thread which is overburdened on average, IFWT will still be positive for some frames. High IFWT is, then, a measurable indicator of poor load balancing. However, low IFWT is not necessarily an indicator of good balancing as it may hide an overburdend main thread. Our conclusions for this experiment are as follows:

1. A high frame-by-frame variance (standard deviation) indicates an ineffective balancing strategy which cannot be easily analyzed using accumulated metrics, nor easily optimized.
2. Accumulated workload (used by Codeiro *et al.*) is generally not a reliable metric, unless frame-by-frame variance is low.
3. A consistent IFWT (across number of clients) indicates a low workloadvariance, and near-optimal scaling.
4. Decreasing IFWT indicates an increasing burden on the main thread.Workload deviation will place a minimum bound on IFWT in this case.
5. IFWT and workload deviation together capture the potential for anyfurther optimization of a particular balancing strategy, and so may be used to help optimize server throughput.
6. LPT and SRR have been shown as the most effective balancing strategies on our platform.

6. Experiment Three: Increasing the number of threads

In this section we re-evaluate LPT and SRR on our second hardware configuration (specified in Table 2) to determine whether their relative performance is consistent across different platforms, and how well they scale up to 6 hardware threads. In addition, we re-examine IFWT in more detail. Figure 11 shows the measured throughput for LPT and SRR with 4, 5, and 6 threads respectively.

It is clear that the results differ somewhat from our previous hardware configuration. In particular, SRR does not perform so well, saturating with 4

threads much earlier at 144 players. Conversely, LPT appears to perform better. It is difficult to draw conclusions as to why this should be the case, as the platform configurations are different in several respects; however, it is clear that performance is not consistent across configurations.

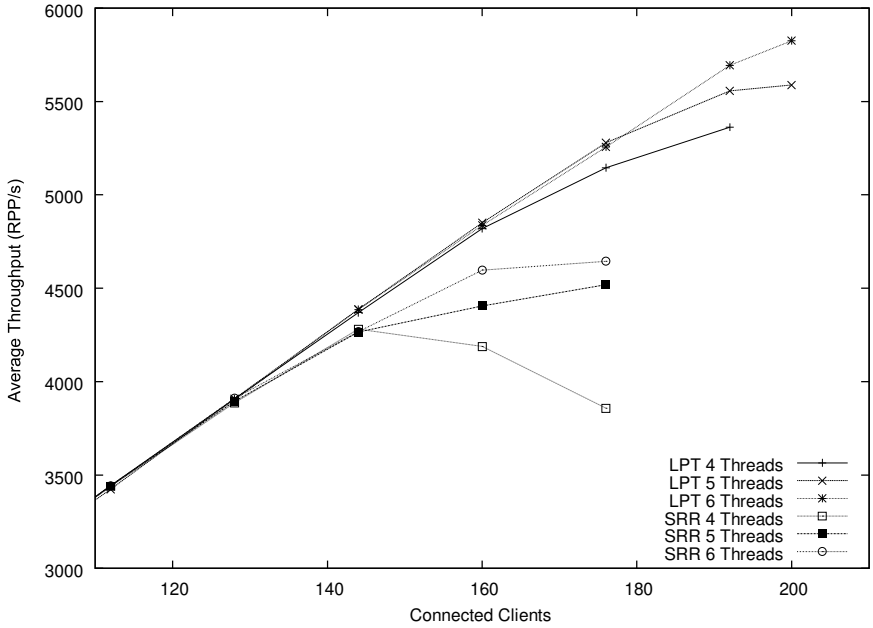

Figure 11: Server throughput for LPT and SRR with four, five and six threads.

Both LPT and SRR show relatively small improvements in performance between 4 and 6 threads. In the case of SRR, saturation occurs later with 6 threads (at 176 rather than 144 clients). However, with LPT the throughput appears to peak around 192 players for 4, 5, and 6 threads, although the replies sent does increase.

The results for IFWT are shown in Figure 12, and the corresponding workload distributions for both LPT and SRR with 6 threads with 176 connect clients is shown in Figure 13. Comparison with the previous section confirms the difference in thread loading: SRR loads the first thread whereas LPT places more weight the second. This is further confirmed by the measurable difference in IFWT.

IFWT for LPT (4,5 and 6 threads) shows a common profile. Initially, between 100-140 connected clients, IFWT falls: at around 140-160 clients it

begins to rise again, prior to reaching server saturation. This suggests that workgroup weight imbalance increases from around 140 clients (so that there is more differential between the first and second threads). This could be a result of, for example, increasing player density. However, the overall frame execution time also increases over this range, and so a corresponding

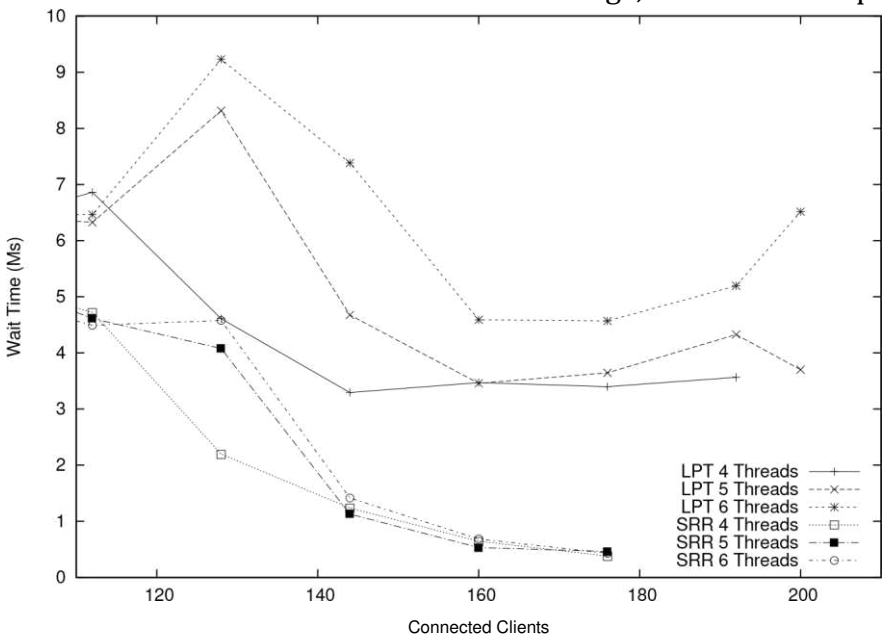

Figure 12: IFWT for LPT and SRR with four, five and six threads.

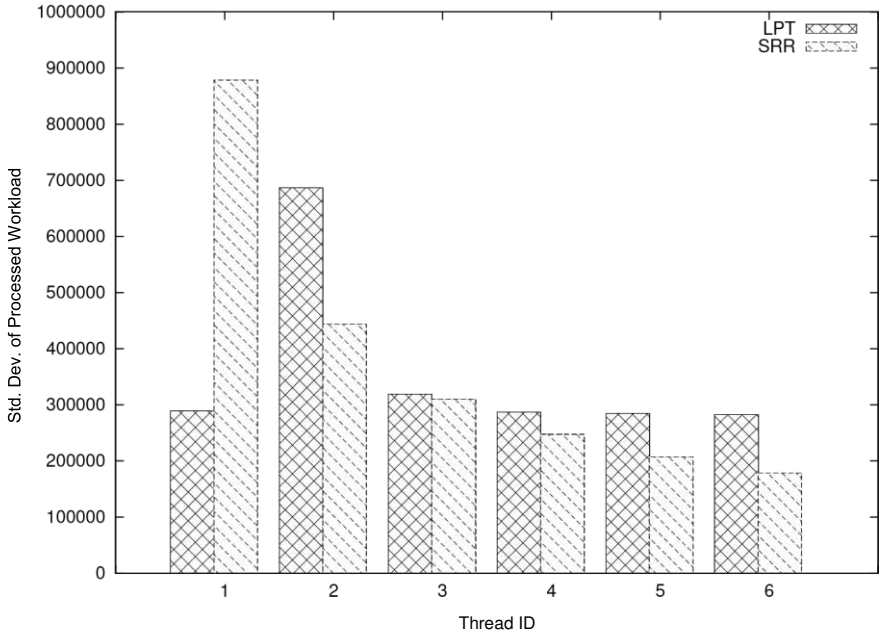

Figure 13: Workload distribution for LPT and SRR with 176 connected clients [6 Threads].

increase in IFWT may, in fact, be expected regardless. This motivates us to introduce the normalized version of IFWT: Total Wait Percentage (TWP), which is calculated as described in section 3.2. Figure 14 shows the measured TWP.

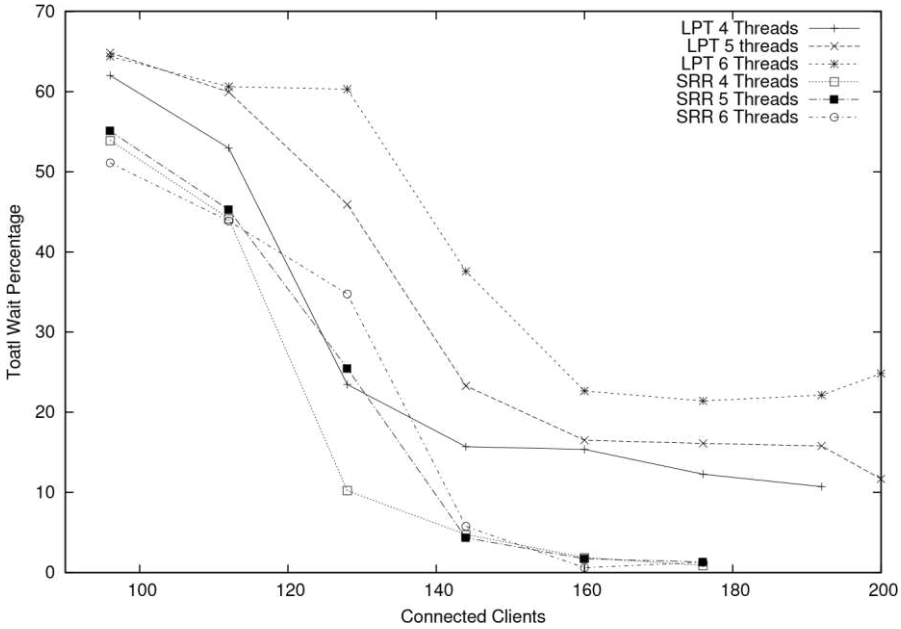

Figure 14: TWP for LPT and SRR with four, five and six threads.

Comparing figures 12 and 14, the value of TWP is significantly more informative. For example, in the case of SRR it is now clear that, as a percentage, the main thread spends close to zero time waiting with more than 160 players (variations become insignificant compared to increasing processing time). On the other hand, LPT shows a significant amount of frame time (between 10% and 25%) for which the main thread is effectively idle. This is consistent with the applied thread penalty, combined with low granularity workgroups, which is also evident in the work load imbalance in Figure 13.

Interestingly, TWP does provide some basis for quantifying the optimization which *could* be achieved by better balancing the thread workloads. For example, with 4 threads and 192 players, the main thread TWP is measured to be around 10%. If that time is caused by additional work load on one thread which could redistributed across all 4 threads, then something like a 7% decrease in frame processing time could potentially be achieved in the best case. This figure increases with the number of threads,

which again implies that workgroup granularity is a limiting factor. In the next section we will examine the effect of reducing this granularity.

7. Experiment Four: Reducing the Workgroup Size

This experiment connects server metrics with game mechanics: we show how metrics can be used to provide designers with a means of predicting how changes to gameplay can impact on performance, scalability, and platform requirements. Whilst accumulated workload distribution alone is not a reliable performance metric, it can highlight systematic imbalances. For example, the higher weight visible on the second thread for LPT is caused not only by the penalty applied to the main thread, but also by a variation in workgroup weights from frame to frame.

The frame-by-frame variation in workgroup weights is related to a number of factors, including the spatial distribution of gameplay. Of key importance is the value of the parameter d_i in Equation 1, which directly affects the size and extent of those workgroups. Cordeiro *et al.* [14] set d_i to a constant 256, which is the *action range* of an entity in Quakeworld. Entities are constrained to interact with other entities within this range, within a single frame (this is a constant value and independent of frame rate). For comparison, the player height is 56 units, a stair step is 18 units, and the maximum player speed is 320 units/second. Using this value guarantees that any objects which can interact in the current frame are processed in the same workgroup; however a lower value would result in smaller workgroups, and so more even load balancing. For example, if the designer decided to reduce the maximum speed of the fastest object, then d_i could also be reduced, reducing workgroup size and potentially improving performance. Indeed, it would be possible to set individual values of d_i on a per-object basis (though this would increase the load-balancing overhead). This type of feedback from implementation testing to design is potentially useful in informing a balance between game play and performance, but is rarely available to designers in practice.

Our final experiment compares server performance at values of 256 and 128 respectively for d_i. We look at both LPT and SRR with 4,5, and 6 threads, and in each case we compare the performance using the two values of d_i, across a range of metrics. Henceforth we denote results obtained for LPT

using smaller workgroups as LPT-S, and, similarly, SRR-S for SRR with small workgroups.

Figure 15 shows the average workgroup sizes with d_i = 256 and 128 respectively, for LPT with 6 threads, as a function of the number of connected clients. Whilst the average size for d_i = 256 increases from 2 to 20 over this range, it remains at around 2 to 3 for d_i = 128. This clearly indicates that reducing d_i has a dramatic effect on group formation, which we might expect to be reflected in other observed server metrics.

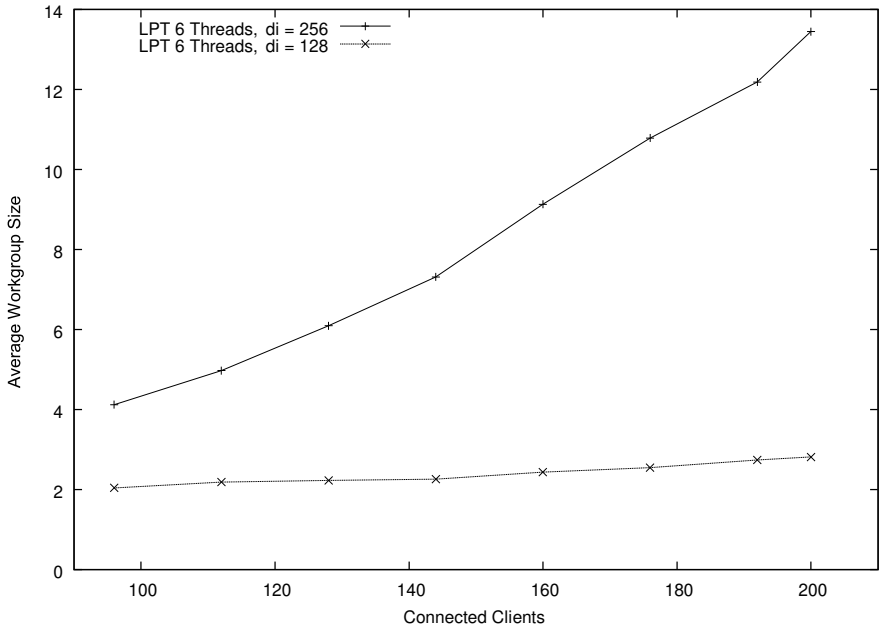

Figure 15: Average workgroup size with modified entity pre-processing step.

7.1. Server Throughput and Workload Distribution

In terms of overall server capacity, the use of smaller workgroup sizes has had a mixed effect. Figure 16 shows that throughput for LPT-S has improved slightly over LPT, although the improvement is minor. However, Figure 17 shows that SRR-S has improved the throughput of the server from 144 to 192 connected clients. The change in throughput for SRR-S is such that it brings the SRR algorithm again to an almost equal performance to LPT (remembering that this was the case with our initial results in Section

4).

The apparent performance parity between LPT-S and SRR-S suggests that the resulting smaller workgroups are contributing to a much more even

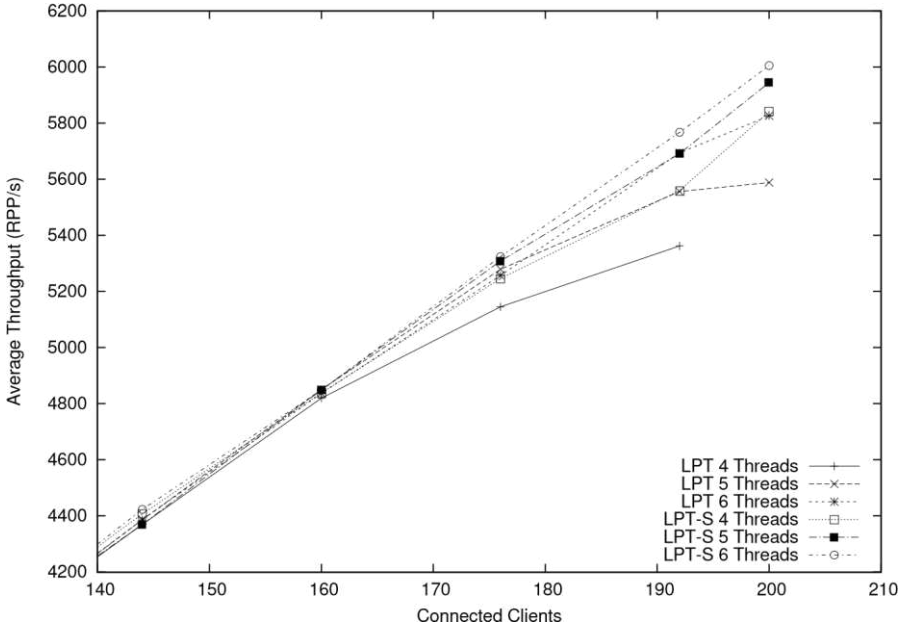

Figure 16: Server throughput for LPT and LPT-S.

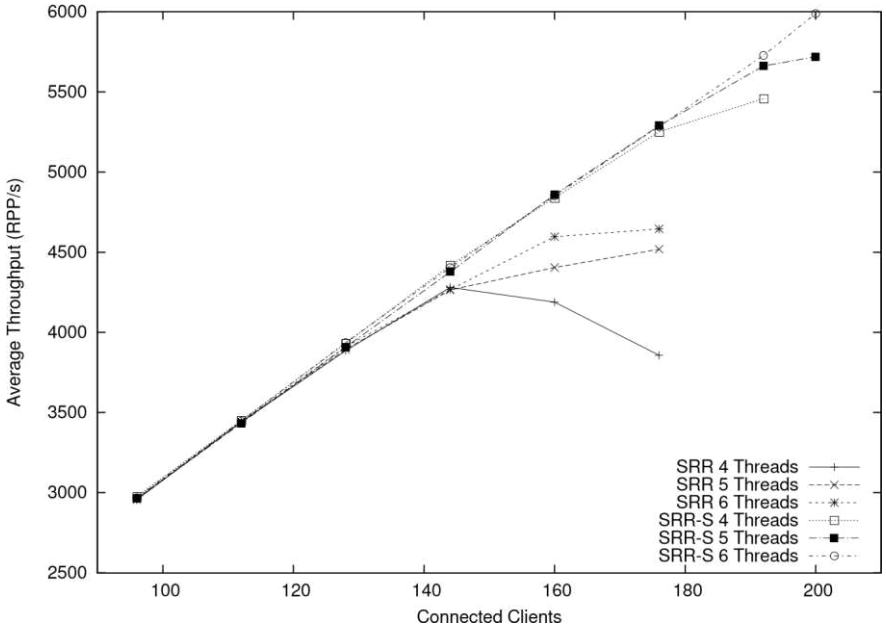

Figure 17: Server throughput for SRR and SRR-S.

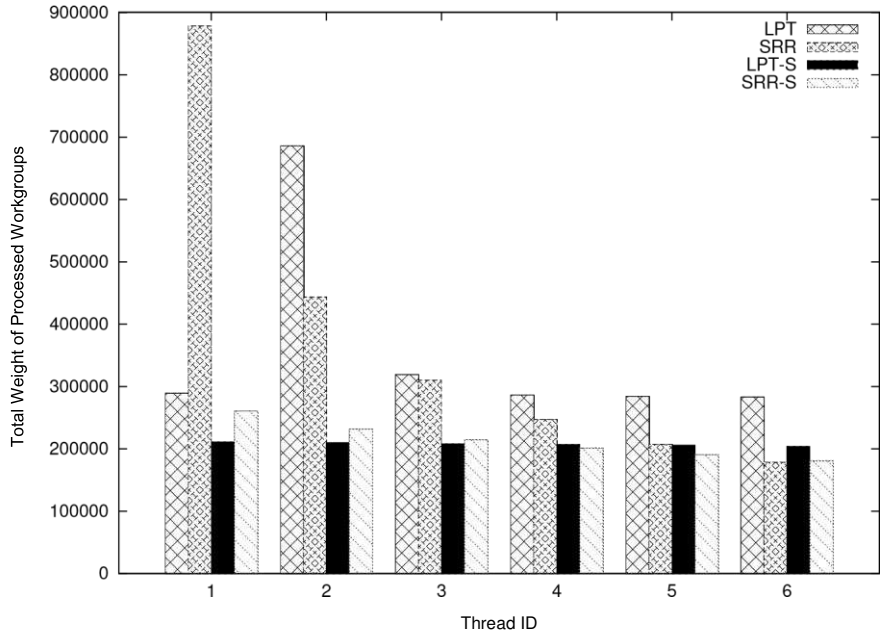

Figure 18: Workload distribution for LPT, SRR, LPT-S and SRR-S with 176 connected clients [6 Threads].

distribution of processing weight across the available threads. Figure 18 shows the workload distribution for all four algorithms, where the distribution for LPT and SRR show the same profiles as seen in Figure 13.

Initial inspection of the workload distributions for LPT-S and SRR-S shows that there appears to be less total weight processed by these algorithms. This disparity is a direct result of the smaller workgroup size. With larger groups used for LPT and SRR, the weight measurement includes entities which may be inactive (for which no input has been received for the current frame), but which are included in workgroups with active entities. With smaller, more fragmented groups, inactive entities are more frequently omitted from processed workgroups, and so not measured as processed weight. Further inspection of Figure 13 shows that the workloads for LPT-S and SRR-S are also much more evenly distributed. Reducing d_i to 128 has the effect of producing a more balanced distribution, producing close to the same performance between LPT-S and SRR-S: the choice of balancing strategy has less impact on performance when balancing a larger number of smaller workgroups. This trend is to some extent predictable; however, what is interesting in this case is that reducing d_i by 50% is enought to mitigate imbalances between different strategies. Being able to quantify this is important in optimizing the game design against server performance.

7.2. TWP for Smaller Workgroups

We conclude by examining the TWP metric for LPT/LPT-S, shown in Figure 19, and SRR/SRR-S shown in Figure 20. Figure 21 shows a comparison of LPT-S and SRR-S. An analysis of these graphs confirms the convergence of performance demonstrated by server throughput and workload processing. LPT-S shows a decrease from the 15–20% range shown by LPT to around 10%, indicating a more even processing balance across the threads. Similarly SRR-S shows an increase from 0 to 10%, as the staircasing distribution seen for SRR is evened out. Based on both of these results, a value of 10% for TWP seems to correspond to optimal performance; we infer that this value reflects the frame-by-frame variation in workload distribution, across the test frames (TWP only measures the wait time of the first thread, and does not record negative values).

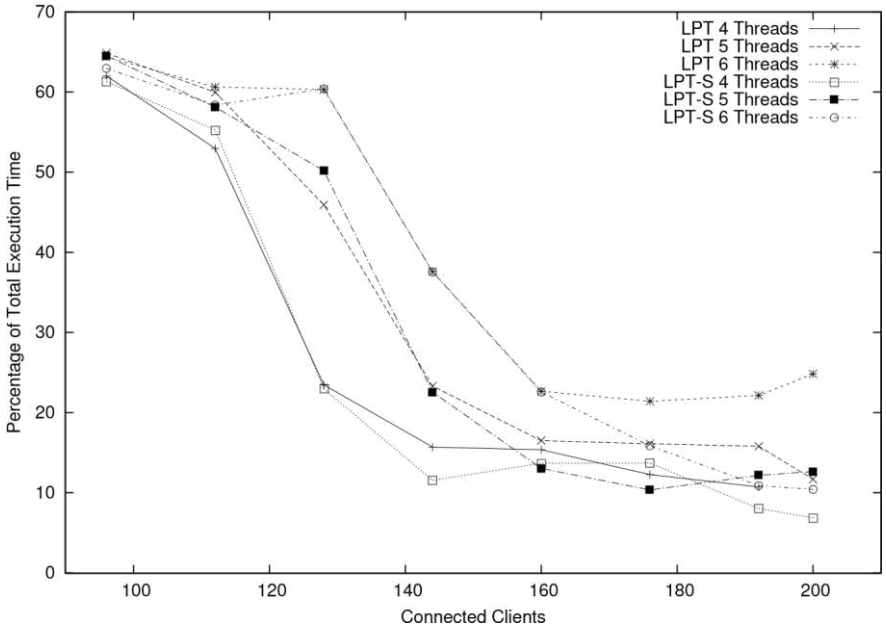

Figure 19: Total wait percentage for LPT and LPT-S

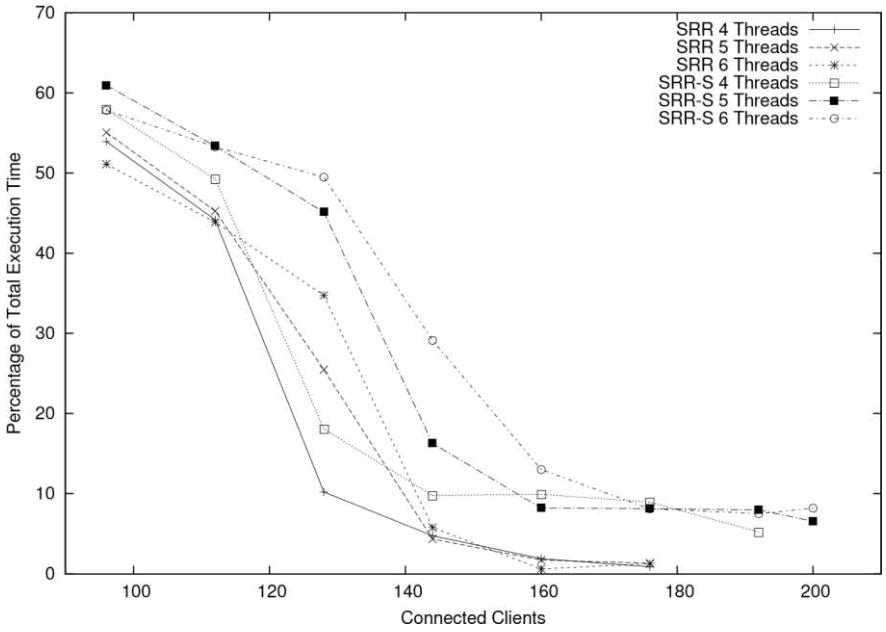

Figure 20: Total wait percentage for SRR and SRR-S

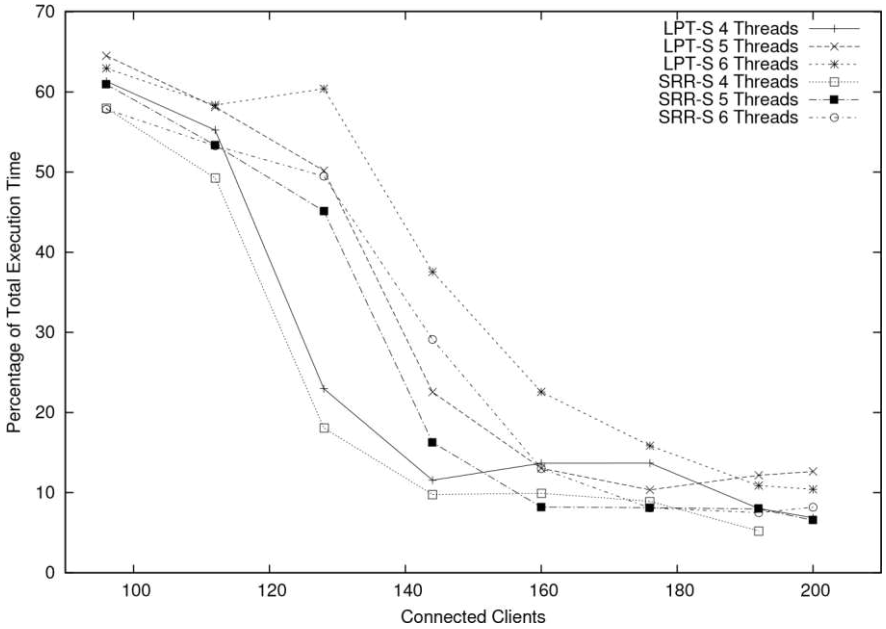

Figure 21: Total wait percentage for LPT-S and SRR-S

8. Conclusion

Our study has focused primarily on identifying informative performance metrics to characterize the behavior of a multicore parallel version of the *QuakeWorld* game server. Existing studies of multicore game servers are sparse: most researchers have used *ad-hoc* metrics with little consideration of how to interpret results, or which are most suitable to evaluate the performance of parallel implementation. We have collated metrics used in these studies, proposed some new ones, and through a series of experiments we have examined their inter-relationships and how they may best be used to investigate the behavior of multicore server processes.

Our work is generalizable: experimental work has focused on thread loadbalancing metrics, and is therefore directly relevant to other real-time game world servers utilizing a stand-alone parallel architecture. Moreover, the lockless threading architecture proposed by Cordeiro *et al.* [14] will transpose to other physically-based game world servers. We concluded our experiments by using our suite of metrics to show how the relationship

between game mechanics and performance can be quantified and used to inform game design.
Our primary conclusions are:

1. We confirm that server throughput, proposed by Abdelkhalek and Bilas [25] also works well in analyzing Cordeiro *et al.*'s architecture [14]. In particular, Saturation, at an identifiable maximum, characterizes the largest number of players which may connect to the server while maintaining optimal performance.
2. Frames per Second, used by Cordeiro *et al.* is bound to throughput, but less effective in identifying peak performance.
3. We have found that accumulated workload, used again by Cordeiro *et al.*, may be used to characterize average weight distribution, but is not reliable as a performance metric due to frame-by-frame variations in distribution.
4. Standard deviation of thread workload is a better indication of performance: high variance indicates an ineffective balancing strategy, and correlates with poor performance in our experiments.
5. IFWT captures aspects of both distribution and variability of workload.We also noted that a variant of IFWT, TWP, can be used to estimate the best possible performance that could be achieved by an optimal balancing strategy.
6. Our experiments indicate that LPT and SRR show the best performance on our implementation. Cordeiro *et al.*'s choice of balancing strategy is shown to be well motivated.
7. We have also showed that varying the size or workgroups had a dramatic effect on performance: in particular, reducing the size interaction radius d_i in Equation 1 helped optimize both LPT and SRR, resulting in convergence. This exemplifies how metrics can be used to predict the effect of changes to game mechanics on server performance.

Game servers are highly sophisticated and complex software applications: understanding their behavior goes hand-in-hand with performance optimization. Leveraging parallel processing architectures still represents a challenge for future work: we see the continued development of

empirical analysis based on server metrics as an area which is not only attracting academic interest, but also has the potential to deliver new and useful tools to industry.

Continuing our own work, we consider that the architecture proposed by Cordeiro *et al.* is immediately applicable to game developers, but has not yet been fully developed. Firstly, the load balancing uses a simple workload weighting model based only on the number of entities in a workgroup. This appears to be an oversimplification, and more accurate weight descriptions may well help to improve balancing (for example, help to eliminate frameby-frame variations in thread processing). Additionally, our experiments in Section 7 suggest that a more accurate maximum interaction range (d_i in Equation 1) would help reduce workgroup size. A more intelligent method of determining d_i could provide significant further optimization.

9. Acknowledgments

The authors would like to thank Daniel Cordeiro of Laboratoire d'Informatique de Grenoble for providing source code to support this study.

References

[1] W.-c. Feng, D. Brandt, D. Saha, A long-term study of a popular mmorpg, in: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games, NetGames '07, ACM, New York, NY, USA, 2007, pp. 19–24. doi:http://doi.acm.org/10.1145/1326257.1326261. URL http://doi.acm.org/10.1145/1326257.1326261

[2] Y. Deng, R. Lau, On delay adjustment for dynamic load balancing in distributed virtual environments, IEEE Transactions on Visualization and Computer Graphics 18 (2012) 529–537.

[3] R. Alexandre, P. Prata, A. Gomes, A grid infrastructure for online games, in: ICIS '09: Proceedings of the 2nd International Conference on Interaction Sciences, ACM, New York, NY, USA, 2009, pp. 670–673. doi:http://doi.acm.org/10.1145/1655925.1656046.

[4] J. Lim, J. Chung, J. Kim, K. Shim, A dynamic load balancing for massive multiplayer online game server, in: Entertainment Computing ICEC

2006, Vol. 4161 of Lecture Notes in Computer Science, 2006, pp. 239–249.

[5] Y. Ahn, A. Cheng, J. Baek, P. Fisher, A multiplayer realtime game protocol architecture for reducing network latency, IEEE Transactions on Consumer Electronics 55 (2009) 1883–1889. doi:10.1109/TCE.2009.5373746.

[6] E. Cronin, B. Filstrup, A. R. Kurc, S. Jamin, An efficient synchronization mechanism for mirrored game architectures, in: NetGames '02: Proceedings of the 1st workshop on Network and system support for games, ACM, New York, NY, USA, 2002, pp. 67–73. doi:http://doi.acm.org/10.1145/566500.566510.

[7] J. Mu¨ller, S. Gorlatch, T. Schro¨ter, S. Fischer, Scaling multiplayer online games using proxy-server replication: a case study of Quake 2, in: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing, ACM, New York, NY, USA, 2007, pp. 219–220. doi:http://doi.acm.org/10.1145/1272366.1272399.

[8] R. Alexandre, P. Prata, A. Gomes, A grid infrastructure for online games, in: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, ICIS '09, ACM, New York, NY, USA, 2009, pp. 670–673. doi:http://doi.acm.org/10.1145/1655925.1656046.
URL http://doi.acm.org/10.1145/1655925.1656046

[9] A. Abdelkhalek, A. Bilas, A. Moshovos, Behavior and performance of interactive multi-player game servers, Cluster Computing 6 (4) (2003) 355–366. doi:http://dx.doi.org/10.1023/A:1025718026938.

[10] G. Deen, M. Hammer, J. Bethencourt, I. Eiron, J. Thomas, J. H. Kaufman, Running Quake II on a grid, IBM Syst. J. 45 (2006) 21–44. doi:http://dx.doi.org/10.1147/sj.451.0021.
URL http://dx.doi.org/10.1147/sj.451.0021

[11] G. J. Armitage, An experimental estimation of latency sensitivity in multiplayer Quake 3, in: 11th IEEE International Conference on Networks (ICON) 2003, 2003, pp. 137–141.

[12] E. Chapresto, K. Mitchell, F. Seron, Capture and analysis of racing gameplay metrics, IEEE Software 28 (2011) 46–52. doi:10.1109/MS.2011.71.

[13] A. Drachen, A. Canossa, Towards gameplay analysis via gameplay metrics, in: Proceedings of the 13th MindTrek, ACM-SIGCHI, 2009.

[14] D. Cordeiro, A. Goldman, D. da Silva, Load balancing on an interactive multiplayer game server, in: A.-M. Kermarrec, L. Boug´e, T. Priol (Eds.), Euro-Par 2007 Parallel Processing, Vol. 4641 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 184–194.

[15] M. Dick, O. Wellnitz, L. Wolf, Analysis of factors affecting players performance and perception in multiplayer games, in: Proceedings of Workshop on Network and Systems Support for Games, 2005.

[16] M. Bredel, M. Fidler, A measurement study regarding quality of service and its impact on multiplayer online games, in: Proceedings of Workshop on Network and Systems Support for Games, 2010.

[17] N. Gildea, Adapting a game engine to take advantage of multi-core processors, Master's thesis, University of Glasgow (2007).

[18] M. Joselli, M. Zamith, E. W. G. Clua, A. Montenegro, R. C. P. LealToledo, L. Valente, B. Feijo´, An architecture with automatic load balancing and distribution for digital games, in: Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment, SBGAMES '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 59–70. doi:http://dx.doi.org/10.1109/SBGAMES.2010.19.
URL http://dx.doi.org/10.1109/SBGAMES.2010.19

[19] M. P. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, R. C. P. Leal-Toledo, P. A. Pagliosa, L. Valente, B. Feij, A game loop architecture for the gpu used as a math coprocessor in real-time applications, Comput. Entertain. 6 (2008) 42:1–42:19. doi:http://doi.acm.org/10.1145/1394021.1394035.

URL http://doi.acm.org/10.1145/1394021.1394035

[20] M. Joselli, M. Zamith, E. Clua, A. Montenegro, R. Leal-Toledo, A. Conci, P. Pagliosa, L. Valente, B. Feijo´, An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu, Comput. Entertain. 7 (2010) 50:1–50:15. doi:http://doi.acm.org/10.1145/1658866.1658869.
URL http://doi.acm.org/10.1145/1658866.1658869

[21] M. Joselli, E. Clua, A. Montenegro, A. Conci, P. Pagliosa, A new physics engine with automatic process distribution between cpu-gpu, in: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, Sandbox '08, ACM, New York, NY, USA, 2008, pp. 149–156. doi:http://doi.acm.org/10.1145/1401843.1401871.
URL http://doi.acm.org/10.1145/1401843.1401871

[22] M. P. d. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, P. A. Pagliosa, L. Valente, Parallel processing between gpu and cpu: Concepts in a game architecture, in: Proceedings of the Computer Graphics, Imaging and Visualisation, CGIV '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 115–120. doi:http://dx.doi.org/10.1109/CGIV.2007.64.
URL http://dx.doi.org/10.1109/CGIV.2007.64

[23] A. Bharambe, J. Pang, S. Seshan, Colyseus: A distributed architecture for online multiplayer games, in: Proceedings of the 3rd ACM/USENIX Symposium on Network Design and Implementation (NSDI06), 2006.

[24] A. Ploss, S. Wichmann, F. Glinka, S. Gorlatch, From a single- to multiserver online game: a Quake 3 case study using RTF, in: ACE '08: Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology, ACM, New York, NY, USA, 2008, pp. 83–90. doi:http://doi.acm.org/10.1145/1501750.1501769.

[25] A. Abdelkhalek, A. Bilas, Parallelization and performance of interactive multiplayer game servers, Parallel and Distributed Processing Symposium, International 1 (2004) 72a.

[26] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, C. Griwodz, C. Griwodz, Lears: A lockless, relaxed-atomicity state model for parallel execution of a game server partition., in: ICPP Workshops, 2012, pp. 382–389.

[27] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguad´e, T. Harris, M. Valero, Atomic Quake: using transactional memory in an interactive multiplayer game server, in: PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, New York, NY, USA, 2009, pp. 25–34. doi:http://doi.acm.org/10.1145/1504176.1504183.

[28] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, Quaketm: parallelizing a complex sequential application using transactional memory, in: ICS '09: Proceedings of the 23rd international conference on Supercomputing, ACM, New York, NY, USA, 2009, pp. 126–135. doi:http://doi.acm.org/10.1145/1542275.1542298.

[29] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, C. Amza, Transactional memory support for scalable and transparent parallelization of multiplayer games, in: EuroSys '10: Proceedings of the 5th European conference on Computer systems, ACM, New York, NY, USA, 2010, pp. 41–54. doi:http://doi.acm.org/10.1145/1755913.1755919.

[30] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, C. Amza, Towards scalable and transparent parallelization of multiplayer games using transactional memory support, in: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 325–326. doi:http://doi.acm.org/10.1145/1693453.1693496. URL http://doi.acm.org/10.1145/1693453.1693496

[31] A. Denault, J. Kienzle, The perils of using simulations to evaluate massively multiplayer online game performance, in: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, ICST (Institute for Computer Sciences, Social-

Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 2010, pp. 4:1–4:8. doi:http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632.
URL http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8632

[32] M. C. Daniel P. Bovet, Understanding the Linux Kernel, third edition Edition, O'Reilly, 2006.