Winter 3-17-2014

Towards Constructing Interactive Virtual Worlds

Francis Chang
Portland State University

10.15760/etd.1649

Towards Constructing Interactive Virtual Worlds

by

Francis Chang

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Wu-chi Feng, Chair
Steve Bleiler
Nirupama Bulusu
Wu-chang Feng
Tim Sheard

Portland State University
2014

ABSTRACT

Networked virtual reality environments including virtual worlds devoted to entertainment, online socializing and remote collaboration have grown in popularity with the rise of commercially available consumer graphics hardware and the growing ubiquity of the Internet. These virtual worlds are typified by a persistent simulated three-dimensional space that communicates over a computer network, where users interact with the environment and each other through digital avatars. Development of these virtual worlds challenges the limits of the networking infrastructure, 3D streaming graphics techniques, and the distributed computing design of the virtual world systems that manages the simulation. In this dissertation, we explore solutions to different aspects of the overall problem of developing a general purpose, networked virtual environment, focusing on the networking and software system issues. Specifically, we show how to improve the networking infrastructure to better support the high packet-rate traffic that is typical of virtual worlds, efficiently stream terrain data for remote rendering, and construct a dynamically adaptive distributed systems framework suitable for virtual world simulations.

ACKNOWLEDGEMENTS

I would like to thank my dissertation supervisor, Professor Wu-chi Feng for his guidance and support at every stage of my progress. The amount of counselling and support I was given was above and beyond what was required to nurture a graduate student. I would also like to thank Professor Wu-chang Feng, who helped me begin the path towards my doctorate.

I am also grateful to the members of my committee, Professor Bleiler, Professor Bulusu, and Professor Sheard, for their counsel and feedback on my dissertation, and their counsel throughout my academic career.

I would like to thank my fellow graduate students for their guidance throughout the years, and especially their encouragement and feedback in the final weeks before my dissertation defense.

I would like to express many thanks to my friends and coworkers at Intel for their advice, collaboration and nurturing support throughout the years.

I am grateful to all my friends I have met in online virtual worlds for their inspiration.

Thank-you to my friends back home who have never let me forget where I am from, and my friends in Portland for their friendship who have made my time in graduate school so enjoyable, especially Chris and Laxmi whose friendship I will always be grateful for.

This dissertation is dedicated to my parents, without whose guidance, support and teaching have made this all possible.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

The evolution and growing ubiquity of the Internet along with the wide distribution of affordable computer graphics hardware has spurred a profusion in the creation of online 3D virtual reality environments. These virtual worlds have initially focused on gaming and entertainment but are evolving to support more complex virtual world applications [strassburger]. The first widely successful virtual worlds were developed for online gaming. Technical development focused on creating an expansive virtual world that allowed a large number of users to simultaneously participate in a real-time gaming experience in a genre of application termed "Massively Multiplayer Online Games (MMOG)" [uo][wow]. In these systems, players connect to remote servers that manage the simulation and interact with the world through virtual avatars. These MMOGs were narrow in focus, allowing only game-specific interaction using pre-downloaded content. This concept has been extended to construct more general purpose virtual environments we refer to as "metaverses", where the interaction between users and their environment is less constrained and more free-form [active][croquet][blue].

These metaverses are characterized by a dynamic, persistent simulation, where the world and the content expressed within it are constantly changing. This introduces a number of problems in presenting a quality experience to the remote user. First, the real-time interactive nature of virtual worlds requires support from the network infrastructure to provide high packet-rate low-latency data delivery to manage a constant stream of world update information [claypool]. Second, because the world environment is expansive, detailed, and dynamic, it cannot be pre-downloaded. World information must be interactively streamed to the remote user [odlyzko]. Thirdly, because the world is free-form, the world simulation workload is dynamic and unpredictable [kinicki]. The virtual world's simulation system must be able to cope with this variation in world activity and adapt to the simulation workload.

1.1 Research Overview

As virtual worlds continue to evolve and gain in popularity, the desire for richer, more expansive, and more detailed virtual worlds continually pressures the computer systems that support them to improve [woodcock]. This thesis focuses on three components to support networked virtual environments:

1) Cache optimization strategies in routers and network appliances through the use of approximate data structures to increase the packet processing capabilities of networking devices
2) Terrain data streaming techniques, focusing on the compressibility of streaming terrain models and developing intelligent streaming and prioritization algorithms for terrain data
3) Distributed system designs to support immersive virtual worlds simulations through dynamic load balancing by using spatial subdivision methods

1.1.1 Network Packet Processing Optimization

The Internet is designed as a best-effort, packet-switched network. Communication between different nodes connected to the Internet is divided into data packets and delivered through a complex network of nodes and routers to reach their destination.

None of the intermediate nodes in the network provide any guarantees to the order, timeliness, or integrity of the packets they process. Even the act of forwarding data packets towards their destination is not guaranteed. This allows the network to be designed from simpler components, which has helped the Internet Protocol to gain world-wide adoption as the de-facto standard for network data exchange. By exploiting this understanding of the network, it is possible to construct more efficient packet processing algorithms without changing the semantics of packet delivery.

The goal of real-time interactivity for networked virtual environments has changed the type of traffic that the network must support. Currently, network appliances are designed to support a smaller number of large data packets that make up the majority of today's Internet traffic, primarily being composed of web, file sharing, and video streaming traffic [cisco]. Remote real-time interactive virtual environments require continuous, low-latency updates, which has a traffic composition that is unlike the majority of Internet traffic. In a packet-switched network such as the Internet, this means that time-sensitive updates from remote virtual worlds bombard the network with a high quantity of small data packets which represent world updates. This increases the packet rate that the network must be able to support [ferreira][feng02].

At the network level, devices such as firewalls, network address translators (NAT), and routers rely on fast packet classification in order to process packets in a timely manner. These services require that packets be classified based on a set of rules before deciding how to process the packet. The result of this classification can be used for something as simple as deciding to admit or reject a packet (in the case of a firewall) or something more complex, such as rewriting the identification markers of a packet (in the case of a NAT). These services require classification algorithms to not only analyze the destination address but also flow identifiers such as source address, layer-4 protocol type, and port numbers. Packet classification is a very complex task and there has been a large amount of work done to try and develop more efficient classification algorithms [gupta]. Still, in the context of high-performance networks, the hardware requirements of performing full classification on each packet at line rates can be overwhelming.

One method of accelerating packet identification is to employ a cache to store the results of previous classification decisions. Since connections on the Internet are discretized into data packets, each connection will generate many packets using identical flow identifiers (unique markers that identify a packet as belonging to a specific connection between two applications communicating over the network). By employing a cache, it is possible to eliminate significant amounts of repeated computation by bypassing the packet classification algorithm, enabling packets to be processed at line rates. It is not unusual for routers on the Internet to be dealing with hundreds of thousands of concurrent flows [trammel], which require larger caches to accommodate the larger volume of distinct flow identifiers. Since larger caches are necessarily slower and monetarily more expensive, the design of a caching algorithm must attempt to use as little memory as possible.

This thesis addresses the problem of building faster, more compact, and more affordable packet classification caches by introducing the idea of creating an approximate cache – a cache that stores inexact representations of data instead of the data itself – that maintains the existing semantics and reliability expectations of the network. This cache can be used in two different ways. In the first way, it can be used to replace traditional

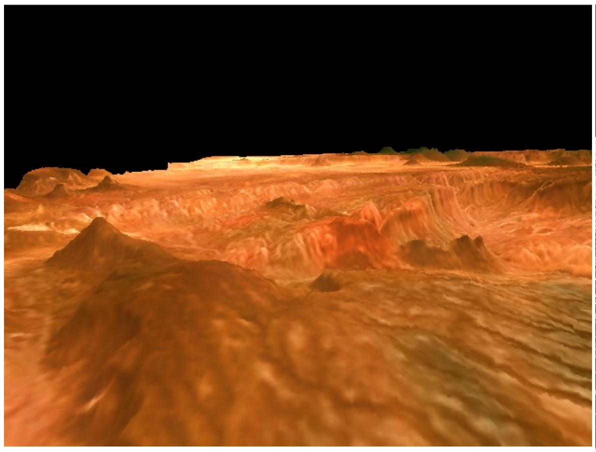


Figure 1: Screenshot of a terrain rendering

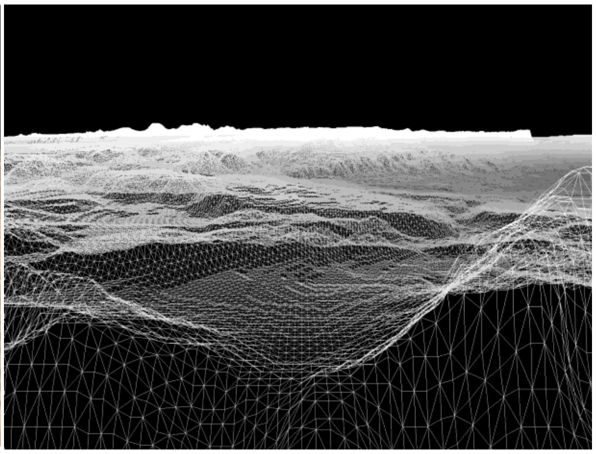


Figure 2: Underlying rendered geometry

exact-precision caches, trading a very small potential misclassification rate in exchange for having smaller, faster caches. We show that the memory footprint required to support the same cache hit rate is reduced by nearly an order of magnitude. In the second, it can be used to augment traditional set-associative caches, allowing them to be accessed more efficiently. By adding a small (much less than 1/10th the size) approximate cache in front of an exact *n*-way set associative cache we can reduce the amount of exact cache memory accessed to service a cache hit by nearly $1/n$. Furthermore, the approximate cache can, with a probability near 100%, determine if a query will miss the exact cache. These designs will allow for the construction of faster, more cost-effective network devices to support the high packet rate traffic that is typical of interactive virtual worlds.

1.1.2 Terrain Data Representation and Streaming

One area of rapid growth in online application usage is areas of virtual mapping, such as Google Earth [gearth] and massively multi-user virtual worlds on both desktop [nielson] and mobile platforms [patro], which also need to render mapping and geographical data (Figure 1, Figure 2). For these types of applications, the data and the users viewing the data are often not co-located, so the world geometry must be delivered through the network for remote rendering and display. The amount of data that describes the world's geometry will exceed the network's ability to deliver it in a timely manner and must be managed in a way that allows the client to receive the data they need to have a visually pleasing experience without overwhelming the network, other applications that share the network, and other components of the virtual world simulation. The remote visualization of the virtual environment requires content and network-aware streaming algorithms to disseminate visualization data to remote viewers in order to provide a time-sensitive high-quality rendering for the users of the virtual worlds.

There is an imperative need for techniques and algorithms that are aware of network constraints and the limits of human perception. Data that is sent through the network that exceeds the ability of the viewer to perceive it is wasted bandwidth. To maximize the user's virtual terrain browsing experience, the order in which remote data is transmitted to the client should be dictated by the viewer's local perception – that is to say, visible features should be prioritized before occluded features, nearby objects favored over distant objects and complex data features sent before uniform data. Models and object geometry should be transmitted in a quality-aware manner that allows information to be transmitted continuously in a compact form that allows clients to view data with progressively increasing quality. This thesis proposes algorithms that use estimates of a terrain's features and visual impact on the viewer to prioritize streaming.

The following are the key observations about the limitations of the network and human perception that guide the design of a well-behaved terrain streaming algorithm:

1. The amount of bandwidth that is available to the terrain streaming algorithm is limited.
2. Large variations in terrain geometry (such as mountains) are more important than more uniform geometry (such as small hills or plains).
3. Terrain near the viewer occupies a larger visual footprint and smaller variations in terrain geometry become magnified as they get closer. This means that terrain geometry near the viewer is more important than distant terrain geometry.

This thesis proposes using a modified progressive JPEG representation to enhance the compressibility of terrain data. By describing the terrain geometry as a height field and dividing it into fixed-area tiles, the data becomes analogous to a grey-scale bitmap image. This representation is amenable to JPEG-style lossy compression that allows the data to be compressed in a way that prioritizes high magnitude frequency data (i.e. cliffs, mountain peaks, and inclines) and de-prioritizes low magnitude frequency data (i.e. plains and flat, featureless areas). By using a progressive JPEG encoding, the terrain geometry is reorganized into tiled refinement layers so that the geometry can be described in varying levels of detail, using a proportionately smaller amount of data. By considering the features of the terrain and the remote user's viewpoint, these tiled refinement layers are streamed to the remote client so that the refinement layer tile that will have the most visual impact on the viewer will be sent first.

1.1.3 Distributed Computing for Virtual Worlds

One fast growing area of computer science is the management of distributed virtual worlds such as Second Life and World of Warcraft [kinicki][rosedale][sl][wow]. An

Figure 3: Screenshot from popular massively-multiplayer online game, World of Warcraft [gamersbin]

example of such a virtual world is shown in Figure 3. Because these virtual worlds can have unbounded size and complexity, it is necessary to develop systems that can distribute the computing load of managing the virtual world over many server computers. Management of an unbounded dataset is a problem because no single computer can process the state of the world in a timely fashion. The only way to manage a large virtual world is to use a distributed system, where the task of managing the virtual world is divided into smaller, more manageable pieces that can each be processed on a single computer.

One way to approach the problem of load balancing a large virtual world is to spatially divide it into regions mapped to multiple computing resources. One simple approach is to divide the world into a regular grid [rosedale] (Figure 4). While this kind of structure does successfully manage to divide the world into smaller pieces, it is not an ideal solution. In a dynamic virtual environment, the computing load is not spread uniformly throughout the world [varvello]. This will lead to some regions being mostly empty, resulting in underutilized computing resources. Some areas will have a high level

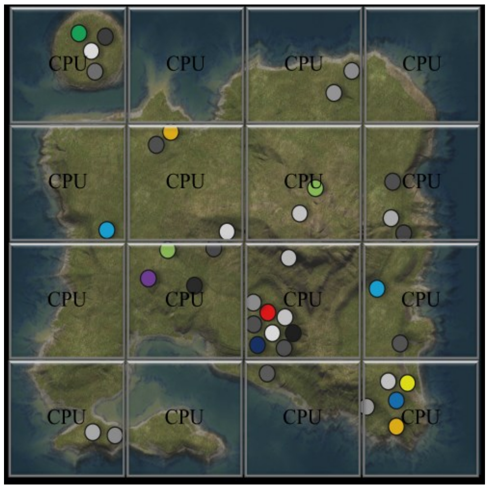


Figure 4: Overhead view of a virtual world using regular grid spatial subdivision. Circles represent entities/players.

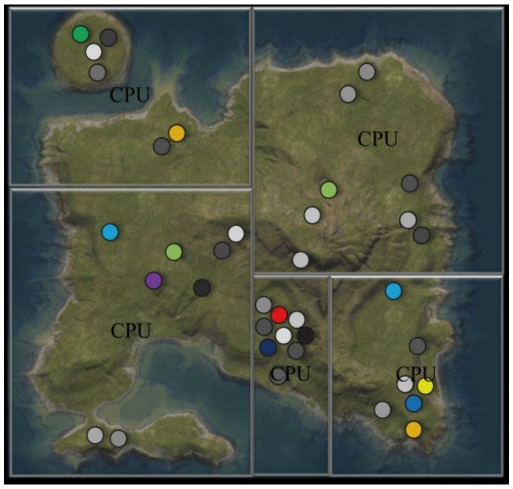


Figure 5: Overhead view of a virtual world using dynamic spatial subdivision. Each rectangular region is managed by a separate server

of activity, resulting in degraded performance because the region's server cannot accommodate the computing load of the world simulation. Spatial subdivision using regular grids also makes an assumption about the virtual scale of the simulation. For example, a region could consist of one square meter, or one square mile. This fixes the expected scale of the simulation. A grid that is scaled appropriately to simulate an ant colony will not be scaled appropriately to simulate larger virtual worlds such as a Disneyland-like theme park.

This thesis addresses this problem of load balancing in virtual world systems by proposing a distributed server infrastructure using a hierarchical dynamic spatial partitioning system. As the distribution of entities within the virtual world move around and cluster together, the system dynamically subdivides the virtual space, assigning more servers to process more densely populated areas (Figure 5). We show that using a simple bintree structure is as effective at efficiently distributing the simulation workload as more sophisticated (but difficult to compute) global knowledge spatial subdivision algorithms.

1.2 Dissertation Overview

The rest of this dissertation is organized as follows.

In Chapter 2, we discuss improving the performance of network devices by introducing a novel approximate caching approach for packet and flow identification.

Chapter 3 presents the design and evaluation of an adaptive virtual terrain streaming protocol that balances the limitations of the network and desire for high quality visualization to deliver terrain geometry in a progressive, quality-aware and adaptive manner.

Chapter 4 explores the construction of the server-side system for virtual world simulation. By using a hierarchical space partitioning algorithm to dynamically assign resources in a distributed computing environment, we design a system that has good performance characteristics using realistic knowledge requirements.

Finally, Chapter 5 summarizes the research contribution of this dissertation, discusses remaining challenges, and outlines future research directions.

Chapter 2 Approximate Packet Classification Caching

As the number of hosts and network traffic continues to grow, the need to efficiently handle packets at line speed becomes increasingly important. Packet classifiers allow in-network devices such as firewalls [qiu], edge routers performing priority marking [stoica], load balancing switches, and network address translators [egevang] to provide differentiated service and access to network and host resources by efficiently determining how packets should be processed. These services require packets to be classified using a set of rules to be applied to packet header information such as the source and destination address, port numbers, and protocol type. The complexity of the packet classification problem and its importance in constructing efficient networks has led to a large volume of work focusing on the development of more efficient classification algorithms [feldmann][gupta], especially concentrating on improving address prefix-matching algorithms [srinivasan][waldvogel]. Bloom Filters have also been used to accelerate exact prefix-matching schemes [dharmapurikar]. However, the requirements of performing a full classification on each packet at current line rates can be overwhelming [partridge]. To keep up with network speeds, some approaches resort to expensive hardware implementations to improve performance [lakshman][xu]. However, there does not appear to be a good algorithmic solution for multiple field classifiers containing more than two fields [baboescu2].

The evolving demands of online gaming and immersive networked virtual environments are also applying pressure to network development. These applications are latency sensitive and require frequent updates as players and entities in those virtual worlds move and interact with the environment and with each other. The update packets in such applications are typically much smaller in size than more traditional bulk-data packets, further increasing the packet processing rate demands on networking hardware, even without increasing overall bandwidth requirements [ferreira][feng02].

It has been well established that memory access delays limit packet classification speeds. While the lookup algorithm itself can be implemented in hardware, the dynamic nature of the classifying rules requires that the classification table be stored in memory whose access latency limits classification speed. Due to the inherent latency of RAM memory access and the need to perform classification lookups at line speed, there is only sufficient amount of time to perform less than half a dozen memory accesses [varghese]. Unfortunately, the best solutions to this problem still require a significantly higher number of memory accesses [gupta].

One effective way to improve classification lookup speed is to avoid performing full classification operations by caching classification decisions and using these previously cached results whenever possible. Whenever a new flow identifier is encountered, a full packet classification decision occurs. The result of this classification decision is cached and the following packets in that flow are classified using the cached values instead of being classified using the slower packet classification engine. Caching improves lookup speeds by taking advantage of the temporal locality inherent in network traffic [claffy].

Unfortunately, packet classification caches must scale up to the total number of flows and it is not unusual for routers on the Internet to concurrently handle hundreds of thousands of flows [trammel]. Because of this, packet classification caches must be reasonably sized in order to maintain high hit rates. The goal of this work is to develop a more scalable packet classification cache, suitable for deployment on the evolving Internet.

Section 2.1 explores related work in the area of packet classification and network caching. Section 2.2 outlines the argument for using an approximate algorithm in the area of packet classification. Section 2.3 introduces the first approximate algorithm, a Bloom filter based approach. Section 2.4 proposes another approximate algorithm, based on set-associative cache framework storing hash digests identifiers. These algorithms are experimentally evaluated in Section 2.5.

2.1 Related Work

A classic approach to managing packetized data streams that exhibit temporal locality is to employ a cache that stores recently referenced items. Packet-switched networks inherently exhibit temporal locality; the arrival of a packet on an Internet link implies a very high probability of the arrival of another packet with the same flow identifier [brownlee][feldmann][mccreary][thompson].

Employing caches to take advantage of this temporal locality has been shown to improve the performance packet classifier significantly in Internet routers [jain][xu]. Network cache design has borrowed concepts from computer architecture (Least-Recently Used (LRU) stacks, set-associative multi-level caches) [jain]. Some caching strategies rely on CPU L1 and L2 cache [partridge] while others attempt to map the IP address space to memory address space to use the hardware TLB [chiueh]. Another approach is to add an explicit timeout to an LRU set-associative cache to improve performance by reducing thrashing [xu]. In addition to leveraging the temporal locality observed on networks, approaches to improving cache performance have applied techniques to compress and cache IP ranges to take advantage of the spatial locality in the address space of flow identifiers [chiueh2][gopalan]. This effectively allows multiple flows to be cached in a single cache entry, so that the entire cache may be placed into small high-speed memory such as a processor's L1/L2 cache.

How well a cache design performs is typically measured by its hit rate for a given cache size. Generally, as additional capacity is added to the cache, the hit rates and performance of the packet classification engine should increase. In a set-associative cache architecture, increasing the level of associativity will improve cache performance, but yields diminishing returns for associativity levels greater than four [li].

Unlike route caches that only need to store destination address information, packet classification caches require the storage of full packet headers. Unfortunately, due to the increasing size of packet headers (the eventual deployment of IPv6 [huitima]), storing full header information can be prohibitively expensive given the high-speed memory that would be required to implement such a cache. It is beneficial to develop a cache architecture that can store more information, without increasing the amount of memory required to support the cache.

2.2 An Approximate Algorithm Approach

Traditionally, cache designs trade off time and space with the goal of balancing the overall cost and performance of the device. This thesis proposes another axis of the design space that has not been previously considered: accuracy. In particular, we quantify the benefits of relaxing the accuracy of the cache on the cost and performance of packet classification caches.

To understand the implications of developing an approximate packet classification cache, we must first consider the design semantics of the Internet. The network is structured as a packet-switched best-effort service, meaning that communication between hosts is divided into packets before being transmitted through the network, with intermediate nodes in the network providing no guarantees about bandwidth availability or the reliability, integrity, timeliness and order of data delivery. The responsibility of ensuring that a data packet is delivered is delegated to the end points, rather than the network infrastructure itself. This simplicity of this design was motivated by a desire to connect many different networks together, communicating in a single common Internet Protocol (IP), without requiring any internal changes to any of the distinct networks connecting to the Internet. By designing a protocol that required few guarantees from the underlying network, it would be possible to connect any type of network (such as ARPANET, Packet Radio and Packet Satellite) to a common Internet [leiner]. While this philosophy has led to the world-wide adoption of IP, this lack of reliability forces the burden of detecting and correcting for network faults to end hosts; the core infrastructure of the Internet is simple and the edges and end points of the network are intelligent. This design philosophy is known as the end-to-end principle [saltzer]. In this scheme, since the end hosts are designed to detect and correct faults, this introduces an opportunity to employ approximate algorithms within the network infrastructure because any errors generated by the network will automatically be rectified at the endpoints of the data communication.

This chapter explores the design of two styles of approximate caches. The first is explored in Section 2.3, and is based on a Bloom filter data structure [bloom]. A model for optimizing Bloom filters for this purpose is explored, as well as extensions to the data structure to support graceful aging, bounded misclassification rates, and multiple binary predicates. This design will yield potential false-positive matches and can store only a limited amount of information on each flow identifier. This design is appropriate for use in firewalls or routers. These types of approximate caches can provide nearly an order of magnitude cost savings at the expense of misclassifying one billionth of packets for IPv6-based networks.

The second design is explored in Section 2.4 and is based on storing hash digests of flow identifiers. It is suitable for situations requiring an arbitrary amount of information to be stored for each flow identifier. This design can also be adapted for use in a multi-level cache, using the digest cache to augment a more traditional set-associative cache to provide improved cache performance without incurring the cost of a probabilistic misclassification. In this scenario, the digest cache is used as a Las Vegas style of approximate algorithm, where the digest cache will always correctly identify locations in the cache where a given flow identifier is not stored and yield high-probability matches.

2.2.1 Dealing with Misclassification

Measurement studies have discovered that between 1 in 1100 to 1 in 32000 TCP packets on the Internet will fail their CRC check, showing that packet corruption has occurred, even though link-layer checksums should only admit error rates of 1 in 4 billion [stone]. Extrapolating, this means that on average, 1 in 16 million to 1 in 10 billion TCP packets will contain an undetectable error. With this in mind, we contend that introducing a packet misclassification probability in the order of 1 in a billion packets will not meaningfully degrade the utility of the network. It is the responsibility of the end system to detect and compensate for errors that may occur in the network [saltzer]. The immediate question that arises when we introduce the possibility of a misclassification is to account for the result of the misclassifications.

Consider the case of a firewall. If F_1, F_2 ... F_q unique flows were to set signatures in an approximate cache that matched the signature to a new flow *F'*, we will accept *F'* as a previously validated flow. In the case that *F'* is a valid flow, no harm is done, even though *F'* would never have been analyzed by the packet classifier. If *F'* is a flow that would have been rejected by the classification engine then there may be more serious repercussions - the cache would have instructed the firewall to admit a bad flow into the network. This case can be rectified for TCP-based flows by forcing all TCP SYN packets through the classification engine. Another solution would be to periodically reclassify packets that have previously been marked as cached. If a misclassification is detected, all bits corresponding to the signature of the flow id could be zeroed. This approach has the drawback of initially admitting bad packets into the network, as well as causing flows which share similar flow signatures to be reclassified.

If an attacker wanted to craft an attack on the firewall to allow a malicious flow, *F'*, into the network, they could theoretically construct flows, F'_1, F'_2 ... F'_q, that would match the flow signature of *F'*. If the firewall's internal hash functions were well known, this could effectively open a hole in the firewall. To prevent this possibility, internal hash functions should not be openly advertised. An additional measure would be to randomly choose the hash functions that the firewall uses. Hash functions can easily be changed periodically as the cache ages, as there is no need to synchronize the hash function with any external host.

In the case of a router, a misclassified flow could mean that a flow is potentially misrouted, resulting in an artificially terminated connection. In a practical sense, the problem can be corrected by an application or user controlled retry. In the case of UDP and TCP, a new ephemeral port would be chosen, constructing a new flow identifier, and network connectivity can continue. If an approximate cache has misclassified a previous flow, it will have no impact on the classification of the new flow. The network is also designed to atomically guard itself from errors. For example, if the misclassification results in a routing loop, the network already protects itself from this error by using the IP time-to-live counter (TTL). If we randomly force cached flows to be re-classified, we can reduce this “fatal” error to a transient one. TCP retransmits and application-level UDP error handlers will make this failure transparent to the user.

Real-time update packets that are characteristic of online gaming and networked virtual worlds are performed with UDP packets for low-latency signalling [ferreira] [feng02]. These protocols are already designed to be resilient to packet loss.

2.3 Approximate Algorithm 1: Bloom Filters

A Bloom filter is a data structure that allows a quick, but approximate test, to see if an identity, x, is a member of a set, S [bloom]. This approach may generate false positives – a Bloom filter may incorrectly report that an identity, x, is a member of the set – but a Bloom filter will never generate false negatives. The Bloom filter is a very space-efficient data structure, which makes it an attractive data-structure from which to construct a cache. Bloom filters were originally invented to store large amounts of static data (hyphenation rules on English words), but have found applications in computer networking [baboescu][mitzenmacher]. Applications range from web cache sharing [fan] to active queue management [feng] to IP traceback [sanchez][snoeren] to resource routing [byers][czerwinski].

The Bloom filter data structure used in this chapter consists of $M = N \times L$ bins. (Each bin consists of one bit.) These bins are organized into L levels with N bins in each level, to create N^L virtual bins (possible permutations). To interact with the Bloom filter, there are L independent hash functions, each associated with one bin level. Each hash function maps an element into one of the N bins in that level. For each element we enter into the Bloom filter, we compute the L hash functions and set all of the corresponding bins to 1. To test membership of any element in our Bloom filter, we compute the L hash functions and test if all of the corresponding buckets are set to 1. See Figure 6 for an example. This approach may generate false positives – a Bloom filter may incorrectly report that an element is a member of the set S.

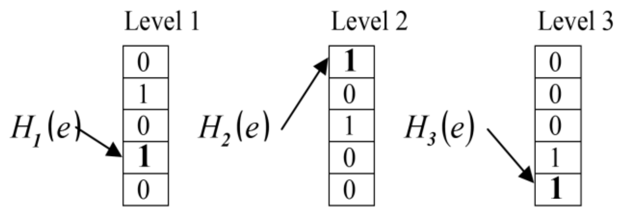


Figure 6: An example: A Bloom filter with *N*=5 bins and *L*=3 hash levels. Suppose we wish to insert an element, *e*.

For optimal performance, each of the *L* hash functions, H_1, H_2… H_L should be a member of the class of universal hash functions [carter]. That is, each hash function should distribute elements evenly over the hash's address space, and for each hash function $H_\varepsilon : e \rightarrow [1 \dots N]$, the probability of collision $H_\varepsilon(a) = H_\varepsilon(b), a \neq b$, is 1/*N*. In practice, we only compute one hash function, $H_\varepsilon : e \rightarrow [1 \dots N^L]$, for each insertion/query operation and simply use different portions of the resulting hash to implement the *L* hash functions.

This definition of a Bloom filter differs slightly from the classical definition [bloom], where each of the *L* hash functions can address all of the *M* bit buckets. This definition of the Bloom filter is often used in current designs due to potential parallelization gains to be had by artificially partitioning memory [feng]. It should be noted that this approach yields a negligibly worse probability of false positives under the same conditions but an equal asymptotic false-positive rate [broder].

2.3.1 Properties of the Bloom Filter

In order to better design and understand the limitations of our architecture, it is important to understand the behavioral properties of a Bloom filter. In particular, we are interested in how the misclassification probability and the size of the Bloom filter will affect the number of elements it can store.

A Bloom filter storing *k* elements has a probability of yielding a false positive of

$$p = \left(1 - \left(1 - \tfrac{1}{N}\right)^{k}\right)^{L} \tag{1}$$

For our purposes, we need to know how many elements, *k*, we can store in our bloom filter, without exceeding some misclassification probability, *p*. Solving for *k* yields

$$k = \frac{\ln(1 - p^{1/L})}{\ln(1 - 1/N)} \tag{2}$$

To simplify this equation, we apply the approximation $1 - 1/N \approx e^{-1/N}$. So constructing $\kappa \approx k$,

$$\begin{aligned} \kappa &= \frac{\ln(1 - p^{1/L})}{\ln(e^{-1/N})} \\ &= -N \ln(1 - p^{1/L}) \\ &= -\frac{M}{L} \ln(1 - p^{1/L}) \end{aligned} \tag{3}$$

From Equation 3 it is clear that the number of elements, *κ*, that a Bloom filter can support scales linearly with the amount of memory *M*. The relative error of this approximation, *κ/k*, grows linearly with the number of hash functions *L*, and decreases with increasing *M*. For the purposes of our application of this approximation, the relative error is negligible. (For $M \geq 1024$ bytes and $L \leq 50$, the relative error is less than 0.35 %.)

Note that solving for *p* in this equation yields the more popular expression [broder, fan, snoeren],

$$p = \left(1 - e^{-\kappa L/M}\right)^{L} \tag{4}$$

2.3.2 Dimensioning the Bloom Filter

Bloom filter design was originally motivated by the need to store spell-checking dictionaries in memory. The underlying design assumption is that the intent is to store a large amount of static data. However, this assumption is not applicable when dealing with dynamic data, such as network traffic. Previous work has attempted to dimension a Bloom filter such that the misclassification rate is minimized for a fixed number of elements [broder].

To apply Bloom filters to the context of driving a cache, we prefer to maximize the number of elements *k* that a Bloom filter can store, without exceeding a fixed maximum tolerable misclassification rate, *p*. To maximize *κ* as a function of *L*, we first take the derivative *dκ/dL*, set it to 0, and solve for *L* to find the local maximum.

$$\begin{aligned}
\frac{d\kappa}{dL} &= \frac{d}{dL}\left(\frac{M}{L}\ln(1-p^{1/L})\right)\\
\frac{d\kappa}{dL} &= \frac{M}{L^2}\left(\ln(1-p^{1/L}) - \frac{p^{1/L}\ln p}{L(1-p^{1/L})}\right)\\
0 &= \frac{M}{L^2}\left(\ln(1-p^{1/L}) - \frac{p^{1/L}\ln p}{L(1-p^{1/L})}\right)\\
0 &= \ln(1-p^{1/L}) - \frac{p^{1/L}\ln p}{L(1-p^{1/L})}
\end{aligned} \tag{5}$$

Now suppose a $u = p^{1/L}$, so $L = \ln p / \ln u$.

$$\begin{aligned}
0 &= \ln(1-u) - \frac{u\ln p}{(\ln p/\ln u)(1-u)}\\
0 &= \ln(1-u) - \frac{u\ln u}{(1-u)}\\
u\ln u &= (1-u)\ln(1-u)\\
u^u &= (1-u)^{1-u}
\end{aligned} \tag{6}$$

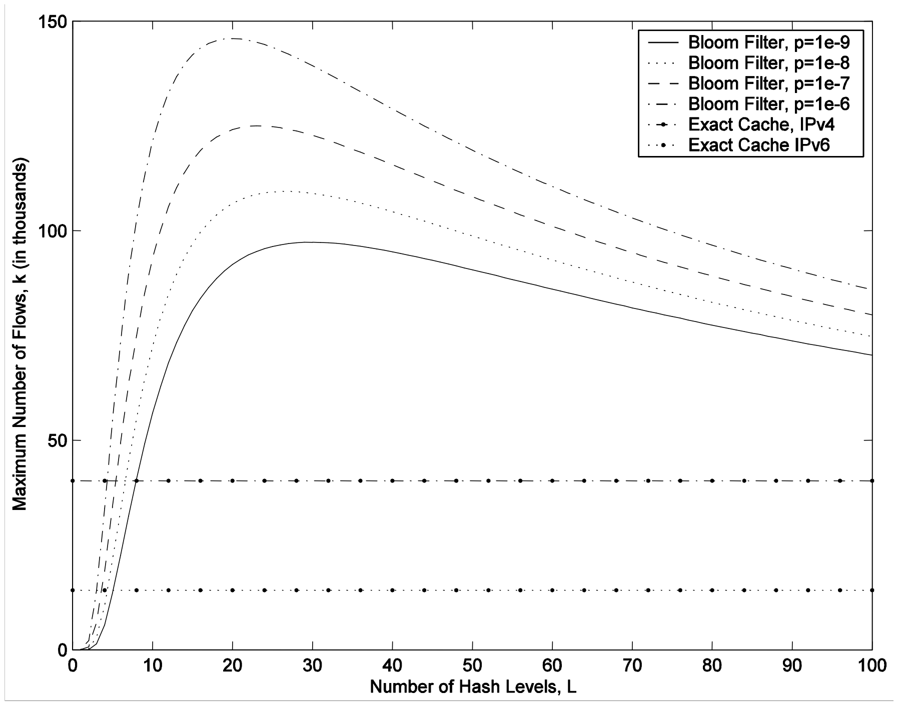


Figure 7: The maximum number of elements that can be stored by a 512KB cache

Since $p \in [0, 1]$ then $u \in [0, 1]$, u only has one solution, $u = ½$, which means κ is maximized for

$$L = -\ln p / \ln 2 = -\log_2 p \qquad (7)$$

This is an interesting result, because it implies that L is invariant with respect to the size of the Bloom filter, M.

The accuracy of this approximation increases as M increases. In our testing, for cache sizes greater than 1KB, this approximation yields no error. In all the simulations presented in this chapter, this approximation and the optimal value of L are equal. Even if we choose a slightly sub-optimal value of L, the difference in the maximum number of flows the Bloom filter can store is negligible. Figure 7 graphs this relationship. For values of L that are near optimal, the number of flows, k, that the Bloom filter can store are nearly identical.

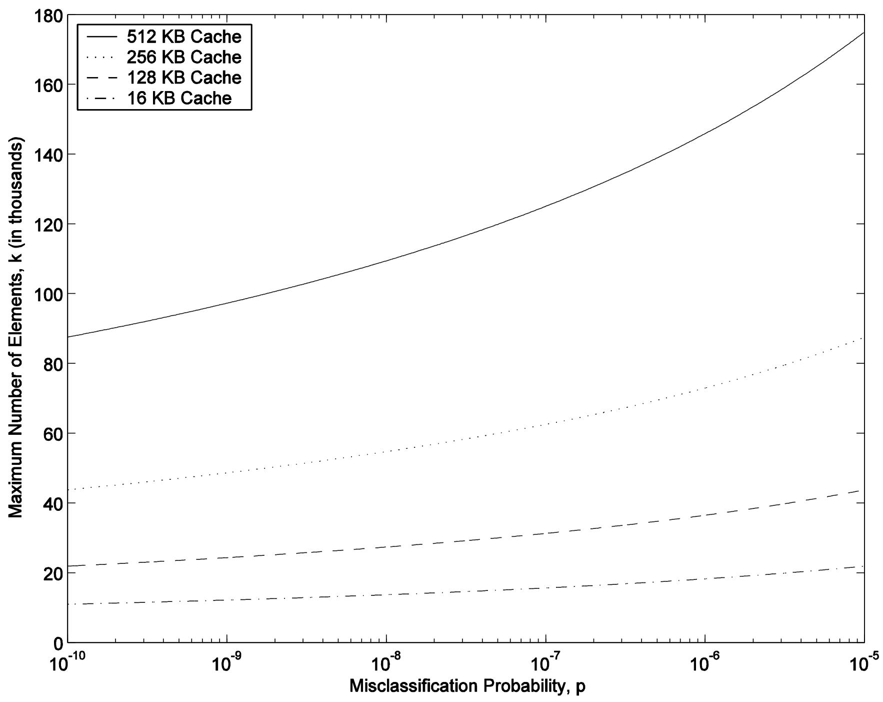


Figure 8: The trade-off between the misclassification probability, *p*, and the maximum number of elements, *k*, using optimum values of *L*.

Figure 8 graphs the relationship between *p* and *k*. We can see that the relationship is roughly logarithmic. This approximation serves as a good guide for ranges of two orders of magnitude or less.

A less obvious implication of this approximation is the relationship between the amount of memory, *M*, the number of elements, *k*, and the probability of a false positives, *p*. Since the optimal choice of *L* is asymptotically invariant with respect to *M*, and κ is proportional to *k*, we can assert that *k* is linearly related to *M*. A visual representation of this relationship is depicted in Figure 9. Note that a Bloom filter cache with a misclassification rate of one in a billion can store more than twice as many flows as an exact IPv4 cache, and almost 8 times as much as an exact IPv6 cache. (Each entry in an exact IPv6 cache consumes almost 3 times as large as an IPv4 entry [huitima].) The

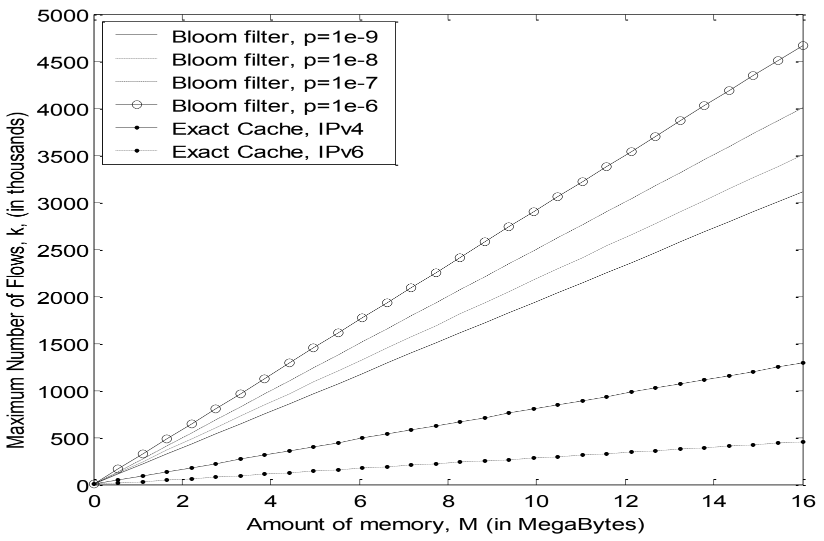


Figure 9: The relationship between the amount of memory, *M*, and the maximum number of elements (flows), *k* that can be stored while maintaining a given misclassification probability

effective storage capacity of the Bloom filter decreases logarithmically with the misclassification rate.

It is also important to recognize that with this scheme, it is possible to store mixed IPv4/IPv6 traffic without making any major changes to our design.

To summarize:

- The optimal value of *L* is invariant with respect to the size of the Bloom filter, *M*.
- *k* and *p* are roughly logarithmically related.
- *k* is linearly related to *M*.

2.3.3 Multiple Predicates

There are a number of applications where multiple binary predicate data may be useful in in a packet classification cache. For example, in the case of a router, the forwarding interface for must be stored along with the flow identifier. Our first extension

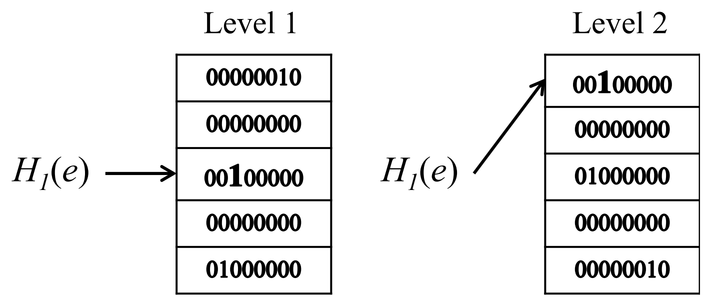


Figure 10: An example: A modified Bloom filter with 5 buckets and 2 hash levels, supporting a router with 8 interfaces. Suppose we wish to cache a flow *e* that gets routed to interface number 2.

to the Bloom filter is to extend its storage capability to a multiple binary predicate data structure. We propose a modification to our existing algorithm that allows us to store multiple binary predicates while preserving the desired original operating characteristics of the Bloom filter cache.

Consider a router with I interfaces. This is analogous to a data structure that records I binary predicates. To store this information, we will construct a cache composed of I Bloom filters. Suppose we are caching a flow, e, that should be routed to the i^{th} interface. We would simply insert e into the i^{th} Bloom filter in our cache. To query the cache for the forwarding interface number of flow e, we will simply need to query all I Bloom filters. If e is a member of the i^{th} Bloom filter, this implies that flow e should be forwarded through the i^{th} interface. If e is not a member of any Bloom filter, e has not been cached. In the unlikely event that more than one Bloom filter claims e as a member, we have an ambiguous result. One solution to this problem is to treat the cache lookup as a miss by reclassifying e. This approach preserves correctness while adding only minimal operating overhead for the small fraction of packets for which this will occur.

The probability of misclassification, p, with this algorithm is

$$p = 1 - \left(1 - \left[1 - (1 - 1/N')^{k'}\right]^{L'}\right)^{I} \tag{8}$$

Solving for *k'*, the maximum number of flows this approach can store, we find

$$k' = \frac{\ln\left(1 - \left[1 - (1 - p)^{1/I}\right]^{1/L'}\right)}{\ln(1 - 1/N')} \tag{9}$$

Using the same technique discussed earlier in Section 2.3.2, we find that k' is maximized when

$$L' = -\frac{\ln\left(1 - (1 - p)^{1/I}\right)}{\ln(2)} \tag{10}$$

The proposed extension to the Bloom filter cache requires increasing the amount of memory accessed by a factor of *I*. As will be shown in Section 2.5.3, additional memory accesses can incur serious performance penalty. However, by taking advantage of the memory bus width and fetching buckets from multiple Bloom filters simultaneously can easily mitigate this disadvantage (Figure 10).

Consider a Bloom filter in which each bucket can store a pattern of *I* bits, where bit *i* represents interface *i*. When adding a packet to the bloom filter, we would only update bit *i* of each bucket. When querying the modified Bloom filter for a flow, *e*, we will take the results from each level of the bloom filter, and AND the results.

2.3.3.1 Multiple Predicates with Non-Uniform Distributions

The equations presented earlier in Section 2.3.3 assume that elements are evenly distributed over the multiple binary predicates. If the elements are not evenly distributed, our modified Bloom filter can become polluted in a short amount of time.

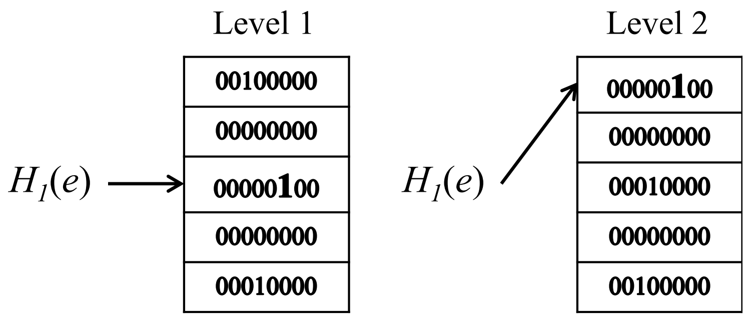


Figure 11: As before, suppose flow *e* is to be forwarded to interface 2. Now, let us suppose that *H'*(*e*) = 3. So *j* = (*i*+*H'*(*e*)) mod *I*=(2+3) mod 8 = 5.

For example, suppose a router that supports 16 interfaces (binary predicates), using 1KB of memory and a misclassification probability of 1e-9. If flows are distributed evenly over the interfaces, this configuration can support 167 elements. Conversely, if 90% of flows set the first predicate, it would require only 13 elements to "fill" this Bloom filter.

To compensate for this deficiency, suppose a new hashing function, $H': e \rightarrow [0 \ldots I - 1]$, and let $j = \big(i + H'(e)\big) \bmod I$. Instead of setting bit *i* in a Bloom filter, we will set bit *j* (Figure 11). This approach ensures that set bits are uniformly distributed throughout the cache, even when the elements are not evenly distributed.

2.3.3.2 Multiple Predicates Compared With Single Predicate Bloom Filters

It is important to examine how the multiple-binary-predicate Bloom cache compares to the single-predicate case. As discussed previously, the single-bit Bloom filter cache can store a maximum of $\kappa = -M/L \ln\big(1 - p^{1/L}\big)$. For an optimized choice of $L = -\ln \mathrm{p} \;/\ln 2$, κ becomes

$$\kappa_{\max} = \frac{M \ln(2)}{\ln(p)} \ln(1 - p^{-\ln 2/\ln p}) \tag{11}$$

The maximum number of flows the modified multi-bit Bloom filter can store is

$$k' = \frac{\ln\left(1 - \left[1 - (1-p)^{1/I}\right]^{1/L'}\right)}{\ln(1 - L'/M)} \tag{12}$$

Applying the approximation $1 - 1/N \approx e^{-1/N}$ we find

$$\kappa' = \frac{M}{L'} \ln\left(1 - \left[1 - (1-p)^{1/I}\right]^{1/L'}\right) \tag{13}$$

When *L'* is optimized, *κ'* becomes

$$\kappa'_{max} = \frac{M \ln 2}{\ln(1 - (1-p)^{1/I})} \ln\left(1 - \left[1 - (1-p)^{1/I}\right]^{v}\right) \tag{14}$$

where

$$v = \frac{-\ln 2}{\ln(1 - (1-p)^{1/I})} \tag{15}$$

Immediately, we can see that the two approaches are still linearly related in *M*. Note here that *I* and *p* are constants. This is an important property, because it means that our proposed algorithm preserves the behavior of the single binary predicate cache.

To better determine the relative performance of the multiple binary predicate and the single-binary-predicate cache approaches, we take the difference in the maximum number of flows that each design will accommodate.

$$\kappa_{max} - \kappa'_{max} = M(\ln 2)^2 \left(\frac{1}{\ln(1 - (1-p)^{1/I})} - \frac{1}{\ln p}\right) \tag{16}$$

For $p \ll 1$, $(1-p)^{1/I} \cong 1 - p/I$, giving

$$\kappa_{max} - \kappa'_{max} = \frac{M(\ln 2)^2}{\ln p} \left(\frac{\ln I}{\ln p - \ln I}\right) \tag{17}$$

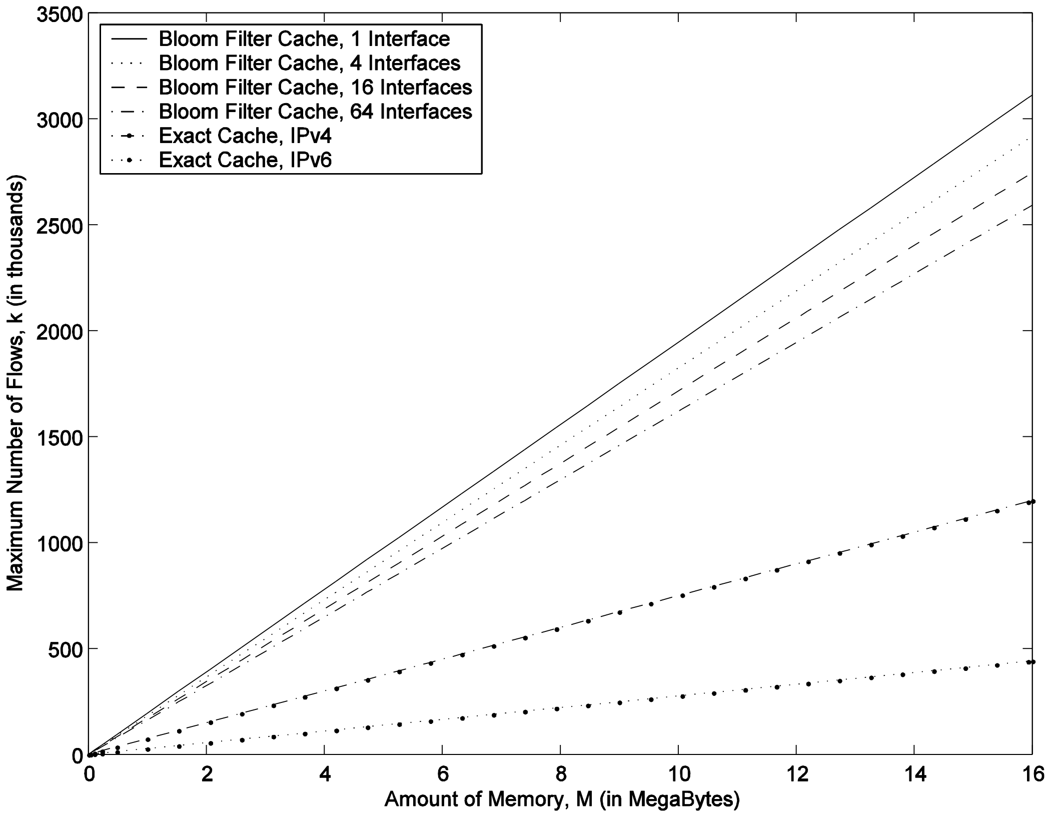


Figure 12: Comparison of storage capacity of various multi-predicate Bloom filters

If I is not very big, as is the case when considering the number of interfaces of a router (for reference, a Juniper T640 routing node has 160 interfaces) then $-\ln p \gg \ln I$, we can approximate by

$$\kappa_{max} - \kappa'_{max} = \frac{M(\ln 2)^2}{\ln p}\left(\frac{\ln I}{\ln p - 0}\right) = \frac{M(\ln 2)^2 \ln I}{(\ln p)^2} \tag{18}$$

This is an overestimate of the difference. So, we can say that, at worst, this approach scales logarithmically with I (for M and p constant).

It is surprising how effective this approach is. The algorithm does not pollute the Bloom filter by setting bits any more than the single-bit approach. However, it is slightly more susceptible to contamination because each membership query examines $L \times I$ bits, as opposed to the L bits of the single binary predicate Bloom filter.

Figure 12 compares the difference in the maximum number of flows that can be stored by a multi-predicate Bloom filter cache. The number of flows that a multiple predicate Bloom filter can store decreases logarithmically with the number of binary predicates.

Note that the multi-predicate solution is a superset of the single-predicate solution – setting I to 1 yields the equations presented in Section 2.3.1.

2.3.4 Bloom Filter Aging

This second extension to the Bloom filter adds the ability to evict stale entries from the cache. Bloom filters were originally designed to store digests of large amounts of static data – adapting this algorithm to gracefully evict elements is required to use this data structure meaningfully in a dynamic environment such as the Internet.

The first step towards developing an algorithm to age a Bloom filter is to decide how much information has already been stored in the cache. A simple method of deciding when the cache is full is to choose a maximum tolerable misclassification probability, p. When the instantaneous misclassification probability exceeds this constant, $(p_{instantaneous} > p)$, we consider the Bloom filter to be "full". We can calculate $p_{instantaneous}$ by using different means. Let $\omega_1, \omega_2, \ldots, \omega_L$ be the fractions of buckets of each level of the Bloom filter that are set. The probability of misclassification is simply the product of ω_i's.

$$P_{misclassification} = \prod_{i=1}^{L} \omega_i \tag{19}$$

This method will accurately estimate the misclassification probability. The drawback to this approach is that it will require counting the exact number of bits we set, complicating later parallel access implementations of this algorithm, as well as adding several per-packet floating-point operations.

We can devise a simpler estimate of $P_{misclassification}$ that does not involve precise bit counting, nor global synchronization, by applying knowledge of the properties of the Bloom filter discussed earlier. We simply need to count the number of flows k' that we have inserted into our Bloom filter. So our estimate of the misclassification probability becomes

$$P_{misclassification} = [1 - (1 - 1/N)^{k'}]^{L} \tag{20}$$

Reversing this equation, and solving for k_{max} we get

$$k_{max} = \left\lfloor \log\left(1 - P_{max}^{1/L}\right) / \log(1 - 1/N) \right\rfloor \tag{21}$$

This estimate also provides the benefit of simplicity of calculation – floating-point arithmetic is no longer required during runtime (since *P*, *N*, *L* are constant), only an integer comparison ($k' > k_{max}$). Additionally, it becomes easier to gauge the behavior of the cache - k' increases proportionally with the number of new flows we observe.

With this information, it is now possible to design an aging strategy for the Bloom-filter cache.

2.3.4.1 Bloom Filter Aging: Cold Cache Approach

This naïve approach to the problem of Bloom filter aging involves simply emptying the cache whenever the Bloom filter becomes “full”. The main advantage to this solution is that it makes full use of all of the memory devoted to the cache, as well as offering a simple implementation while maintaining a fixed worst-case misclassification probability.

The disadvantages, however, are quite drastic when considering the context of a high-performance cache:

- While the cache is being emptied, it cannot be used.
- Immediately after the cache is emptied, all previously cached flows must be re-classified, causing a load spike in the classification engine.
- Zeroing out the cache may cause a high amount of memory access.

This approach mainly serves as a reference point to benchmark further algorithm refinement.

2.3.4.2 Bloom Filter Aging: Double-Buffering

If we partition the memory devoted to the cache into two Bloom filters, an active cache and a warm-up cache, we can more gracefully age our cache. This approach is similar to the one applied in Stochastic Fair Blue [feng]. The basic algorithm is given in Figure 13. The goal of this approach is to avoid the high number of cache misses immediately following cache flush, which occurs when the cache is full and older, stale entries must be evicted. By switching to a background cache, we can start from a “warmed-up” state. This approach can be thought of as an extremely rough approximation of LRU.

However, this approach also has its drawbacks:

- Doubling the memory requirement to store the same number of concurrent flows, as compared to the cold-cache case.

```
when a new packet arrives
    if the flow id is in the active cache
        if the active cache is more than ½ full
            insert the flow id into the warm-up cache
        allow packet to proceed
    otherwise
        perform a full classification
        if the classifier allows the packet
            insert the flow id into the active cache
        if the active cache is more than ½ full
            insert the flow id into the warm-up cache
        allow packet to proceed
    if the active cache is full
        switch the active cache and warm-up cache
        zero out the old active cache
```

Figure 13: Pseudocode for double-buffer aging algorithm for Bloom filters

- Zeroing out the expired cache still causes a load spike in the use of the memory bus (although it is a smaller spike). This can be partially mitigated by slowly zeroing out memory.
- If the instantaneous number of concurrent flows, k_{inst}, is greater than k_{max}, this system will observe severe thrashing. Spikes in cache miss rates may be observed whenever $k_{inst} > k_{max} / 2$, depending on flow duration and packet inter-arrival rates
- The simplest, naive implementations of this algorithm will double the number of memory accesses required to store a new flow. This performance loss can be

recovered by memory aligning the two bloom filters, so that fetching a word of memory will return the bit states of both Bloom filters.

2.4 Approximate Algorithm 2: Digest Caches

In this section, we propose the notion of digest caches for efficient packet classification. The goal of digest caches is similar to Bloom-filter caches proposed in Section 2.3; it trades some accuracy in flow identification in exchange for increased performance.

There are two primary limitations of this Bloom filter cache design. First, each Bloom filter lookup requires *N* independent memory accesses, where *N* is the number of hash levels of the Bloom filter. For a Bloom filter optimized for a 1 in a billion packet misclassification probability, *N*=30. Second, no mechanism exists to recover the current elements in a Bloom filter, preventing it from using efficient cache replacement mechanisms such as LRU.

Digest caches, however, allow traditional cache management policies to be employed to better manage the cache over time. Instead of storing a Bloom filter signature of a flow identifier (source and destination IP addresses & ports and protocol type), it is only necessary to store a hash of the flow identifier, allowing for smaller sized cache entries. This idea is extended to accelerate exact caching strategies by building multi-level caches with digest caches in Section 2.4.4.

Network cache designs typically employ simple set associative hash tables, an idea that borrowed from traditional memory management systems design. The goal of the network cache is to quickly determine the operation or forwarding interface that should be used, given the flow identifier. Hashing the flow identifier allows traditional network processors to determine what operation or forwarding interface should be used while examining only a few of entries in the cache. One significant limitation of exact- match caches for flow identifiers is the need to store large flow identifiers (e.g. 37 bytes for an IPv6 flow identifier) with each cache entry. This limits the number of flows that can be stored in a cache and increases the time necessary to find information in the cache.

The most important property of a digest cache is that it stores only a hash of the flow identifier instead of the entire flow identifier. The goal of the digest is to significantly reduce the amount of information stored in the cache, in exchange for a small amount of error in cache lookups. Digest caches can be used in two ways. First, they can be used as the only cache for the packet classifier, allowing the packet classifier caches to be small. Second, they can be used as an initial lookup in an exact classification scenario. This allows a system to quickly partition the incoming packets into those that are in the exact cache and those that are not, as well as identifying likely match locations in the exact cache.

Digest caches are superior to Bloom caches in two ways. First, cache lookups can be performed in a single memory access. Second, they allow direct addressing of elements, which can be used to implement efficient cache eviction algorithms such as LRU.

2.4.1 Dimensioning the Digest Cache

The idea of a digest cache is to compare compact hashed flow identifiers to match cached flows, instead of comparing the larger full flow identifiers. In a sense, this scheme trades the accuracy of the cache for a reduced storage requirement. Cache memory is partitioned in a similar manner to a traditional, set-associative cache. When dimensioning the set-associative cache, we need to decide what level of associativity to use. Previous work has demonstrated that higher cache associativity yields better cache hit-rates [jain][li]. However, unlike a traditional exact set associative cache, in the case of the digest cache, an increase in the degree of associativity must be accompanied by an increase in the size of the flow identifier's hash to compensate for the additional probability of collision. If the digest is a c-bit hash, and we have a d-way set associative cache, then the probability of cache misidentification is

$$p \approx \frac{d}{2^c} \tag{22}$$

Equation 22 can be described as follows: Each cache line has d entries, each entry of which can take 2^c values. A misclassification occurs whenever a new entry has coincidentally the same hash value as any of the existing d entries. We must employ a stronger hash to compensate for increasing collision opportunities (associativity).

Figure 14 graphs the number of flows that a 4-way set associative can store, assuming different misclassification probability tolerances. The maximum number of addressable flows increases linearly with the amount of memory and decreases logarithmically with the packet misclassification rate.

2.4.2 Theoretical Comparison of Bloom Filters with Digest Caches

To achieve a misclassification probability of one in a billion, a Bloom filter cache must use 30 independent hash functions to use memory optimally. This allows us to store a maximum of $k_{bloomcache}$ flows in our cache,

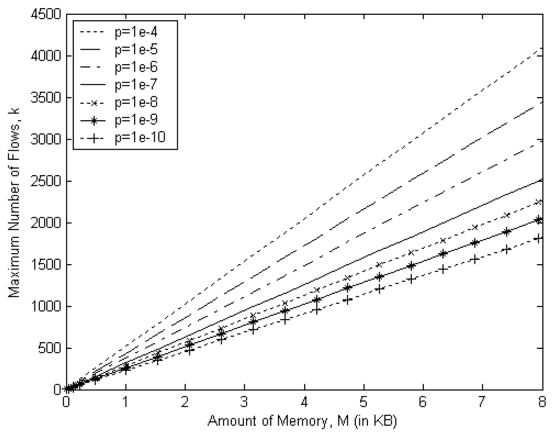


Figure 14: Maximum number of flows that can be addressed in a 4-way set associative digest cache, with different misclassification probabilities, *p*

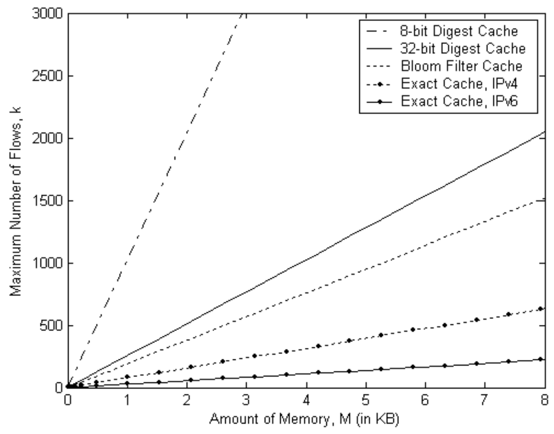


Figure 15: Comparison of storage capacity of various caching schemes. The Bloom filter cache assumes a misidentification probability of one in a billion, which under optimal conditions is modeled by a Bloom filter with 30 hash functions.

$$k_{bloomcache} = \frac{\ln\left(1 - p^{1/L}\right)}{\ln(1 - L/M)} \tag{23}$$

where L = 30, the number of hash functions, M, the amount of memory in bits, and p, the misidentification probability. To directly compare this with a digest cache, the maximum number of flows that our scheme can store, independent of the associativity, is given by

$$k_{digest} = \frac{\mathrm{M}}{\mathrm{c}} \tag{24}$$

where the required number of bits in the digest function is given by

$$c = \lceil \log_2(d/p) \rceil \tag{25}$$

This relation between $k_{bloomcache}$ and k_{digest} dependent on p, the misidentification probability and d, the desired level of cache set associativity.

Figure 15 compares the storage capacity of both caching schemes. Both schemes linearly relate storage capacity to available memory, but it is important to note that simply storing a hash is more than 35% more efficient in terms of memory use than a Bloom filter, for this application. One property that makes a Bloom filter a useful data

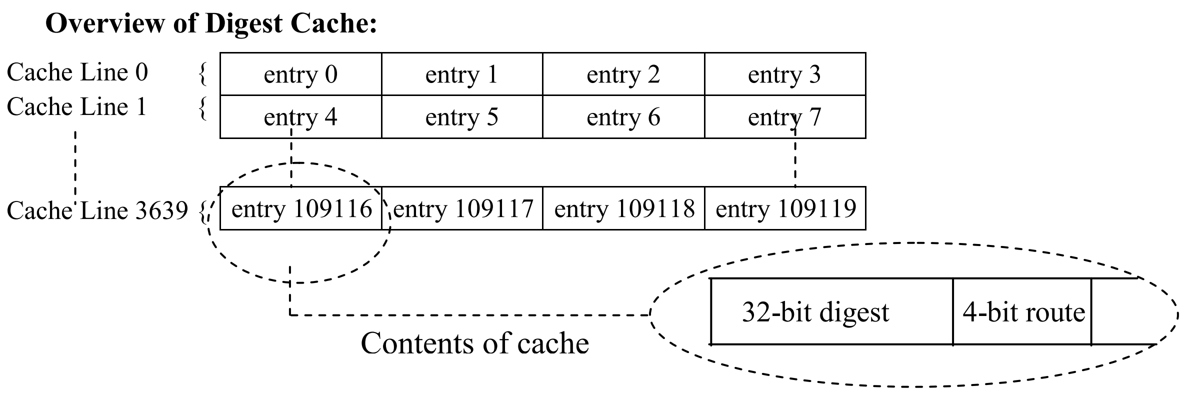


Figure 16: An overview of 64KB 4-way set associative digest cache, with a misclassification probability of 1 in a billion. This cache services a router with 16 interfaces.

structure is its ability to insert an unlimited number of signatures into the data structure, at the cost of increased misidentification. However, since we prefer a bounded misclassification rate, this property is of no use to the solution to our problem.

2.4.3 A Specific Example of a Digest Cache

To illustrate the operation of a digest cache, we will construct an example application of a digest cache. Consider a router with 16 interfaces and a set of classification rules. We begin by assuming that we have 64KB of memory to devote to the cache and wish to have a 4-way associative cache that has a misclassification probability of one in a billion. These parameters can be fulfilled by a 32-bit digest function, with 4 bits used to store per-flow routing information. Each cache entry is then 36 bits long, making each cache line 144 bits (18 bytes). 64KB of cache memory partitioned into 18-byte cache lines, gives a total of 3640 cache lines, which allows our cache to store 10920 distinct entries. A visual depiction of this cache is given in Figure 16.

Now, consider a sample trace of the cache, which is initially empty. Suppose 2 distinct flows, *A* and *B*.

1. Packet 1 arrives from flow *A*.
    a. The flow identifier of *A* is hashed to $H_1(A)$ to determine the cache line to look up. That is, H_1 is a map from flow identifier to cache line.
    b. *A* is hashed again to $H_2(A)$ and compared to all four elements of the cache line. There is no match. The result, $H_2(A)$, is the digest of the flow identifier that is stored.
    c. *A* is classified by a standard flow classifier, and is found to route to interface 3.
    d. The signature $H_2(A)$, is placed in cache line $H_1(A)$, along with its routing information (Interface 3).
    e. The packet is forwarded through interface 3.
2. Packet 2 arrives from flow *A*.
    a. The flow identifier of *A* is hashed to $H_1(A)$ to determine the cache line to look up.
    b. *A* is hashed again to $H_2(A)$ and compared to all four elements of the cache line. There is a match, and the cache indicates the packet should be forwarded through interface 3.
    c. The packet is forwarded through interface 3.
3. Packet 3 arrives from flow *B*.
    a. The flow identifier of *B* is hashed to $H_1(B)$ to determine the cache line to look up. Coincidentally, $H_1(A) = H_1(B)$.

b. B is hashed again to $H_2(B)$ and compared to all four elements of the cache line. Coincidentally, $H_2(A) = H_2(B)$. There is a false-positive match, and the cache indicates the packet should be forwarded through interface 3. The probability that this sort of misclassification occurs is approximately $4/2^{32} \approx 10^{-9}$.

c. The packet is forwarded through interface 3.

In the absence of false-positive matches, this scheme behaves exactly as a 4-way set associative cache with 14560 entries (3640 cache lines). Using an equivalent amount of memory (64 KB) a cache storing IPv4 flow identifiers will be able to store 4852 entries (1213 cache lines), and a cache storing IPv6 flow identifiers will be able to store 1744 entries (436 cache lines).

The benefit of using a digest cache is two-fold. First, it increases the effective storage capacity of cache memory, allowing the use of smaller, faster memory. Second, it reduces the memory bandwidth required to support a cache by reducing the amount of data required to match a single packet. As intuition and previous studies would indicate, a larger cache will improve cache performance [jain][li][partridge]. To that end, in this example, the deployment of a digest cache would have the effect of increasing the effective cache size by a factor of between 3 and 8.

2.4.4 Exact Classification with Digest Caches

Digest caches can also be used to accelerate exact caching systems, by employing a multi-level cache (Figure 17). A digest cache is constructed, in conjunction with an exact cache that shares the same dimensions (in number of cache lines and set associativity).

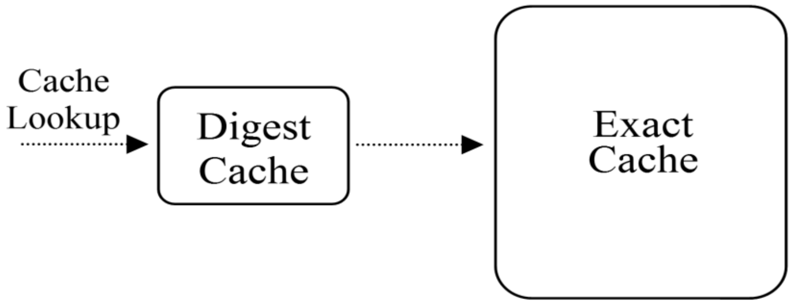


Figure 17: A multi-level digest-accelerated exact cache. The Digest cache allows filtering potential hits quickly, using a small amount of faster memory.

While the digest cache only stores a hash of flow identifiers, the exact cache stores the full flow identifier. Thus, the two hierarchies can be thought of as “mirrors” of each other.

A *c*-bit, *d*-way set associative digest cache implemented in a sequential memory access model will be able to reduce the amount of exact cache memory accessed (due to cache misses) by a factor of

$$p_{miss\_savings} = \frac{1}{2^c} \tag{26}$$

while the amount of exact cache memory accessed by a cache hit is reduced by a factor of

$$p_{hit\_savings} = \frac{1}{d} + \frac{1}{2^c} \times \frac{d-1}{d} \tag{27}$$

The intuition behind Equation 27 is that each cache hit must access the exact flow identifier, while each associative cache entry has an access probability of 2^{-c}. Note the digest cache allows for multiple entries in a cache line to share the same value because the exact cache can resolve collisions of this type. Since this application relies on hashing strength only for performance acceleration and not for correctness, it is not necessary to have as strong a misclassification rate.

Trace Length	3600 s
Number of Packets	974613
UDP Packets	671471
TCP Packets	303142
Number of Flows	32507
Number of TCP Flows	30337
Number of UDP Flows	2170
Avg. Flow Length	23.2654 s
Avg. TCP Flow Length	13.8395 s
Avg. UDP Flow Length	155.041 s
Longest Flow	3599.95 s
Avg. Packets/Flow	29.9816
Avg. Packets/TCP Flow	9.99248
Avg. Packets/UDP Flow	309.434
Max # of Concurrent Flows	268

Table 1: Bell Labs research trace

Trace Length	3600 s
Number of Packets	15607297
UDP Packets	10572965
TCP Packets	5034332
Number of Flows	160087
Number of TCP Flows	82673
Number of UDP Flows	77414
Avg. Flow Length	10.2072 s
Avg. TCP Flow Length	11.2555 s
Avg. UDP Flow Length	9.08774 s
Longest Flow	3600 s
Avg. Packets/Flow	97.4926
Avg. Packets/TCP Flow	60.8945
Avg. Packets/UDP Flow	136.577
Max # of Concurrent Flows	567

Table 2: OGI OC-3 trace

A multi-level 8-bit 4-way set associative digest-accelerated cache will incur a 4-byte first level lookup overhead. However, it will reduce second level memory access cost of an IPv6-bit cache miss look up from 148 bytes to 37.4 bytes, and a cache miss look up from 148 bytes to .6 bytes. Assuming a 95% hit rate, the average cost of cache lookups is reduced to 4 bytes of first level cache and 35.6 bytes of second level cache.

2.5 Performance Evaluation of Approximate Caching Strategies

Two network traces were used to evaluate the effectiveness of the proposed caching strategies. Each of the two datasets represents a one-hour network trace. The first of the datasets was collected by Bell Labs research, Murray Hill, NJ. This dataset was made available through a joint project between NLANR PMA and Internet Traffic Research Group [bell]. The trace was from a 9 Mb/s link, consisting only of IP traffic, serving a staff of 400 people. The second trace was a non-anonymized trace collected at the OGI

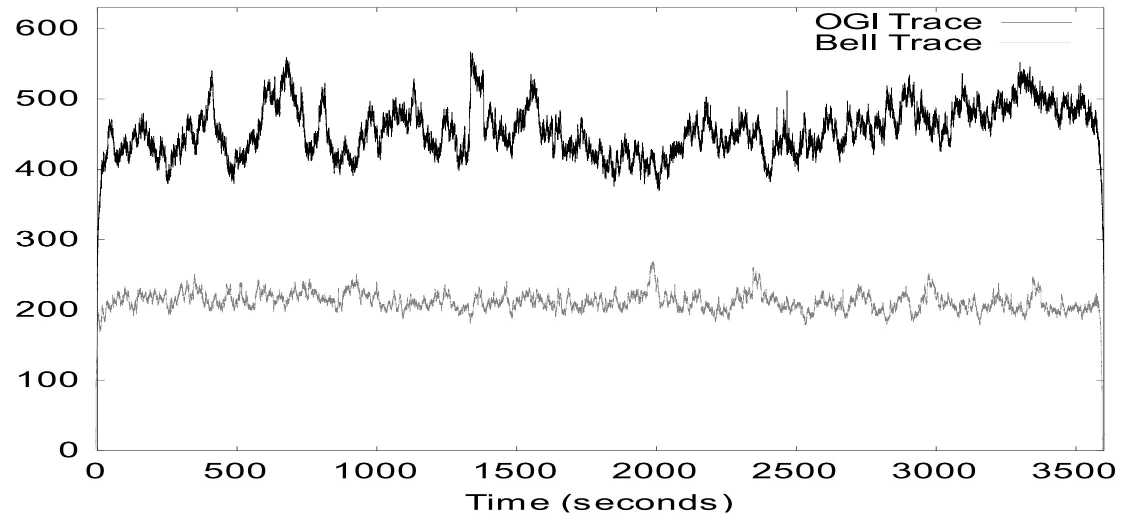


Figure 18: Number of concurrent flows in test data sets

OC-3c link. This link connects with Internet2 in partnership with the Portland Research and Education Network (PREN). This trace captured a portion of an active Half-life game server (an example of an interactive virtual world), whose activity is characterized by a moderate number (~20) of long-lived small high packet-rate UDP flows. Table 1 and Table 2 present a summary of the statistics of these two datasets. A graph of the number of concurrent flows is shown in Figure 18. The Bell trace is much smoother, while the OGI trace contains roughly twice as many concurrent flows.

For the purposes of our analysis, a bi-directional flow is considered as 2 independent flows. A flow begins when the first packet bearing a unique 5-tuple (source IP address, destination IP address, protocol, source port, destination port) arrives at the node. A flow ends when the last packet is observed, or after a 60 second timeout. This number is chosen in accordance with other measurement studies [fraleigh] and observations in the field [iannaccone][mccreary].

	Bell Trace	OGI Trace
Hit Rate	0.9714	0.9877
Maximum misses (over 100 ms intervals)	6	189
Variance of misses (over 100 ms intervals)	1.35403	17.4375
Average misses (over 100 ms intervals)	0.7749	5.8434

Table 3: The results of a perfect cache

As a reference benchmark, we introduce the idea of a perfect cache – a fully associative cache, with an infinite amount of memory. This cache takes only the theoretical minimum cache misses (compulsory cache misses). The fundamental performance statistics are reported in Table 3. The Bell trace is characterized by having more long-lived low-traffic flows, and so receives relatively few cache misses over a given time interval. Despite this, the cache hit rate is just 97%, because of the relatively few packets per flow there are. The OGI trace has roughly twice as many concurrent flows, and is characterized by having higher-traffic, short lived flows. As a result, the OGI trace has nearly 13 times as many cache misses, but achieves a cache hit rate of 99%.

For a comparison with exact caching schemes, we simulate a fully associative cache using an LRU replacement policy. LRU was chosen because of its near-optimal caching performance in networking contexts [jain]. This simulation is intended to represent best-case exact caching performance, even though it is infeasible to implement a fully associative cache on this scale.

2.5.1 Bloom Filter Cache Evaluation

For the reference implementations in this study, we use the SHA1 hash function [sha]. It should be noted that the cryptographic strength of the SHA1 hash does not

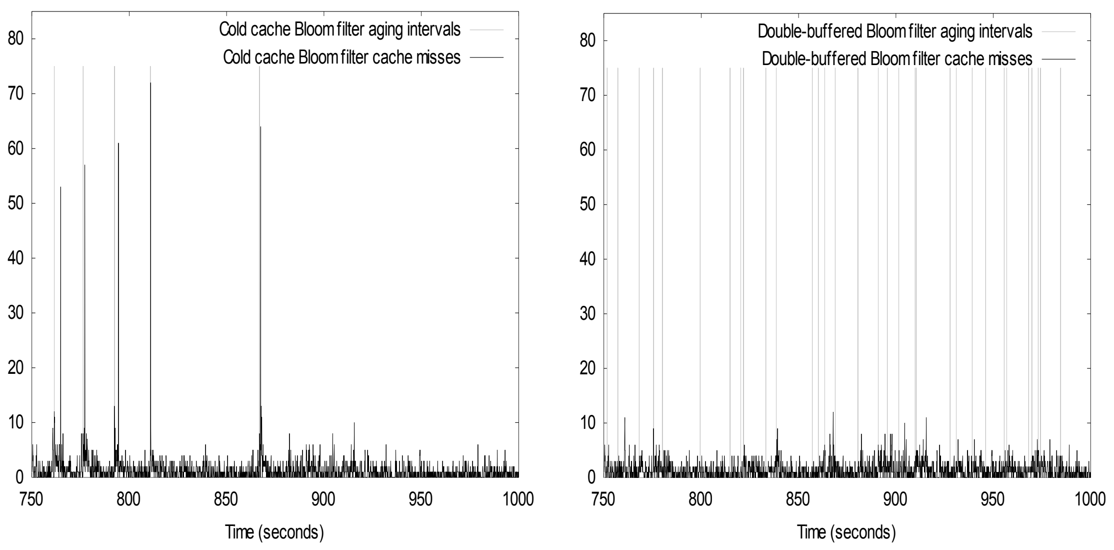


Figure 19: Comparing cold cache and double-buffered bloom caches using 4 KB of memory (Bell dataset)

increase the effectiveness of our implementation, because the hashed result does not have to be resistant to cryptographic inspection. Any universal hash functions will be equally effective for this use [carter]. It is important to recognize that other, faster hashing algorithms exist, and using a hardware-based hashing implementation is possible. For example, the IXP1200 architecture [ixp] has a hardware hashing unit that is suitable for this use, and can complete a hashing operation every nine clock cycles.

For the purposes of this study, we use a misclassification probability of 1 in a billion. The reasoning behind this choice of misclassification probability is presented in Section 2.2.1. This misclassification rate should have a completely negligible effect on the utility and performance of the network.

2.5.1.1 Bloom Filter Cold Caching Evaluation

With the Bell dataset, using 4 KB of cache memory and a misclassification probability of 1e-9 the cold cache performs reasonably with respect to the overall cache hit rate. The optimal dimensions for a Bloom filter this size should have 30 hash

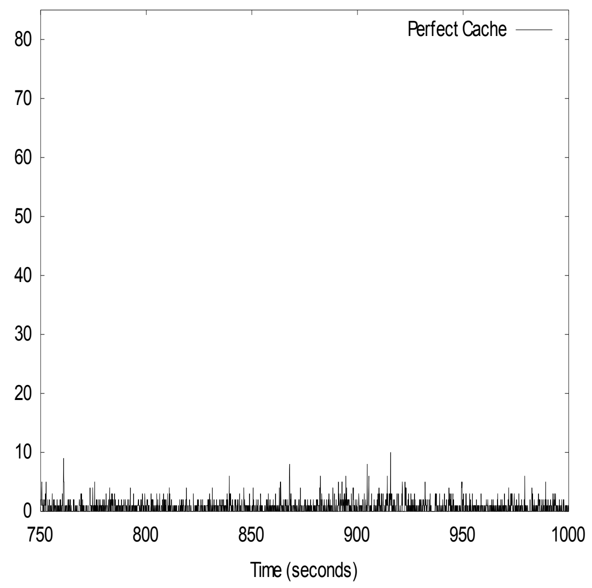


Figure 20: Cache Misses using a perfect cache (Bell dataset)

functions, storing a maximum of 611 flows. Throughout the 1-hour trace, there were no misclassifications and an overall cache hit-rate of 95.1529%. Aggregated over 100ms intervals, there were a maximum of 8 cache misses/100ms, with an average of 1.31668 and a variance of 10.3272.

Figure 19 illustrates the cache misses during a portion of the trace. We can see that emptying the cache corresponds to a spike in the amount of cache misses that is not present when using a perfect cache (Figure 20). This spike is proportional to the number of concurrent flows at the time of cache flushing. This type of behavior will apply undue pressure to the classification engine, resulting in overall performance degradation.

2.5.1.2 Bloom Filter Double-Buffering Cache Evaluation

Using a double-buffered approach can smooth the spikes in cache misses associated with suddenly emptying the cache. Double-buffering effectively halves the amount of immediately addressable memory, in exchange for a smoother aging function. As a result,

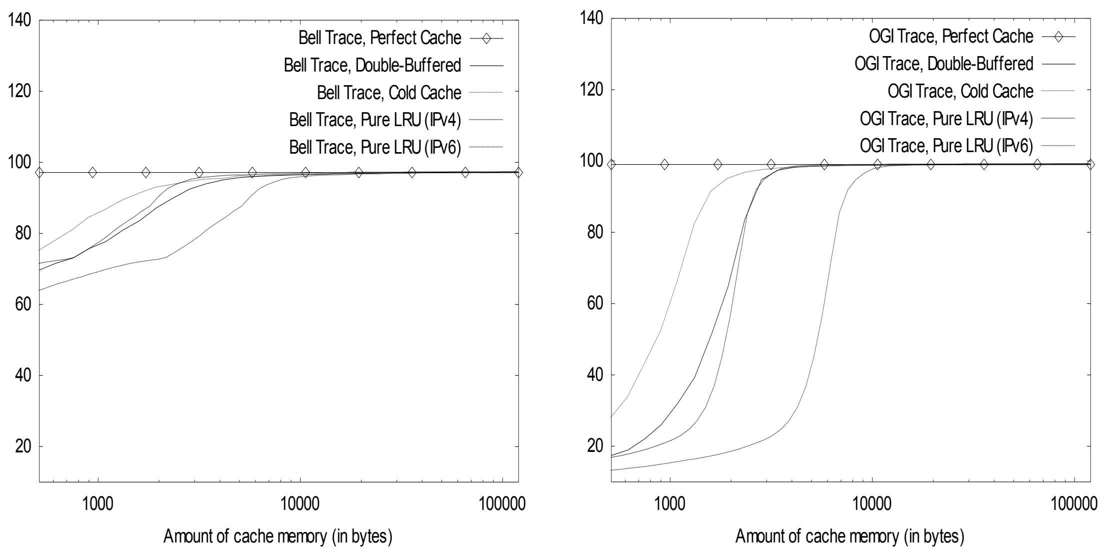


Figure 21: Cache hit rates as a function of memory, M

this bloom filter was only able to store 305 flows for a 4096 byte cache, in comparison with the 611 flows of the cold-cache implementation. This implementation also had a slightly lower hit rate of 95.0412% with the Bell dataset. However, we succeeded in reducing the variance to 5.43722 cache misses per 100ms, while maintaining an average cache miss rate of 1.34251 per 100ms. Reviewing Figure 19, we can see that the correspondence between cache aging states and miss rates does not correspond to performance spikes as prevalently as in the cold cache implementation.

This implies that the double-buffered approach is an effective approach to smoothing out the performance spikes present in the cold cache algorithm. To better quantify the "smoothness" of the cache miss rate, we graph the variance, and average miss rates (Figure 21 and Figure 22). From these graphs, we observe that for a memory-starved system, the cold-cache approach is more effective with respect to cache hit-rates. It is surprising how effective this naïve caching strategy is, with respect to overall cache performance. Moreover, we note that it performs better than both an IPv6 and IPv4 exact

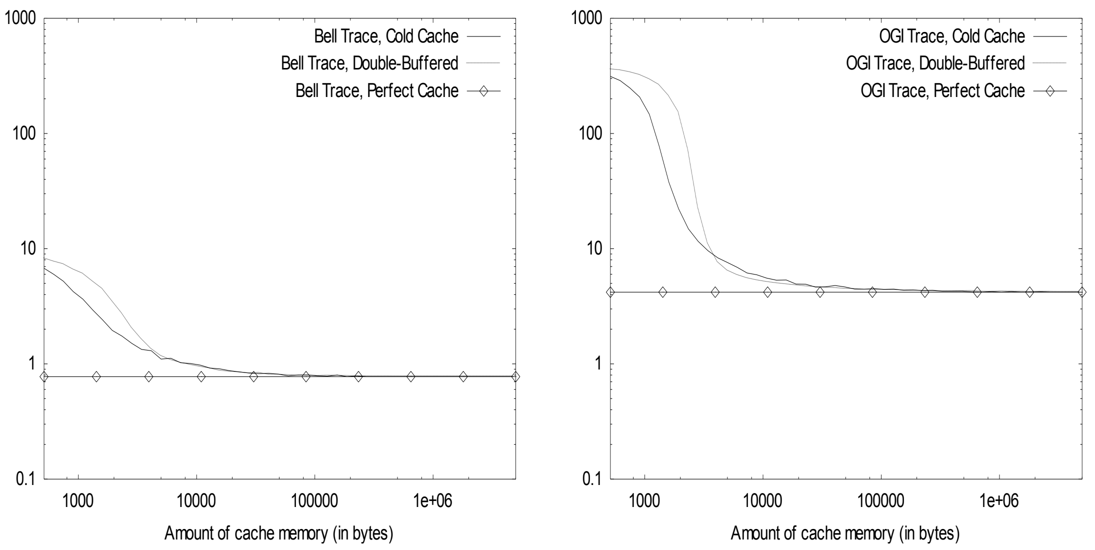


Figure 22: Average cache misses as a function of memory, *M* (aggregate over 100ms timescales)

cache, with both datasets for a memory starved cache, and keeps pace as memory improves. As the amount of memory increases, we can see that the double-buffered approach is slightly more effective in reducing the number of cache misses.

Looking to Figure 23, we observe that the variance in miss rates decreases much faster in the double-buffered case than in the cold-cache approach. This is because of the removal of cache miss “spikes” that occur during cache flushing, due to the need to suddenly repopulate the cache. It is interesting to note that in the OGI trace, the variance actually increases, before it decreases. Comparing Figure 22 and Figure 23, this implies that for a very memory-starved system, the variance is low because the cache miss rate is uniformly terrible.

Comparing the double-buffered approximate cache implementation to exact caching gives comparable performance when considering an IPv4 exact cache even though the approximate approach can cache many more flows. This is due to the imprecision of the aging algorithm – an LRU replacement policy can evict individual flows for replacement,

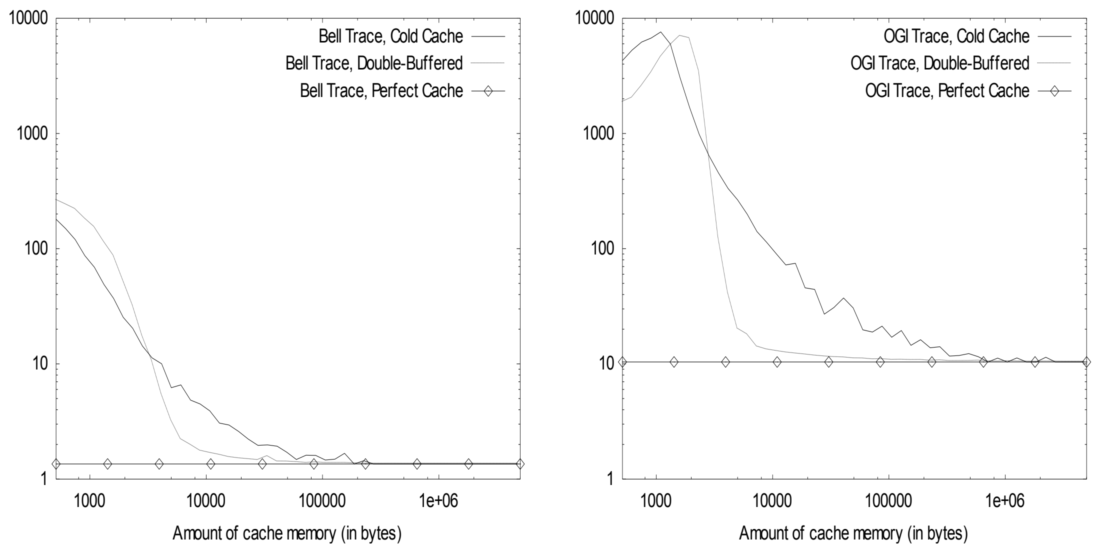


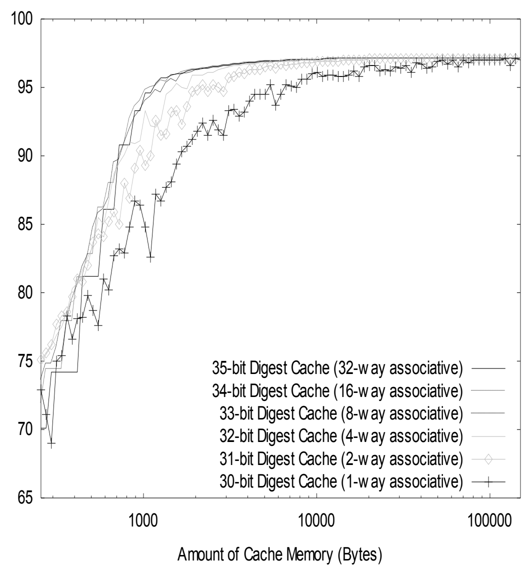
Figure 23: Variance of cache misses as a function of memory, *M* (aggregate over 100ms timescales)

whereas a double-buffered approach must evict ½ of the cached flows at a time. However, when considering IPv6 data structures, this disadvantage is overshadowed by the pure amount of storage capacity a Bloom filter can draw upon.

In all of these experiments, the behavior of each of the systems approaches the theoretical optimum cache performance as memory increases. This implies that our algorithm is correct and does not suffer fundamental design issues.

2.5.2 Digest Cache Performance Evaluation

The digest cache presented in this evaluation was chosen to be a four-way set associative hash table, using 32-bit flow identifier digests. Each lookup and insertion operation requires a single 16-byte memory request. An LRU cache replacement algorithm was chosen, due to its low cost complexity and near-optimal behavior [jain]. Figure 24 graphs the behavior of digest caches with different set associativities. By increasing the level of set-associativity of the cache, thrashing is reduced because of the reduction in cache contention. However, increasing the level of set-associativity also



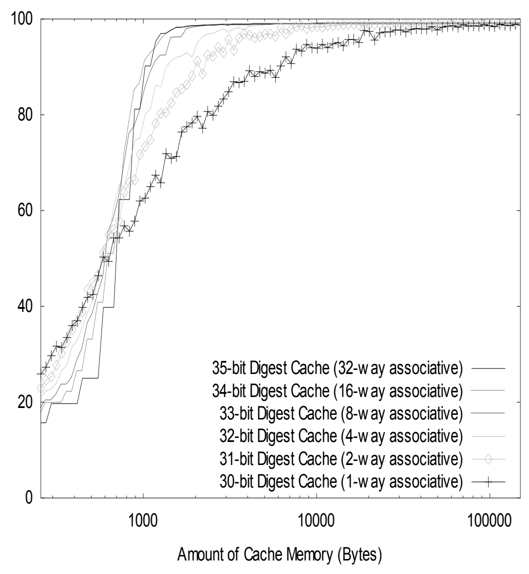
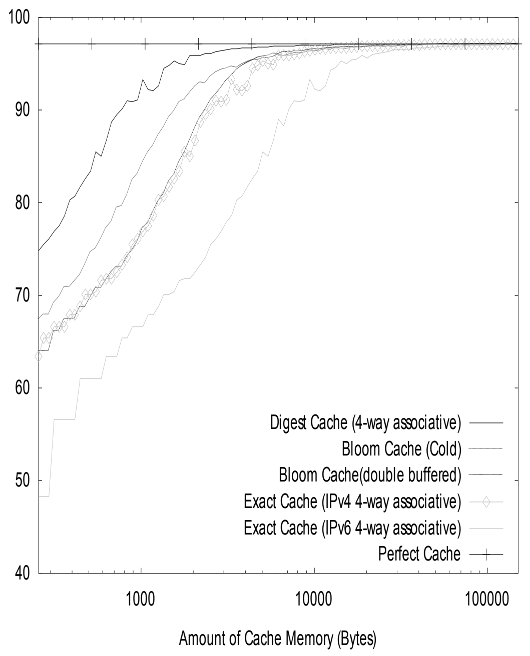


Figure 24: Hit Rates for digest caches, as a function of memory for various set associativity, assuming a misclassification rate of 1 in a billion

increases the amount of memory required to support that level of associativity, as well as the amount of cache memory that must be examined on each cache query operation. We observe from the graph that the effective performance of the cache increases very little after the level of set-associativity increases past four. This is consistent with other experimental observation [li].

We also compare our cache against a traditional four-way set associative layer-4 IPv4 and IPv6 based hash tables. Each lookup and insertion operation requires a single 52-byte or 148-byte memory request, respectively. Hashing for all results presented in this evaluation was accomplished with a SHA-1 hash [sha].As is with the Bloom filter evaluation (Section 2.5.1), the cryptographic strength of the SHA-1 hash is not an important property of an effective hashing function in this domain and it is sufficient that it is a member of the class of universal hash functions [carter].



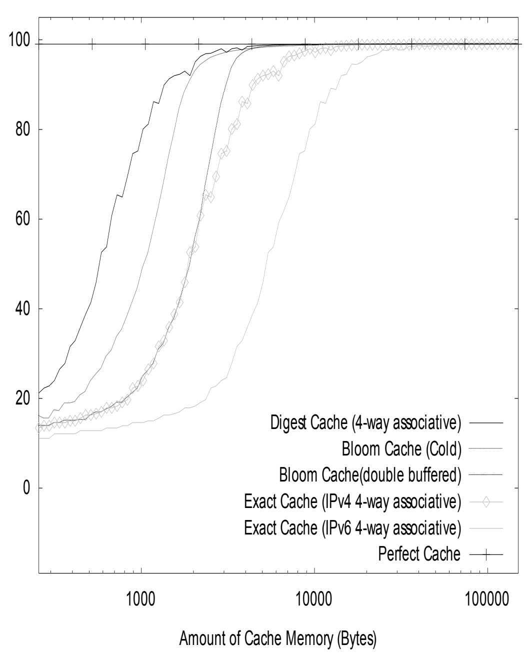


Figure 25: Cache hit rates as a function of memory, M. The Bell trace is on the left, the OGI trace is on the right

2.5.2.1 Digest Cache Results

In evaluating the performance of the caching systems, we must consider two criteria; we must examine the overall hit-rate as well as the smoothness of the cache miss rate. A cache that has large bursts of cache misses has low utility, because it places a high amount of stress on the packet classification engine.

Figure 25 graphs the resulting hit rate of various caching strategies, using the sample traces. As expected, the digest cache scores hit-rates equivalent to an IPv6 based cache ten times its size. More importantly, the digest cache still manages to perform well when compared with a Bloom filter cache. The digest cache yields an equivalent hit rate of a cold-caching Bloom filter 50-80% its size, and out-performs a double-buffered Bloom filter cache 2-3 times its size.

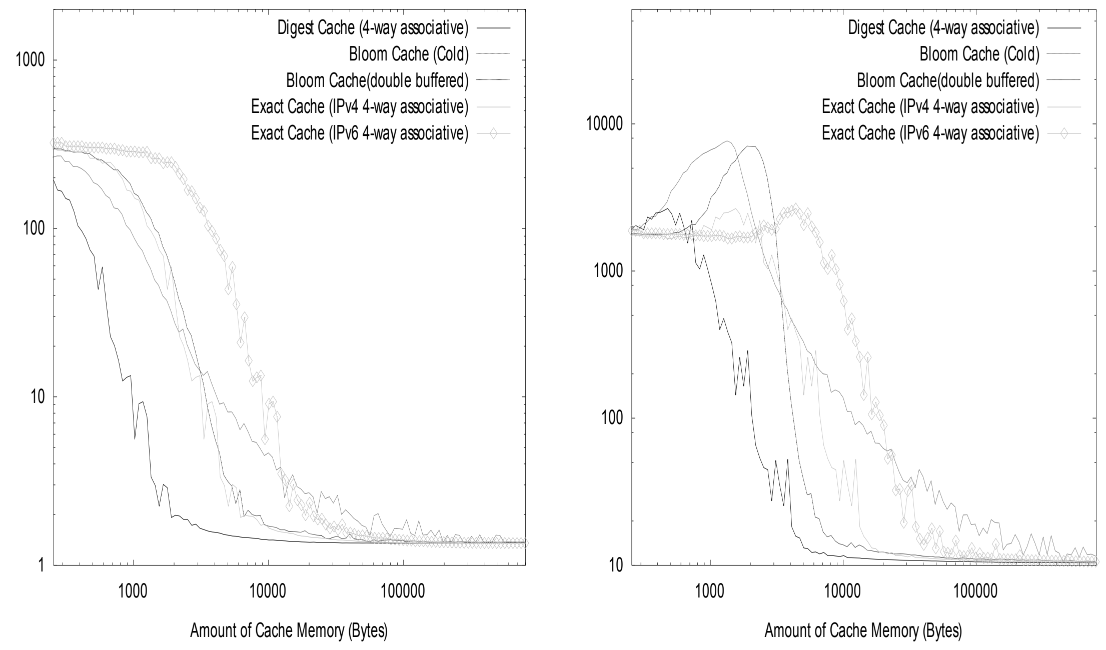
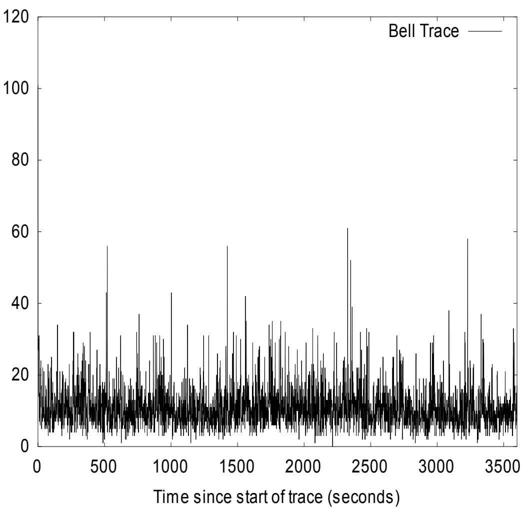


Figure 26: Variance of cache misses as a function of memory, *M* (aggregate over 100ms time scales). The Bell trace is on the left, the OGI trace is on the right .

Figure 26 graphs the variance of cache miss rates of the different caching approaches, aggregated over 100ms intervals. As can be observed from the two traces, a digest cache gives superior performance, minimizing the variance in aggregate cache misses. For extremely small cache sizes, the digest cache exhibits a greater variance in hit rate than almost all other schemes. This can be attributed to the fact that the other algorithms, in this interval, behave uniformly poor by comparison.

As the cache size increases, this hit rate performance improves, and the variance of cache miss rates decreases to a very small number. This is an important observation because it implies that cache misses in these traces are not dominated by bursty access patterns, which would place a high amount of pressure on the packet classifier.

To consider a more specific example, we have constructed a 2600 byte 4-way set associative digest cache. This number was chosen to be coincidental with the amount of



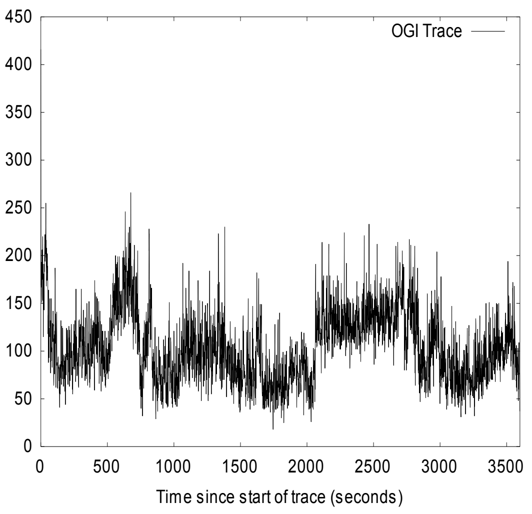


Figure 27: Cache miss rates aggregate over 1 second intervals, using a 2600 byte 4-way set associative digest cache. The Bell trace gave a 95.9% hit rate, while the OGI trace achieved a 97.6% hit rate.

local memory available to a single IXP2000 family micro-engine [ixp]. Figure 27 presents a trace of the resulting cache miss rate, aggregated over one-second time intervals. This graph represents the number of packets a packet classification engine must process within one second to keep pace with the traffic load. As can be observed from the plot, a packet classification engine must be able to classify roughly 60 packets per second (pps) in the worst case for the Bell trace, and 260 pps in the worst case for the OGI trace. Average packet load during the entire trace is 270.7 and 4335.4 pps for the Bell and OGI traces respectively. The peak packet rate for the Bell trace approached 1400 pps, while the peak rate for the OGI trace exceeds 8000 pps.

By employing a 2600 byte digest cache, the peak stress level on the packet classification engine has been reduced by a factor of between 20 and 30 for the observed traces.

2.5.3 Hardware Specific Implementation

A preliminary implementation on Intel's IXP1200 Network Processor was constructed, to estimate the amount of processing overhead an approximate cache would add [ixp]. The IXP1200 has a 3-level memory hierarchy: scratchpad, SRAM and SDRAM, each having 4KB, 16MB and 256MB respectively. Scratchpad memory is the fastest of the three, but does not support queued memory access – subsequent scratchpad memory accesses block until the first access is complete. The IXP micro-code allows for asynchronous memory access to SRAM and SDRAM. The typical register allocation schema allows for a maximum of 32 bytes to be read per memory access.

The hardware tested was an IXP1200 board, with a 200 MHz StrongARM, 6 packet-processing micro-engines and 16 Ethernet ports. The implementation's input buffers were kept constantly filled, and we monitored the average throughput of the system. A simple micro-engine level layer-3 forwarder was implemented as a baseline measurement. A cache implementation was then grafted onto the layer-3 forwarder code base. A null-classifier was used, so that we could isolate the overhead associated with the cache lookup function. No aging function was used for the Bloom Filter caches. The cache was placed into SRAM because the scratchpad does not support queued memory access which prevents the multi-processing power of the IXP design to be used, and the SDRAM interface does not support atomic bit-set operations which are needed to resolve concurrency issues in the multi-processing design.

The performance of our implementation was evaluated on a simulated IXP1200 system, with 16 virtual ports. The implementation's input buffers were kept constantly filled, and we monitored the average throughput of the system. The Bloom Filter

Number of Hash Levels	No-Miss Cache Throughput	All-Miss Cache Throughput
0	990 Mb/s	990 Mb/s
1	830 Mb/s	770 Mb/s
2	772 Mb/s	748 Mb/s
3	740 Mb/s	733 Mb/s
4	612 Mb/s	605 Mb/s

Table 4: IXP implementation overhead for Bloom filter caches

implementation was modified into two separate configurations to generate performance results. The first configuration simulates a no-miss cache. Each packet processed by the IXP would go through a full cache lookup with no shortcutting. Regardless of the result, the packet would be processed without adding (or re-adding) its signature to the cache.

In the second implementation, an all-miss cache was also constructed. All processed packets would always have their packet header digests added to the Bloom filter, regardless of the result of the packet lookup. The design ensured that no flow identifier was successfully matched, and each packet required an insertion of its flow ID into the cache. The code was structured in a way to disallow any shortcutting or early negative membership confirmation. This was done so that the worst possible performance of a Bloom filter cache could be ascertained. In this manner, we can determine an upper and a lower bound on the IXP's performance, presented in Table 4. The trace was composed entirely of small, 64-byte packets as is typical of virtual world update packets.

This implementation used the hardware hash unit. In this case, four hashes are as computationally expensive to calculate as one, because we simply use different portions of a single hashing result to implement multiple hash functions. This implementation appears to be SRAM limited – in the logged traces, we often note that the SRAM access queues are filled, stalling even asynchronous SRAM accesses.

A digest-cache was also constructed. Using the experimental results as a guide, a four-way set associative digest cache using 32-bit flow identifier digests placed in SRAM was constructed. This implementation was able to maintain a sustained average no-miss throughput of 803 Mb/s and all-miss throughput of 797 Mb/s. This implementation is roughly equivalent to a Bloom Filter using two hash levels, which is an inefficient design for a Bloom Filter using a targeting misclassification rate of one in a billion – ideally, there would be 30 hash levels. Given a 16MB SRAM store, and a 1e-9 misclassification rate, this Bloom filter could only store 2122 flow identifiers (Equation 23). A digest cache with the same memory constraints could store 4194304 flow identifiers, and would be easier to augment to store more flow meta-information, such as output routing interface numbers, or NAT flow ID translation parameters.

The IXP is far from an ideal architecture to implement a Bloom filter in large part due to its lack of small, high-speed bit-addressable on-chip memory. Ideally, a Bloom filter would be implemented in hardware that supports parallel access on bit-addressable memory [sanchez].

2.6 Conclusion

Online virtual worlds typically have different packet traffic distributions than the majority of Internet traffic, and are characterized by having frequent small updates. To help process the high packet-rate traffic generated by these applications, network devices can employ a packet classification cache. In this chapter, we have proposed and explored two different mechanisms for efficiently and effectively using memory, given a slightly relaxed accuracy requirement. Performance of any existing flow caching solution that employ exact caching methods can be dramatically improved by employing these techniques, at the sacrifice of a small amount of accuracy. With the deployment of IPv6 and the storage required to support the caching of its headers, such a trade-off will become increasingly important. The digest caching solution proposed in this chapter is able to service nearly an order of magnitude more flows than its exact-caching counterpart by allowing a cache misclassification rate of one in a billion.

This technique can be applied to the design of a novel 2-level exact cache which can take advantage of a hierarchical memory structure to accelerate exact caching algorithms. By adding a small (approximately 1/40th the size) digest cache to a more traditional *n*-way set associative IPv6 cache, we can reduce the amount of exact cache memory that is required to be accessed on a cache hit by $1/n$, and by a cache miss to nearly 0.

The digest caching approach is superior to the Bloom filter approximate caching algorithm, in both theoretical and practical performance while also addressing the shortcomings in the Bloom Filter cache design without introducing any additional drawbacks.

2.7 Future Work

The work presented in this chapter assumes that memory used to support the packet classification cache is homogeneous, or in the case of using a digest cache as an acceleration structure for an exact cache, potentially using two different kinds of memory with different access speeds and sizes. There is room to be explored in modifying this algorithm to be better suited to taking advantage of heterogeneous memory types. If the memory used to support the packet classification cache is accessed through a multi-level cache (as would be the case in a modern CPU cache) it may be possible to reorganize the cache to provide better spatial locality so that the multi-level cache is more effective. It may also be possible to use different kinds of memory, such as content addressable memory to accelerate this algorithm.

Chapter 3 Terrain Data Representation and Streaming

Virtual reality systems [active][croquet][gearth][wwind][wow][sl] have risen in popularity with readily available high-speed networking and affordable consumer computer graphics processing hardware. However, the deployment of networking hardware has not kept pace with the increasing quality, quantity and complexity of visualization data. Even with these advances, the increasing level of detail of virtual environments can easily consume any additional gains in bandwidth. In a virtual world or client/server video game, world information such as buildings, map and terrain are stored in a remote central server. Clients or players connecting to the virtual space will need to download the virtual world in a manner that maximizes interactivity – the world should be progressively streamed to minimize pre-buffering delay and should quickly converge to a high-quality scene rendering. In a large dynamic virtual world, the environment data must be downloaded on-demand, rather than before runtime, because the data can be prohibitively large and constantly changing.

To provide a high quality interactive experience for the user, we need to devise techniques and algorithms that are aware of networking limitations to deliver a maximal quality model of the world for the remote viewing client to render in as short a time as possible. These models should be transmitted progressively, in a compact form that allows the quality of the models to increase as time goes forward and more data is transferred. To best prioritize the portions of graphical data to send, the server should consider the remote client's viewpoint.

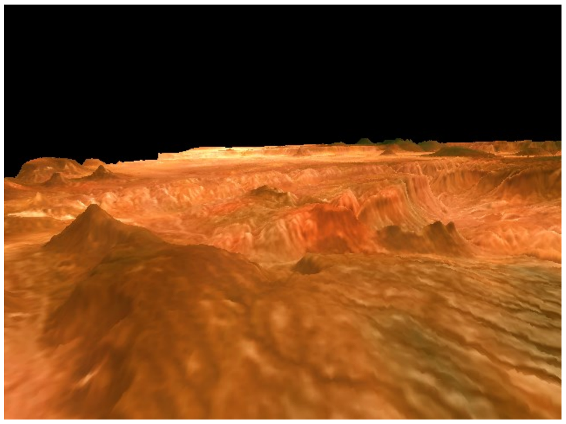


Figure 28: Screenshot of a terrain fly-through

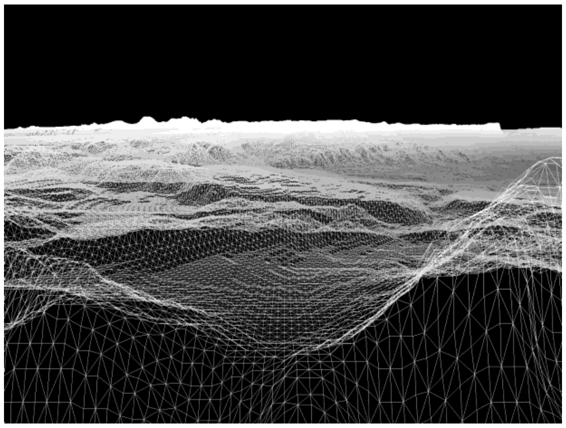


Figure 29: Underlying rendered geometry

It is important to realize that streaming virtual world information cannot use traditional video streaming algorithms. In a video stream, the data is quality-adaptive to available bandwidth, but is relatively constant in data rate and the data can be discarded after the user has viewed that portion of video. Video streaming has a more constant cost, in terms of network bandwidth requirements. In a virtual world context, data that is sent to the client can be cached and used for future use as the user explores and revisits that section of the world, only needing to update the cache when the world changes. Streaming data for virtual worlds will not require as much bandwidth after the user has finished downloading all the data about their surrounding environment.

Streaming computer graphics data is challenging because of the need to retrieve large triangle meshes before any display can begin. In this chapter, we focus on the streaming delivery of terrain data for remote viewing. Figure 28 shows a scene with a rendered terrain for viewing by a user and Figure 29 shows the underlying triangle mesh that represents the height field of that terrain. Due to bandwidth limitations, it is not possible to send the triangle/height field data all at once, and so we need to carefully choose what data to send to the client and how to prioritize the data that we do send. To achieve maximal quality remote rendering, two key techniques should be employed: First, distant terrain details and data that are outside the user's viewing frustum should be transmitted with a low priority. Second, terrain data that has a larger impact on the client's view, such as rocky terrain with many distinguishable features and areas nearer the viewer, should be streamed to the client with high priority.

This thesis proposes borrowing techniques from the lossy image compression domain to implement a novel technique of progressive terrain streaming over a network. By using a progressive streaming format to represent the height fields for terrain data, this approach allows client viewers to begin rendering the visual field almost immediately while capturing the essence of the terrain being represented. As the visualization progresses, more detail is streamed to a viewing client to improve the rendering accuracy of the scene. Using the Grand Canyon terrain data set [usgs], we compare the efficacy of using an approximate-algorithm based approach with a lossless terrain streaming algorithm. Our results show that we can present the user with a high quality interactive experience with smaller delay than would be possible using an exact-representation approach.

The goal of this work is to provide a quality-aware framework for remote 3-D rendering of height fields in a client/server model. In this study, the basic assumptions for modeling the interactive virtual world system are that local storage and computing power are large relative to network bandwidth, the network is reliable, and the network delivers all packets with minimal latency. These assumptions are chosen to reflect the goal of this research – to construct an algorithm that can deliver a high-quality 3D reconstruction of a terrain over constrained network infrastructure. The architectural model we follow is to construct a single server and client. The server stores all the world data and transmits it to the client in a quality-aware manner. The client is responsible for rendering the scene and sending viewer update information to the server.

Section 3.1 explores related work in the areas mesh simplification, and network streaming computer graphics. Section 3.2 outlines an experimental framework for evaluating the efficacy of a terrain streaming algorithm. Section 3.3 describes non-streaming reference algorithms, from which to compare the performance of proposed algorithms. Section 3.4 gives an exact-representation terrain streaming algorithm. Section 3.5 proposes and evaluates various strategies for an approximate algorithm approach to the terrain streaming problem.

3.1 Related Work

In practice, to display terrain data to a viewer, terrains are rendered to a display as triangle meshes (Figure 29). This is often a large amount of information, which is too dense for a computer's graphics hardware to render in real-time.

From the computer graphics field, a significant amount of work has been done to allow variable level-of-detail (LOD) rendering to alleviate the burden on computer graphics hardware, using progressive meshing techniques. Most of the work in this area focuses on arbitrary 3-dimensional meshes, as opposed to specific optimizations for height fields, which are explored in this chapter. Moreover, the viewer's perspective is usually not taken into account, resulting in suboptimal viewer-independent streaming algorithms [allies][chen][isenburg]. These techniques are done only to reduce the burden on the local graphics rendering hardware and cannot easily be adapted to rendering partial data, view-dependent refinement, and data that needs to be transmitted across a limited-bandwidth network to render.

When dealing with triangle-based meshes, one technique to simplify the overall geometrical complexity of a model is to use triangle decimation [schroeder]. This technique simplifies triangle-based meshes by combining adjacent triangles into a single larger and simpler triangle. For local real-time terrain rendering, this triangle-decimation technique has been adapted so that a terrain mesh can be simplified in real-time, termed "Real-time Optimally Adapting Meshes" (ROAM) [duchaineau][turner]. This approach organizes data into a binary triangle tree that allows progressive refinement of the data by visiting deeper nodes of the tree. By balancing the triangle rendering budget with the estimated visual importance of refining a specific area of the triangle mesh, a simplified viewer-adapted mesh representing the terrain can be rendered in real-time.

A network-aware transport protocol has been shown to significantly improve the speed and quality of progressive streaming in image data by explicitly modelling packet loss and performing out-of-order data processing [raman]. This approach improves the latency of progressive refinement. However, it does not consider view-dependent prioritization of regions of interest which can be inferred from an understanding of the three dimensional nature of the streaming data.

Several systems have been implemented for the streaming of computer graphics data. Second Life [sl] is a massively multiplayer online dynamic virtual world that allows users to explore a large three-dimensional space, where players can create, interact with, and exchange virtual items. Objects are described using a primitive constructive solid geometry model. Terrain and map information are sent in 16x16 tiles using a non-progressive JPEG-like encoding. In Second Life, the tiles are frustum-culled, so that only potentially visible tiles are sent, and tiles closest to the viewer are delivered first. This terrain encoding is not very efficient and is described in more detail in Section 3.3.1. These encoded tiles are used to render the terrain using a simple triangle-splitting algorithm based on an exponential distance metric. In OpenSimulator (Second Life's open-source counterpart), tiles are sent in row-major order (typewriter fill) with no consideration to the viewer's location or orientation [opensim].

In its original incarnation, Google Earth, a client/server virtual mapping program, used a similar approach, using frustum-culled non-progressive terrain-tile streaming system. Unlike Second Life, the streaming algorithm did not prioritize tiles based on the proximity to the viewer, sometimes resulting in distant terrain geometry being sent before nearby terrain data.

For streaming terrain, multi-resolution bitmaps for progressive rendering have also been employed. The data can be organized in a quad-tree structure, with each child node representing a refinement of one-quarter of the space [reddy]. This approach only considers viewer's location into account when streaming, without considering the visual importance of the existing geological features in the data. This approach has been extended by considering terrain complexity and culling terrain tiles outside the viewer's frustum, but not by prioritizing information based on viewer distance [tsai].

The strip mask approach to terrain streaming scheme divides the terrain into square tiles, attempting to pre-cache visible areas around the viewer [pouderoux]. This approach tries to minimize computational complexity for CPU-constrained devices by compiling terrain patches into display lists which can be quickly re-rendered by graphics hardware on successive frames. There is no data compression used in this approach, and progressive refinement is accomplished through sending triangle strips representing a new refinement level. More detail geometry can only be added on a per-patch basis, not on a per-vertex basis.

3.2 Framework for Experimental Evaluation

To evaluate the performance of a terrain streaming algorithm, we have constructed a simulated client viewing a fly-through of a landscape streamed from a remote server. This gives us the ability to analyze the visual quality of the streaming simulation, as viewed by a real interactive user (Figure 28). The client is implemented in OpenGL, rendering various fly-throughs of the terrain, in a 640 x 480 viewport. The rasterized output rendering by the client is recorded at 25fps and captured video rendering is compared with a reference ideal video rendering. The reference ideal video rendering is constructed by pre-downloading the entire terrain dataset, and rendering the fly-through in full quality, without any LOD simplification.

The results of these simulations are compared with this reference ideal video rendering using the peak signal to noise ratio (PSNR) metric. This PSNR metric will represent the visual quality of the streamed simulation compared to its exact fully-detailed representation. PSNR metrics are commonly used to measure the quality of video and image reconstruction when employing lossy compression codecs and is calculated by comparing a lossy reconstruction of an image and its original, and is represented using a logarithmic scale expressed in decibels (dB). Given an original image, *A*, which is *m* × *n* pixels in size, and reconstructed image, *B*, the mean squared error, *MSE*, is

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(a_{i,j} - b_{i,j}\right)^2 \tag{28}$$

where $a_{i,j}$ is the color value of pixel *i,j* of image *A* and $b_{i,j}$ is the color value of pixel *i,j* of image *B*. Then PSNR is defined as

$$PSNR = 10 \cdot \log_{10}\left(\frac{R^2}{MSE}\right) \tag{29}$$

where *R* is the maximum pixel value. In the case of an 8-bit image, $R = 255$.

The reference trace is constructed by running the simulation using a full-detail (unlimited network bandwidth with zero latency) rendering of the fly-through. Subsequent simulations are compared with this reference trace using the PSNR metric which represents the visual quality of the streamed rendering.

3.2.1 Underlying Network Assumptions

The experimental framework is completely simulated in a stand-alone executable – the simulation models a network with zero latency and a bandwidth of 56kbps. Given our 25fps capture rate, this effectively allows 380 bytes of data to be delivered between frames. The choice of a 56kbps stems from the idea that terrain data should only consist of a portion of a true virtual simulation's network stream. In a realistic scenario the data stream would include information such as objects, vehicles, buildings, textures and avatars, which sometimes must be transferred using a limited-bandwidth mobile wireless network link [blue].

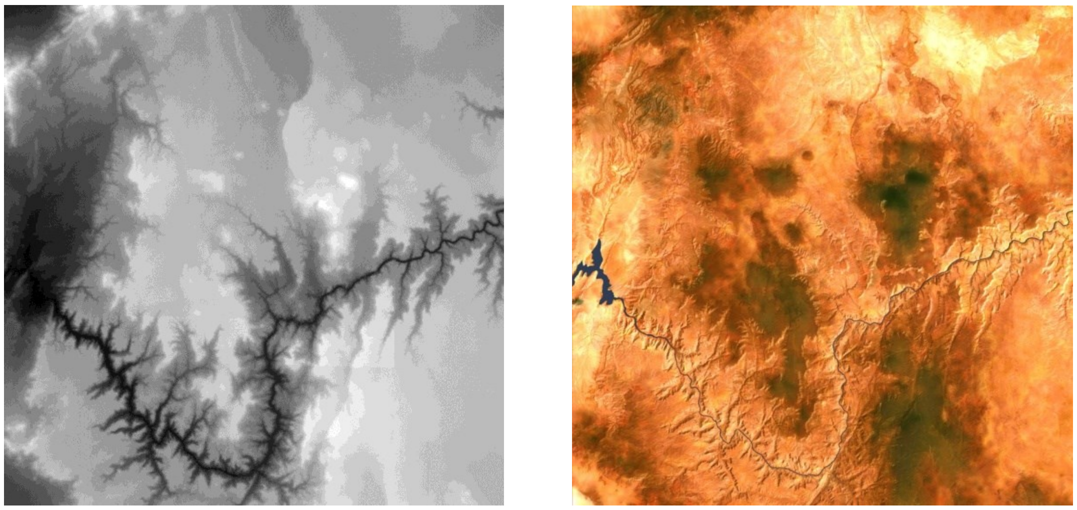


Figure 30: USGS dataset of the Grand Canyon. The height field information is on the left (lighter shades represent higher altitude) and lighting information is on the right.

This simulation deals only with terrain geometry. Texture, material and lighting information is not sent. In practice, this information can be generated procedurally. In such approaches, texture is inferred from the terrain geometry and need not be sent over the network.

3.2.2 Simulation Dataset

The simulation dataset used in this set of experiments is the Grand Canyon dataset from The U.S. Geological Survey (USGS) with processing by Chad McCabe of Microsoft Geography Product Unit [usgs]. The subset of this dataset used for simulation was based on a 2048x2048 grid with 8-bit height posts (Figure 30), representing an area of roughly 15000 km^2. This dataset was chosen because it expresses many characteristics of importance when considering terrain data, such as rocky mountainous regions, cliffs, canyons, and relatively flat plains. To test the streaming framework, we designed three representative walk-throughs to measure the performance of the various algorithms under different scenarios.

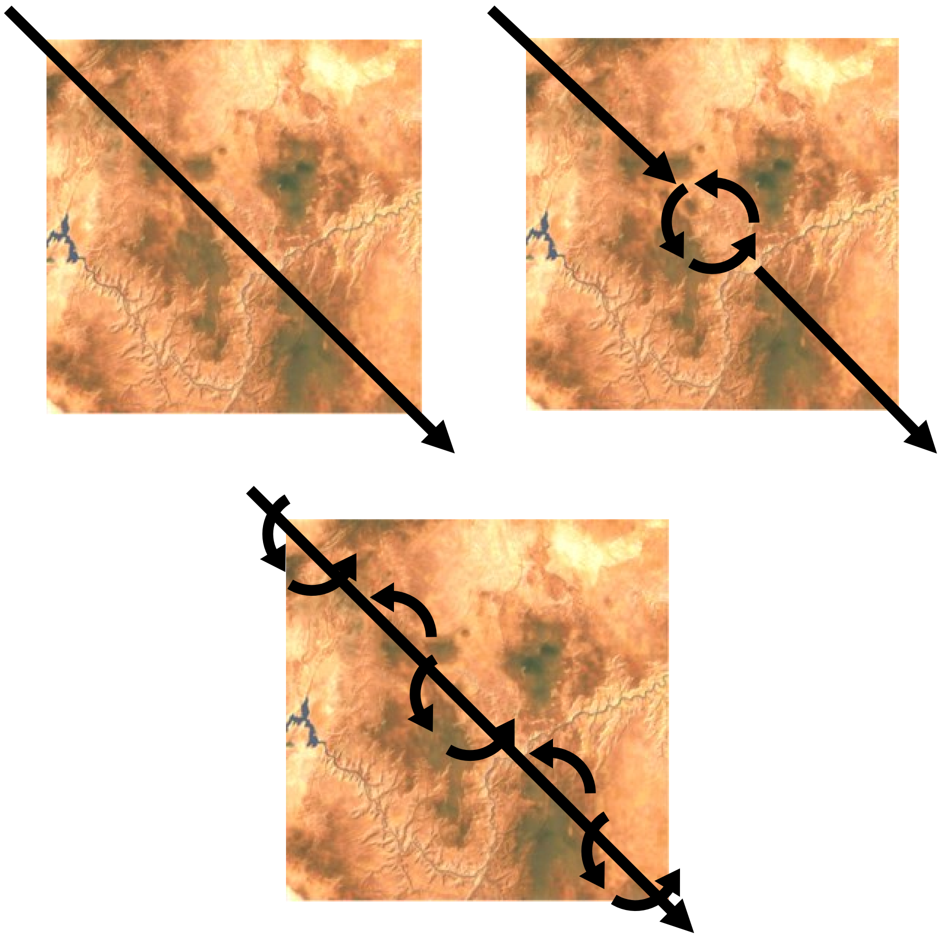


Figure 31: The three flythrough test scenarios. The arrows represent the path and direction of the viewer over the terrain.

The simplest terrain flythrough simulation we use simply crosses the simulated grid diagonally from corner to corner (Figure 31). This crossing is accomplished over 2048 rendered frames. The second flythrough also traverses the terrain from corner to corner (Figure 31). This flythrough is augmented by pausing in the center of the map to rotate the viewer 360 degrees. This requires the streaming system to cope with a changing client orientation. The total length of this simulation is 2768 frames. The third walk-through traverses the grid diagonally while continually panning over the terrain (Figure

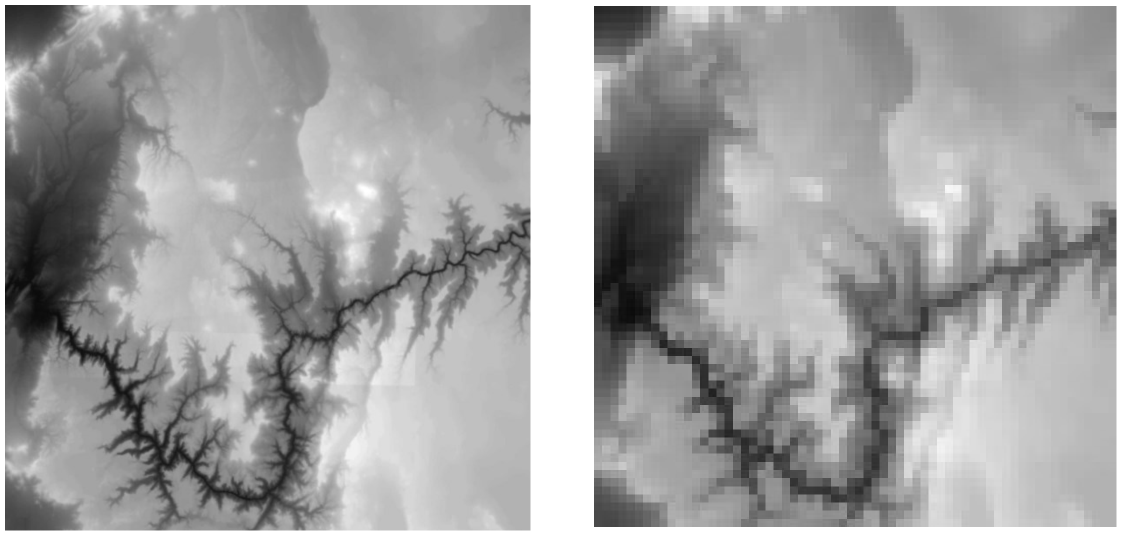


Figure 32: Visual representation of input terrain data (left) down-sampled to a 64x64 image (right).

31) over 2048 frames. This is the most demanding of the three walk-throughs, requiring the streaming solution to adapt to a constantly changing viewer location and orientation.

3.3 Reference Algorithms

Before considering any terrain streaming solution, it is important to first construct reference implementations in order to evaluate the efficacy of any proposed terrain streaming algorithms.

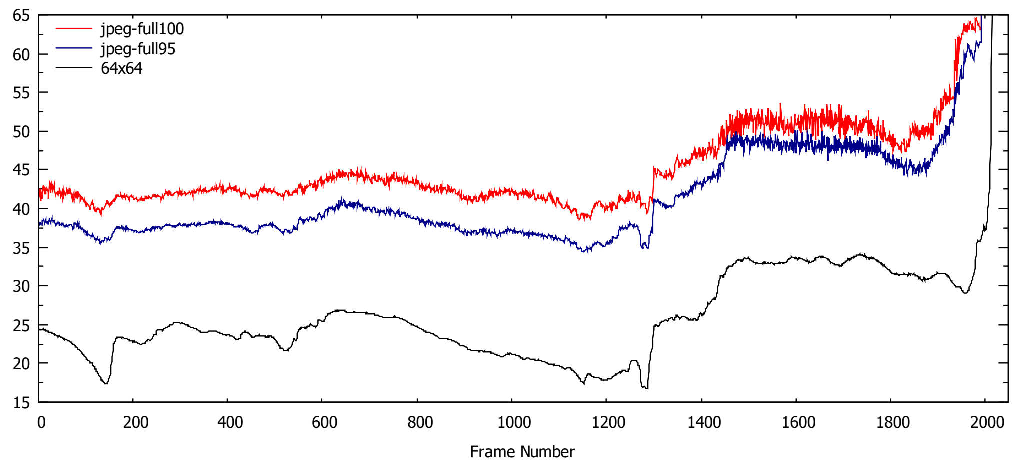
3.3.1 Non-Streaming Reference Algorithms

A very simple and naïve compression algorithm is to down-sample the entire 2048x2048 dataset to 64x64 (4KB of uncompressed data, less than 3KB of lossless PNG compressed data) (Figure 32). This can be transferred in approximately half a second with a 56.6 kbps connection. Instead of progressively streaming the terrain data from the server to the client, this first reference implementation simply pre-loads the simplified representation of the terrain at the viewer, and renders this terrain. Data points between samples are linearly interpolated to recover a 2048x2048 terrain dataset. If the dataset is treated as an 8-bit greyscale image, compared to the original raw dataset, this compression gives us a PSNR of 23.7015 dB. This algorithm represents the simplest, near worst-case performance simulation. The result of the simulation when conducted with this algorithm is graphed as *64x64* in Figure 33. In flythrough #1 and #2, the PSNR trends towards infinity towards the end of the simulation because the viewer travels passed the end of the terrain and there is no more data to display. In this case, the simulation displays the (lack of) data perfectly.

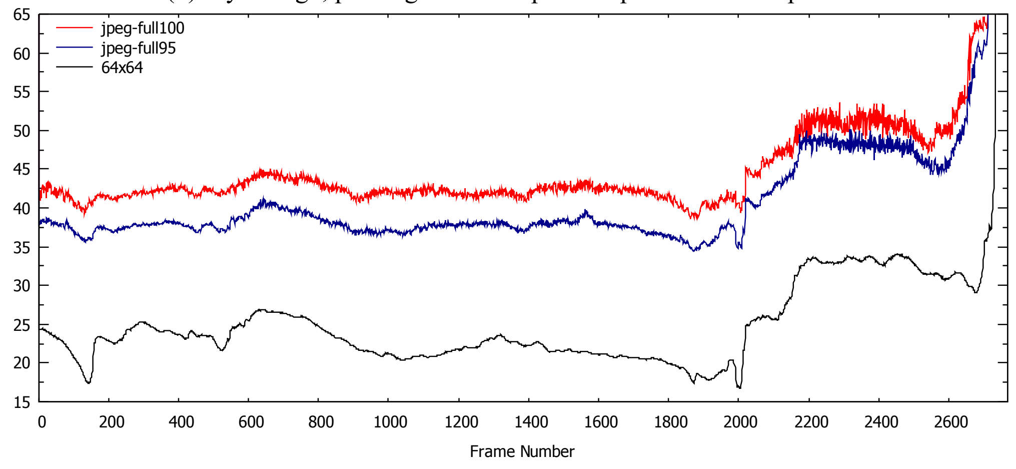
A less naïve approach would be to encode the entire terrain data set using JPEG greyscale image compression, with each of the height posts encoded as an 8-bit luminosity value. The JPEG encoding of the entire 2048x2048 Grand Canyon dataset, using a quality encoding level of 100 (maximum quality), compressed as a collection of 32x32 tiles, resulted in 1245870 bytes (1216KB) of data with a PSNR of 59.7080 dB. If the entire dataset is not subdivided into tiles, and encoded as a single JPEG file, the output file size is 985 KB. This means that encoding overhead incurred by tiling the data is roughly 20%. The importance of representing the terrain as a collection of tiles will be explored in Section 3.5.

The result of the simulation using this non-streaming approximate reconstruction is graphed as *jpeg-full100* in Figure 33. This simulation is representative of a near-ideal terrain representation, with minimal loss of fidelity. The results are very good – the PSNR of the measured simulation indicates that it continually maintains a high-quality rendering.

(1) Flythrough



(2) Flythrough, pausing at the midpoint to perform a 360˚ pan



(3) Flythrough with a continuous 360˚ pan

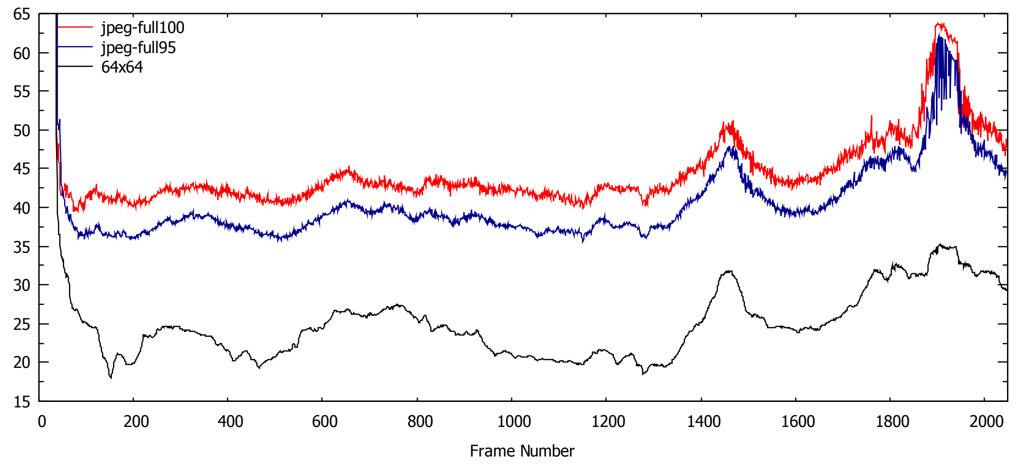


Figure 33: Simulation results for non-streaming algorithms. Frame number is on the X axis. The rendering quality of the frame (PSNR in dB) is on the Y axis. Higher values are better.

If the same JPEG compression technique is used with a JPEG encoding quality value of 95, a comparison of the original data yields a PSNR of 53.7115dB, using a 746550 byte (729KB) approximate representation of the terrain. The result of this simulation is graphed as *jpeg-full95* in Figure 33. As expected, the more compact representation of the terrain data yields a simulation that performs slightly worse than *jpeg-full100*, but more crucially, yields a simulation trace that performs with a PSNR quality metric of no less than 35 dB, which is nearly human indistinguishable from a perfect representation. Figure 34 shows a visual representation of how the image deterioration appears for a 35 dB (PSNR) signal. As can be seen in these figures, the difference is minimal. Artifacts of the lossy compression typically manifest themselves at the edges of sharp elevation changes (such as mountains, cliffs, and rocky hill-sides) due to DCT quantization. In JPEG compression, the high-frequency DCT coefficients are quantized to achieve high levels of compression.

For comparison of coding efficiency, using the 2048x2048 Grand Canyon dataset (4096 KB uncompressed) was only compressed to 3533 KB using Second Life's terrain encoding algorithm [sl]. A PSNR difference of 57.9752 dB was measured between the Second-life encoded terrain data, and the original uncompressed data. The Second Life terrain encoding system does not yield a very high compression ratio (1.16:1), even when compared to JPEG at quality level 100 (3.37:1). Despite this compression-rate disparity, it does not seem to give a better representation of the terrain data either (PSNR of 57.9752 dB) compared to JPEG at quality level 100 (PSNR of 59.7080 dB).

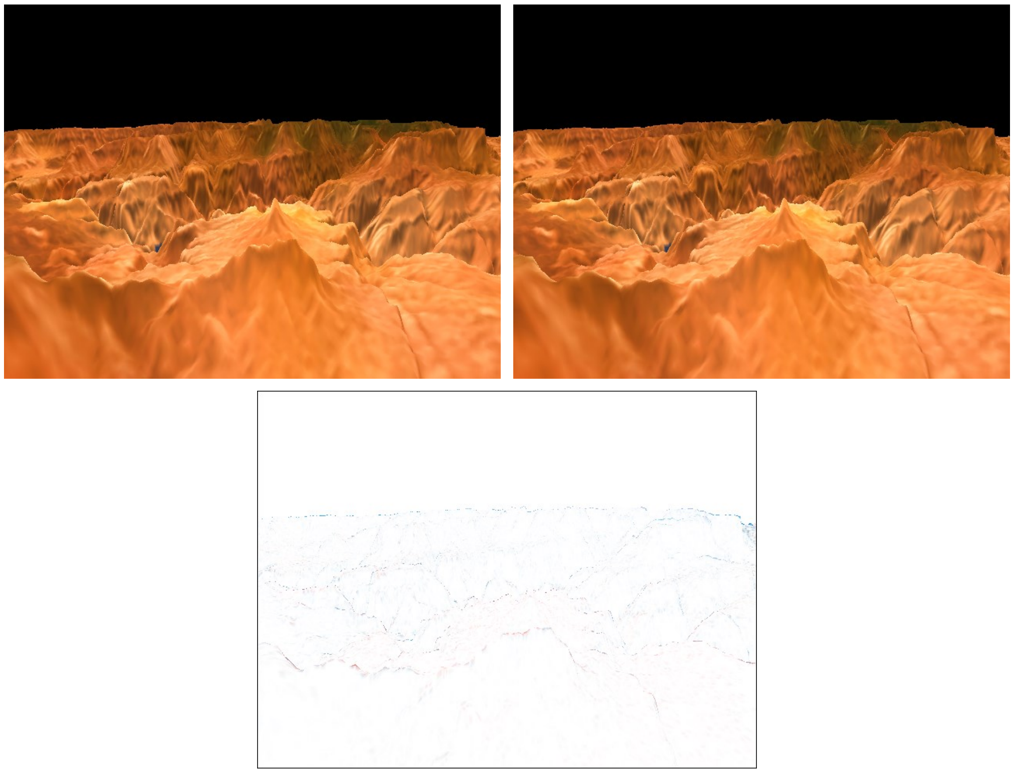


Figure 34: Example showing the degradation in a 35.0dB PSNR rendering. Top Left: Image captured with full-detail terrain representation. Top Right: Reconstruction using compressed approximate terrain representation. Bottom: Pixel difference of both images.

3.4 Exact Representation Terrain Streaming Algorithms

Before any lossy compression algorithm can be considered, it is important to understand the limits of what an exact algorithm solution are. To this end, a lossless terrain streaming algorithm has been constructed, based on the ROAM adaptive meshing algorithm [duchaineau]. While ROAM was originally designed for mesh simplification in order to minimize the number of simultaneous rendered triangles, it can be modified for network streaming by allowing ROAM to only introduce refinement triangles without removing previously rendered high-detail triangles. The goal of this reference

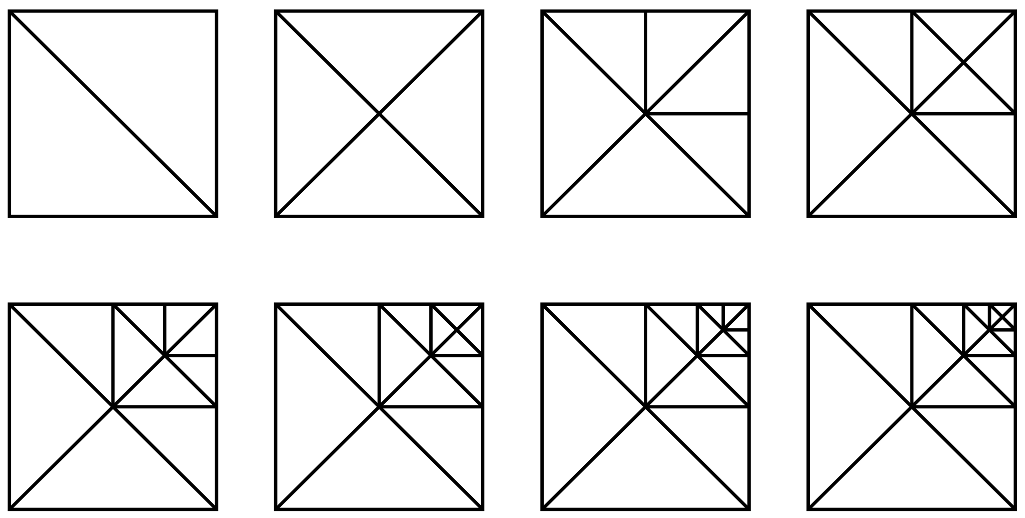


Figure 35: The recursive splitting of triangles in a ROAM terrain patch (overhead view). This example illustrates progressive refinement to add detail to the upper right-hand of the tile. Each vertex represents a rendered height post.

implementation is to represent the result of employing an intelligent streaming algorithm using a verbose, non-lossy data representation.

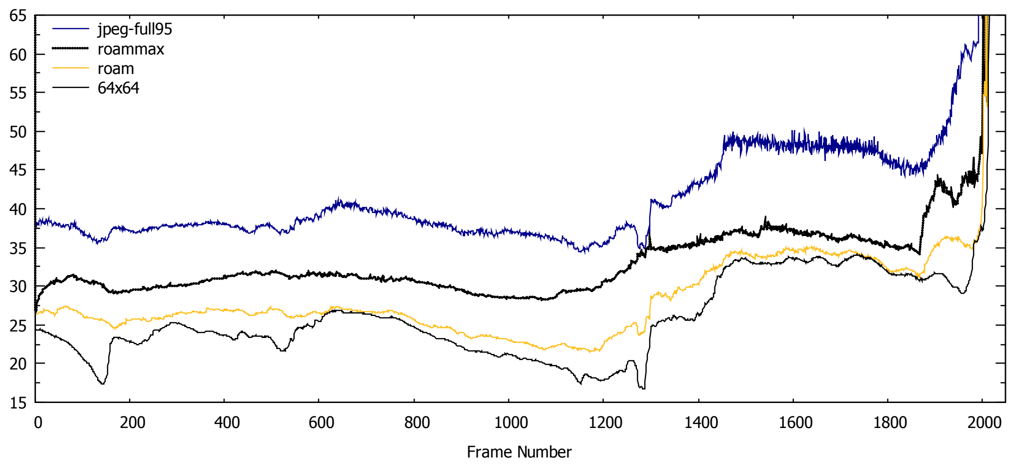
ROAM employs a triangle decimation technique for expressing more detail in a triangle-based mesh – triangles are repeatedly and recursively split into right-angle isosceles triangles to add additional vertices. In the coarsest representation, a ROAM patch is represented by two triangles. To render a more detailed mesh, a triangle may be split into two children triangles, introducing an additional vertex (Figure 35). Triangles are always split in pairs, to prevent the formation of T-junctions – visual cracks in the triangle mesh, formed when two neighboring triangles are rendered at incompatible detail levels.

In practice, a ROAM terrain mesh is represented in memory by a binary tree, with each node representing a triangular area. Each triangle is in turn represented by two smaller triangles that form the descendants of each node. This data structure is referred to as a binary triangle tree (BTT). The BTT is constructed so that travelling down the branches of the tree represents progressive refinement of the terrain mesh, with each step increasing the visual detail that is presented to the user.

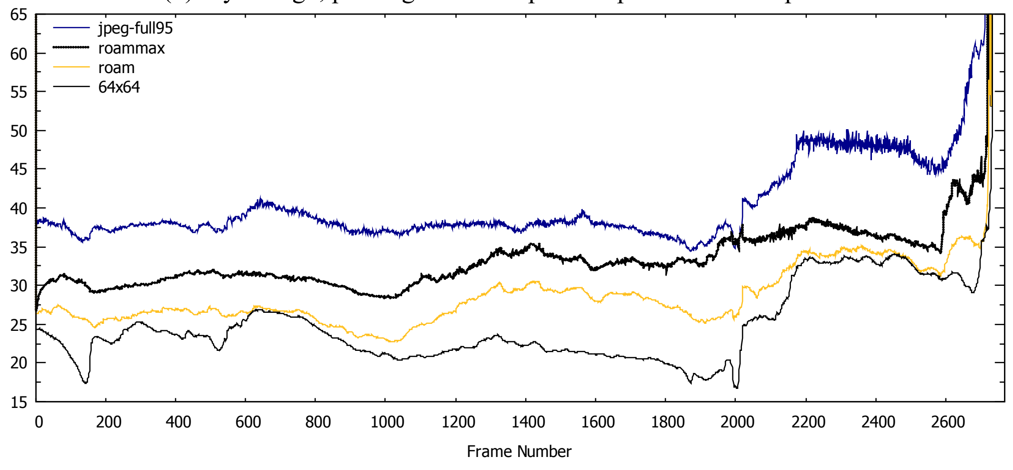
In the implementation presented in this chapter, there are two BTTs representing the ROAM-encoded terrain mesh – one on the server, and one on the client. Initially, the server's BTT will be fully populated with the full terrain geometry, while the client's BTT will contain only the coarsest representation. Over the course of the simulation, the client populates its BTT until the server's entire BTT is transmitted, at which point the entire terrain can be rendered from cache without the need to query the network. The server constructs a BTT node/vertex stream to send to the client, based on the viewer's location and orientation, using a distance-variance metric for vertex prioritization. This is similar to the way standard ROAM implements progressive refinement.

BTT node/vertex streaming priority is calculated by finding the variance in height of all the child vertices and dividing by the distance of the node from the viewer, forming a score for each node in the terrain mesh. This score represents a measure of the visual difference between the current terrain representation, and the fully-detailed terrain. Every node that has not yet been transmitted is placed in a priority queue for streaming to the client. The scores of the nodes that have not yet been transmitted are recalculated on each frame of the simulation and reprioritized, to ensure that the most crucial data is sent to the client.

(1) Flythrough



(2) Flythrough, pausing at the midpoint to perform a 360˚ pan



(3) Flythrough with a continuous 360˚ pan

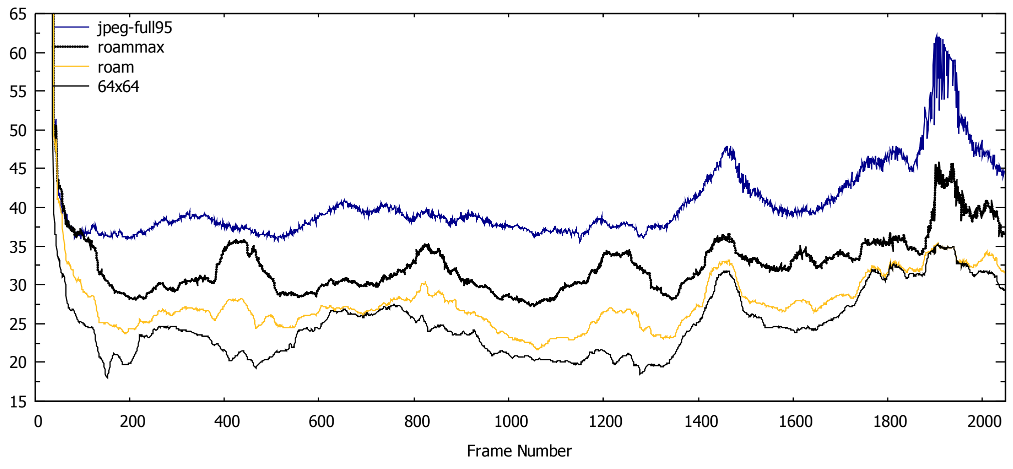


Figure 36: Simulation results for exact representation streaming algorithms. Frame number is on the X axis. The rendering quality of the frame (PSNR in dB). Higher values are better.

It is important to note that this implementation is extremely server resource intensive and impractical to deploy in a real-world system. This implementation serves only to represent a best case exact representation streaming algorithm. Because the server must mirror the state of the binary triangle tree of each client the server is streaming to, this solution demands a significant amount of server resources to implement. This may also present a potential synchronization problem if a client viewer is restarted using a previously cached partially complete BTT.

The performance of the ROAM-based exact representation streaming algorithm is illustrated as *roam* in Figure 36. This simulation counts each vertex as 4-bytes of data (1 byte for height, 3 bytes for XY positional information). This simulation represents the effect of organizing the terrain data in a streaming-friendly manner, without applying any compression. Although the quality of the rendering is easily recognized as imprecise to the human eye, it represents a significant improvement over *64x64*, the non-streaming 4KB reference algorithm.

For comparison, we have also simulated *roammax*, which is the same algorithm, but counts each vertex as only 1 byte of data (Figure 36). This value was chosen in accordance with observed compression factors optimal under optimal conditions [alliez]. This four-fold improvement in compression results in a significantly increased image quality – at times, it is almost impossible to differentiate between the original full-detail rendering and the results of streamed simulations. This suggests that at this level, more bandwidth is incredibly helpful in improving the quality of the experience.

The most significant feature of the ROAM-based algorithm is that it provides vertex-level explicitness, allowing the streaming solution to add vertices/detail where they are needed most. This allows the information flow to be quickly adapted to account for viewer location and orientation changes.

The rendering quality of the streaming algorithms in flythrough #3 tends to oscillate with a period of 360 frames. This is because the viewer rotates through a 360º pan every 360 frames, and so begins viewing terrain that has been progressively refined in the previous rotation cycle.

The ROAM-based streaming techniques exhibit "popping" artifacts – temporal discontinuities formed by the sudden introduction of a new vertex to the terrain mesh. These artifacts are not captured by our PSNR metric, but may prove distracting to the viewer. The visual impact of these artifacts can be lessened by introducing new vertices using a geomorphing technique to smooth the geometric transition between mesh refinement levels [hoppe].

3.5 Approximate Terrain Representation Streaming Algorithms

The motivating observation of this research is that above a certain quality level, human beings lack the ability to perceive changes in data quality. Any data that is sent that exceeds this threshold is wasted. We address this problem by using an approximate representation of the terrain geometry. The landscape (Figure 30) will be represented as a collection of 2-dimensional tiled bitmaps. In this approach, height-fields will be represented as image data and compressed using a greyscale JPEG [jpeg]. Thus, the pixel luminosity in the image will relate linearly to the height at a given location on our map. This representation will efficiently encode terrain data because terrain data is fairly smooth (modulo cliffs). The JPEG representation used in this chapter to represent height-field data is an 8-bit grey-scale JPEG. Thus, each data point (height information) must be expressed as an 8-bit integer value. To allow this 8-bit tiled bitmap to describe arbitrary terrain, each tile can be given a scaling factor (difference between highest and lowest point) and an offset (value of the lowest point).

The entire terrain is divided into smaller, square bitmaps and compressed using JPEG encoding in progressive mode to allow progressive refinement as data is streamed to the client. By representing the terrain as a collection of small tiles, instead of a single large tile, this allows the server to send terrain information in varying levels of detail (or even not at all if it is not visible to the user) for different areas of the landscape.

JPEG encoding is based on a discrete cosine transformation (DCT) which first transforms two-dimensional data in the spatial domain to a frequency domain. The frequency coefficients are quantized, which is why JPEG encoding is very compressible and lossy. The quantizing factors are weighted to give a higher priority to high frequency information over lower frequency information. This is because the human eye perceives more detail in high-frequency information than smooth gradients. This observation is true of both photographic images (such as edges of objects) and terrain features (such as the shapes of mountains and cliffs).

For progressive JPEG encoding, DCT coefficients are grouped into different refinement layers, with the first layers giving the low-frequency coefficients, and adding detail with refinement layers that describe the higher-frequency coefficients. In our implementation, each terrain tile is encoded into six progressive refinement layers (Figure

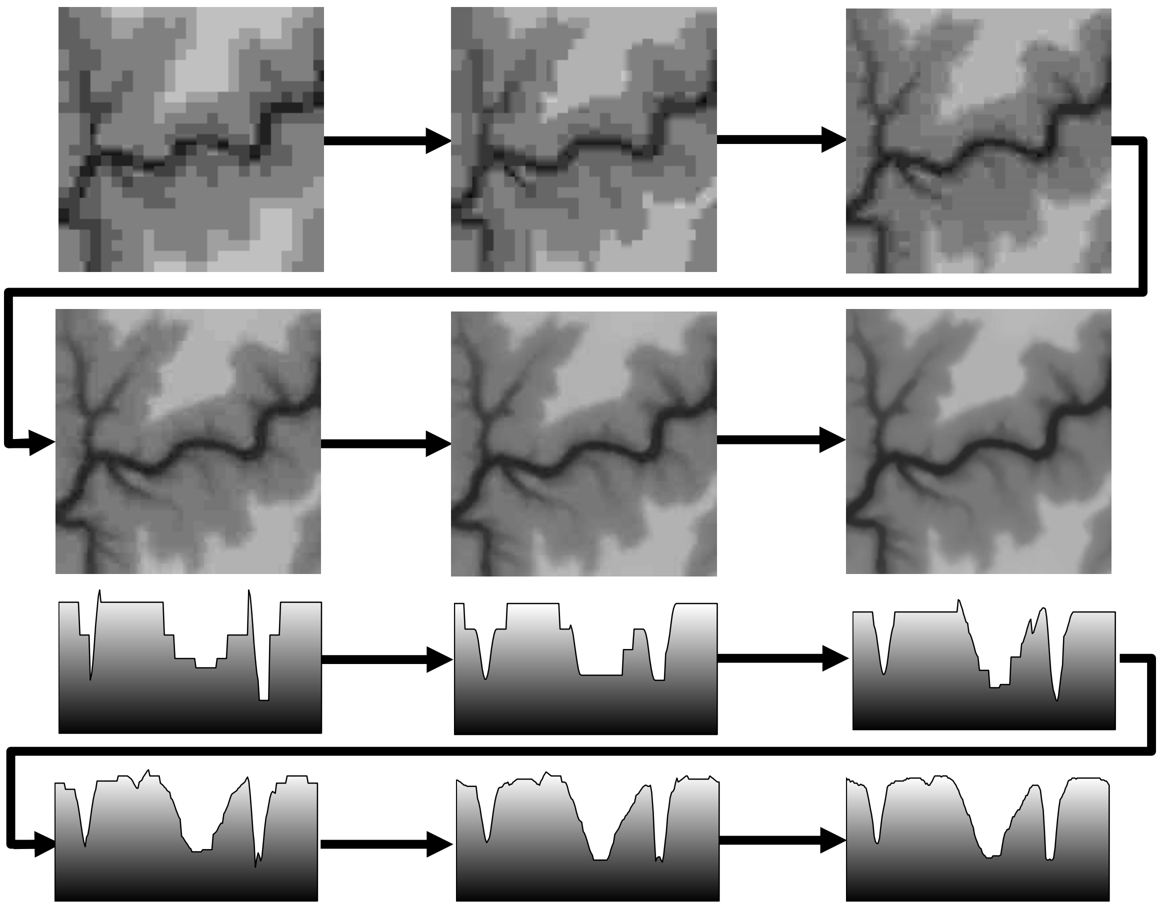


Figure 37: Top: Progressive refinement of a JPEG image. This image represents an actual land geometry tile in the Grand Canyon simulation, with lighter shades representing higher elevation. Bottom: A side view of the center cross section of the same map, undergoing progressive refinement.

37). As more terrain data is retrieved, the detail of the representation of the terrain increases.

Tiles that are outside the viewer's frustum are not transmitted to the client while visible tiles are all transmitted using an equal share of the available bandwidth. By employing JPEG compression, this solution explicitly trades accuracy of height field data to benefit from a more compact representation. It is the goal of this work to find a compact representation of the data that degrades the rendered visualization in a minimal way, so that the difference is not perceivable to the human eye.

3.5.1 Simple Approximate Terrain Representation Approach

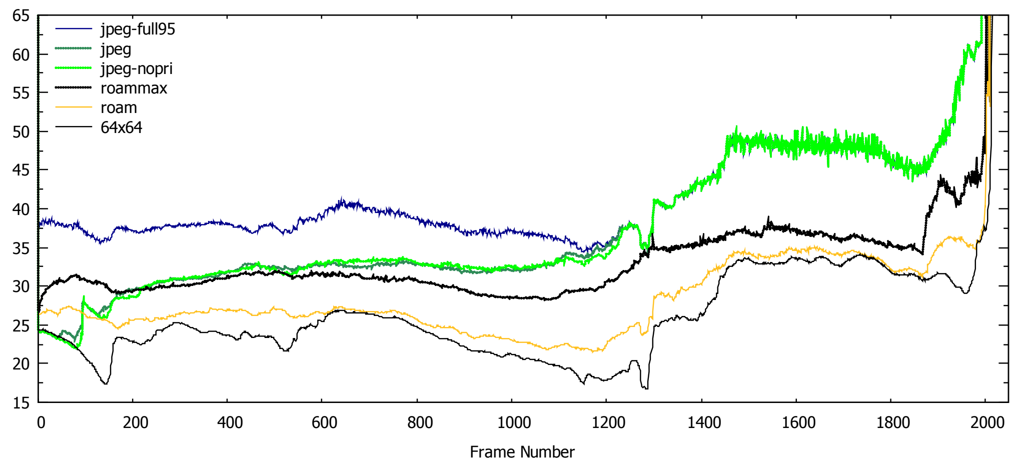
The first attempt at constructing an approximate algorithm for streaming terrains uses a JPEG representation of the terrain information, using a compression quality level of 95. From previous experiments (Section 3.3.1), this is the most compact JPEG representation of the data that maintains a minimum rendered display quality of 35dB (Figure 33), which is near the limits of human perception (Figure 34).

In this implementation, the entire terrain is divided into 64^2 square bitmaps (terrain tiles) and compressed using JPEG encoding in progressive mode to allow progressive terrain refinement as data is streamed to the client (Figure 37). An initial 64x64 point coarse representation is first sent to the viewing client before the streaming algorithm begins. This representation is identical to the data used for the non-streaming *64x64* reference algorithm, which represents the worst-case performance of this approach.

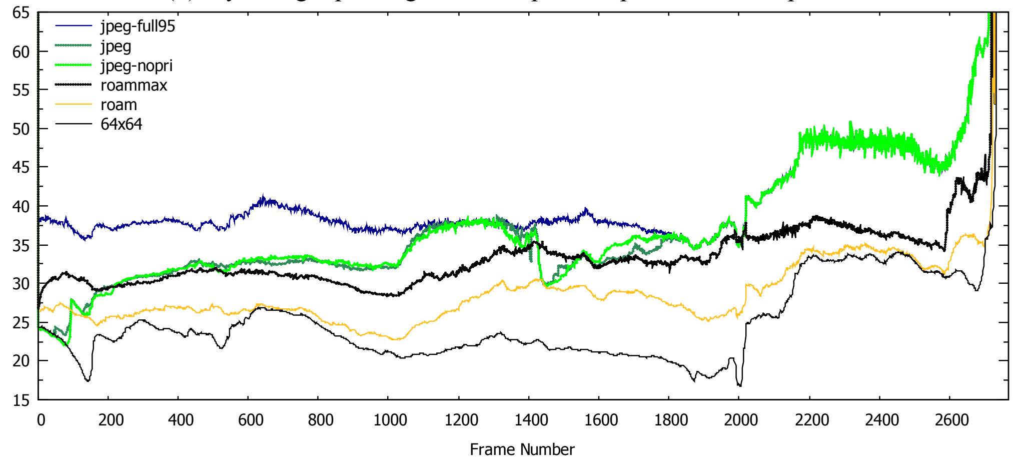
If the streaming algorithm is able to fully download the entirety of the terrain data, the performance becomes identical to the non-streaming *jpeg-full95* reference algorithm introduced in Section 3.3.1. *jpeg-full95* represents the best-case performance of this JPEG-based approach.

In the simplest version of this algorithm, all visible terrain tiles are streamed with equal priority. Tiles that are outside the client's viewing frustum are not downloaded to the client. This approach is termed *jpeg-nopri*, and its simulation results are graphed in Figure 38. In an extension to the *jpeg-nopri* algorithm, visible tiles are prioritized with respect to their distance from the viewer and the size of the compressed tiles:

(1) Flythrough



(2) Flythrough, pausing at the midpoint to perform a 360˚ pan



(3) Flythrough with a continuous 360˚ pan

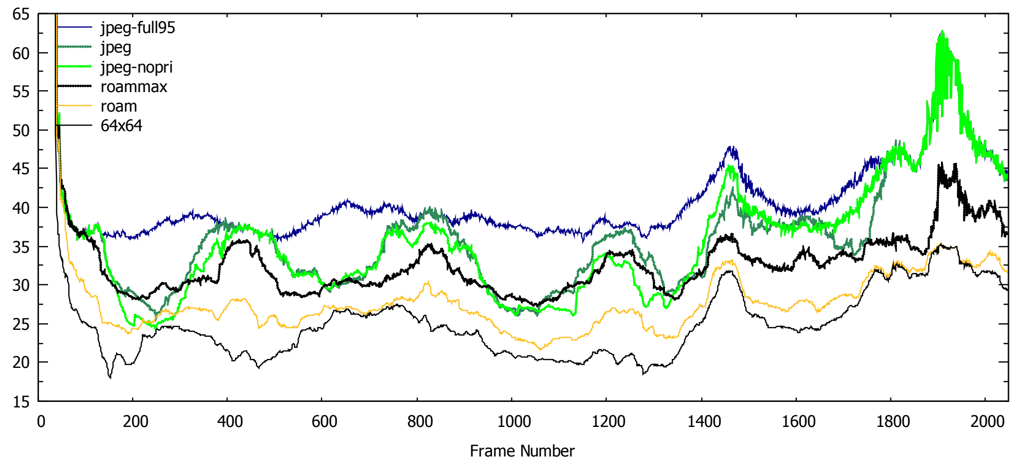


Figure 38: Simulation results for initial approximate representation streaming algorithms. Frame number is on the X axis. The rendering quality of the frame (PSNR in dB). Higher values are better.

$$\mathit{tile\ importance} = \frac{\mathit{data\ size\ of\ tile}}{\mathit{distance\ from\ viewer}} \tag{30}$$

The proximity of the tile to the viewer is used to determine its visual weight, while the size of the compressed tile is used as a coarse metric to determine the tile's geometric complexity. The bandwidth from the server is divided among visible tiles in proportion to the score yielded from Equation 30. The results of this approach are graphed as *jpeg* in Figure 38. This prioritization is similar to the display-list streaming algorithm presented by Pouderoux & Marvie [pouderoux]. However, their simplification model focuses on using a multi-resolution strip mask display structure rather than highly-compressed progressively refined terrain data and lacks the fine-grain streaming properties possessed by the *roam* and *roammax* algorithms.

The server overhead for implementing these streaming solutions is much smaller than the ROAM-based algorithms introduced in Section 3.4. This is because the calculations for determining priority streaming order are coarser-grained and only require a greatly simplified understanding of client state. The server only needs to keep track of the viewer's location and orientation in addition to the number of bytes already streamed to each JPEG tile instead of the state of the client's entire BTT. This drastically reduces the demand on the server's resources, making it an algorithm that is suitable to deploy on production systems.

Both *jpeg* and *jpeg-nopri* perform well and, as expected, are bounded by the best and worst case simulations (*64x64* and *jpeg-full95*). Both approximate algorithms usually out-perform the exact representation algorithms. Although *jpeg-nopri* can do better than *jpeg* when the view frustum mispredicts the future importance of JPEG tiles, the *jpeg* algorithm usually gives slightly better in the worst cases because it prioritizes information closer to the viewer.

The most surprising result of these two simulations is that the performance of *jpeg* and *jpeg-nopri* behave very similarly. Further investigation revealed that the prioritization implementation was weak – the streaming policy attempted to enforce byte-level streaming prioritization fairness on a per-frame timescale. This policy did not allow sufficient freedom for high-priority tiles to receive a significant larger share of available bandwidth, so the prioritization metric was rendered nearly useless. Despite this lack of intelligent streaming, approximate representation techniques always yield a better result than the *roam* simulation, and almost always better results than *roammax* except in 3 key areas:

1) The beginning of flythrough #1 and #2. Due to the lack of a good prioritization mechanism, the JPEG-based algorithms are not able to quickly adapt to the newly initialized viewer and prioritize the transmission of terrain information closer to the viewer. ROAM's vertex-level explicitness allows it to quickly send the most important pieces of terrain data to the client. The superiority of the data prioritization exhibited by the ROAM-based algorithms allowed them to deliver the most relevant data to the client, in a smaller amount of time.
2) At frame 1450 of flythrough #2, the JPEG-based algorithm's render quality drops to the level of *roam*. At this point, a distant mountain range that is not visible for the majority of the simulation rotates into view. Immediately prior to this event, the terrain quality level was the same as *jpeg-full95* – that is to say all visible

tiles were fully downloaded. Investigation into this event revealed that when all visible terrain tiles were completely downloaded, the JPEG-based streaming implementations would arbitrarily choose information to stream. A more appropriate use of bandwidth would have been to begin transmitting terrain information that is important (ie. large mountain ranges) that are not immediately visible to the viewer. The *jpeg* algorithm required almost 2 seconds to recover from this misprediction. The superior data prioritization and fine-level granularity in *roam* and *roammax* allow them to more quickly adapt to the viewer's changing field of view. This prioritization advantage allows the *roammax* streaming algorithm to keep pace with the JPEG compressed stream in some cases, despite using a less efficient data encoding.

3) During flythrough #3, the viewer's camera is constantly being rotated. This is the most detrimental case to the *jpeg* streaming algorithm. Because of the weak prioritization metric it is unable to cope with the constantly changing view. The high JPEG compression rate exaggerates the result of stream prioritization. The results are better than *roammax* when the viewing areas of the terrain that have previously been viewed (due to compression efficiency) but worse than *roam* when the future tile importance is not correctly predicted.

The most significant result is that despite the lack of an intelligent prioritized streaming technique, the JPEG based algorithms usually yield superior results to even the *roammax* approach. This implies that a high compression rate is more important to the visual quality of the simulation than intelligent prioritization of data. This phenomenon will become more pronounced in systems with large network latency, due to less accurate prediction by the ROAM-based prioritization mechanism because the server will not be able to react as quickly to changes in the position and orientation of the viewer. An increased compression rate is effectively increasing the available bandwidth as more data can be sent over any given time interval. However, a less accurate JPEG approximation of the terrain also means that the simulation will not converge on as high of quality of rendering as an uncompressed data stream because of the loss in data/signal quality. The terrain will not be rendered as accurately as when all of the data has been transmitted without loss.

During subjective examination of the rendered output, JPEG “ringing” artifacts are not easily observed – the quality increase in the streaming simulation tends to be fast enough that small inaccuracies are removed before they become too close and apparent to the viewer. However, blocking artifacts from neighboring terrain tiles being rendered at different detail levels can be distracting.

3.5.2 Prioritize Streaming for Approximate Terrain Representation

As an extension to the approximate streaming algorithms presented in Section 3.5.1, this section explores the use of a more involved understanding of the progressive JPEG image format. More specifically, progressive JPEG stores data in multiple refinement layers with increasing quality. The base level refinement layer is equivalent to a JPEG encoded with a low quality setting, with each layer improving the image (terrain data representation) quality. A terrain tile can only be considered of uniform quality when an entire refinement layer has been processed. With this in mind, we can construct a streaming algorithm that calculates a tile’s priority based on a desired refinement level instead of the byte size of the terrain tile. This also simplifies the complexity of our streaming protocol. Instead of streaming many terrain tiles simultaneously, we can construct a protocol that sequentially streams JPEG refinement layers. This will significantly reduce the amount of control information that the terrain streaming protocol must synchronize.

We can also exploit our understanding of how the terrain data is used in order to construct a better streaming algorithm. In this study, the goal is to have more accurate terrain representation to promote a better visual experience from the perspective of a 3D viewer. Details near the viewer are important because closer objects are visibly larger on the viewer's display. Terrain features that are far away from the viewer are disproportionately important if they contribute to the landscape's skyline – if the silhouette of a distant mountain is absent or particularly soft, it can present a noticeable display inconsistency.

The ROAM-based streaming algorithm accounts for these properties by considering the deviation between the rendered terrain and the true terrain from the viewer's orientation and position, and then adding vertices as required. Unfortunately, a similar solution is not possible using a JPEG-based streaming algorithm because vertex-level granularity is not expressible. However, it is possible to apply a coarse approximation of this streaming prioritization.

At each step in the rendering process, the position and orientation of the viewer is known to the terrain streaming server, in addition to the maximum height of the features in each terrain tile. A tile will contribute to the rendered skyline if the angle between the viewer, the tile, and the horizon is greater than the tiles in front of and behind it (Figure

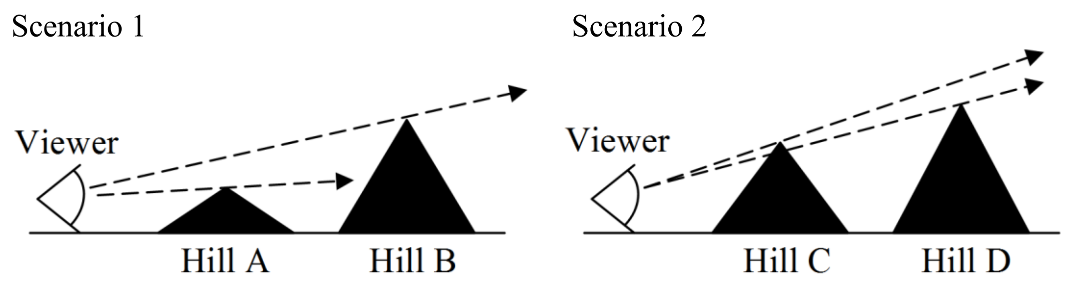


Figure 39: Example: Two different streaming scenarios. In the first scenario, the more distant Hill B is more likely to contribute to the rendered horizon. In the second scenario, the nearby Hill C is more likely to contribute to the rendered horizon.

39). Skylines are especially important to consider, because visually, disparity between the true skyline and the rendered skyline are the easiest to notice by a human viewer, and this inconsistency is well reflected in the PSNR evaluation metric.

Silhouette edges have a disproportionately important contribution to the overall image accuracy because they guide our mental reconstruction of an object's shape. Many level-of-detail (LOD) simplification algorithms recognize this property [luebke]. A prioritization algorithm that favors distant terrain features (terrain tiles) if they contribute to the silhouette edge of the skyline will result in better quality terrain renders. This can be approximated by calculating the angle between the viewer and the highest peak in each terrain tile.

Whenever the viewer's position changes, the angular distance of each tile with respect to the horizon is recalculated, with greater angles representing tiles that are higher on the client's viewscreen. The list of visible tiles is sorted by angular distances to each terrain tile's peak to generate a ranking of tiles that are likely to contribute to the landscape's skyline. The terrain tile with the highest angular height is ranked first.

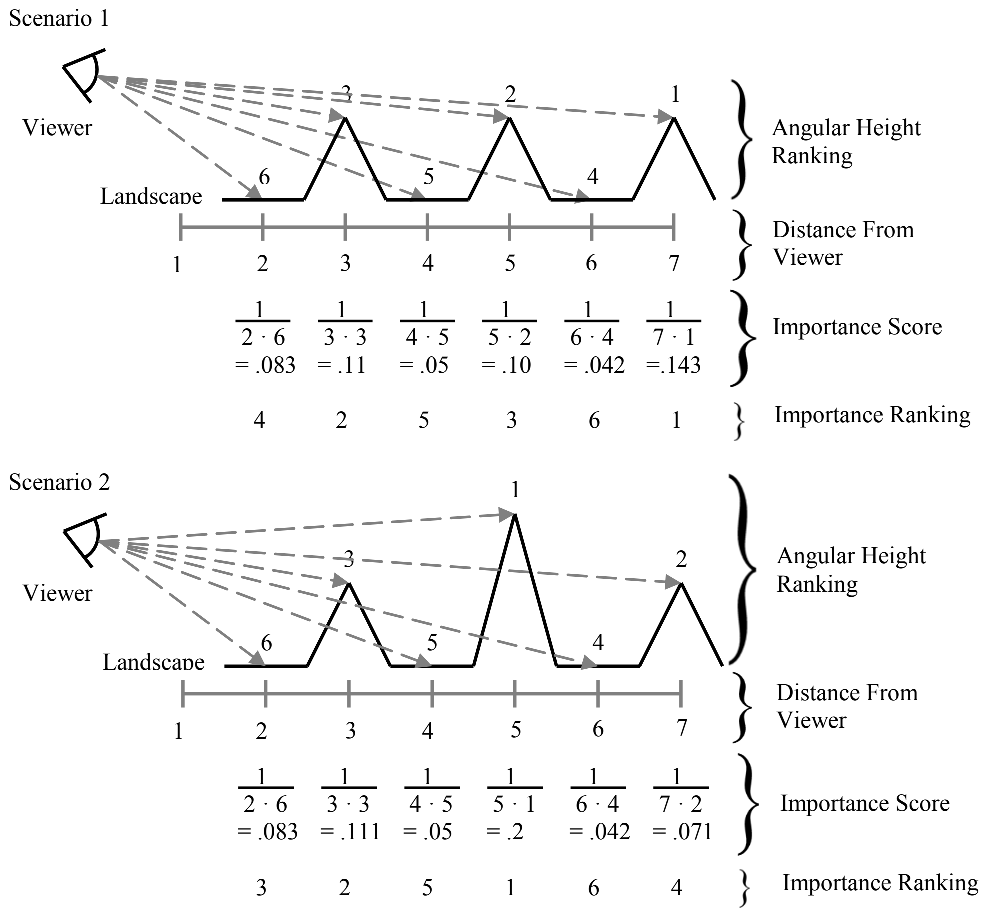


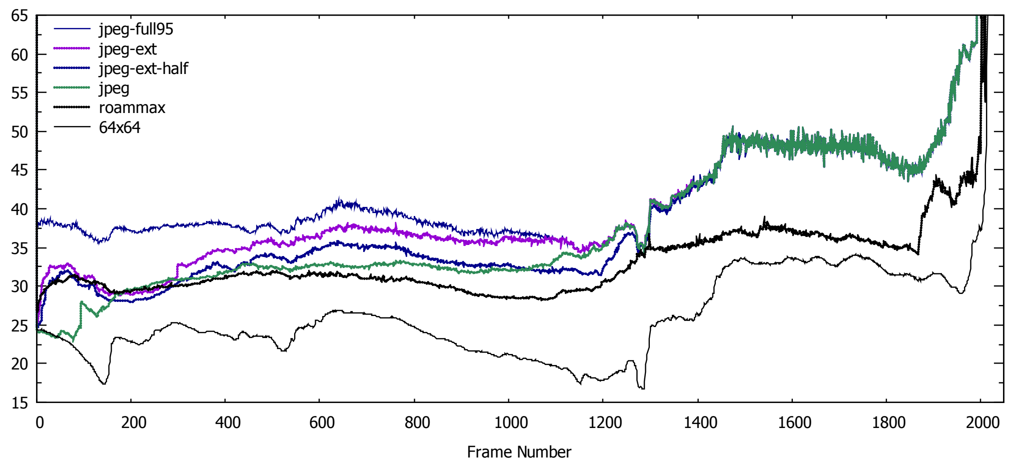
Figure 40: Two scenarios demonstrating the results of the extended priority scoring in jpeg-ext. Note that in this example, we do not take the square root of the priority score because we are only presenting a one-dimensional example.

These two metrics are inverted and multiplied to form a tile importance score:

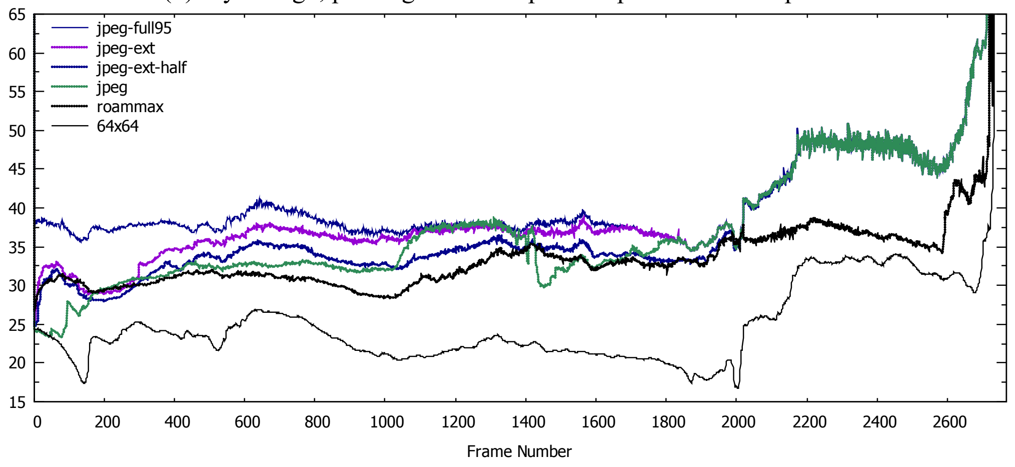
$$tile\ importance = \frac{1}{\sqrt{distance\ from\ viewer\ \times rank\ of\ angle\ to\ horizon}} \tag{31}$$

The square root of the result is taken because number of tiles that must be considered grows roughly quadratically with the scale of the terrain we render (due to the two-dimensional nature of a set of terrain tiles). A simple one-dimensional example is presented in Figure 40.

(1) Flythrough



(2) Flythrough, pausing at the midpoint to perform a 360˚ pan



(3) Flythrough with a continuous 360˚ pan

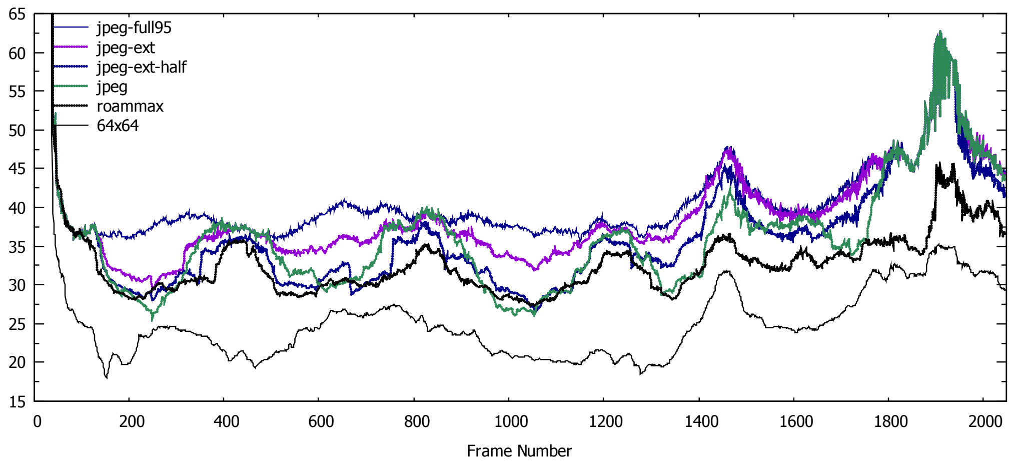


Figure 41: Simulation results for approximate representation with more intelligent prioritized streaming algorithms. Frame number is on the X axis. The rendering quality of the frame (PSNR in dB). Higher values are better.

To apply this understanding to the JPEG-based streaming algorithm, the tile importance score (Equation 31) is scaled between [0, *n*], where *n* is the maximum JPEG refinement level (in this particular implementation, $n = 6$). This normalized score represents the desired progressive JPEG refinement level. To choose a specific tile to stream, the server locates the visible terrain tile that has the greatest desired JPEG refinement level less the previously transmitted JPEG refinement level and transmits the next JPEG refinement layer. If all visible tiles have already been fully transferred, the same metric is applied to tiles outside the viewer's frustum.

An additional advantage of this approach is that it allows the terrain server to be almost completely stateless. If the client is initialized with knowledge of the maximum heights of every tile (4KB uncompressed with this data set), it can independently calculate stream prioritization and simply send the server requests for tile refinement layers – the server does not need to know anything about the state of the viewer. This solution is significantly more scalable, allowing a terrain streaming server to be simpler, and serve more concurrent clients.

The result of the JPEG-based streaming solution with the extended prioritization scoring metric is graphed as *jpeg-ext* in Figure 41. The results of using a more sophisticated streaming metric are uniformly positive – it is almost as good as the better of *jpeg* and *roammax* in the worst case and significantly better than both in the best case. This effectively addresses each of the three of the weakness of *jpeg* discussed in Section 3.5.1. The streaming prioritization used in *jpeg-ext* combines the quick-reacting nature and feature identification of the ROAM-based exact representation algorithms with the high compression coding efficiency of JPEG.

JPEG Refinement Level	**1**	**2**	**3**	**4**	**5**	**6**
Avg. size: JPEG Quality 95	80.325	124.924	56.5547	129.415	16.313	216.521
Std. Dev.: JPEG Quality 95	13.367	56.501	27.272	37.651	0.728	58.592
Avg size: JPEG Quality 100	88.332	124.998	70.049	213.088	16.330	598.873
Std. Dev: JPEG Quality 100	14.109	56.449	45.147	89.085	0.744	109.639

Table 5: Sizes (in bytes) of JPEG compressed terrain tiles.

To further demonstrate the effectiveness of the algorithm, we introduce *jpeg-ext-half* (Figure 41). This simulation uses the same algorithm and data as *jpeg-ext*, but penalizes it by only allowing it use of half the bandwidth available to the other streaming simulations. Surprisingly, it is still able to keep pace with both *jpeg* and *roammax* in most cases. Even with just half the bandwidth, it avoids the worst-case behavior characteristics exhibited by *jpeg*, such as *jpeg*'s unreactiveness in flythrough #2 when the distant mountain range rotates into view (as discussed in Section 3.5.1). *jpeg*'s results are superior to *jpeg-ext-half*'s performance only when its bandwidth advantage allows it to converge to a fully downloaded state first.

3.5.3 Understanding the Characteristics of JPEG Representation

As revealed in Section 3.5.1, the efficiency of JPEG encoding is of particular importance to this work as it has allowed the even approximate terrain streaming algorithms with weak data prioritization to provide a better remote visualization experience than exact terrain streaming algorithms in most cases.

Table 5 charts the average size of each refinement layer across all of JPEG encode tiles. On average, JPEG compressed tiles using a quality level of 95 are 594.053 bytes and JPEG compressed tiles using the maximum quality level 100 are 1081.669 bytes. The reported sizes discard headers and other information which is identical in every

Compression	Size	PSNR Quality
64x64 Representation	4KB	23.7015 dB
SL/Opensim	3533 KB	57.9752 dB
Progressive JPEG level 1	193 KB	40.3551 dB
Progressive JPEG level 2	318 KB	47.9664 dB
Progressive JPEG level 3	388 KB	49.984 dB
Progressive JPEG level 4	601 KB	53.258 dB
Progressive JPEG level 5	617 KB	53.3564 dB
Progressive JPEG level 6	1216 KB	59.708 dB

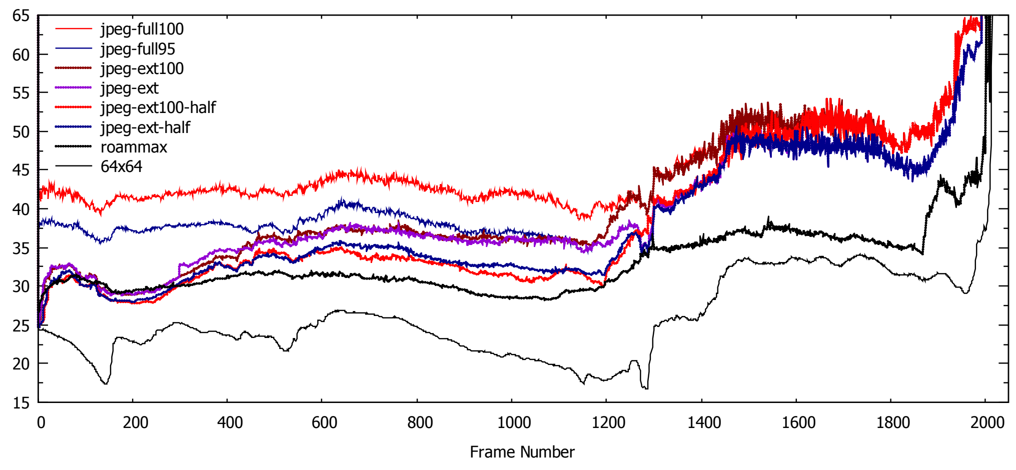
Table 6: Compressed representation of terrain, as well as the PSNR comparison with the original, raw data. JPEG-compressed representations use a quality level of 100.

compressed tile (135 bytes). Any streaming algorithm need not burden itself with transmitting duplicate data. For reference, an uncompressed tile is 4096 bytes.

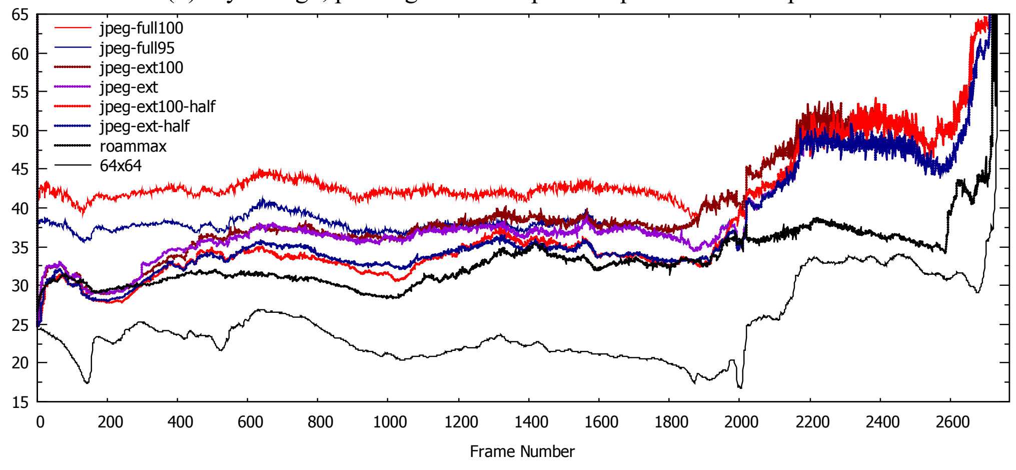
Although JPEG compression at quality level 100 is not as compact as JPEG compression at quality level 95, more than half of its data size footprint is represented only in refinement level 6, the highest level of progressive refinement. The average JPEG compressed tile at quality level 100 is only 512.797 bytes at refinement level 5, which is noticeably smaller than the average fully refined JPEG compressed tile at quality level 95 (594.053 bytes).

Table 6 charts the visual quality of JPEG compression at the different refinement layers, using a JPEG quality level of 100. At JPEG refinement layer 5, the quality of the representation is 53.3564 dB, which is nearly indistinguishable from the fully refined JPEG compressed map using a quality level of 95 (53.7115dB). These observations indicate that a progressive JPEG using a quality level of 100 is as efficient at encoding data as a progressive JPEG using a quality level of 95, and can deliver a superior final representation of the data.

(1) Flythrough



(2) Flythrough, pausing at the midpoint to perform a 360˚ pan



(3) Flythrough with a continuous 360˚ pan

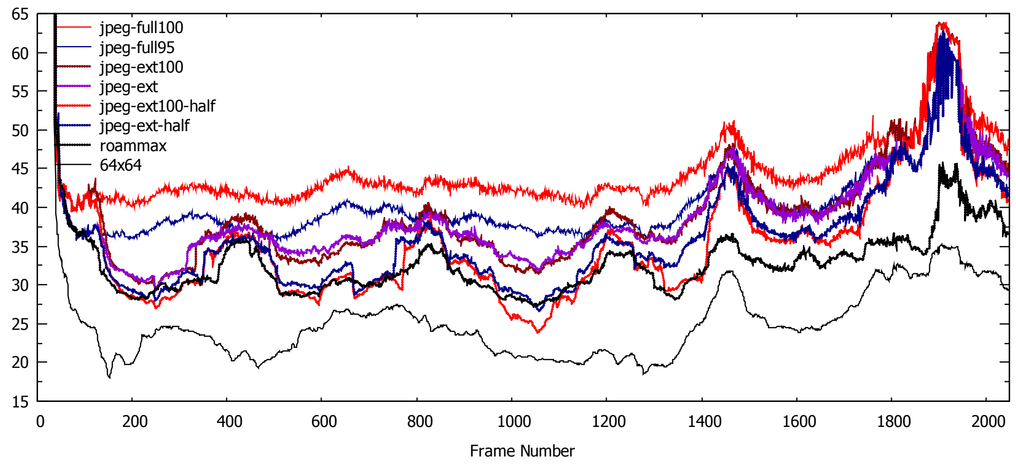


Figure 42: Simulation results for high-quality approximate representation streaming algorithms with intelligent streaming. Frame number is on the X axis. The rendering quality of the frame (PSNR in dB). Higher values are better.

With this new insight, we re-examine the previous decision to use JPEG compression at quality level 95, instead of its maximum quality level 100 encoding. The *jpeg-ext* algorithm presented in Section 3.5.2 (JPEG encoded terrain tile streaming with extended priority scoring) is modified to use a JPEG compressed tiles with a quality level of 100, termed *jpeg-ext100*. The results are graphed in Figure 42. A simulation of this algorithm using half as much bandwidth was also run, and the results are graphed as *jpeg-ext100-half*.

For the most part, the *jpeg-ext100* and *jpeg-ext* (and their half-bandwidth counterparts *jpeg-ext100-half* and *jpeg-ext-half*) perform very similarly. The most significant performance deviation occurs when *jpeg-ext* (and *jpeg-ext-half*) has downloaded all relevant data, and their rendering quality is bound by the maximum detail in the final JPEG refinement layer. At this point, *jpeg-ext100* (and *jpeg-ext100-half*) can continue streaming more terrain detail information, providing a superior quality terrain rendering visualization.

3.6 Conclusion

To address the need for online virtual worlds to display landscapes on remote viewing clients, we have proposed a lossy streaming architecture suitable for the streaming of 3-dimensional terrain data. In the first stage of the *jpeg-ext100* algorithm, we reduce the height-field data representing the virtual terrain to a bitmap image, using JPEG to compress it to approximately one quarter of the original size. The data is compressed in a lossy manner that is indistinguishable by a human eye from the uncompressed version. By exploiting the structure of JPEG compression, our knowledge of the shape of the terrain, and the viewer's position and orientation, we are able to design a nearly stateless streaming algorithm that prioritizes the portions of the terrain data that are more relevant to the viewer. After initially connecting to the virtual world and beginning streaming, the quality of the remote rendering rapidly improves and becomes nearly indistinguishable from an uncompressed, pre-downloaded terrain after approximately 20 seconds in our experimental scenarios. When compared to *roammax*, a reference exact-representation algorithm with impractical computing requirements and vertex-level expressivity, *jpeg-ext100* yields a 5-15 dB PSNR improvement in the quality of the rendering for the majority of the experimental scenarios after the initial 20 second start-up. During the initialization phase, *roammax* and *jpeg-ext100* yield similar results in terms of rendered image quality. Because *roammax* has vertex-level granularity, it is effective in adding detail to where it is needed most. However, as the simulation progresses, the more compact data representation used by *jpeg-ext100* begins to dominate in terms of rendered level of detail.

The experimental results here demonstrate the importance of achieving a high data compression ratio in order to provide high-quality streaming terrain. This further underscores the importance of adopting lossy encoding techniques, which can yield much higher compression rates than the non-lossy approaches. By exploiting the structure of the compressed data, *jpeg-ext100* intelligently prioritizes data delivery while requiring very little state information to be stored at the server, greatly simplifying server design and server computing load, allowing *jpeg-ext100* to be an excellent candidate for deployment in virtual-world streaming systems.

3.7 Future Work

The techniques proposed here are suitable for deployment in its current state as a streaming only solution. However, because client processing power is not unlimited, it must be used in conjunction with a client-side LOD simplification algorithm for real-time display. There may be a way to design a client-side LOD simplification algorithm that applies understanding of the level of representational accuracy expressed in the underlying compressed streaming data in order to design an efficient client-side rendering algorithm.

The JPEG-based algorithms presented here only allows terrain to be presented in as many levels of detail as is practical with JPEG compression (6-8 levels in practise) and assumes that the highest level of detail supported in the terrain is uniform. For some styles of virtual world, it may be more appropriate to use multi-level hierarchy of terrain tiles, or even using an irregular mesh instead of a regular grid of height-posts to represent the terrain.

At present, the material and lighting information is not considered. It may be possible to design a way to send this information that combines this information with the terrain geometry for a more efficient, unified streaming algorithm. Combining terrain streaming with general 3D model streaming is also an open problem. Objects in the environment (such as buildings) may occlude portions of the terrain, making it less important to stream those parts of the terrain.

Chapter 4 Distributed Simulation Architecture for Virtual Worlds

Virtual reality systems [active][croquet][sl][opensim] have risen in popularity with readily available high-speed networking and affordable consumer computer graphics processing hardware. One significant problem of designing 3D virtual worlds such as a metaverse (a dynamic and persistent virtual online shared space where users interact through digital avatars) is developing a scalable architecture that can manage millions of simultaneous users in an interactive 3D environment with dynamic content. This chapter presents XPU (Extremely Partitioned Universe), a hierarchical client-server architecture for developing highly scalable Metaverses. This design addresses the problem of dynamically partitioning the world to manage network and computing resources. Unlike massively multiplayer online games (MMOGs) which strive to simplify their universe by optimizing their implementation for a specific game environment, metaverses are characterized by a generalized approach to the problem of 3D worlds. These designs seek to promote unconstrained user-generated content for services such as social networking, collaboration, scientific experimentation, e-commerce, marketing, gaming, education and training. The unconstrained nature of metaverses requires a different style of architecture to manage computing and networking resources than regular online games which are fixed in content complexity and distribution. Because the world is dynamic and constantly changing, especially as users move in the virtual space, the infrastructure cannot pre-allocate computing resources to service different regions of the world as is the style in all currently deployed virtual worlds [chen3]. The infrastructure must be constantly adaptive, allocating computing resources when and where they are needed in the simulation. In practice, a large portion of the virtual space is unused for the majority of the time, while others have disproportionately high amount user traffic, which makes intelligently adaptive simulation management essential [varvello].

XPU is an architecture designed with the goal of managing 3D virtual space and content in a client-server situation. The following are the design requirements for this architecture:

- The system must follow a client-server architecture. In this way, the service provider can guarantee security, availability and adequate resource provisioning.
- The available computing power is large, but no single computer can support the entire computing load. The state of the world is so vast and dynamic no single entity can even have sufficient global knowledge of the world manage computing resources to address the load balancing problem.
- The virtual environment is a free-form universe and cannot make strong assumptions about the distribution of content in the metaverse, which will be driven by dynamic user activity. The population is large and unpredictable, and the architecture must accommodate flash crowds as well as vast unused or unpopulated spaces.

It is the goal of XPU to be an architecture for metaverse-like entities and to be a foundation for all types of MMO virtual simulations including online gaming and 3D social networks.

4.1 Related Work

There are many examples of massively multiplayer virtual spaces that each employ distinct solutions to the problem of managing vast virtual spaces that need to service a high number of simultaneous clients.

In MMOGs, sharding is a popular approach to broadly partition the user base into disjoint copies of the world. In this model, replication is easy because users belonging to one shard cannot interact with users in other shards [uo][wow]. Load balancing is accomplished by restricting the number of simultaneous users in each replicated shard. In these environments, only a minimal amount of functionality is delegated to the server to simplify their operation, allowing them to accommodate a large number of simultaneous users. For example, generalized physics and dynamic content are usually omitted.

Croquet [croquet] is a decentralized approach to the problem of virtual spaces relying on a peer-to-peer synchronization protocol to distribute the contents of the virtual space. A single croquet instance can become congested with many simultaneous users since there is no mechanism to subdivide existing space.

Active Worlds [active] is another Metaverse-like virtual world that allows dynamic content creation, including a simplified scripting interface. The Active World universe hosts hundreds of worlds which can be traversed by users, where each world is hosted on a single server. This architecture has no mechanism to allow a world to grow beyond a single server's ability to manage the world's resources.

Second Life [sl][kumar][rosedale] and its open-source counterpart OpenSimulator [opensim] are metaverse-like worlds that allow users to explore and create dynamic content in a three-dimensional space. This space is partitioned into square 256x256m regions, each managed by a separate region simulator (sim) process. Each sim is tied to a specific region of land and cannot be repartitioned to react to a changing workload. This is the primary reason that scaling is such a difficult problem in this architecture. Larger spaces are created by placing sims adjacent to one another. Shards or instancing is not supported.

Several dynamic load balancing algorithms for virtual worlds based on spatial subdivision have been proposed. Since the optimal solution for load balancing is NP-complete, it is necessary to devise a more practical approach [liao]. Different topologies of fixed grid spatial subdivision strategies have been explored, such as triangular, square, hexagonal and brickworks [presetya]. These systems are not as scalable as spatial subdivision approaches using hierarchical grids. Either dynamic resource allocation is not present, or it involves moving server processes around so that unloaded servers can time-share a single CPU. The frameworks that do dynamic server allocation first divide the world into regular-shaped cells (squares and hexagons) [chen3] [ahmed]. These cells are moved between servers to perform dynamic load balancing. These approaches are inherently not scale-free as a single overloaded cell cannot be repeatedly subdivided until it only contains a managable workload. It is also difficult to add new servers to overloaded areas, because the algorithms are designed to shed load to neighbors, which themselves may be overloaded.

Heirarchical subdivision using binary region splitting has been attempted using Opensim and Sirikata as virtual world test platforms [liu][cheslack]. The analysis for the work presented by Liu et al. focus on workload completion times for measuring the effectiveness of the approach. The performance analysis for Sirikata focused on packet rate and server discovery latency. The approach in the Sirikata platform uses the heirarchical structure to perform visibility estimation using solid angle queries to simplify the streaming load and for message routing. The heirarchical structure is also used to perform view aggregation and model simplification of larger subregions for graphical streaming.

ALVIC approaches Metaverse design by using quad-tree subdivision for partitioning logic servers and employing many proxy servers to hide the network topology from clients [quax]. This work focuses on streaming services to manage clients and does not consider the computing load required to manage the virtual world simulation itself.

Another spatial subdivision approach based on Voronoi partitions has been explored [hu]. This approach reacts to increasing load by introducing server nodes near high activity areas and relying on a Voronoi partition to allocation virtual space to a node. The number of users assigned to a single region cannot be easily controlled using this technique because the load balancing mechanism can only seek to reduce the amount of server load by assigning more servers to an area, without directly considering the distribution of users. In this system, the shape of subdivided regions is very irregular, so the number of regions meeting at a single point is unbounded and can lead to more complex synchronization issues [liu2]. The region shapes formed by Voronoi partitioning can degenerate into long, thin wedge shapes, and the borders between regions move drastically as two nearby Voronoi control points pass by each other. An advantage to Voronoi partitioning is that it is scale-free and is able to adapt to any granularity of simulation. This analysis presented in this work focuses on servicing client traffic rather than the operational cost of running the simulation.

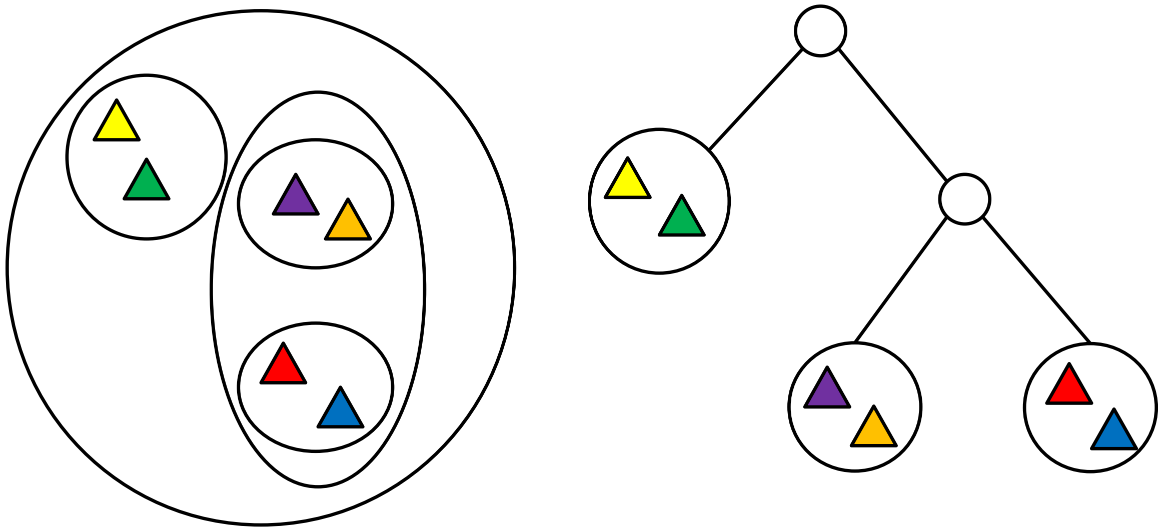


Figure 43: An example of a two-dimensional hierarchical bounding volume. Triangles represent objects, circles represent bounding volumes. The virtual space is represented on the left, and the associated heirarchy is represented on the right.

The Project Darkstar (Sun Gamer Server Technology framework) approach to accommodating massive world state avoids spatial subdivision in favor of storing object and world state in a massive database [darkstar]. Actions on objects are performed through the database. While this approach allows additional processing power to be added easily, it discards any sense of spatial locality that makes the processing of virtual worlds more efficient.

4.2 Approach to Distributed Systems for Virtual World

To manage the allocation of dynamic objects in virtual space, XPU borrows fundamental tree data structures from computer graphics. Modern ray-tracers rely on acceleration structures to manage scene and world data to minimize computationally expensive collision and lighting calculations. One classic approach to this problem is to divide space into hierarchical bounding volumes (HBV) [rubin] (Figure 43). In this approach, the 3D space is divided into hierarchies and arranged in a tree structure. Child nodes represent space encompassed by the parent, with leaves being atomic renderable objects such as triangles and spheres. *kd*-trees are a more restrictive type of spatial partitioning, only allowing partitioning planes to subdivide space perpendicular to the canonical 3-space axis, resulting in a binary space partitioning (BSP) tree. This data structure is successfully used in modern ray-tracing algorithms [reshetov].

The core design motivation of XPU is the assumption that no single computer has enough resources to manage the entire Metaverse simulation. XPU provides a convenient load splitting and management mechanism to distribute computation over a set of servers. At the core of the XPU architecture is the XPU tree. The XPU tree is very similar to an HBV tree. The most significant difference between the XPU and the HBV tree is that leaves in an XPU tree represent virtual regions instead of objects. Each leaf node in the tree is a region managed by a separate server process. Just as in all HBVs, parent nodes must completely encompass the space occupied by child nodes. The root node in the XPU tree represents the entire virtual space, with world simulation being handled at the leaf nodes of the tree, processed by region simulators (sims).

For this discussion, a sim is a single server computer. To distribute the workload of managing the XPU world, each sim can divide its managed space in two and delegate responsibility of managing a sub-region of its space to a new sim, allocated from a pool of servers. Just as in a *kd*-tree, the space managed by the child nodes is expressed by a partition plane, aligned perpendicularly to either the x, y or z axis. The left child is responsible for managing all objects on one side of the first partition plane while the right child is responsible for managing all objects to the opposite side of the partition plane.

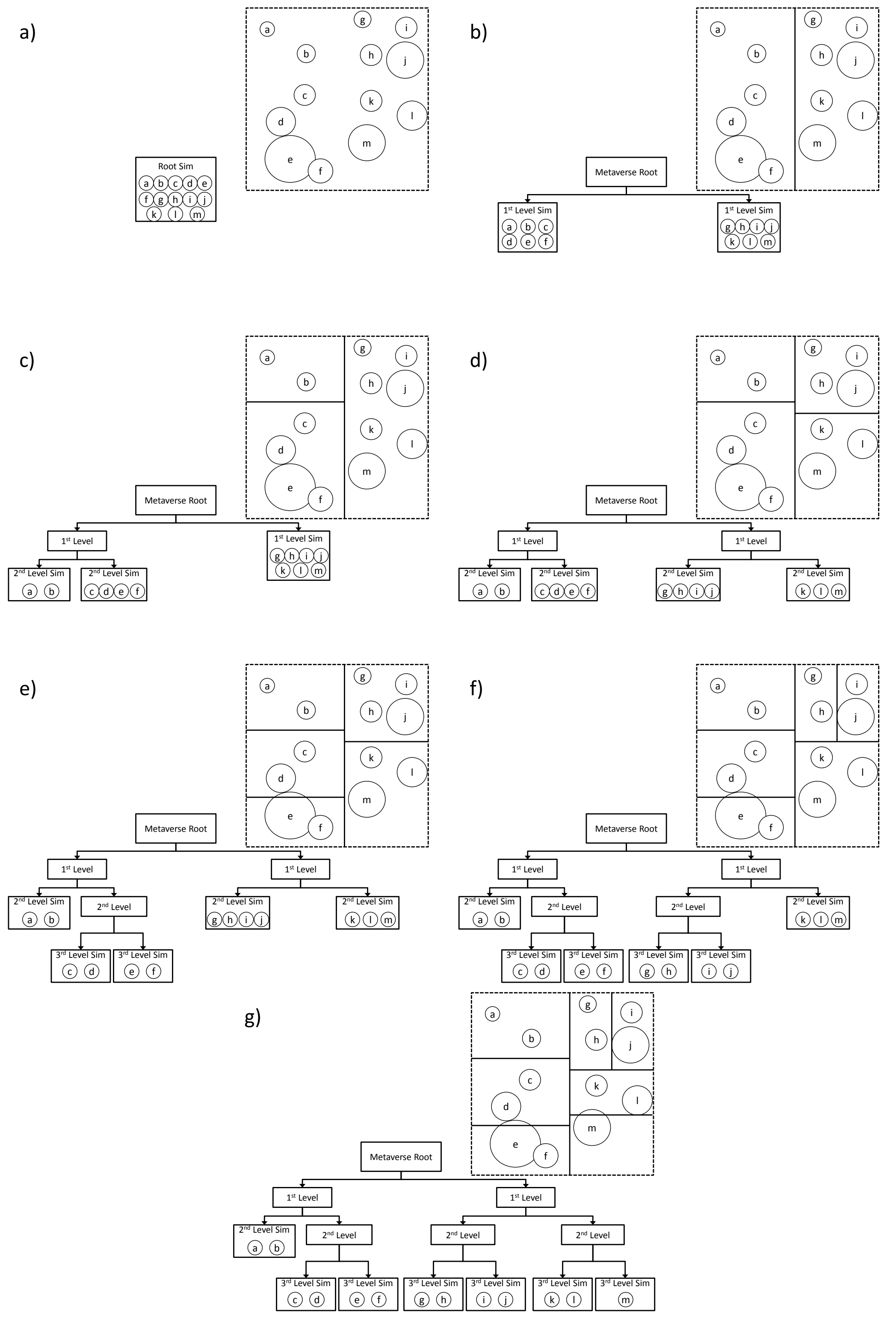


Figure 44: Recursive spatial subdivision of a virtual world (upper right) resulting in a heirarchical topology (lower left). Users and dynamic objects in the world are represented with circles.

Figure 44 illustrates the recursive construction of an XPU graph in a 2 dimensional Cartesian space. The simulation begins with all users and dynamic objects in the world being simulated by a single server. The world repeatedly subdivides until every sim manages only the objects it has sufficient computing resources to accommodate.

4.2.1 The kd-tree Structure for Virtual World Partitioning

The *kd*-tree is a useful structure for spatial subdivision for several reasons. First, because of its hierarchical nature, it naturally adapts to varied population size and density. This divide-and-conquer approach repeatedly subdivides space until an appropriate object density is selected. By selecting the locations of the partitions between cells, the number of objects assigned to each region and server can be controlled with high precision.

Each region is partitioned using axis-aligned boundaries, so a rectangular region shape is always formed. Rectangular regions are useful because they are convex shapes, which have more desirable properties when used for spatial subdivision in virtual world contexts. These properties will be explored in more depth in Section 4.6. This can be contrasted with other types of spatial partitioning which can give wedge shapes, non-convex shapes, and even complex shapes.

kd-tree spatial partitioning limits the number of regions that meet at a point to a maximum of four. It is undesirable to have many regions meeting at a single point in virtual world spatial partitioning because a moving object crossing over this point may have to undergo multiple boundary crossing / server migration events to reach its destination.

It is also possible to take advantage of the hierarchical structure of the *kd*-tree to use as infrastructure for message routing and graphical LOD simplification [cheslack].

4.3 XPU Load Balancing

The most significant motivation to XPU design is the need to divide and distribute processing load of a metaverse over many servers. A property of XPU volumes inherited from its HBV-derived structure is that all objects will be fully enclosed by a bounding sub-volume. This is an important property because it allows the processing of objects to be assigned to a hierarchy of logical regions. A leaf node in the XPU tree represents a region simulator (a single server computer) that is responsible for the management and processing of objects in its enclosing volume.

The two most significant operations in managing XPU systems are node splitting and joining. When a region simulator is overwhelmed by an implementation-specific definition of load, it can choose to split its workload between two child sims (Figure 44). For this operation, the XPU system will assign a new simulator (from a pool of idle servers) to the task, and delegate a sub-region of the Metaverse to the newly allocated sim to manage. The converse operation is simpler – when two sibling leaf simulators have a sufficiently small combined workload, one sim synchronizes its state with its neighbor which takes over the management of their combined virtual world volumes. The now vacated child sim can rejoin the pool of idle simulators. In this manner, the XPU tree is constantly balancing the simulation load of the virtual world over a cluster of servers.

The goal of the XPU management system is to maximize the performance of the interactive virtual world, given a workload and realistic computing constraints. A sim can only accommodate a finite number of users and dynamic objects before it cannot guarantee real-time performance. When two users in different simulators interact across a region boundary, the two sims must communicate over the network, which incurs a network cost. When a dynamic object traverses from a region managed by one sim to another region, the management and processing of the object is transferred to a new sim. Its information must be synchronized and marshalled across the network, which again, incurs a cost.

The challenge of this work is to develop an algorithm that will efficiently allocate computing resources to manage the simulation in a way that minimizes the computing and networking cost of supporting the virtual world. An unconstrained virtual world that supports an unbounded number of users can easily grow to a prohibitively large scale so that no single computer or entity can handle the task of managing computing resources for the entire world. It is for this reason that a distributed algorithm must be developed, so that this problem can be approached without the requiring knowledge of the entire world state.

4.4 XPU Simulation Workload

To evaluate the performance of the XPU system, a simulation workload representing activity in a virtual world is required to compare the efficacy of different approaches to virtual world management. For the purposes of this evaluation, the world is represented by a square region, with users and dynamic objects moving around in this space. In this simulation, users and dynamic objects are modelled identically, and are interchangeable with respect to overall virtual world modelling and performance evaluation.

Currently no completely freeform virtual world exists, so it is not possible to use a real-world trace of user activity in a deployed virtual world. All existing traces of user behavior in virtual worlds are constrained by the architectural limitations of the system which XPU seeks to remove. Because of these reasons, a synthetic workload must be constructed to evaluate the performance of the XPU system. To model an expansive virtual world, we begin by assuming that the world can be contained by a large square region and we model the movement patterns of objects over 100000 time steps.

The synthetic virtual world simulation workload is motivated by the following observations:

- Users move around in the world.
- Users and content tend to cluster together, rather than be evenly distributed throughout the world. Groups of users attract new users and have "flocking" tendencies.
- The session times for users follow a heavy-tailed distribution [chang]. Peak load varies drastically from minimum load.

The first two characteristics are reminiscent of an *n*-body simulation (a simulation of celestial bodies moving in space, accelerating due to gravitational interaction). It is for this reason that an *n*-body simulation was chosen as the basis of the synthetic evaluation workload. *n*-body simulations are characterized by moving objects with natural clustering behavior, as groups of objects naturally come together under the influence of gravity. To characterize user session times, new dynamic objects are introduced into the simulation following a Pareto distribution, with time-limited life, also following a Pareto distribution. A Pareto distribution is chosen for its characteristic long tail distribution which matches patterns observed in user session times [chang]. The generator for this distribution is given by:

$$Par(\alpha,\beta) = \left\lfloor \beta \left(\frac{1}{U(0,1)}\right)^{1/\alpha} \right\rfloor \tag{32}$$

where *U(0,1)* gives a random number following a uniform distribution in the interval [0,1). The objects themselves are initialized with a random mass, also following a Pareto distribution. This is to promote natural clustering behavior so that some objects will disproportionately attract other objects. The initial velocity of dynamic objects is given by a Gaussian distribution, using the Box-Muller method

$$Gauss(\mu,\sigma) = \mu + \sigma\sqrt{-2\,ln(s)/s} \tag{33}$$

where $s = z_0^2 + z_1^2$, $z_0 = 2 \cdot U(0,1) - 1$, $z_1 = 2 \cdot U(0,1) - 1$ and $s < 1$.

The parameters for this simulation are chosen to model the population of a moderately sized virtual world. For the purposes of the analysis presented in this chapter, the population of the virtual world we are simulating has an average of roughly 17,000 users. More detailed information on the world population is given in Appendix A.

4.4.1 Simulation Workload Variations

Because different styles of virtual worlds will exhibit different overall user behavior, several synthetic workloads have been constructed to model varying movement patterns of users in virtual worlds. To create points of interest and promote object clustering, we introduce the idea of *fixed attractors*. These are objects with a large mass that do not move under the influence of simulated gravity and have an unlimited lifetime. These fixed attractors exist only to influence the motion of dynamic objects in the simulation using gravitational attraction. Dynamic objects are preferentially initialized near fixed attractors, using a Gaussian distribution to promote clustering behavior near these points of interest. The following are the names and descriptions of these workloads:

- *No Fixed Attractor*: This workload does not contain any fixed attractors. Objects are initially placed in a wide Gaussian distribution centered about the center of the world. This workload exhibits a relatively small amount of clustering behavior, with objects modestly preferring to gather near the center of the world, to larger dynamic objects, and to other clusters of objects.
- *Single Attractor*: This workload contains a single fixed attractor placed at the center of the world. Dynamic objects are initially placed in a tight Gaussian distribution centered about the center of the world. This workload is characterized by dynamic objects strongly preferring to cluster near the center of the world.
- *Single Attractor (Broad Initial Location)*: This workload contains a single fixed attractor placed at the center of the world. Dynamic objects are initially placed in a wide Gaussian distribution centered about the center of the world. This workload is characterized by dynamic objects moderately preferring to cluster near the center of the world.
- *Row-lined Attractors*: In this simulation workload, several uniformly sized fixed attractors are placed regularly in a row along the center of the virtual world. Dynamic objects in this workload tend to move along a line bisecting

the world. This world represents a world where content is designed along a single line or street, similar to the Metaverse described in Neal Stephenson's seminal science fiction novel on virtual worlds, "Snow Crash".

- *2x2 Attractors*: This simulation contains four uniformly sized fixed attractors, placed at the center of the four quadrants of the virtual world. This is analogous to a world where the centers of activity are uniformly distributed throughout the world.
- *Circular Motion Attractor*: This simulation contains a single fixed attractor that gradually traverses the world in a circular path. This workload is analogous to a virtual world where users preferentially flock to a moving center of interest.
- *Random Attractors*: This simulation contains nine fixed attractors of varying sizes, distributed in an initially random pattern around the world. This workload is analogous to virtual worlds where content (and hence locations of user congregation) is placed in an uncoordinated fashion, resulting in an irregularly distributed traffic pattern.

More detailed statistics, graphs and diagrams describing these seven workloads are located in Appendix A. These synthetic workloads will serve as a basis for evaluating virtual world performance in different scenarios.

4.5 Performance Metrics

To evaluate the performance of the XPU system on the various workloads described in Section 4.4, we introduce four metrics to describe the overall performance of this system under those workloads. These metrics quantify the amount of computing resources that are required to manage the simulation, the level of service provided by the simulators, and the amount of network communication and inter-server synchronization required to support the simulation. For the experimental results in this chapter, simulation results are only recorded after the first 10000 time cycles have elapsed. This is to allow the simulation achieve a kind of "steady state" and disregard initialization cost. In a persistent virtual world, the continuous operational efficiency of the system is much more important than initialization cost of an entire world, which happens relatively infrequently.

4.5.1 Number of Servers Metric

The total number of sims allocated to manage the world simulation is designated as λ. This represents the number of servers (computing resources) that are allocated by XPU to support the virtual world simulation. A lower number of servers are preferred so that the virtual world can be more cost-effectively managed.

4.5.2 Server Crossing Metric

The total number of server crossings, δ, is the measure of resources used in each time step to transfer the management and processing of objects from one sim to another. This represents the cost of synchronizing the state of an object to another server, which consumes networking resources. A server crossing can occur due to the movement of an object across the border between two sims, or when sims split or merge, necessitating the transfer of objects between servers. An ideal result is $\delta = 0$, with higher numbers indicating an increased number of objects that must be migrated between sims, incurring higher communication and synchronization costs. In practice, the overhead of performing an object migration can be significant [liu] and care should be taken to minimize the number of server crossings.

4.5.3 Spatial Locality Score

The spatial locality score is a measure of the virtual world management system's ability to allocate co-located objects together in the same sim. This is important because objects that are co-located are more likely to interact, and the interaction cost of two objects is lower if they are managed by the same server. If two objects in different sims interact, this will incur a communication cost between two servers. The spatial locality score, ω, can be thought of an estimate of the amount of inter-sim object-to-object interaction in the virtual world. This is estimated by modelling a probability of interaction between two objects as

$$p = \frac{1}{1 + |o_i - o_j|^3} \tag{34}$$

where o_i and o_j are the locations of object i and j. If two objects are in exactly the same location, the probability of interaction is estimated to be 1. The probability that two objects will interact in a given time step decreases cubically with the distance between the two objects. This is consistent with the observation that two users or dynamic objects are less likely to interact the farther they are from each other. Intuitively, as the distance, *d*, from an object increases, the volume enclosed by a sphere of radius *d* also increases cubically.

The overall spatial locality score, ω, is obtained by summing over the probability that all two objects in different simulators will interact.

$$\omega = \sum_{(o_i,o_j)\in O} \frac{1}{1+\left|o_i - o_j\right|^3} \tag{35}$$

where $O = \{ (o_i, o_j) \mid o_i \text{ and } o_j \text{ not in the same sim} \}$. An ideal score is $\omega = 0$, with higher scores representing the need to perform a higher amount of inter-sim interactions.

Note that this metric is not scale-free. If the same distribution of objects is rescaled from a small size (e.g. a dozen people in a conference room) to a large size (e.g. a dozen people in a forest) this metric will yield completely different estimates of interaction probability.

4.5.4 Overload Score

Overload score reflects the amount of insufficiently allocated computing power to support the simulation. We assume that each server has a fixed amount of computing power and can support the processing of a maximum of *m* dynamic objects. For the purposes of this evaluation, m=32. This number was chosen in accordance with the number of simultaneous users that a single server can process in currently deployed real-world systems [sl]. If a sim is currently managing *n* dynamic objects, and $n > m$, then it is said to be overloaded. $n \cdot (n - m)/m$ is a measure of the degree of overload of the sim. It is the number of objects in the simulator experiencing the overload condition, multiplied by the degree of overload of the simulator. Summing this result over all active simulators yields the overall overload score of the simulation.

$$\theta = \sum_{n \in \{n_s > m\}} n \cdot \frac{n-m}{m} \tag{36}$$

where n_s is the number of objects in simulator s. In the experiments presented in this chapter, the virtual world management system is constructed so that no object experiences overload, so $\theta = 0$. This is to make the results of the experiments tractable, so that the results of different XPU sim allocation algorithms can be directly matched to server crossings, spatial locality score and required number of servers.

4.6 XPU Sim Allocation Algorithms

The main challenge of this work is developing a partitioning algorithm that intelligently subdivides the world using the hierarchical bounding structure provided by XPU. The overall goal of a subdivision algorithm is threefold: It should dynamically subdivide the world in a way that provides enough computing power to satisfy the processing demands of managing dynamic objects in the world while requiring as few servers as possible. Where possible, it should construct regions so that nearby objects are allocated to the same sim to minimize inter-sim interaction cost. Objects should be migrated between sims as little as possible to reduce the overhead of synchronizing state over the network.

In a real deployable system, this algorithm is difficult to construct because the world size is unbounded, so no single server has the storage or processing power to know the location of all objects in the world. This section will begin by exploring several global-knowledge algorithms (where the state of the entire world is known to a single server), before using a distributed algorithm that does not rely on global knowledge to partition the world using the XPU tree.

4.6.1 kd_split XPU Algorithm

This first approach is a global-knowledge algorithm referred to as *kd_split* and is based on a *k-d* tree. The algorithm is very simple – it recursively partitions the world so that the number of objects on each side of the partition is even. The orientation of the partition is chosen to be orthogonal to the longest dimension of a rectangular region, bisecting it, so that region shapes will be predisposed to constructing square-like regions rather than strips of long rectangular regions.

Square regions are preferred to rectangular regions because they allow nearby objects to be grouped closer together. The efficacy of this approach is reflected in the spatial locality score. More squarely shaped regions will also allow the region to have a shorter border relative to their area, which reduces the likelihood that a moving object will traverse a region border, reducing the occurrences of server crossings due to object movement across a region boundary.

To ensure that a sim is never overloaded, a sim is always subdivided when the number of objects it contains exceeds maximum number of dynamic objects, *m* that can be fully accommodated by the sim.

When two neighboring sims, s_1 and s_2 are underutilized, the objects from s_2 should be transferred to s_1 which will take over the responsibility of managing the region formerly managed by s_2. s_2 can then be returned to the pool of unused servers. This merge operation reduces the number of active sims, λ, reducing the computing resources

Workload	λ (avg.)	Δ (avg.)	ω (avg)
No Fixed Attractor	1020.7	530.4	511.0
Single Attractor (Broad Initial Location)	978.8	784.1	1422.0
Single Attractor	1024.0	1519.7	8628.9
Row-lined Attractors	1013.6	1182.6	4585.1
2x2 Attractors	1030.9	1278.9	3278.8
Circular Motion Attractor	1024.0	2214.9	5502.5
Random Attractors	1009.1	1462.1	4509.0

Table 7: kd_split performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

committed to supporting the virtual world simulation. The merging of two sims also improves the efficiency of inter-object interaction, because objects formerly managed by s_2 can now interact directly with objects managed by s_1 without incurring the overhead of communicating between servers. This efficiency will be reflected in a lowered spatial locality score, ω. Finally, reducing the number of sims will reduce the number of dynamic objects that need to be moved from one sim to another due to object movement, which incurs a synchronization cost as the object's state must be transferred from one sim to the next. Fewer server crossing events, δ, will occur because there will be fewer sims managing a given area for moving objects to cross into.

While merging underutilized sims is beneficial for the overall efficiency of the virtual world, this merge operation also incurs a cost as all dynamic objects managed by s_2 will need to be transferred to s_1. Transferring an object from one sim to another due to a merge operation is the same as transferring an object from one sim to another due to the object moving over the boundary between two sims, so the cost of a merge operation will be reflected in an increase in the number of server crossings, δ. Care must be taken to balance the cost of merge operations versus the benefit of merging sims. Through experimentation later discussed in Section 4.6.6, a reasonable choice of merge condition is when two neighboring sims contain less than ¾ m=24 dynamic objects.

The results of using the *kd_split* algorithm to manage the different evaluation workloads are reported in Table 7. These performance results will serve as a baseline for comparison with other XPU partitioning algorithms. To understand the limitations of *kd_split*, we begin by recognizing that it always constructs a balanced tree where there are an equal number of dynamic objects managed by both sides of each sub-tree. This has the effect of ensuring that the XPU tree always has the structure of a complete tree, so there will always be 2^n active sims for some natural number, n. 2^n active sims will always be able to support $2^n \cdot m$ dynamic objects. This is insufficient granularity to ensure efficient allocation of resources, and the average total usage of assigned computing resources is just 52-55% for all workloads. However, of all the algorithms analyzed in this chapter, *kd_split* did have the best mininimum number of active sims during all workloads, because it is very efficient at allocating sims whenever the total number of dynamic objects is just under $2^n \cdot m$.

kd_split also yields unimpressive results with respect to the spatial locality score. This is for two reasons: First, because of the inefficient allocation of computing resources as mentioned above, dynamic objects are spread over an unnecessarily high number of servers. Second, this algorithm has a tendency to develop long rectangular region shapes as the simulation progresses, which are poor for exploiting spatial locality in the distribution of objects. Long rectangular regions will group objects that are near each other on one axis, but not necessarily near one another in space. The structure of XPU tree when using *kd_split* is very stable (because sims are largely underutilized) so every sim is very long-lived, very rarely allocating new sims to manage the virtual world. Since the only mechanism that prefers the construction of more square-like regions in the algorithm operates only when new sims are allocated, over time region shapes have a tendency to devolve into more thin and long rectangular regions.

The most serious shortcoming of the *kd_split* is the high number of server crossings that occur when using this approach. This is in part due to the long thin rectangular regions that tend to develop. Long rectangular regions have larger borders relative to their interior area and are biased to cause dynamic objects to cross over sim boundaries as they travel about the virtual world. However, the most significant contributor to the high number of server crossings occurs due to how rigidly *kd_split* forces the XPU tree to be completely balanced. To accomplish this, the algorithm frequently moves objects between sims, which incurs a server crossing cost. This approach generates very few server crossings from sim split and merge operations, because sims managed by *kd_split* rarely split or merge.

4.6.2 kd_split_mincross XPU Algorithm

The most significant downfall of the *kd_split* algorithm is the high number of server crossings it produces from its rigid enforcement of tree balancing. The general approach of *kd_split_mincross* is to partition the world to minimize the number of server crossings at each time step.

At the beginning of every simulation cycle this algorithm analyzes the XPU tree in a depth-first traversal, determining where to move the partition that minimizes the number of dynamic objects that will be moved between the two sub-trees. This calculation

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	1012.3	146.0	485.9
Single Attractor (Broad Initial Location)	986.2	395.0	1370.1
Single Attractor	974.5	1100.9	8264.3
Row-lined Attractors	966.5	424.1	4312.1
2x2 Attractors	995.9	845.4	3127.3
Circular Motion Attractor	939.0	1755.9	5283.7
Random Attractors	962.0	983.2	4298.5

Table 8: kd_split_mincross performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

requires global knowledge of the location of all objects in the virtual world simulation which makes it impractical for real-world deployment. In cases where there is more than one choice of minimal sim-crossing partitions, the algorithm chooses the one that will most equally balance the number of objects managed by the two sub-trees of the XPU node. In this way, the shape of the tree is biased to being more like *kd_split* when possible. To prevent the formation of a degenerate or highly unbalanced tree, we only allow partitions where one sub-tree manages at most twice the number of dynamic objects as its neighbor. As with all the experiments reported in this chapter, a sim is split when it manages more than m=32 dynamic objects (to avoid overloading the server), and merged when neighboring sims contain less than ¾ m=24 objects.

The performance results of *kd_split_mincross* are reported in Table 8. This algorithm demonstrates a dramatic improvement over *kd_split* in terms of the number of server crossings, reducing it by 21-72%. The most significant improvement occurs in the *No Fixed Attractor* workload because the objects in this workload are more evenly distributed throughout the world relative to the other workloads. This distribution allows the sim allocation algorithm more latitude to choose partitions that reduce the number of server crossings. More densely clustered workloads such as *Single Attractor* and *Circular Motion Attractor* benefit less from the improved partition choice in *kd_split_mincross* because these workloads have more dynamic objects moving around in dense clusters (which will necessarily be more heavily partitioned) causing server crossings to occur.

The number of allocated servers is marginally improved over *kd_split* due to the less rigid enforcement of a fully balanced tree. The reduced number of sims is also beneficial to a lowered spatial locality score, because the objects are managed by a smaller number of sims. Also, since sims are more often re-allocated as the population of dynamic objects change, this has the effect of producing region shapes that more closely approximate a square because regions are always split on the long edge. This is beneficial in reducing the number of server crossing incurred because of object motion, due to a shorter border length relative to enclosed area, and reducing the spatial locality score because nearby objects are more frequently partitioned in a single sim.

4.6.3 centersplit_mincross XPU Algorithm

One notable observation of the behavior of the previously presented XPU algorithms is that having square-shaped regions is beneficial for reducing the number of server crossings and the spatial locality score. This motivates the construction of a new global knowledge algorithm, *centersplit_mincross*. This algorithm behaves exactly as *kd_split_mincross* does, but instead of preferring partitions that better balance the tree, it prefers partitions that more evenly divide the space managed by two neighboring sub-trees. This bias will allow regions to remain more squarely shaped as the simulation progresses.

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	930.3	137.4	427.1
Single Attractor (Broad Initial Location)	896.2	384.7	1314.1
Single Attractor	901.4	1107.5	8144.5
Row-lined Attractors	954.2	444.4	4123.3
2x2 Attractors	927.6	845.6	3045.8
Circular Motion Attractor	917.1	1777.5	5208.4
Random Attractors	913.2	971.4	4192.5

Table 9: centersplit_mincross performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

A summary of the results of *centersplit_mincross* running on the various evaluation workloads are presented in Table 9. Overall, the impact on the number of server crossings relative to *kd_split_mincross* was minimal. The most significant improvement was exhibited on the *No Fixed Attractor* workload because this workload has a more uniform object distribution than the other workloads, which allows for the formation of larger, squarer regions. Dynamic objects in this workload are also slower moving and so are not crossing between sims as frequently. Fast moving objects will incur the cost of many server crossing operations, regardless of the sim shape. A noticeable improvement was exhibited in the spatial locality score for all workloads. As predicted, the squarer region shapes allow more nearby objects to be allocated to the same region, decreasing the number of objects that must interact across a sim boundary. *centersplit_mincross* was also more effective at reducing the number of sims required to support the simulation. Just as with *kd_split*, the bias in *kd_split_mincross* towards creating a fully balanced tree prevented some sims from reaching the merge threshold, leaving more servers underutilized. By removing this bias in *centersplit_mincross*, simulators were allocated in a way that allowed them to more fully utilize their computational capacity to manage

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	932.9	138.5	428.8
Single Attractor (Broad Initial Location)	905.1	397.0	1342
Single Attractor	1006.9	1220.1	8665.8
Row-lined Attractors	1066.0	422.6	4093.8
2x2 Attractors	965.5	893.7	3143.7
Circular Motion Attractor	1086.9	1960.0	5771.5
Random Attractors	961.8	1038.4	4340

Table 10: centersplit_unbalanced_mincross performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

dynamic objects. Having a well-balanced tree is not very useful for efficiently allocating server resources because we prefer that the leaves of the tree (sims) to be nearly at capacity rather than having a balanced tree with leaves (sims) being evenly loaded and under capacity.

A more careful examination of the simulation shows that the partition selection algorithm was occasionally being constrained by the balancing requirement of *centersplit_mincross* where one sub-tree can only contain at most double the number of dynamic objects as its neighbor. To examine the effects of this requirement, a modification to this algorithm was explored, termed *centersplit_unbalanced_mincross*. This algorithm chooses partitions exactly as *centersplit_mincross* does but without the balancing requirement. Each sub-tree is only required to contain at least one object.

The performance summary of this modified algorithm is reported in Table 10. *centersplit_unbalanced_mincross* does not perform as well as *centersplit_mincross*. By completely removing the balancing requirement many sims went highly underutilized. This increased the number of sims required to support the virtual world, which in turn, caused the spatial locality scores and the number of server crossings to increase.

From this analysis, we conclude that *centersplit_mincross* represents a global-knowledge algorithm that does well in reducing the server crossing cost of managing a metaverse in an XPU-style architecture.

4.6.4 clustersplit and clustersplit_mincross XPU Algorithms

The key to improving the spatial locality of an XPU partition algorithm is to group nearby objects together in the same region since nearby objects are much more likely to communicate than distant objects. The section introduces *clustersplit*, an XPU partitioning algorithm that uses a simplified k-means cluster analysis to allocate clusters of objects to a single region. This is a global knowledge algorithm that is impractical to deploy on a real-world system but serves as a reference algorithm to evaluate the efficacy of other approaches.

As with all XPU algorithms, *clustersplit* recursively partitions the world, splitting each sim along the shortest axis of the enclosed region at each level of the XPU tree, splitting sims when they reach the computational capacity of a server to support them. *clustersplit* seeks to minimize the squared error of the locations of all the objects in each sub-region with respect to the centroid of objects in the sub-region. Suppose the set of all objects in a region, O, is to be split into two sub-regions containing the sets of objects, O_L and O_R where $O_L \cup O_R = O$. Define the centroid of a region to be the geometric average of all object locations in a set:

$$centroid(O) = \frac{1}{|O|} \sum_{o \in O} location(o) \tag{37}$$

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	931.6	1185.9	333.3
Single Attractor (Broad Initial Location)	899.4	1778.3	1139.1
Single Attractor	890.6	2390.6	7470.6
Row-lined Attractors	913.9	1699.2	3368.7
2x2 Attractors	916.8	2347.2	2644.8
Circular Motion Attractor	907.5	2969.6	4631.6
Random Attractors	898.7	2786.3	3749.9

Table 11: cluster_split performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

The *clustersplit* partitioning algorithm will choose the partition that minimizes the mean squared error of all of the locations of all objects in each sub-region with respect to its centroid:

$$MSE = \frac{1}{|O|}\left(\sum_{o\in O_L}|location(o) - centroid(O_L)|^2 + \sum_{o\in O_R}|location(o) - centroid(O_R)|^2\right) \tag{38}$$

The results of using this partitioning algorithm on the evaluation workloads are reported in Table 11. With respect to the previously explored algorithms, *clustersplit* performs approximately on par with the best algorithms in terms of the number of sims, and bests all previous algorithms in terms of spatial locality score. This indicates that the previously explored algorithms leave room for improvement in this area. Unfortunately, *clustersplit* performs abysmally in terms of server crossings, incurring a performance degradation of roughly 70% to 800% compared to *centersplit_mincross*. This is because *clustersplit* ruthlessly moves objects across sims to try and preserve groups of objects in

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	939.6	147.5	418.2
Single Attractor (Broad Initial Location)	907.9	397.6	1302
Single Attractor	900.4	1116	8124.2
Row-lined Attractors	939	459.5	3951.8
2x2 Attractors	926.2	858.9	3012.1
Circular Motion Attractor	919.2	1792	5189.9
Random Attractors	907	990.9	4174.9

Table 12: clustersplit_mincross performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

their own simulator. It is for this reason that *clustersplit* is an unsuitable algorithm for XPU partitioning.

To blend the benefits of a partition algorithm that is capable of identifying clusters of objects while avoiding the penalty of an unacceptable amount of server crossings, a new algorithm, *clustersplit_mincross*, is explored. This algorithm is constructed in a similar fashion to *centersplit_mincross*. As before, *kd_split_mincross* is used as an algorithmic framework, but instead of preferring partitions that better balance the tree, it prefers partitions that are closer to the ideal cluster split. This bias will augment the *kd_split_mincross* algorithm so that region partitioning will better preserve clusters in a single sim.

The results of running *clustersplit_mincross* on the evaluation workloads are reported in Table 12. With respect to the number of sims required for simulation, this partitioning algorithm performs as well, with numbers on par with *centersplit_mincross*. Since region shapes are not as square as *centersplit_mincross*, *clustersplit_mincross* incurs more server crossings because region borders are longer relative to their enclosed area, which increases the chances that a moving dynamic object will cross a region boundary in its regular course of travel. With respect to the spatial locality score, *clustersplit_mincross* performs better than *centersplit_mincross*, because it better preserves clustered objects in a single region, which reduces the need for nearby dynamic objects to interact across region boundaries. *clustersplit_mincross* represents an exact, global-knowledge algorithm that does well to minimize the spatial locality score in a metaverse managed with an XPU style framework.

4.6.5 bintree XPU Algorithm

We now turn our attention to the development of a distributed algorithm that does not require global knowledge. The goal of this algorithm is to use a simplified understanding of the world so that the partitioning algorithm is computable in a real implementation and does not rely on global knowledge. As discovered in the analysis of the global knowledge algorithms (Section 4.6.1-4.6.4), it is preferable to have square-shaped regions. The reason for this preference is two-fold: Firstly, a square (compared to a rectangle) has a smaller border, relative to its enclosed volume. This will lower the number of server crossings incurred due to the motion of dynamic objects, because there will be fewer borders to cross. Secondly, in a square region, the maximum distance between two objects in a region will be shorter than that of two objects in a rectangular shaped region. This will be beneficial in lowering the spatial locality score. To accomplish this, we introduce the bintree data structure [shaffer]. A bintree is similar to a quadtree, but instead of recursively partitioning a square region into four congruent square sub-regions, it partitions regions into two congruent rectangular sub-regions. This structure is employed by the *bintree* partitioning algorithm.

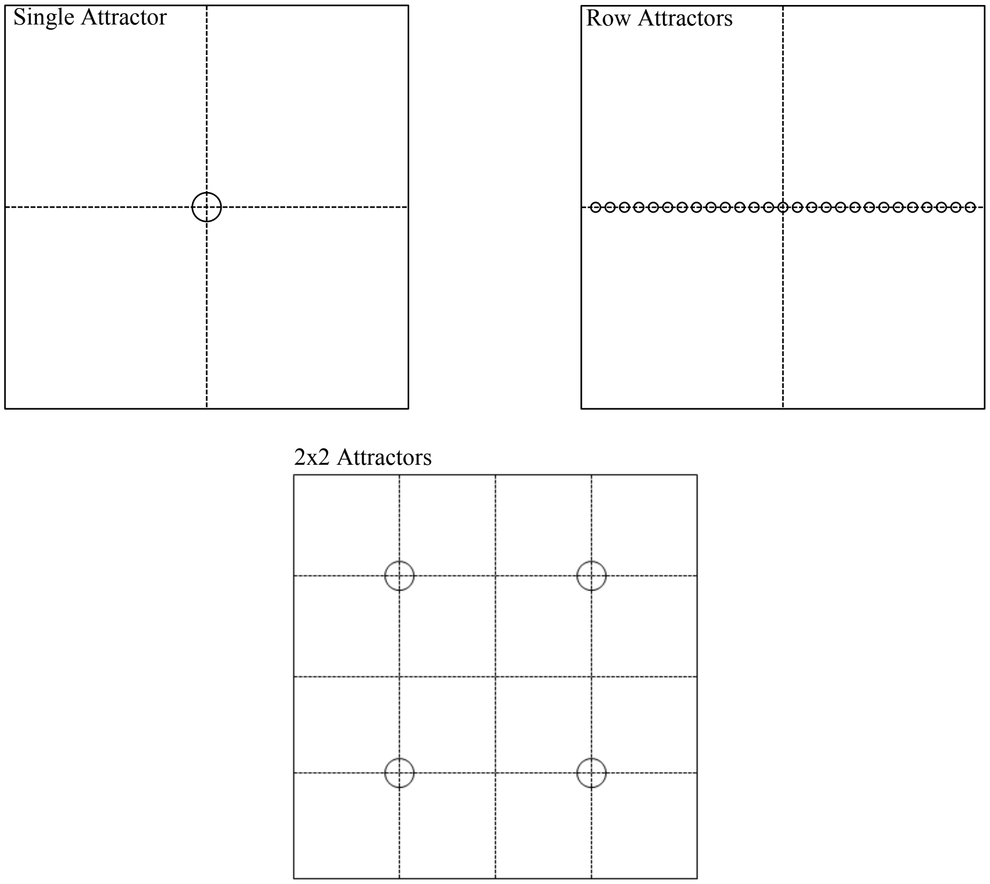


Figure 45: The square-shaped virtual world, using different fixed attractor configurations. Circles represent fixed attractors. The dotted lines represent the world after being partitioned by the first few levels of bintree region subdivision.

The key simplification used by *bintree* that allows it to operate in a real-world scenario is that at each partitioning stage the algorithm only requires a greatly simplified understanding of the world state. Instead of requiring knowledge of the locations of all objects in a given region before being able to choose a partition, it only needs to determine if the region should be split or not. This is much easier to compute in a real system than previously discussed global-knowledge algorithms because it does not consider the location of objects when constructing region partitions – it simply divides each region in half, approximating the ideal region division. At each stage of spatial subdivision, the *bintree* algorithm need only know how many dynamic objects are contained in a region before choosing to split or merge regions. As before, *bintree* will split sims containing more than *m*=32 dynamic objects and merge neighboring sims containing less than ¾ *m*=24.

The construction of several of the evaluation workloads represent near worst-case scenarios for *bintree* (Figure 45). In the *Single Attractor* and *Single Attractor (Broad Initial Location)* workloads, there is a large fixed attractor at the very center of the virtual space. This is especially disadvantageous for *bintree* because the region borders are fixed and unmoving, so the first two levels of region subdivision will partition the space so that the single large attractor is placed at the corner of four regions. This will insure that dynamic objects moving about the center of gravity will incur a high number of server crossings. In the *Row-lined Attractors* workload, the attractors are placed in a row. After the first two levels of region subdivision, all of the fixed attractors will be placed on a region border. Moving dynamic objects near the fixed attractors will therefore have a higher predisposition of crossing a sim border as they move. In the *2x2 Attractors* workload, after just four levels of region subdivision, all of the fixed attractors will be placed at the corner of four regions. These four fixed attractor configurations disadvantage *bintree*, relative to the previously discussed global-knowledge algorithms because they do not have fixed region borders and so will not be exposed to these pathological worst-case scenarios.

Workload	λ (avg.)	δ (avg.)	ω (avg.)
No Fixed Attractor	929.9	133.1	391
Single Attractor (Broad Initial Location)	895.5	371.5	1242
Single Attractor	900.7	1058.9	7815.6
Row-lined Attractors	989.2	399.2	3906.1
2x2 Attractors	927.6	809.4	2869.3
Circular Motion Attractor	909.6	1715.7	4891.5
Random Attractors	908.8	932.2	3969.1

Table 13: bintree performance. The number of servers (λ), server crossings (δ) and spatial locality score (ω) reported here are averages over the 100000 timestep workload.

The results of running the *bintree* algorithm on the evaluation workloads are reported in Table 13. Compared to the best of the global-knowledge algorithms, *clustersplit_mincross* and *centersplit_mincross*, *bintree* performs well, even on workloads that represent worst-case scenarios for *bintree*. In terms of the number of servers required to support the simulation, *bintree* does roughly as well as the better of *centersplit_mincross* and *clustersplit_mincross*. One weaknesses of *bintree* is that it can only divide regions in half spatially, making it inefficient in dealing with situations where the density of objects increases by more than a factor of two for neighboring sub-regions. If a large number of objects only occupy a very small portion of the virtual space, *bintree* must repeatedly subdivide the space before it can begin creating very small sub-region to distribute the object management load onto multiple sims. This process will create several under-utilized sims. This coarseness of region subdivision is demonstrated in the *Row-lined Attractors* workload because the vast majority of dynamic objects are concentrated about the center axis of the virtual space, and *bintree* must create several under-utilized regions before it can begin subdividing regions where objects are heavily concentrated (Figure 46).

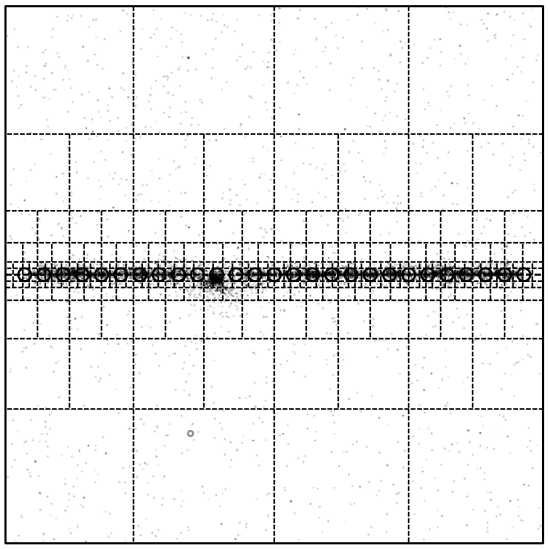


Figure 46: The square-shaped virtual world, using the Row-lined Attractors configuration. The smaller circles and dots represent dynamic objects, and the larger circles represent fixed attractors. The dotted lines represent the world after being partitioned with bintree. Note the larger, underutilized regions.

bintree performs well when compared to *centersplit_mincross* (the best global knowledge algorithm in terms of server crossings) and *clustersplit_mincross* (the best global knowledge algorithm in terms of spatial locality), showing between 3-12% improvements in all cases. This is due to *bintree*'s rigid adherence to square-like regions, which are an excellent heuristic to reducing server crossings and spatial locality score.

4.6.6 Choosing the XPU Merge Constant

All the algorithms discussed in Section 4.6 depend on a fixed merge constant to determine when the load shared between two neighboring simulators is low enough to merit merging the workload managed by two neighboring simulators into a single simulator. If the merge constant is set too high, the partitioning algorithm will seek to merge simulators too frequently. This merge operation incurs a significant operational cost because all the objects in one simulator will need to be transferred to the other, and so should be minimized [liu]. This cost is reflected in the server crossing metric, as each object must be moved between simulators during the merge operation. If the merge constant is set too low, the partitioning algorithm will merge simulators too infrequently, resulting in a higher than necessary number of servers being allocated to manage the simulation. A higher number of servers imply that the dynamic objects of the simulation will be distributed across more simulators, resulting in a worse spatial locality score. A higher number of servers will also result in more server crossing operations because dynamic objects move and a higher number of active simulators mean that there are more regions for these objects to cross into.

This merge constant is determined experimentally, by running the full simulation with the split constant fixed at m=32 (the maximum number of dynamic objects each simulator can manage without becoming overloaded) and varying the merge constant between 2 to 32. These experiments are reported in Appendix B. A merge constant of ¾ m was found to be a reasonable choice for minimizing sim crossings and excessive merge/split operations.

4.7 Results of Fixed Square Grid Spatial Subdivision

To better understand the effectiveness of dynamic spatial partitioning, we have measured the performance of the various workloads when using a fixed square grid pattern that is typical of Second Life and OpenSim (Figure 47) [sl][opensim]. Unlike the dynamic partitioning algorithms explored in Section 4.6, this style of fixed spatial partitioning must be set at the beginning of the experiment and cannot be changed as the

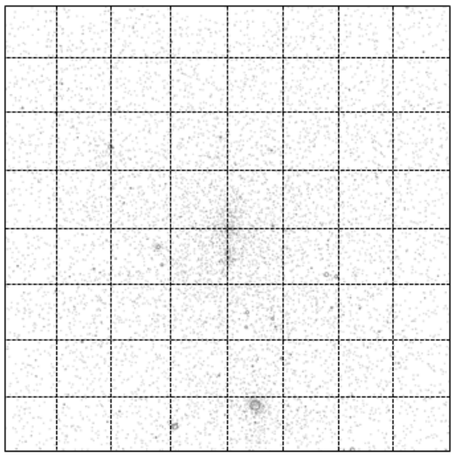


Figure 47: The square-shaped virtual world, using a regular square grid spatial subdivision strategy. The dotted lines represent the borders between regions.

simulation progresses. The administrator of this sort of system must have a way of anticipating the kind of traffic that their virtual world will receive in order to properly dimension their virtual world. To limit the number of dynamic objects assigned to a server, as the algorithms presented in Section 4.6 do, the grid size would have to be set to match the region of highest density during the entire simulation workload.

In these experiments, we consider square grids with between 900 servers (similar to the average number of servers used by *bintree* and the other dynamic spatial subdivision algorithms) and 35344 servers (more servers than objects). When the numbers of servers used is small, the number of server crossings and the spatial locality score is better than what is observed when using the dynamic partitioning approaches by up to a factor of two. However, this advantage comes at a cost – the system becomes highly overloaded. At times, some objects are allocated to servers that are managing more than ten times its maximum load. The dynamic partitioning techniques in in Section 4.6 never assign more objects to a server than it can accommodate and so have no overload.

The only way to reduce the amount of overload in this kind of system is to add more servers by dividing the world into smaller regions. While this does reduce the amount of overload, it does not eliminate it in any of the tested workloads and scenarios. As the number of servers assigned to manage the virtual world increases, the spatial locality score and the number of server crossings also increases (worsens). Fixed grids using more than 60^2 servers show no advantage to fixed grid partitioning in terms of the reported metrics, while still exhibiting a high amount of overload. It was not possible to completely eliminate overload in the test scenarios by using more servers in a finer grid pattern. The full set of experimental results is presented in Appendix C.

4.8 Conclusion

In this chapter we have proposed and described XPU, a hierarchical space partitioning architecture used to distribute a simulation workload in infinitely scaling chunks so that simulation workload requirements can be met. XPU borrows acceleration structures from computer graphics and develops new uses and algorithms to leverage these structures to support a dynamically scaling virtual world load balancing system using distributed computing. These algorithms consider the inherent cost in networking and communication operations when distributing computing over several servers. It has been shown that the *bintree* spatial subdivision algorithm has good performance characteristics relative to global-knowledge algorithms and fixed grid partitioning techniques over several types of metaverse workloads. Using the simple heuristic of favoring square-shaped regions, and subdividing the world so that the activity within each region can be mapped to a single server, the perimeter of the region relative to its area is minimized, and the region shape allows nearby objects to be mapped to a single server. This reduces the number of server crossings that are required to manage an object travelling along a path, and decreases the amount of communication that must occur between servers for inter-object interactions. Because this heuristic is so simple and does not require global knowledge, *bintree* is a suitable algorithm for deployment in a dynamically load-balanced distributed virtual world.

As the density of object clusters increases, it becomes more difficult to choose good partitions and all spatial partitioning schemes will suffer. To improve performance for these cases, it is essential to increasing the number of objects that can be processed by a server, which would effectively increase region size. This can be accomplished by using more powerful servers, optimizing the server [bowman] or separating the services that the server provides [quax].

4.9 Future Work

One of the limitations in XPU is that it can only support rectangular regions. This can be a limiting factor in cases where it is desirable to have irregular region shapes to better subdivide the interaction. One potential means of addressing this is to allow regions to subdivide beyond what is necessary, allowing a single server to manage multiple regions. This would have the effect of segregating virtual world activity into irregularly shaped regions composed of rectangular sub-regions. This would also have the benefit of allowing the structure of the world to efficiently adapt to virtual world content distributions where the density of the content varies more than exponentially. Another way to approach this problem is to use a structure based on skip quadtrees or compressed quadtrees [eppstein].

The XPU virtual space is currently limited by its initial size. The virtual space represents a square or rectangle and there is no mechanism to allow the virtual space to grow or shrink. An extension to XPU that allows the virtual world to represent an infinitely extending and unconstrained space will provide additional freedom to the system.

The evaluation metrics used to evaluate XPU can also be expanded to consider a more detailed simulation workload. For example, the estimation of spatial locality considers all objects as having equal weight. In many realistic scenarios, dynamic objects are not all equally important. For example, a speaker at a conference carries more influence than an audience member, and an aircraft carrier is visually more imposing than a rowboat.

Currently, XPU only uses proximity between dynamic objects as an estimator of the likelihood of inter-object interaction. In a more advanced simulation, it might be possible to use a more sophisticated estimator, such as mutual visibility. For example, two people in close proximity to one another are less likely to interact if they are separated by a wall.

The analysis in this chapter regarding server splitting and merging does not consider that the migration of many simultaneous objects can add a processing delay, as many objects will need to be synchronized over the network between two servers. This delay can be significant in real-world scenarios. One way to mitigate this cost is to gradually split/merge neighboring regions, amortizing the server crossing cost over time.

The algorithms explored in this chapter are purely reactive and do not make any attempt to predict future load levels and object distribution patterns. By considering the velocity of objects, it would be possible to predict future object distribution in the world, and so it may be possible to take advantage of this prediction to perform better spatial subdivision. It may also be possible to employ user-directed hints or machine learning techniques to predict object distribution patterns. For example, if a user schedules a recurring event with a significant number of participants every day at noon, it would be possible to use this information to predict the amount of computing resources required to manage a specific area in the virtual world. The spatial partitioning system could choose to pre-allocate servers to manage that region in anticipation of the predicted traffic.

Chapter 5 Conclusion and Future Work

This dissertation concludes with a summary of the contributions made in this thesis, the lessons learned, as well as discussion of the direction for future work.

5.1 Summary

The work presented in this dissertation explores algorithm and systems software design at different layers of an interactive networked 3D virtual application, ranging from the base networking layer, to client/server graphics streaming protocols, to the design of a distributed virtual-world back-end server architecture. These all address different aspects of designing a truly unconstrained immersive virtual world.

By analyzing the semantics of the Internet and considering the networking requirements of a virtual world, we have demonstrated how to build simpler and quicker routers and network devices to better support the packet processing demands of a real-time networked virtual environment. In reconsidering the necessity of supporting exact, predictable semantics over a best-effort network, we have determined that allowing packet classifiers to misclassify packets with a tiny probability gives us the freedom to design a more efficient packet classification cache. By extending the best-effort semantics inherent in IP networking upwards to the transport layer, we have proposed a novel design for packet classification caches that probabilistically manages packet processing decisions, using much simpler hardware, without modifying the overall semantics of networking design. This approach of using an approximate cache can also be extended to be used purely as an acceleration structure to augment the performance of an exact cache. With this solution, it is possible to optimize a system so that common cases are handled more quickly without the penalty of probabilistic errors.

By considering the constraints of the networking infrastructure and the desire to support a high-quality remote rendering, we have constructed a novel streaming algorithm and data representation to better support remote visualization for virtual terrain rendering. The proposed algorithm adapts to the remote viewer's perspective and the features of the terrain so that a high-quality representation of the virtual terrain stored on a server can be accurately rendered on a remote client. We have demonstrated how to design more efficient, practical ways of streaming terrain data by discarding detail while preserving the overall visual experience. Because the human eye is tolerant of minor deviation in perceptual information, the difference in the true, exact representation of the terrain, and the rendered solution is imperceptible to a human observer. Graphical information is reorganized and prioritized so that the most prominent visual information is transmitted before adding fine detail so that a coarse approximation of the scene can be rendered before rapidly converging on a finely detailed rendering.

XPU, a design for a systems architecture for metaverse-style virtual environments is presented. XPU allows for the distributed management of an expansive virtual world simulator by using a hierarchical spatial partitioning technique. This design allows the virtual world to perform load balancing in reaction to the changing and dynamic nature of the virtual world simulation. This design considers the computational and communication costs of managing the distributed simulation in order to deliver an efficient, scalable system. The result is a virtual world architecture that is both practical to deploy and has demonstrably good performance characteristics, relative to impractical global-knowledge solutions.

5.2 Future Directions

We have addressed more specific future work in the individual chapters. Here, we will focus on the larger issues and challenges that will arise as a result of the demand for richer, more interactive virtual world experiences.

One area of active research and development is the user interface to interact with virtual worlds. The oncoming availability of affordable, low-latency virtual reality head-mounted displays promises to spur a new profusion in virtual reality applications [oculus]. These types of displays have been combined before with devices that allow users to express natural motion, such as omnidirectional treadmills, to provide a more involved experience [darken]. Capturing non-verbal communication (such as facial expressions and body language) is an on-going topic of research that will allow users to project a more complete and expressive avatar in virtual environments [yang2]. Non-traditional user interfaces, such as devices using gesture recognition, haptic feedback and olfactory stimuli also present possibilities for interacting with virtual environments in a deeper way [kortum].

The majority of the tools used to develop virtual world content (such as 3D modelling packages, text-based document editors, bitmap manipulation software, and avatar customizers) are run offline and later imported into the virtual world. While easier to construct, this content development workflow disrupts the immersive nature of virtual worlds and presents obstacles to seamless remote collaboration. There would be benefits to designing a unified framework to allow the integration of external content development tools with virtual environments so that consistency and the sense of virtual world immersion is maintained [croquet].

A more specific extension to this thesis would be to consider how to build a system that is capable of large-scale distributed graphical simplification of a dynamic world. For a remote client to be able to view and interact with a vast virtual world that is unlimited in detail and complexity, a practical way of summarizing the world using level-of-detail reduction algorithms to present a simplified visual representation of the world is necessary. The work and techniques presented in this dissertation can form the framework for a solution to this problem; it is possible to leverage a hierarchical spatial subdivision method (such as XPU) to generate hierarchical level-of-detail simplification for remote clients [cheslack]. It may also be possible to use this structure in conjunction with image-based rendering techniques to present a compact and efficient representation of a large virtual world [chaudhuri].

There is also the question on how to allow communication between independently administrated virtual worlds. Some existing virtual world architectures such as OpenSimulator and Active Worlds allow users to travel between different systems, but each virtual world exists in its own disjoint virtual space. As virtual worlds evolve and become ubiquitous, it is desirable to develop paradigms where different virtual worlds can choose to express different relationships with each other. For example, two virtual world systems could choose to be arranged as neighbors in a combined virtual space, or one virtual world may be fully embedded inside another. These relationships can be even more complicated to express if virtual worlds can represent a non-uniform or non-Cartesian space.

BIBLIOGRAPHY

[active] Active Worlds, http://www.activeworlds.com/

[ahmed] D.T. Ahmed and S. Shirmohammadi. Uniform and Non-Uniform Zoning for Load Balancing in Virtual Environments, In Proceedings of the International Conference on Embedded and Multimedia Computing, Cebu, Philippines, August 11-13, 2010.

[alliez] P. Alliez and C. Gotsman. Recent Advances in Compression of 3D Meshes. In Proceedings of the Symposium on Multiresolution in Geometric Modeling, September 2003.

[baboescu] F. Baboescu and G. Varghese. Scalable Packet Classification. In Proceedings of ACM SIGCOMM 2001, pages 199-210, August 2001.

[baboescu2] F. Baboescu, S. Singh, and G. Varghese. Packet Classification for Core Routers: Is There an Alternative to CAMs?. In Proceedings of IEEE Infocom 2003.

[bell] Passive Measurement and Analysis Project, National Laboratory for Applied Network Research (NLANR), available at http://pma.nlanr.net/Traces/Traces/

[bloom] B.H. Bloom. Space/time tradeoffs in hash coding with allowable errors. Communications of ACM 13(7): 422-426, July 1970.

[blue] Blue Mars, http://www.bluemars.com

[bowman] C. M. Bowman, D. Lake, and J. Hurliman. Designing Extensible and Scalable Virtual World Platforms. In Proceedings of the Extensible Virtual Worlds Workshop (X10), 2010.

[broder] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: a survey. 40th Annual Allerton Conference on Communication, Control, and Computing, Allerton, IL, October, 2002

[brownlee] N. Brownlee and M. Murray. Streams, Fows and Torrents. In Proceedings of the Passive and Active Measurement Workshop, April 2001.

[byers] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed Content Delivery Across Adaptive Overly Networks. In Proceedings of ACM SIGCOMM 2002, pages 47-60, August 2002.

[carlson] S. Carlson. Algorithm of the Gods. In Scientific American, March 1997.

[carter] L. Carter and M. Wegman. Universal classes of hash functions. Journal of Computer and System Sciences, pages 143-154, 1979.

[chang] F. Chang and Wu-chang Feng. Modeling Player Session Times of On-line Games. In Proceedings of the Workshop on Network and Systems Support for Games (NetGames), Redwood City, California, May 2003.

[chaudhuri] S. Chaudhuri, D. Horn, P. Hanrahan, and V. Koltun. Image-Based Exploration of Massive Online Environments. Stanford University Computer Science Technical Report, CSTR 2009-02, 2009.

[chen] B. Chen and T. Nishita. Multiresolution Streaming Mesh with Shape Preserving and QoS-like Controlling. In Proceedings of ACM 2002 International Conference on 3D Web Technology, February 2002.

[chen2] K. Chen, P. Huang, C. Huang and C. Lei. Game traffic Analysis: An MMORPG perspective. In Computer Networks, Vol. 50, No. 16, pages 3002--3023, November 2006.

[chen3] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality Aware Dynamic Load Management for Massively Multiplayer Games. In ACM SIGPLAN Symposium on PPoPP, pages 289–300, New York, NY, USA, 2005.

[cheslack] E. Cheslack-Postava, T. Azim, B.F. Mistree, D. R. Horn., J. Terrace, P. Levis, and M. J.Freedman. A scalable server for 3d metaverses. In Proceedings of the USENIX Annual Technical Conference, June 2012.

[chiueh] T. Chiueh and P. Pradhan. High Performance IP Routing Table Lookup Using CPU Caching. In Proceedings of IEEE INFOCOMM'99, New York, NY, March 1999.

[chiueh2] T. Chiueh and P. Pradhan. Cache Memory Design for Network Processors. In Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA), 2000.

[cisco] Cisco Systems. Cisco Visual Networking Index: The Zettabyte Era – Trends and Analysis.. White Paper, May 29, 2013.

[claffy] Kimberly Claffy, Internet traffic characterization. Ph.D. Thesis, San Diego, 1994.

[claypool] Mark Claypool. Network Characteristics for Server Selection in Online Games. In Proceedings of the ACM/SPIE Multimedia Computing and Networking (MMCN) Conference, San Jose, California, USA, January 2008.

[croquet] Croquet Project. http://www.opencroquet.org/

[czerwinski] S. Czerwinski, B.Y. Zhao, T. Hodes, A.D. Joseph, and R. Katz, R. An Architecture for a Secure Service Discovery Service. In Proceedings of MobiCom-99, pages 24-35, N.Y., August 1999.

[darken] R. Darken, W. Cockayne. and D. Carmein. The Omnidirectional Treadmill: A Locomotion Device for Virtual Worlds. In Proceedings of the ACM Symposium on User Interface Software and Technology, pages, 213–221, 1997.

[darkstar] Sun, Game Server Technology White Paper, http://www.sun.com/solutions/documents/white-papers/me_sungameserver.pdf

[dharmapurikar] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. In Proceedings of ACM SIGCOMM'03, Karlsruhe, Germany, August 2003.

[duchaineau] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich and M. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In Proceedings of the 8th Conference on Visualization, pages 81-89, 1997.

[egevang] K. Egevang and P. Francis. IP Network Address Translator. RFC 1641, May 1994.

[eppstein] D. Eppstein, M. T. Goodrich and J. Z. Sun. The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data. International Journal of Computational Geometry & Applications. April 18, 2008.

[fan] L. Fan, L, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. IEEE/ACM Transactions on Networking (TON), volume 8, issue 3, pages 281-293, June 2000.

[feldmann] A. Feldmann and S. Muthukrishnan. Tradeoffs for Packet Classification, In Proceedings of IEEE Infocom, pages 1193-2002, March 2000.

[feng] W. Feng, D. Kandlur, Saha D, and K. Shin. Blue: A New Class of Active Queue Management Algorithms. University of Michigan CSE-TR-387-99, April 1999.

[feng02] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning Online Games: A Traffic Analysis of a Busy Counter-Strike Server. In Proceedings of the Internet Measurement Workshop, November 2002.

[ferreira] M. Ferreira and R. Morla. Second Life In-World Action Traffic Modeling. In Proceedings of the 20th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), 2010.

[fraleigh] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi. Packet-Level Traffic Measurements from a Tier-1 IP Backbone. Sprint ATL Technical Report TR01-ATL-110101, Burlingame, CA., USA, November 2001.

[gamersbin] Gamersbin – MMO Screenshots. http://www.gamersbin.com/

[gearth] Google Earth. http://earth.google.com/

[gopalan] K. Gopalan and T. Chiueh. Improving Route Lookup Performance Using Network Processor Cache. In Proceedings of Supercomputing '02, ACM/IEEE Conference on Supercomputing, pages 1-10, 2002.

[gupta] P. Gupta and N. McKeown. Algorithms for Packet Classification. IEEE Network Special Issue, volume 15, number 2, ages 24-32, March/April 2001.

[hoppe] H. Hoppe. Progressive meshes. In ACM SIGGRAPH 96, August 1996.

[hu] S. Hu and K. Chen. VSO: Self-organizing Spatial Publish Subscribe. In Proceeings of the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2011), October 2011.

[huitima] C. Huitima. IPv6: The New Internet Protocol (2nd Edition). Prentice Hall, 1998.

[iannaccone] G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring Very High Speed Links. In Proceedings of ACM SIGCOMM Internet Measurement Workshop, San Francisco, CA, November 2001.

[ixp] Intel IXP1200 Network Processor, http://www.intel.com/design/network/products/npfamily/ixp1200.htm

[isenburg] M. Isenburg, and P. Lindstrom. Streaming Meshes. LLNL tech. report UCRL-TR-211608, April 2005.

[jain] R. Jain. Characteristics of Destination Address Locality in Computer Networks: a Comparison of Caching Schemes, Computer Networks and ISDN Systems, 18(4), pages 243-254, May 1990.

[jpeg] Independent JPEG Group, http://www.ijg.org/

[lakshman] Lakshman, T. V. and Stiliadis, D., High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching. In Proceedings of the ACM SIGCOMM 1998, pages 203-214, August, 1998.

[leiner] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S Wolff. A Brief History of the Internet. In ACM SIGCOMM Computer Communication Review, volume 39 issue 5, pages 22-31, October 2009.

[li] K. Li, F. Chang, D. Berger, and W. Feng. Architectures for Packet Classification Caching. In Proceedings of the 11th IEEE International Conference on Networks (ICON), 2003.

[liao] C. Liao and Y. Chung. Tree-Based Parallel Load-Balancing Methods for Solution Adaptive Finite Element Graphs on Distributed Memory Multicomputers. IEEE Trans. Parallel and Distributed Systems, vol. 10, no. 4, Apr. 1999.

[liu] H. Liu and M. Bowman. Scale Virtual Worlds Through Dynamic Load Balancing. In Proc. of 14th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, 2010.

[liu2] H. Liu, M. Bowman, and F. Chang. Survey of State Melding in Virtual Worlds. ACM Computing Surveys (CSUR), volume 44, issue 4, number 21, August 2012.

[luebke] D. Luebke and C. Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. Proceedings of SIGGRAPH 97. pp. 199-208, 1997

[kinicki] J. Kinicki and M. Claypool. Traffic Analysis of Avatars in Second Life, In Proceedings of the 18th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), May 2008.

[kortum] P. Kortum, HCI Beyond the GUI: Design for Haptic, Speech, Olfactory, and Other Nontraditional Interfaces. Morgan Kaufmann, Burlington, MA, 2008.

[kumar] S. Kuma r, J. Chhugani, C. Kim, D. Kim, A. Nguyen, C. Biania, Y. Kim, P. Dubey. Chracterization and Analysis of Second Life Virtual World. IEEE Computer Graphics & Applications, (March 2008)

[mccreary] S. McCreary and k. claffy. Trends in Wide Area IP Traffic Patterns - A view from Ames Internet Exchange. Technical Report, CAIDA, 2000.

[mitzenmacher] M. Mitzenmacher. Compressed Bloom Filters. In Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing , pages 144-150, 2001

[nielson] Nielson Games – Gameplay Metrics, http://www.nielson.com

[oculus] Occulus VR, Irvine, CA, http://www.oculusvr.com

[odlyzko] A. M. Odlyzko, Internet Traffic Growth: Sources and Implications, Optical Transmission Systems and Equipment for WDM Networking II, B. B. Dingel, W. Weiershausen, A. K. Dutta, and K.-I. Sato, eds., Proceedings of SPIE, volume 5247, pages 1-15, 2003.

[opensim] OpenSimulator, http://opensimulator.org/wiki/Main_Page

[partridge] C. Partridge, P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. Troxel, D. Waitzman, and S. Winterble. A 50-Gb/s IP Router. IEEE/ACM Transactions on Networking (TON), volume 6, issue 3, pages 237–245, June 1998.

[patro] A. Patro, S. Rayanchu., M. Griepentrog, Y. Ma, and S. Banerjee. The Anatomy of a Large Mobile Massively Multiplayer Online Game. ACM SIGCOMM Computer Communication Review, 42(4), 479-484, 2012.

[pouderoux] J. Pouderoux, J. Marvie. Adaptive Streaming and Rendering of Large Terrains Using Strip Masks. In Proceedings of ACM GRAPHITE 2005, November 2005.

[presetya] K. Prasetya and Z. Wu. Performance Analysis of Game World Partitioning Methods for Multiplayer Mobile Gaming. In Proceedings of the Workshop on Network and Systems Support for Games (NetGames), October 2008.

[qiu] L. Qiu., G. Varghese, S. Suri. Fast Firewall Implementations for Software and Hardware-Based Routers. In Proceedings of ACM SIGMETRICS 2001, Cambridge, Mass, USA, June 2001.

[quax] P. Quax, J. Dierckx, B. Cornelissen, G. Vansichem, and W. Lamotte. Dynamic Server Allocation in a Real-Life Deployable Communications Architecture for Networked Games. In Proceedings of the Workshop on Network and Systems Support for Games (NetGames), October 2008.

[raman] S. Raman, H. Balakrishnan, M. Srinivasan. An Image Transport Protocol for the Internet. In Proceedings of the International Conference on Network Protocols, November 2000.

[reddy] M. Reddy, Y. Leclerc, L. Iverson, N. Bletter. TerraVision II: Visualizing Massive Terrain Databases in VRML. In IEEE Computer Graphics and Applications, vol. 19, no. 2, pp. 30-38, 1999.

[reshetov] A. Reshetov, A. Suoupikov and J. Hurley. Multi-Level Ray Tracing Algorithm. ACM Transactions on Graphics (TOG) – Proceedings of ACM SIGGRAPH 2005, volume 24, issue 3, pages 1176-1185, 2005.

[rosedale] P. Rosedale, C. Ondrejka C. Enabling Player-Created Online Worlds with Grid Computing and Streaming. Gamasutra, September 2003.

[rubin] S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In Computer Graphics (Proceedings of SIGGRAPH 80) vol. 14, pp.110-116, 1980.

[saltzer] J.H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. In Proceedings of the Second International Conference on Distributed Computing Systems. Paris, France. April 8–10, 1981. IEEE Computer Society, pp. 509-512, 1981.

[sanchez] Sanchez, L., W. Milliken, A., Snoeren, F. Tchakountio, C. Jones, S. Kent, C. Partridge, and W. Strayer. Hardware Support for a Hash-Based IP Traceback. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition, June 2001.

[schroeder] Schroeder, William J., Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. In Proceedings of ACM SIGGRAPH '92 Proceedings of the 19th Anual Conference on Computer Graphics and Interactive Techniques, pages 65-70, 1992.

[sha] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995

[shaffer] Shaffer, C.A., Juvvadi, R., and Heath, L.S. A Generalized Comparison of Quadtree and Bintree Storage Requirements. Image and Vision Computing 11, 7 , pages 402–412, 1993.

[sl] Second Life, www.secondlife.com

[snoeren] A.C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, Kent, S.T., and W. T. Strayer. Hash-based IP Traceback. In Proceedings of the ACM SIGCOMM 2001 Conference, volume 31:4 of Computer Communication review, pages 3-14, August 2001.

[srinivasan] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and Scalable Layer Four Switching. In Proceedings of ACM SIGCOMM'98, pages 191–202, September 1998.

[stoica] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks. In Proceedings of ACM SIGCOMM, September 1998.

[stone] Stone, J., Partridge, C. When the CRC and TCP Checksum Disagree, In Proceedings of the ACM SIGCOMM 2000 Conference (SIGCOMM-00), pages 309-319, August 2000.

[strassburger] S. Strassburger, T. Schulze, and R. Fujimoto. Future Trends in Distributed Simulation and Distributed Virtual Environments. Peer Study Final Report. Version 1.0. Ilmenau, Magdeburg, Atlanta, 2008.

[thompson] K.Thompson, G.Miller, and R.Wilder. Wide-area Internet Traffic Patterns and Characteristics. IEEE Network, 1997.

[trammell] B. Trammell and D. Schatzmann. On Flow Concurrency in the Internet and its Implications for Capacity Sharing. In Proceedings of the 2012 ACM workshop on Capacity sharing. ACM, 2012.

[tsai] F. Tsai, H. Liu, J. K. Liu, K. H. Hsiao. Progressive Streaming and Rendering of 3D Terrain for Cyber City Visualization. In Proc. 27th Asian Conference on Remote Sensing, October 2006.

[turner] B. Turner. Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. http://www.gamasutra.com/features/20000403/turner_01.htm

[uo] Ultima Online, http://www.uo.com

[usgs] USGS and C. McCabe. Grand Canyon Terrain. www.cc.gatech.edu/projects/large_models/gcanyon.html

[varghese] G. Varghese. Detecting Packet Patterns at High Speeds. SIGCOMM Tutorial, August 2002.

[varvello] Matteo Varvello, Stefano Ferrari, Ernst Biersack, Christoph Diot. Exploring Second Life. IEEE/ACM Transactions on Networking (TON), volume 19 issue 1, February 2011.

[wachter] C.Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In Proceedings of the Eurographics Symposium on Rendering, pages 139--149, 2006.

[waldvogel] M. Waldvogel, G. Varghese, J. Turner and B. Plattner. Scalable High Speed IP Routing Lookups. In Proceedings of SIGCOMM '97, pp. 25–36, September 1997.

[woodcock] B. Woodcock. Total MMOG Active Subscriptions. http://www.mmogchart.com/Chart4.html

[wow] World of Warcraft. Blizzard Entertainment, Irvine, CA. http://www.worldofwarcraft.com

[wwind] NASA World Wind. worldwind.arc.nasa.gov

[xu] J. Xu, M. Singhal, and J. Degroat. A Novel Cache Architecture to Support Layer-Four Packet Classification at Memory Access Speeds. In Proceedings of INFOCOM, pages 1445–1454, 2000.

[yang] S. Yang, C.S. Kim, and C.-C. Jay Kuo.View-Dependent Progressive Mesh Coding for Graphic Streaming. In Proceedings of SPIE Vol. 4518, Multimedia Systems and Applications IV, pages 154-165, November 2001.

[yang2] Y. Xiao, J. Yuan and D. Thalmann. Human-Virtual Human Interaction by Upper Body Gesture Understanding . In Proceedings of the VRST '13 19th ACM Symposium on Virtual Reality Software and Technology, pages 133-142, 2013.

Appendix A Virtual World Simulation Workload Summary

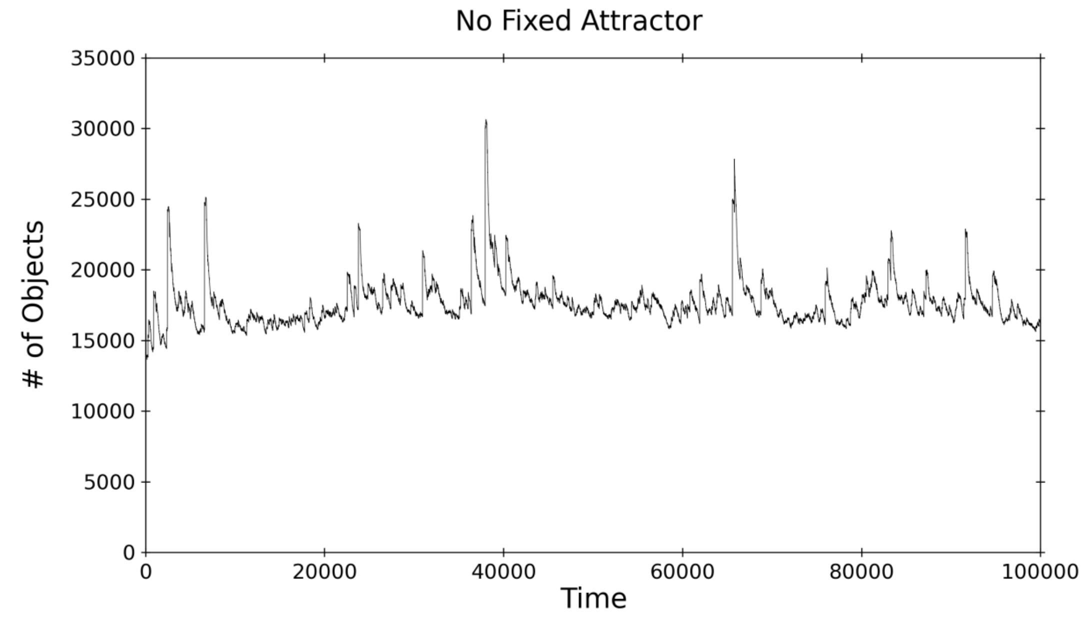


Figure A.1: Number of active objects during No Fixed Attractor workload.

Statistic	Min	Max	Avg	Harmonic Mean	Standard Deviation
# of objects	13684	30669	17706.1	568933.2	1651.2
# of new objects	5	12630	14.8	1284393533.3	84.8
# of deleted objects	1	184	14.7	758495173.3	5.2
Max spatial locality score	1248.6	8607.5	2043.4	5177924.0	625.7

Table A.1: Key statistical summary for No Fixed Attractor workload.

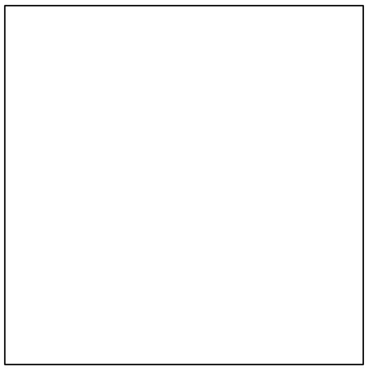


Figure A.2: Locations of fixed attractors in the square-shaped virtual world in the No Fixed Attractor workload.

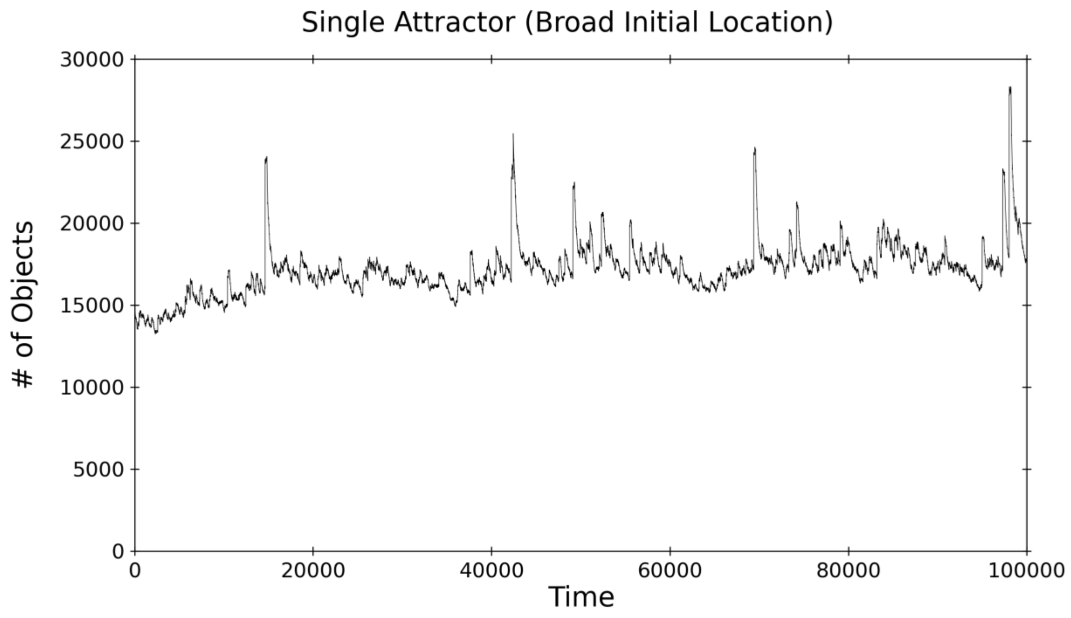


Figure A.3: Number of active objects during simulation run. Minimum: 13255, Maximum: 28344, Average: 17203, Harmonic Mean: 585885, Standard Deviation: 1598

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	13255	28344	17203.3	585885.6	1599.0
# of new objects	5	9948	14.4	1283322107.5	70.0
# of deleted objects	1	177	14.4	773096108.9	4.8
Max spatial locality score	2051.4	12356.1	3750.9	2793868.8	922.6

Table A.2: Key statistical summary for Single Attractor (Broad Initial Location) workload.

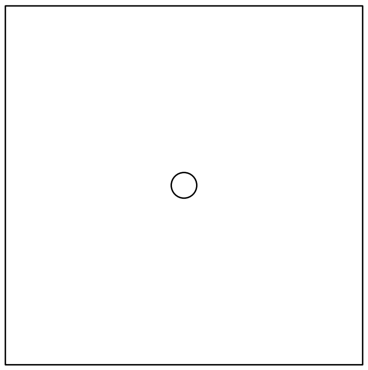


Figure A.4: Locations of fixed attractors in the square-shaped virtual world in the Single Attractor (Broad Initial Location) workload.

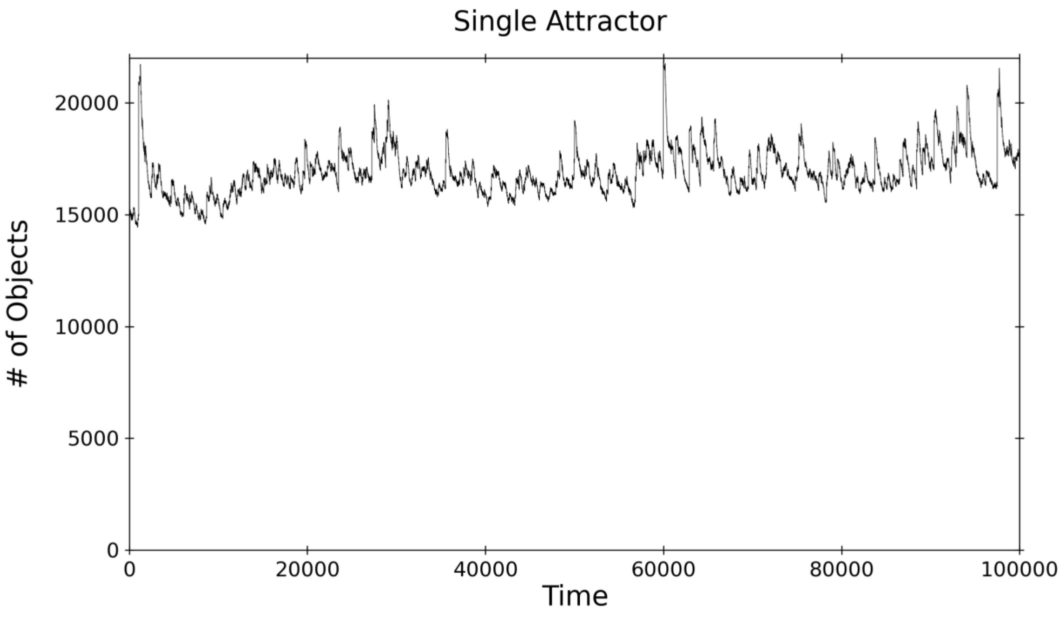


Figure A.5: Number of active objects during simulation run. Minimum: 14454, Maximum: 21865, Average: 16932, Harmonic Mean: 592479, Standard Deviation: 980

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	14454	21865	16932.5	592479.2	980.2
# of new objects	5	5998	14.1	1284046118.0	48.2
# of deleted objects	1	166	14.1	784552211.7	4.2
Max spatial locality score	7246.0	49111.5	15828.6	678272.3	4567.0

Table A.3: Key statistical summary for Single Attractor workload.

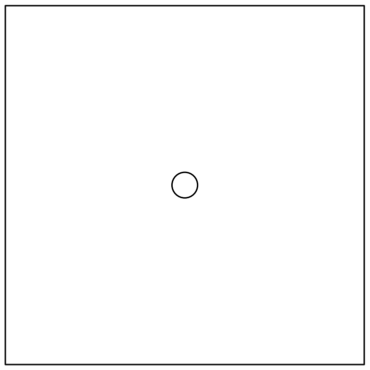


Figure A.6: Locations of fixed attractors in the square-shaped virtual world in the Single Attractor workload.

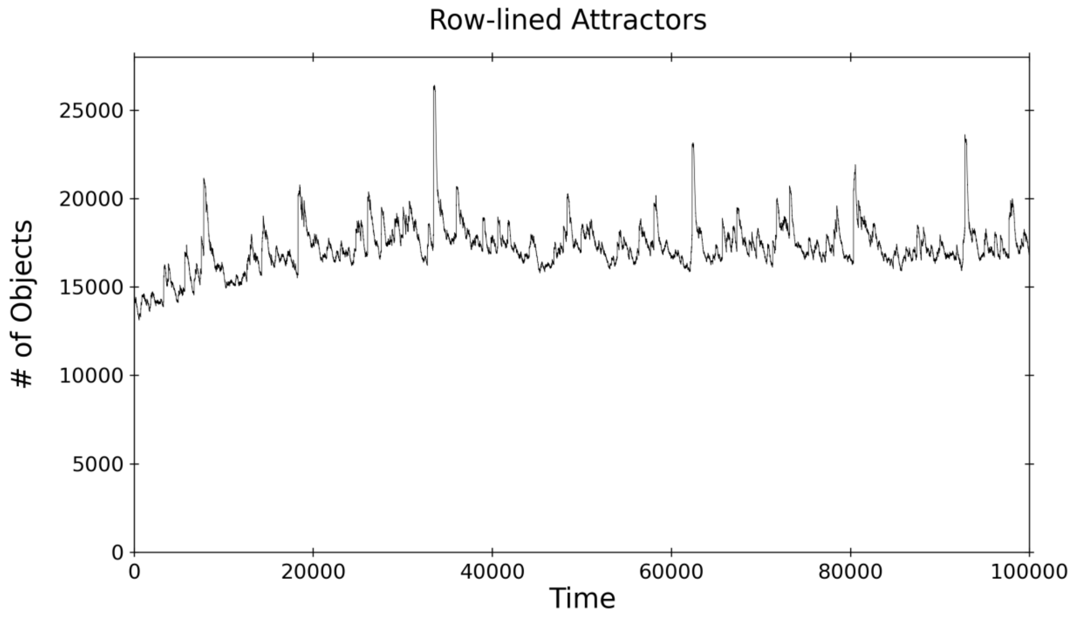


Figure A.7: Number of active objects during simulation run. Minimum: 13147, Maximum: 26430, Average: 17284, Harmonic Mean: 581962, Standard Deviation: 1351

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	13147	26430	17284.7	581962.8	1351.9
# of new objects	5	8411	14.3	1281927759.4	59.7
# of deleted objects	1	162	14.3	773443329.8	4.5
Max spatial locality score	5815.4	43491.6	10599.6	1000589.6	3241.6

Table A.4: Key statistical summary for Row-lined Attractors workload.

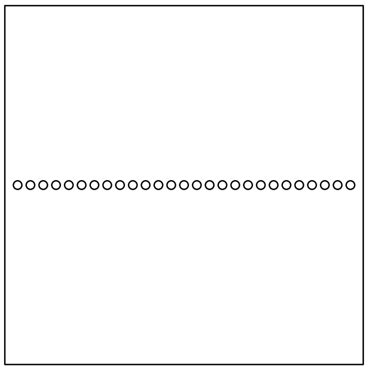


Figure A.8: Locations of fixed attractors in the virtual world in the Row-lined Attractor workload.

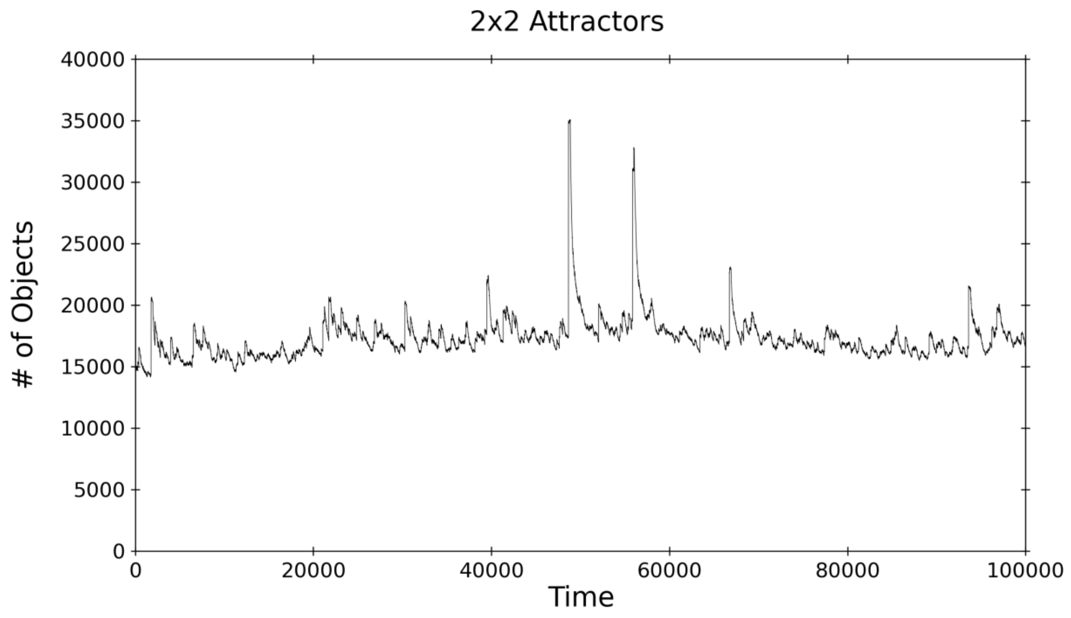


Figure A.9: Number of active objects during simulation run. Minimum: 14171, Maximum: 35115, Average: 17400, Harmonic Mean: 579473, Standard Deviation: 1826

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	14171	35115	17400.7	579473.8	1826.4
# of new objects	5	17508	14.5	1286045852.8	86.5
# of deleted objects	1	164	14.4	772014163.1	5.2
Max spatial locality score	4833.1	59387.7	7780.3	1355710.4	3080.2

Table A.5: Key statistical summary for 2x2 Attractors workload.

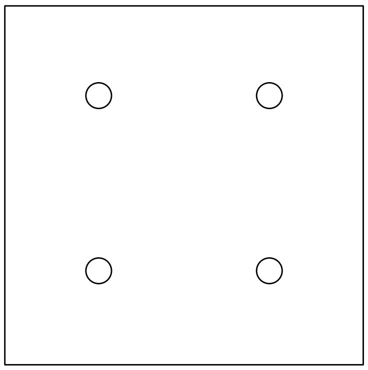


Figure A.10 Locations of fixed attractors in the square-shaped virtual world in the 2x2 Attractor workload.

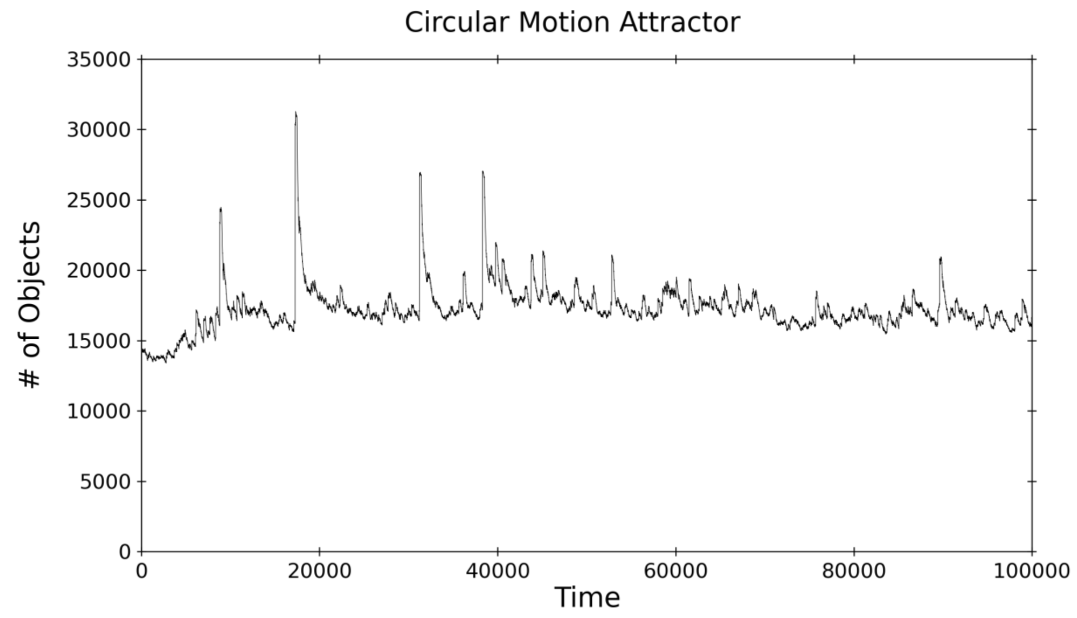


Figure A.11: Number of active objects during simulation run. Minimum: 13427, Maximum: 31314, Average: 17314, Harmonic Mean: 582347, Standard Deviation: 1695

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	13427	31314	17314.0	582347.5	1695.7
# of new objects	5	14035	14.4	1284976804.7	78.6
# of deleted objects	1	171	14.4	775668291.7	5.0
Max spatial locality score	6840.5	59880.9	11854.4	882160.0	3472.3

Table A.6: Key statistical summary for Circular Motion Attractor workload.

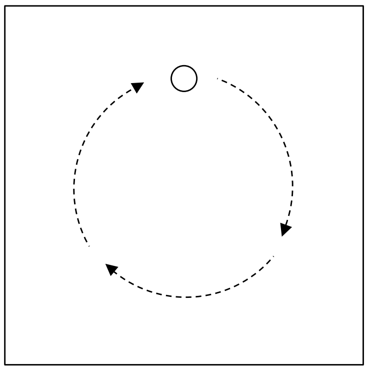


Figure A.12: Locations of fixed attractors in the square-shaped virtual world in the Circular Motion Attractor workload. This single fixed attractor moves in a circular path around the world.

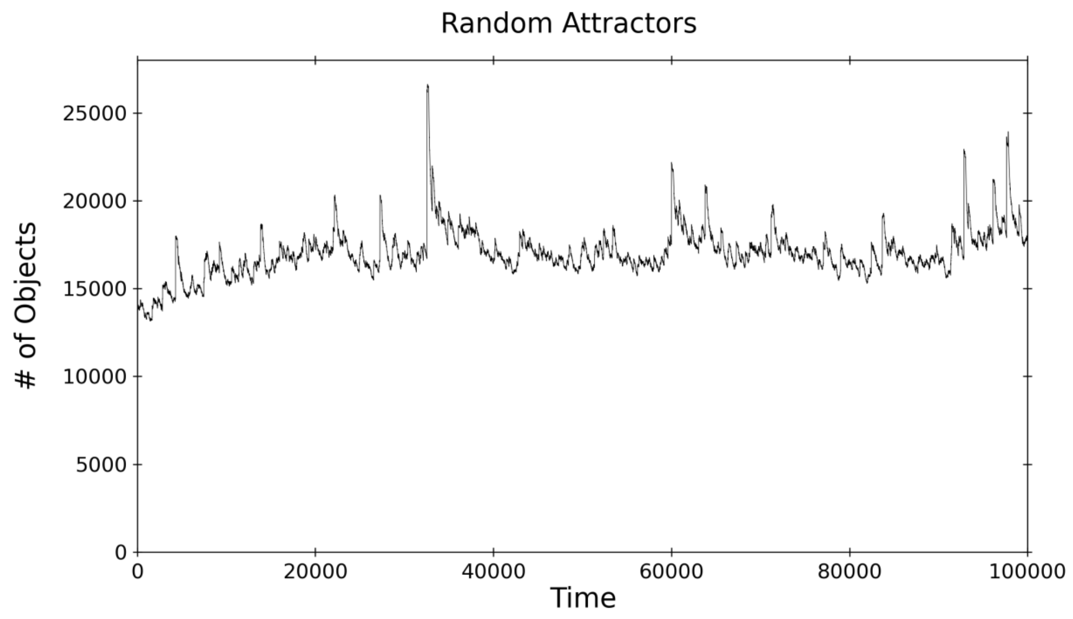


Figure A.13: Number of active objects during simulation run. Minimum: 13170, Maximum: 26649, Average: 17073, Harmonic Mean: 589293, Standard Deviation: 1378

Statistic	**Min**	**Max**	**Avg**	**Harmonic Mean**	**Standard Deviation**
# of objects	13170	26649	17073.5	589293.5	1378.2
# of new objects	5	9650	14.3	1285591932.3	60.7
# of deleted objects	2	172	14.2	778846177.5	4.5
Max spatial locality score	5614.7	50314.6	9745.9	1076399.8	2805.6

Table A.7: Key statistical summary for Random Attractors workload

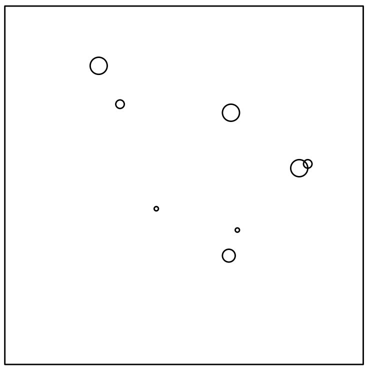


Figure A.14: Location of objects in the Random Attractors workload.

Appendix B Extended Performance Results for Virtual World Simulation

In every graph in this section, each data-line represents a partitioning algorithm with the split constant chosen to be 32 (the maximum number of objects a simulator can manage without becoming overloaded) while varying the merge constant. The choice of merge constant ranges between 2 to 32. The performance metrics reported here are described in detail in Section 4.5.

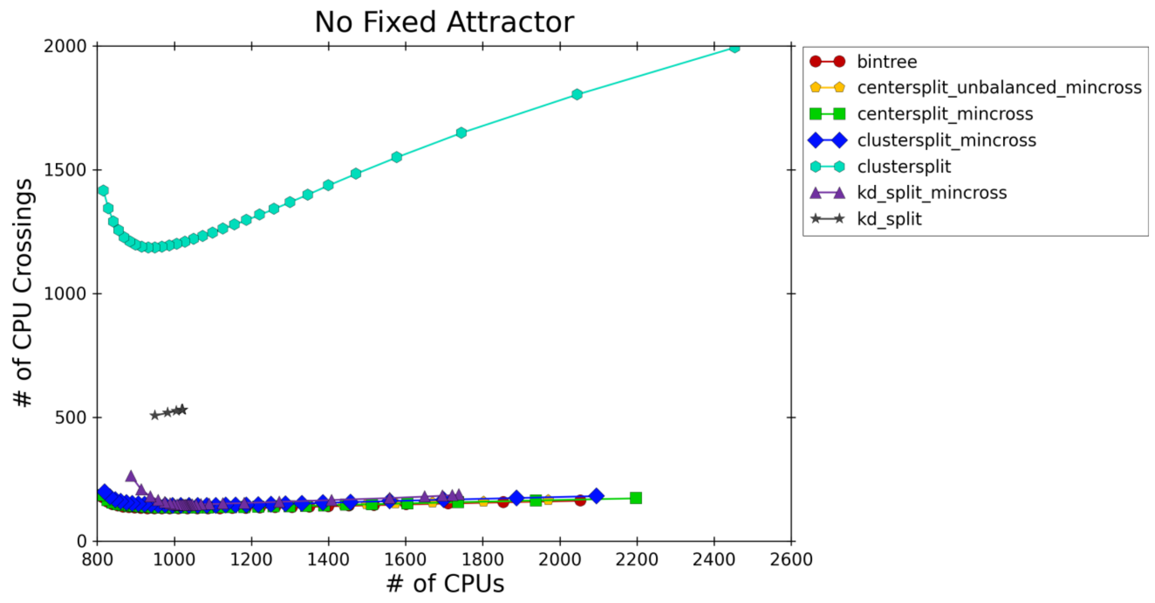


Figure B.1: No fixed attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

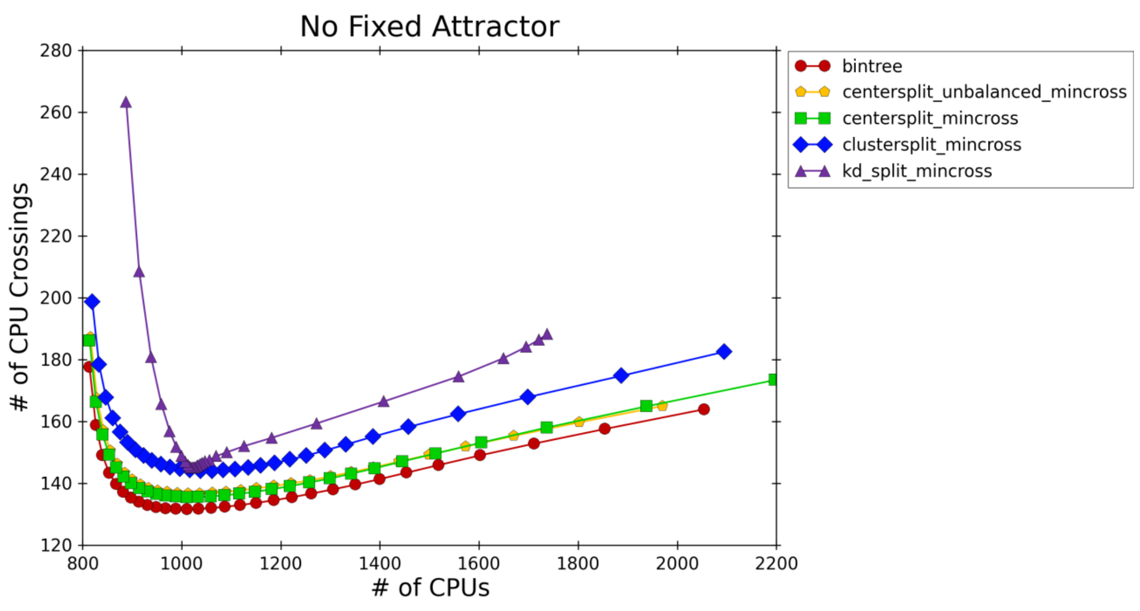


Figure B.2: No fixed attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

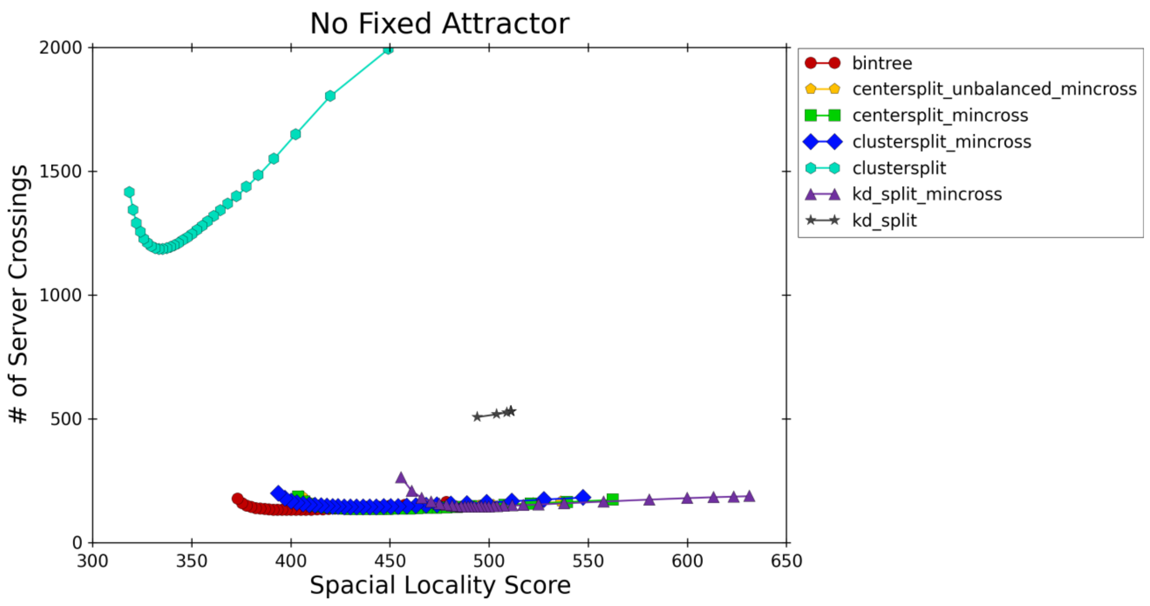


Figure B.3: No fixed attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

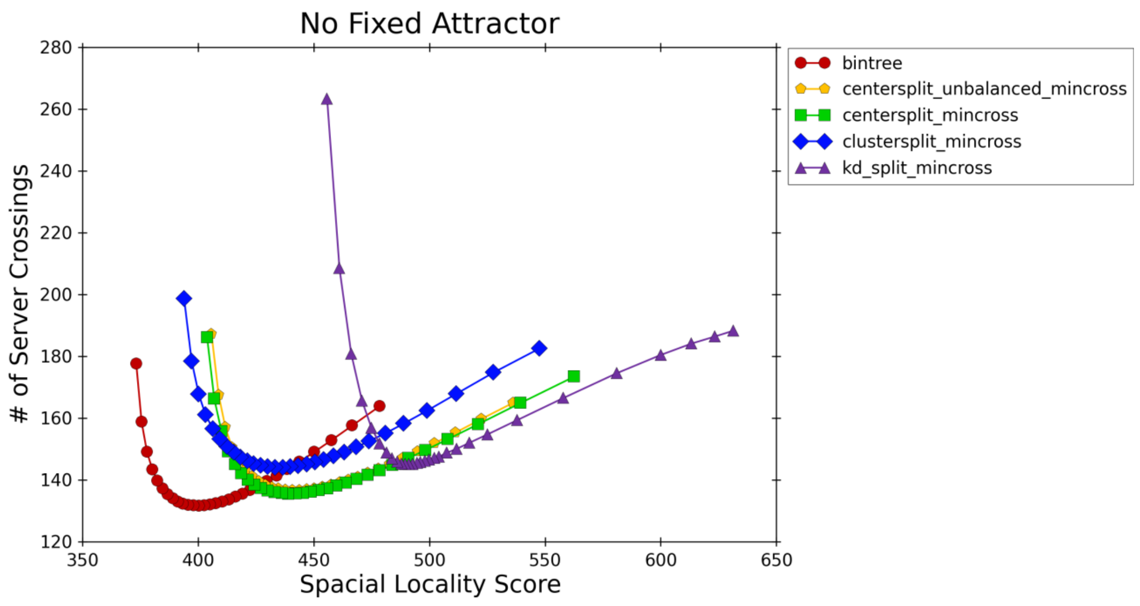


Figure B.4: No fixed attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

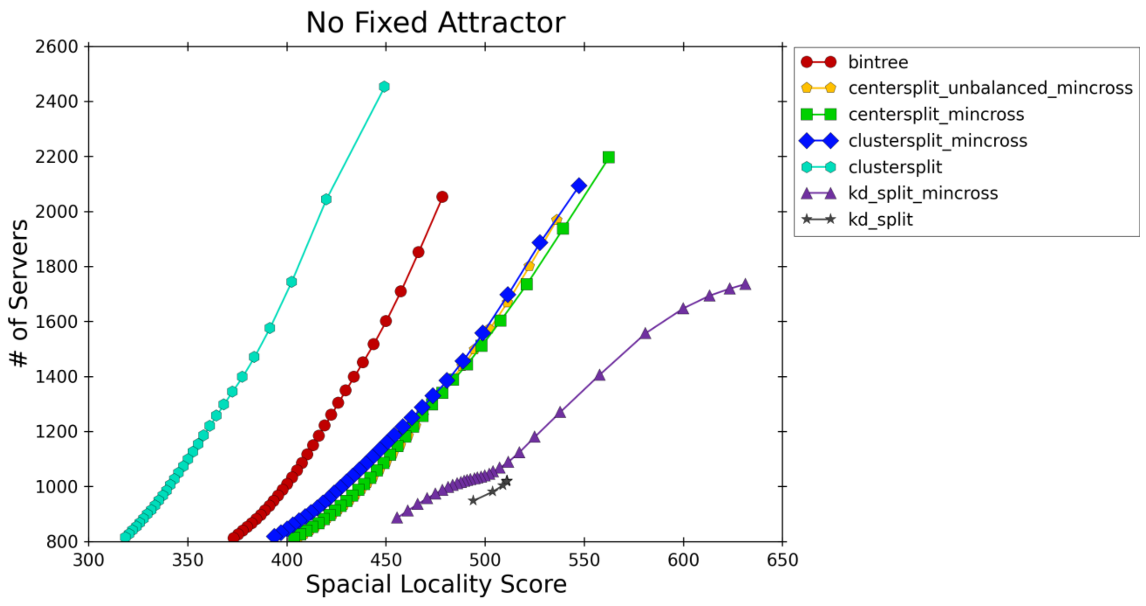


Figure B.5: No fixed attractor workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

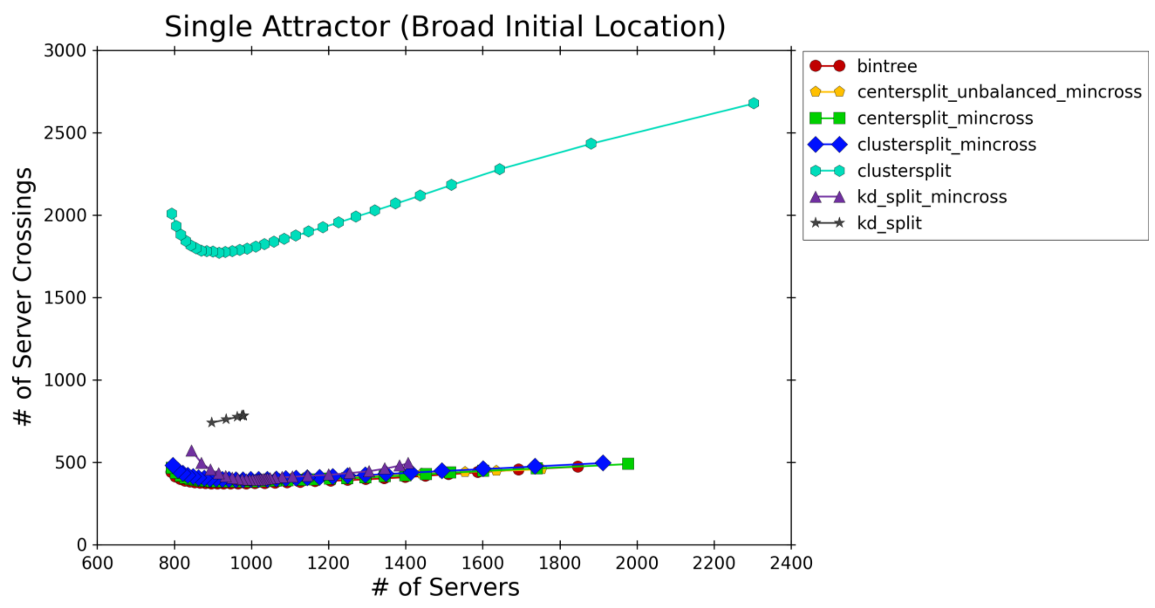


Figure B.6: Single attractor (broad initial location) workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

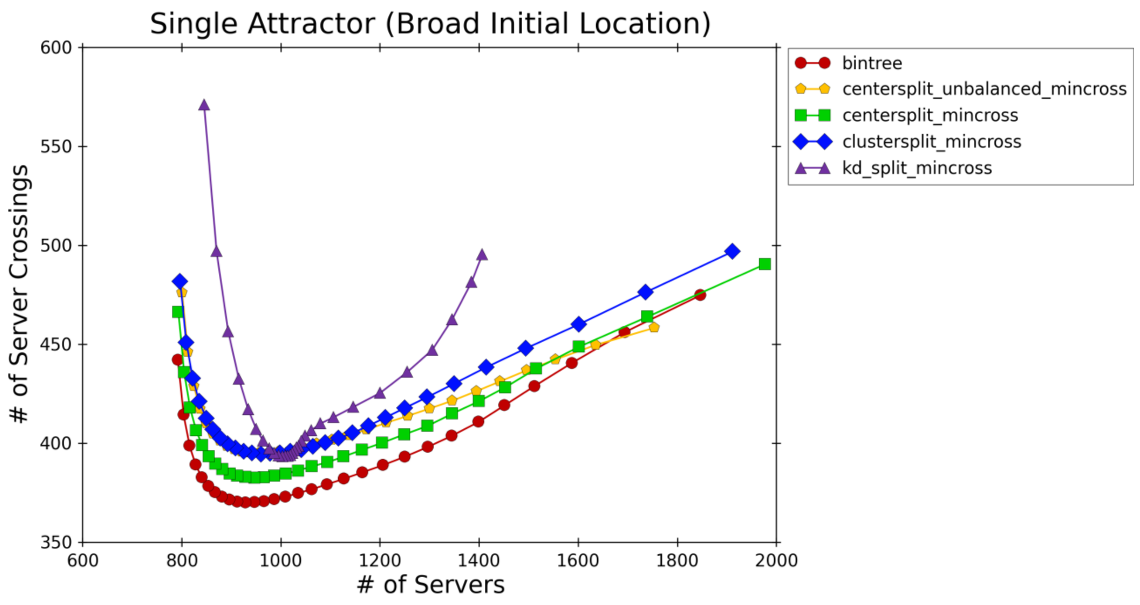


Figure B.7: Single attractor (broad initial location) workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

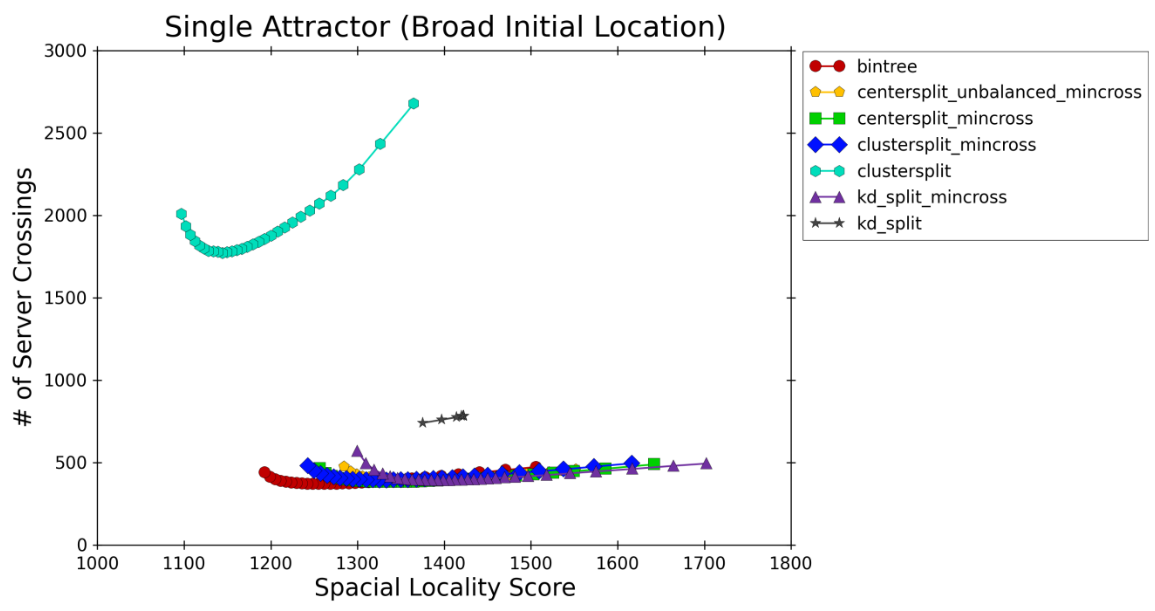


Figure B.8: Single attractor (broad initial location) workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

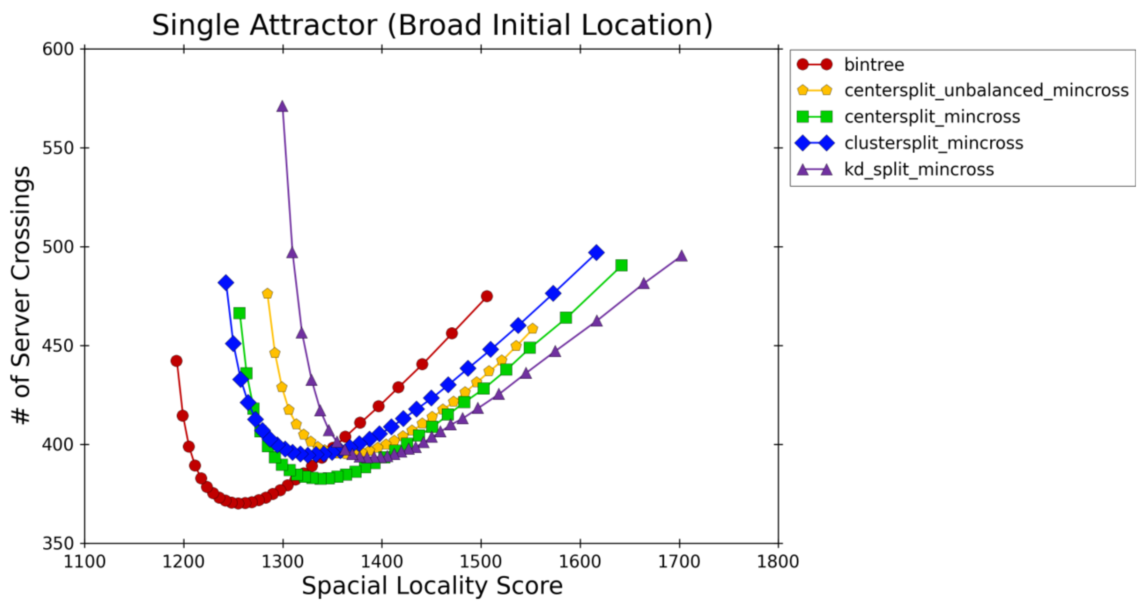


Figure B.9: Single attractor (broad initial location) workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

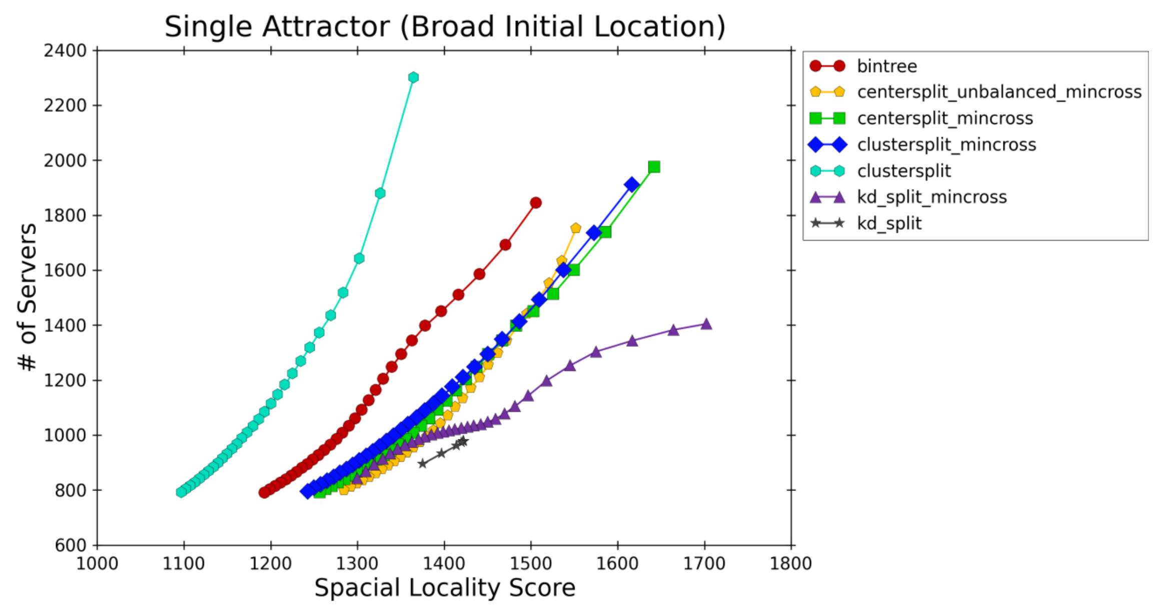


Figure B.10: Single attractor (broad initial location) workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

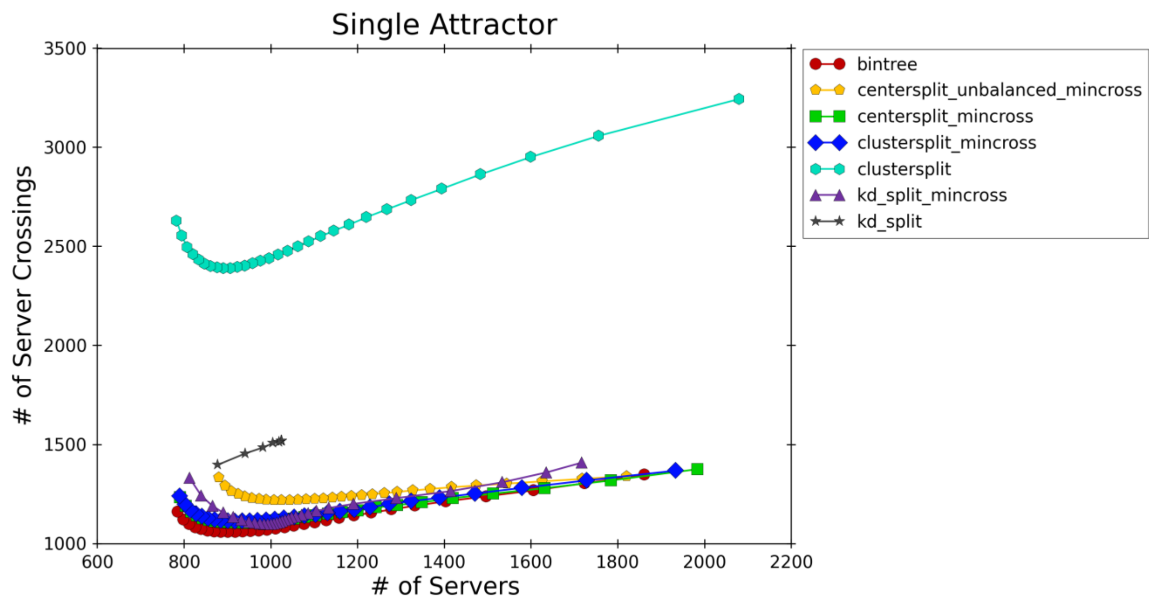


Figure B.11: Single attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

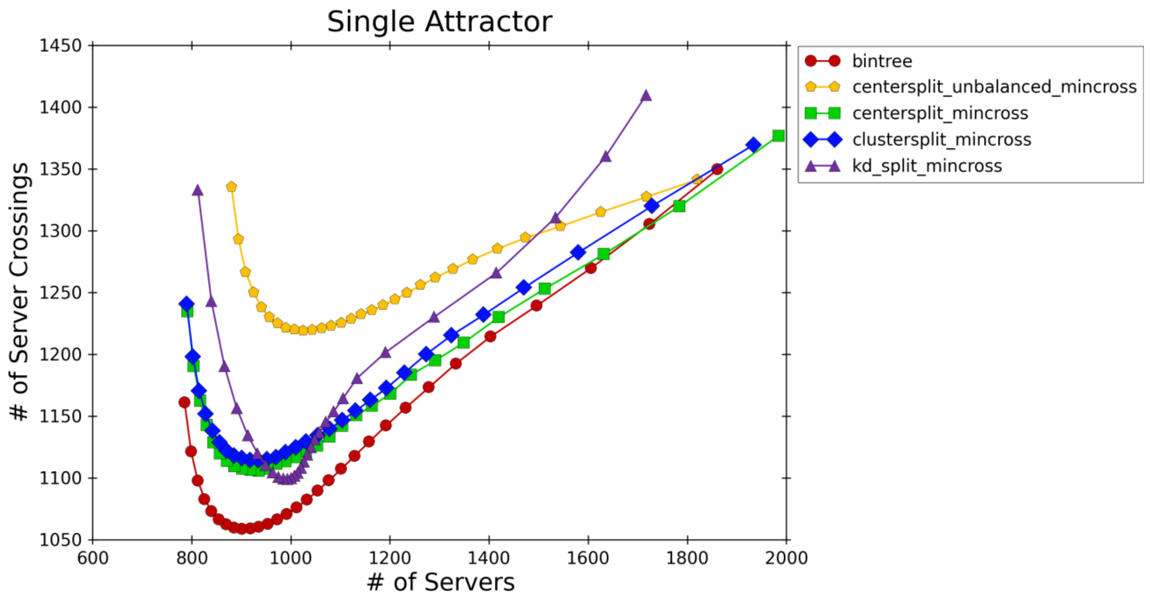


Figure B.12: Single attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

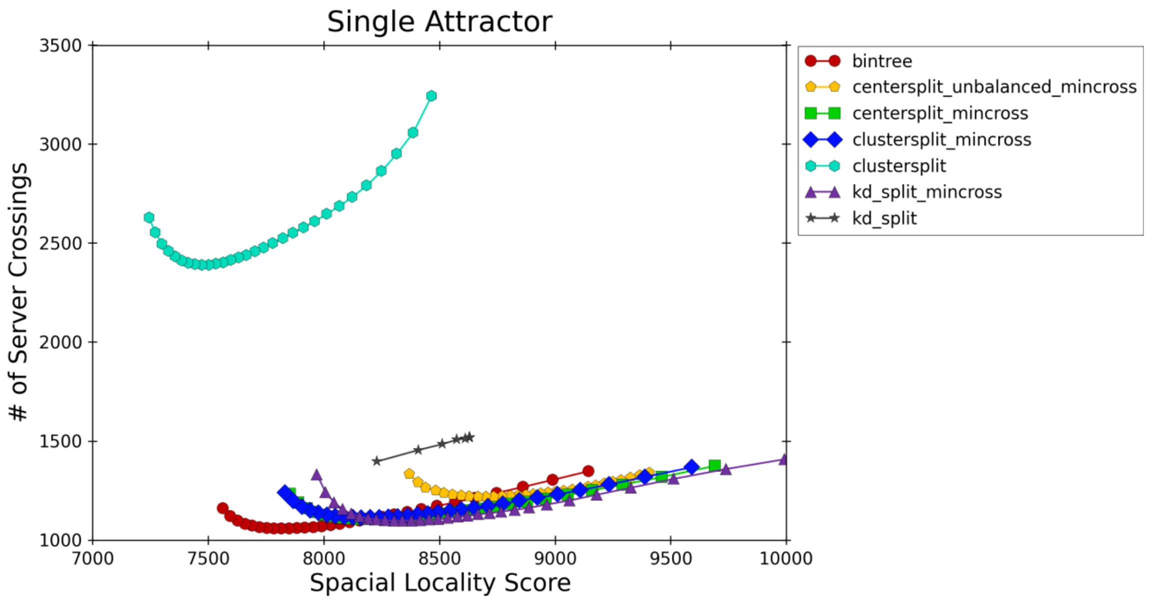


Figure B.13: Single attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

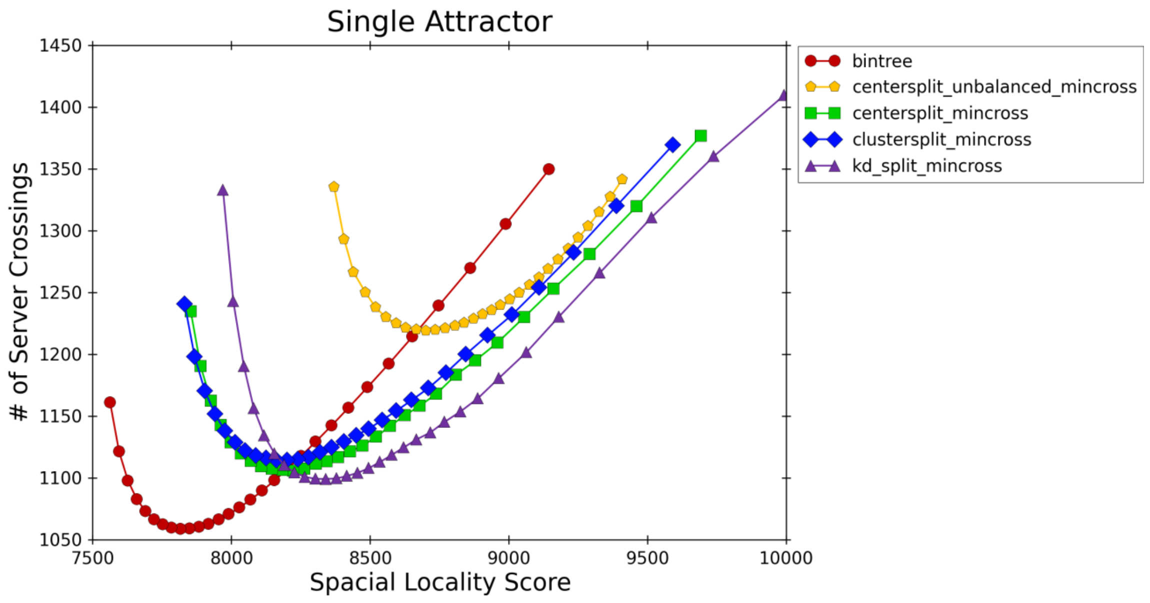


Figure B.14: Single attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

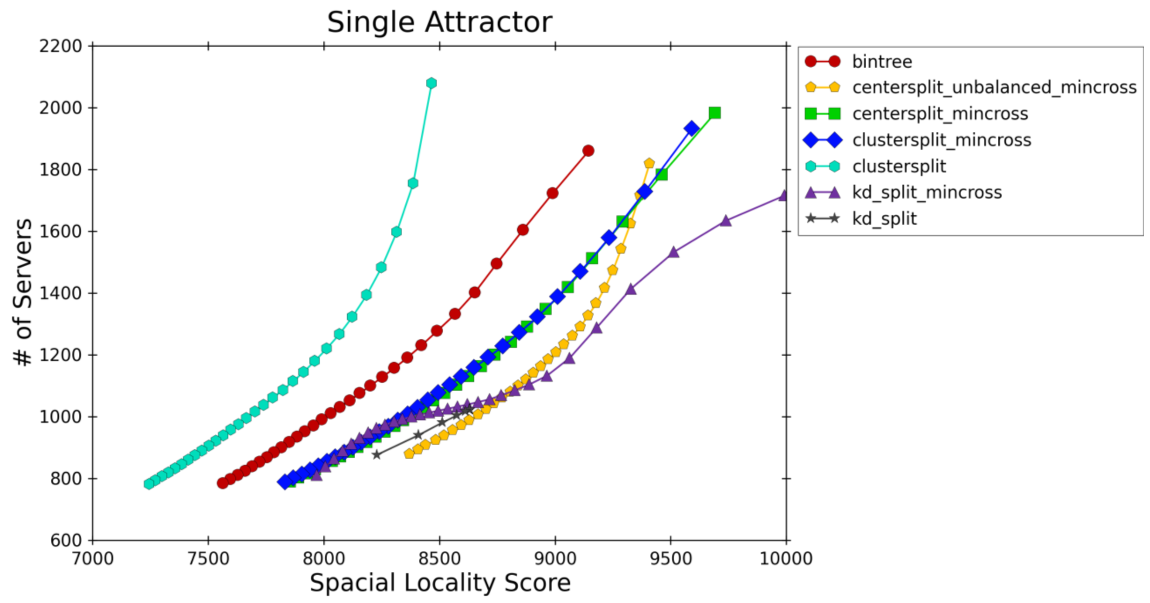


Figure B.15: Single attractor workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

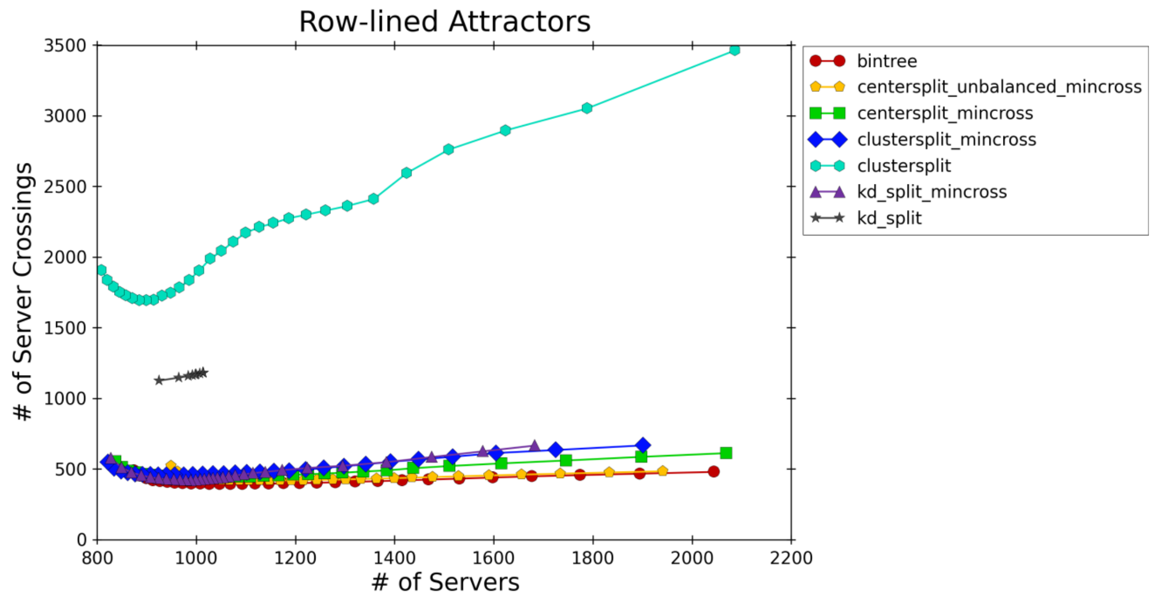


Figure B.16: Row-lined attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

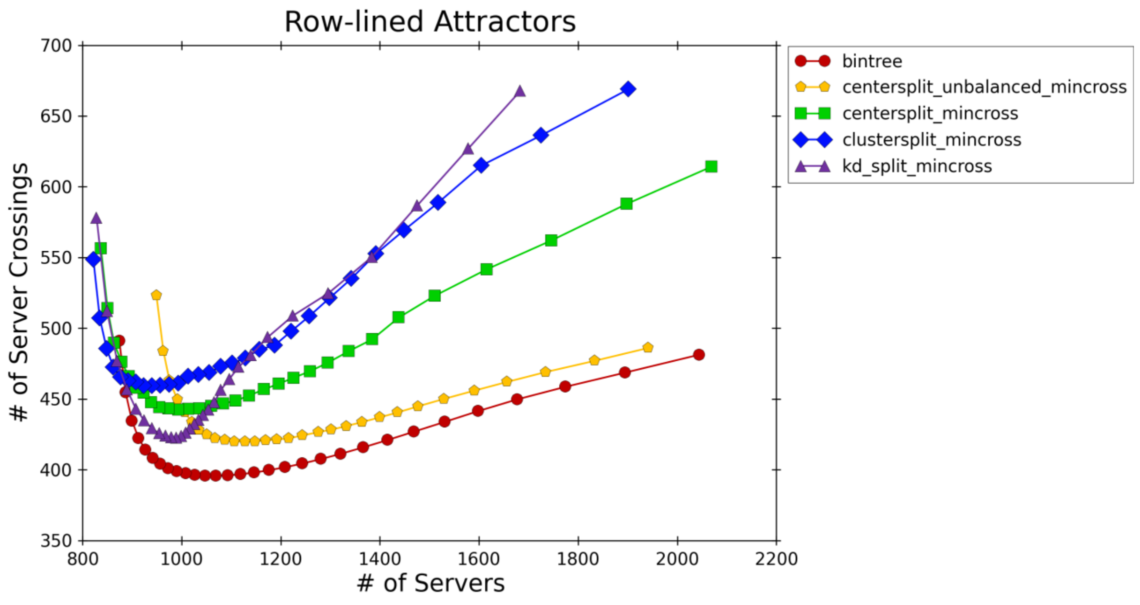


Figure B.17: Row-lined attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

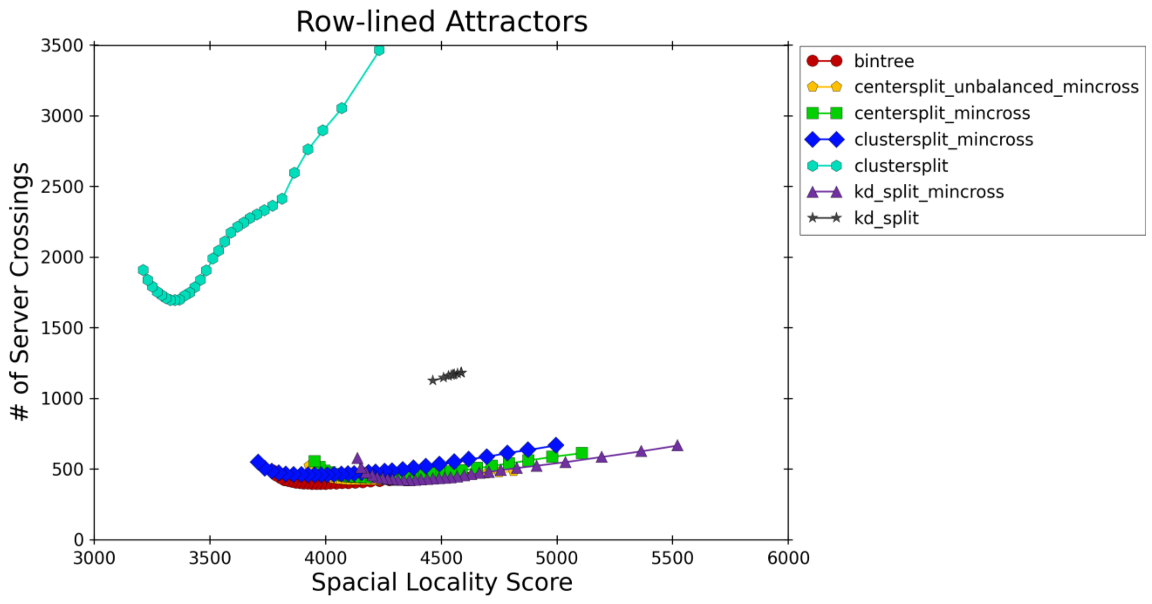


Figure B.18: Row-lined attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

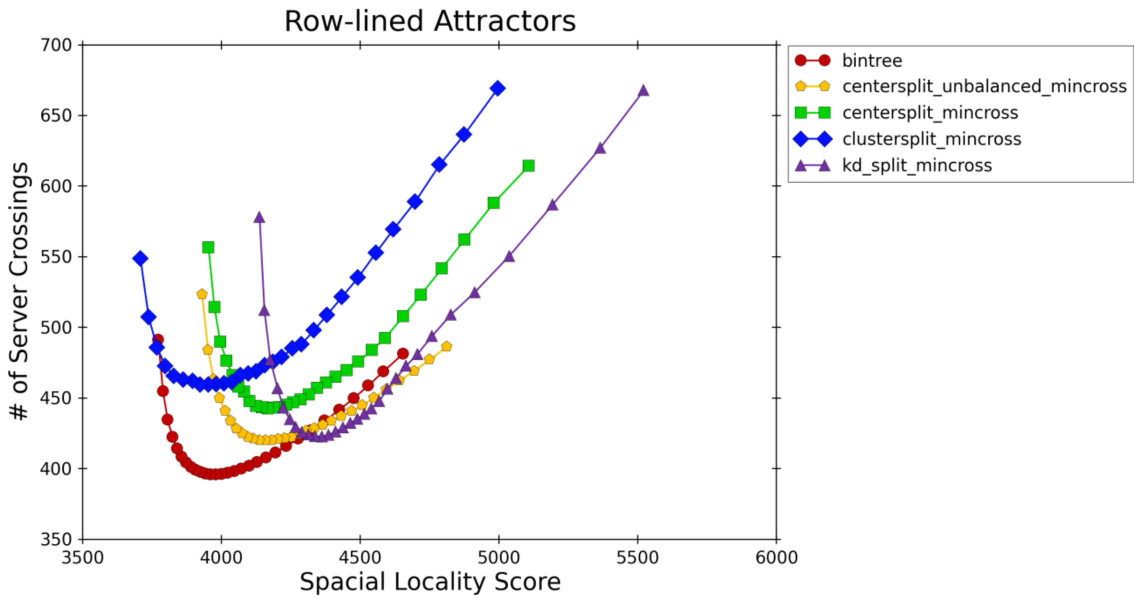


Figure B.19: Row-lined attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

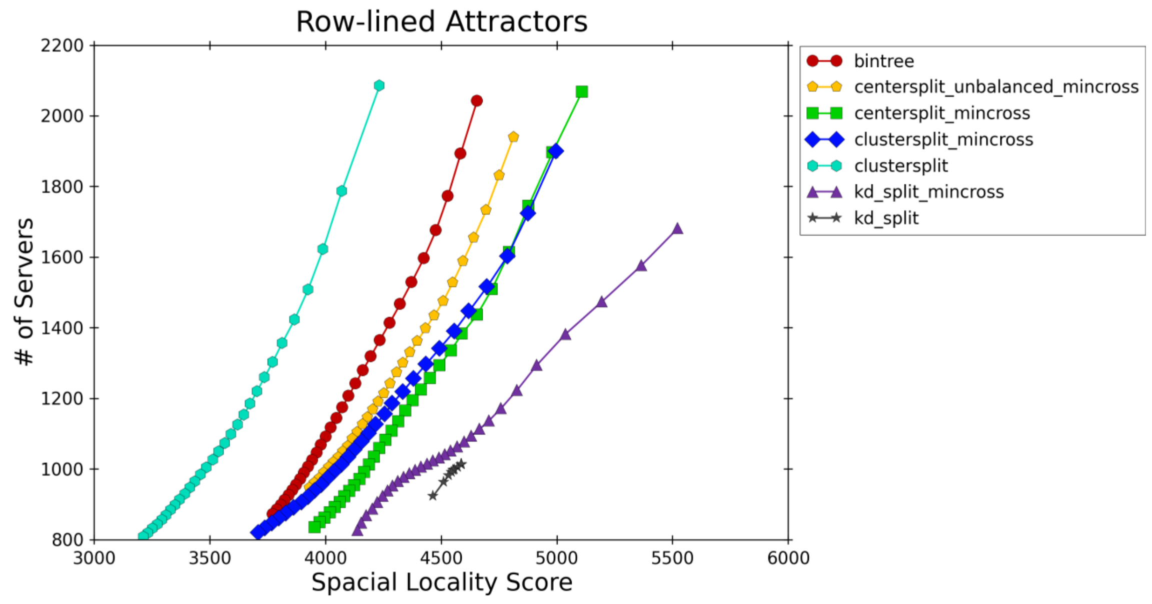


Figure B.20: Single attractors workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

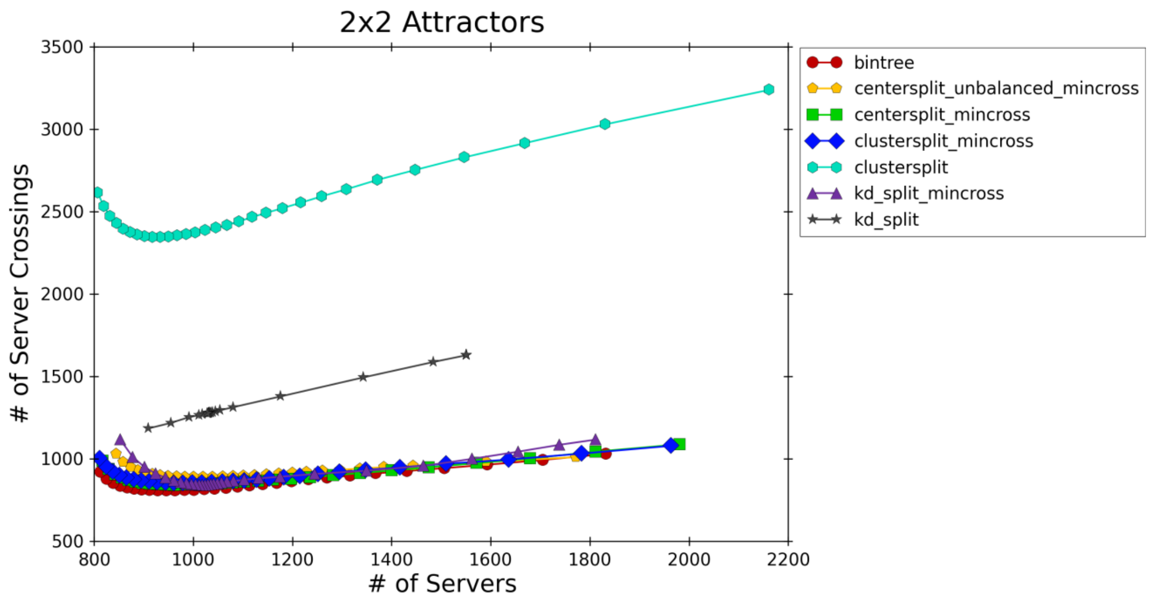


Figure B.21: 2x2 attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

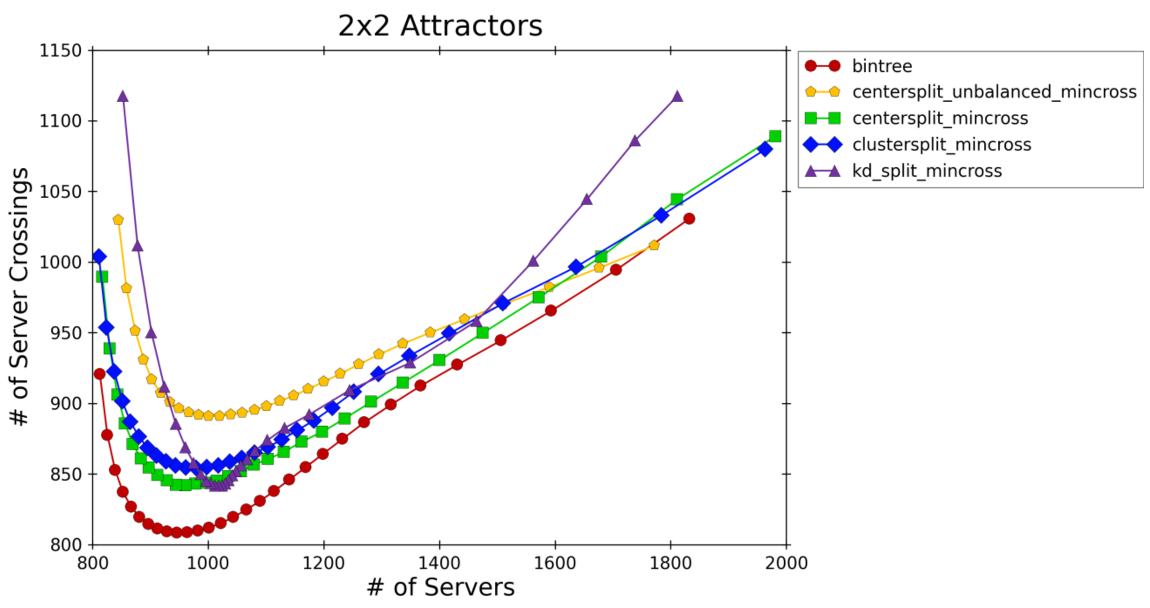


Figure B.22: 2x2 attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

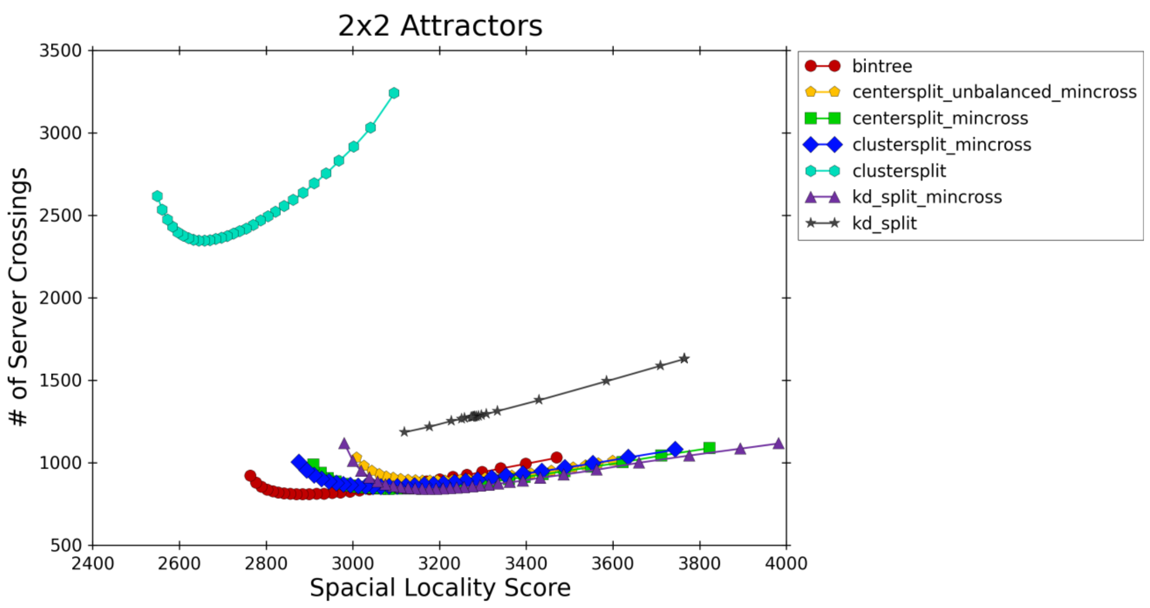


Figure B.23: 2x2 attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

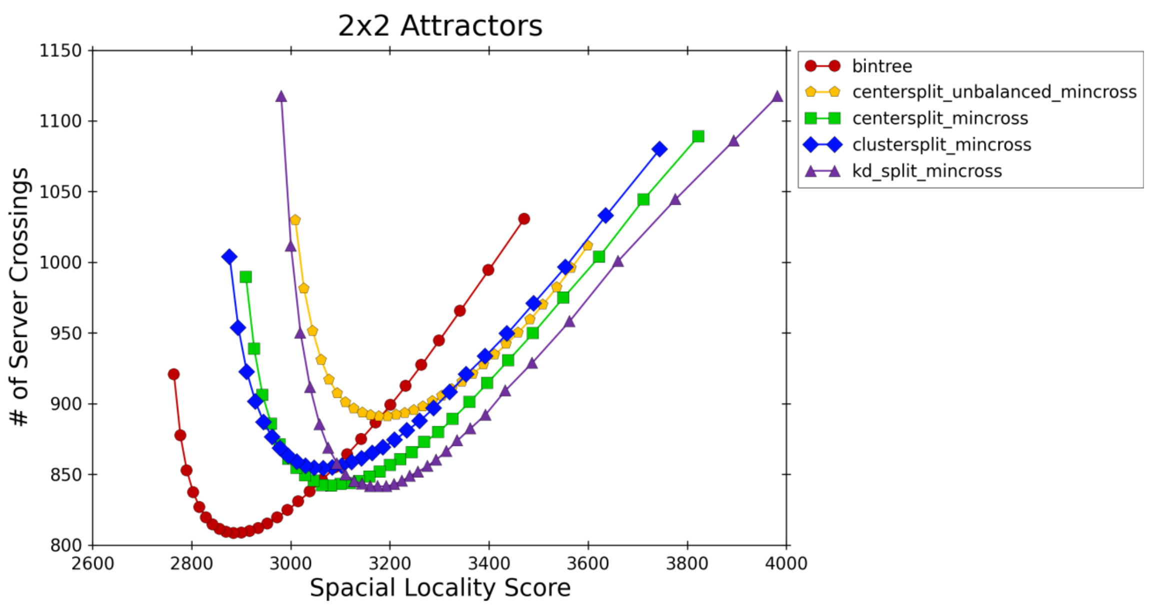


Figure B.24: 2x2 attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

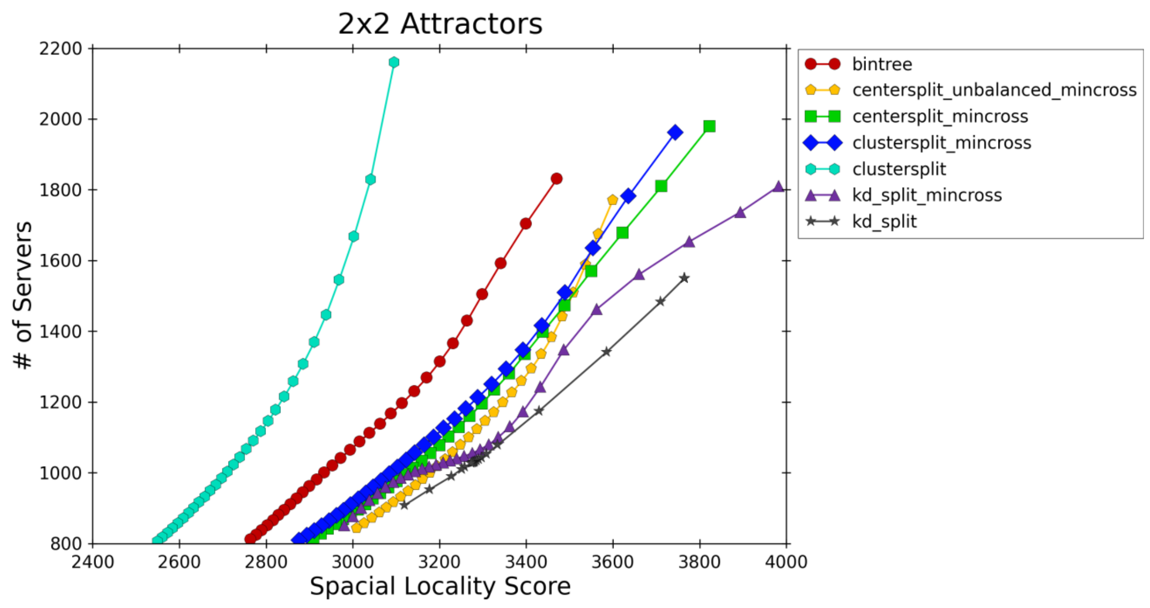


Figure B.25: 2x2 attractors workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

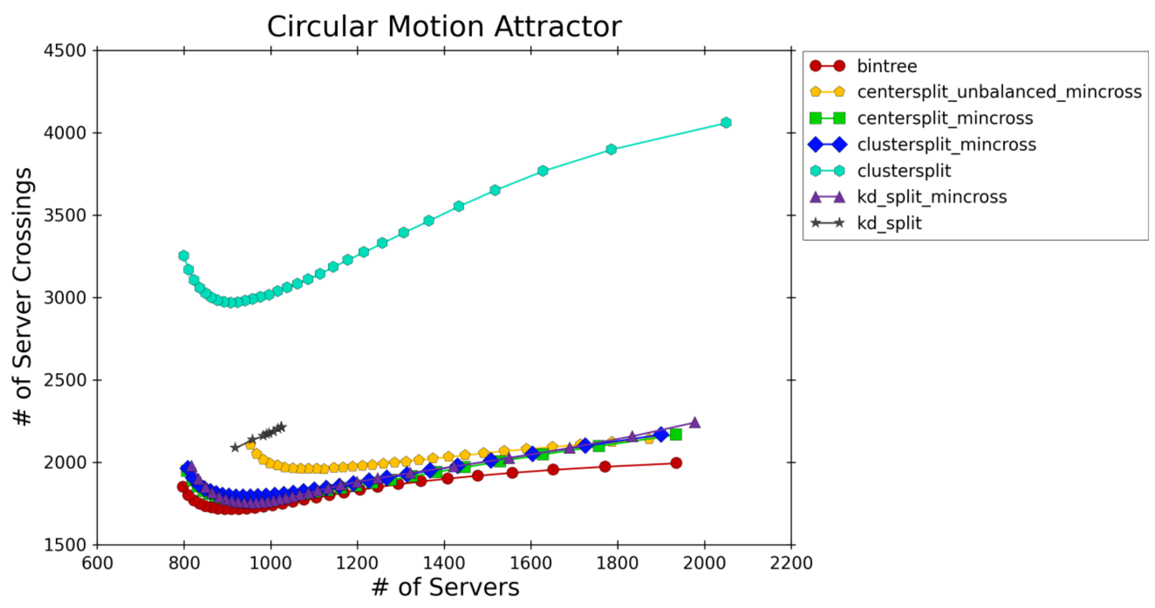


Figure B.26: Circular motion attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

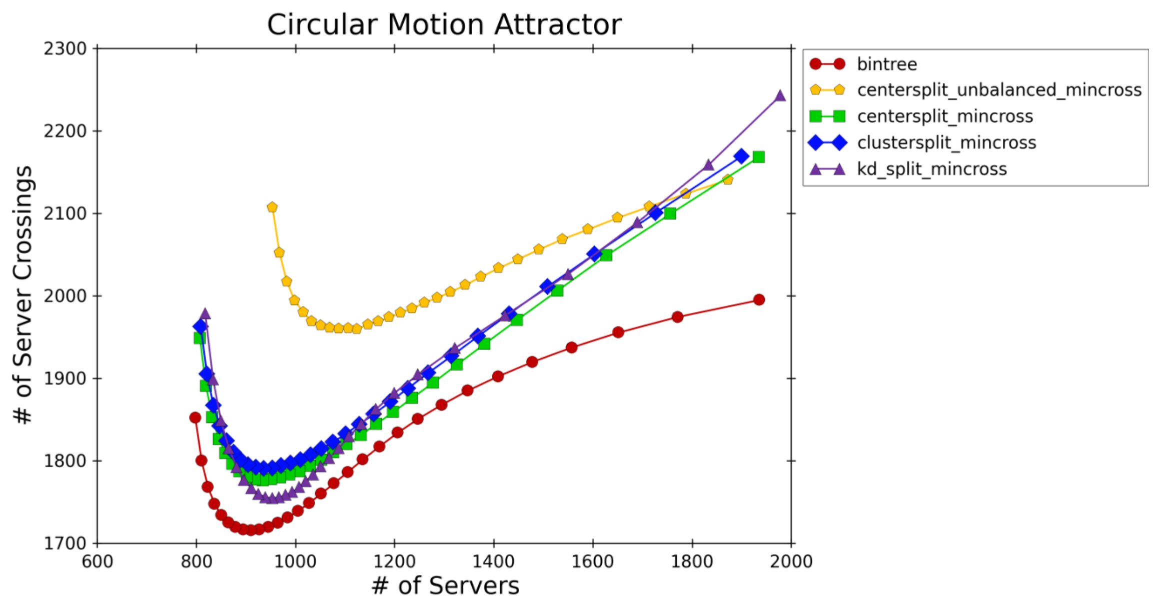


Figure B.27: Circular motion attractor workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

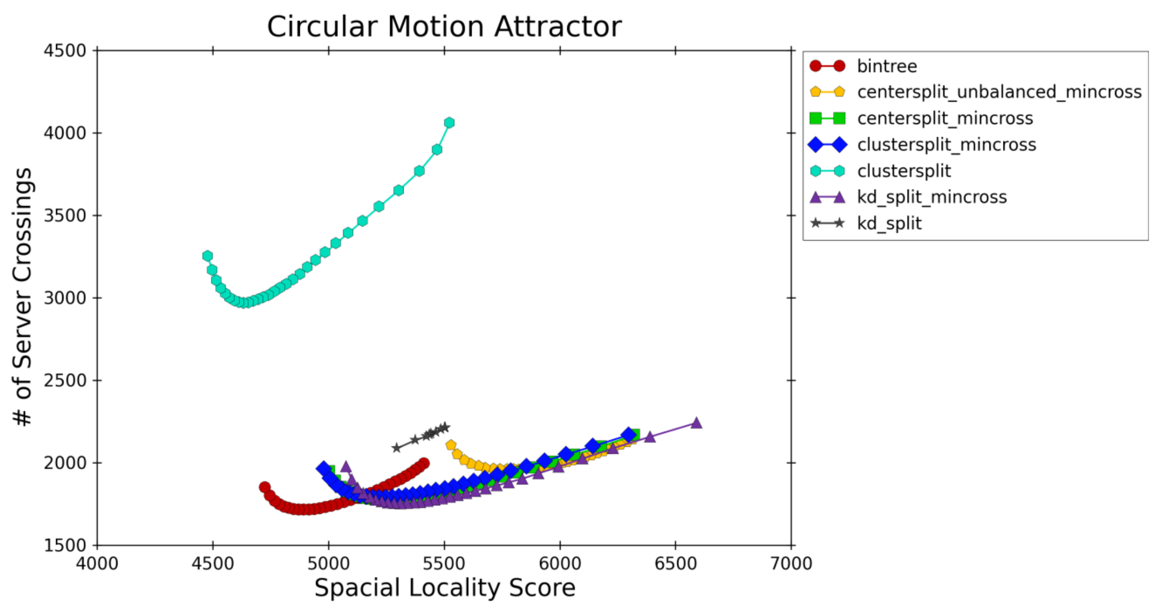


Figure B.28: Circular motion attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

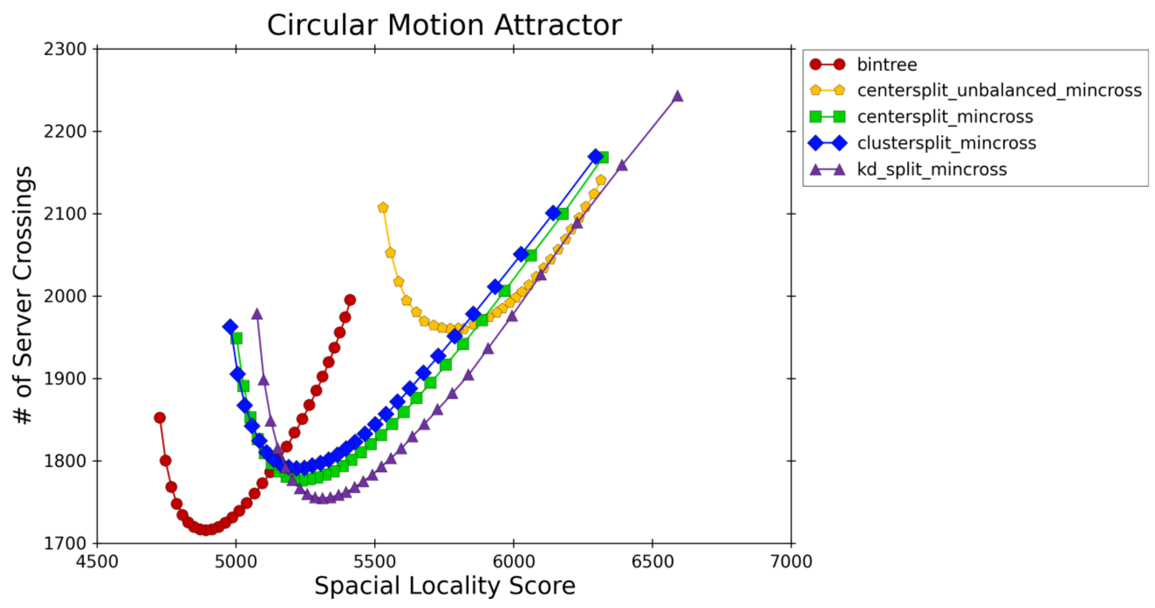


Figure B.29: Circular motion attractor workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

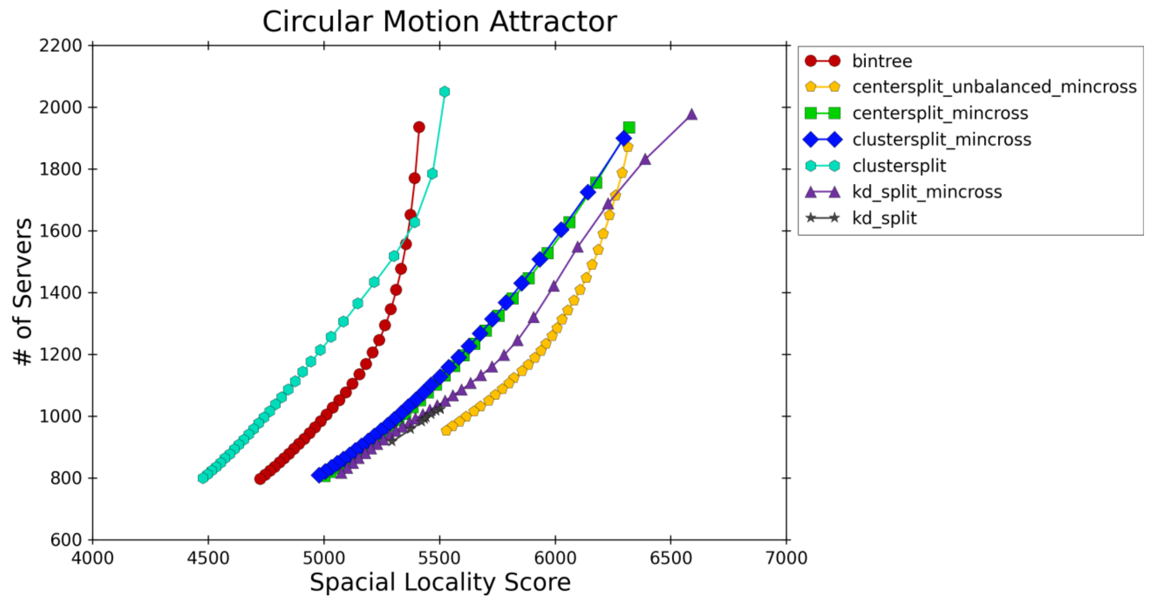


Figure B.30: Circular motion attractor workload. Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

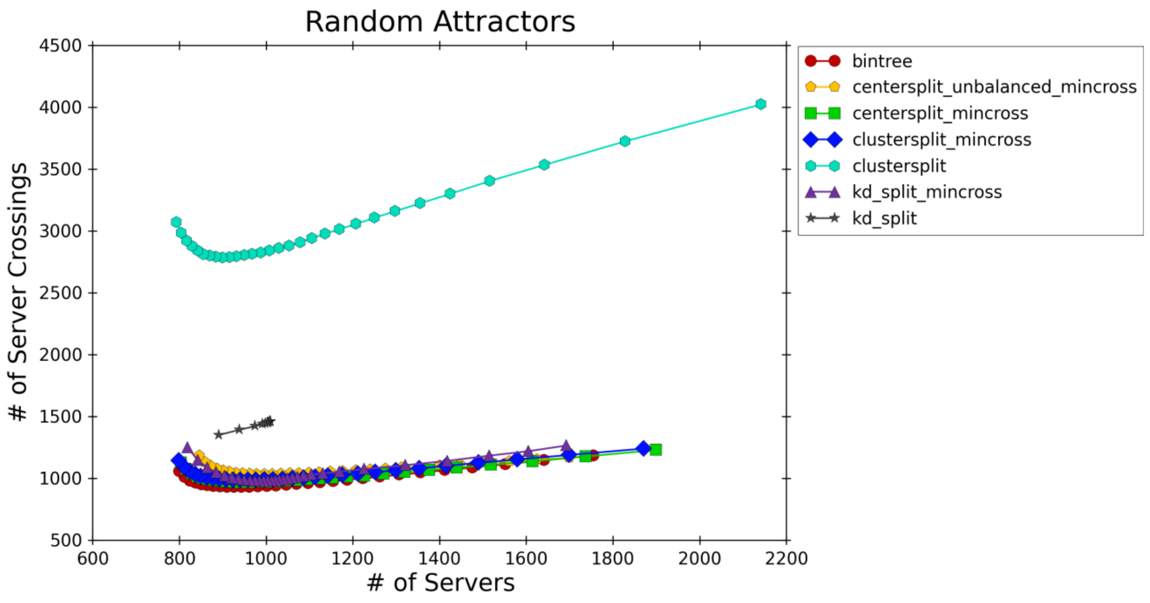


Figure B.31: Random attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

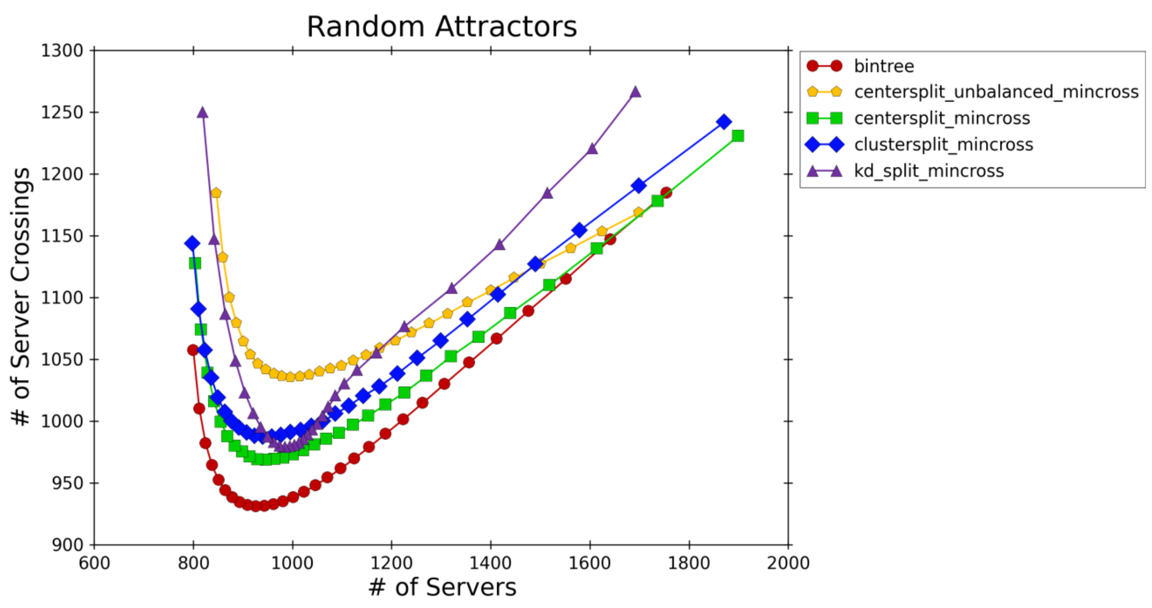


Figure B.32: Random attractors workload. Average number of server crossings per time interval vs. the average number of servers required to manage the simulation. Better performance is towards the bottom left on this graph.

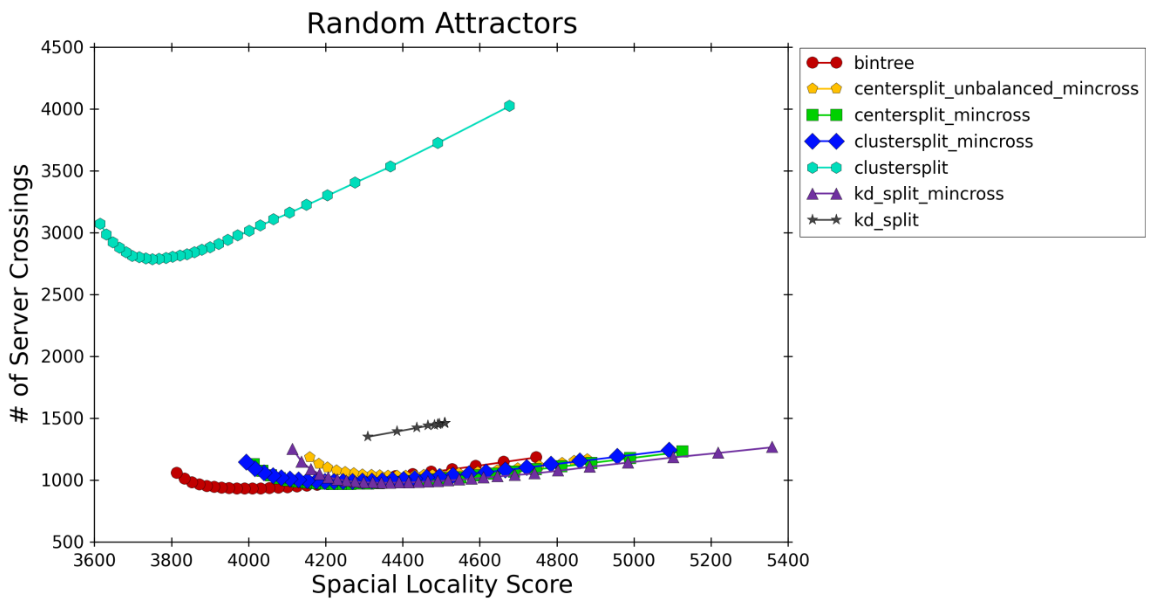


Figure B.33: Random attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

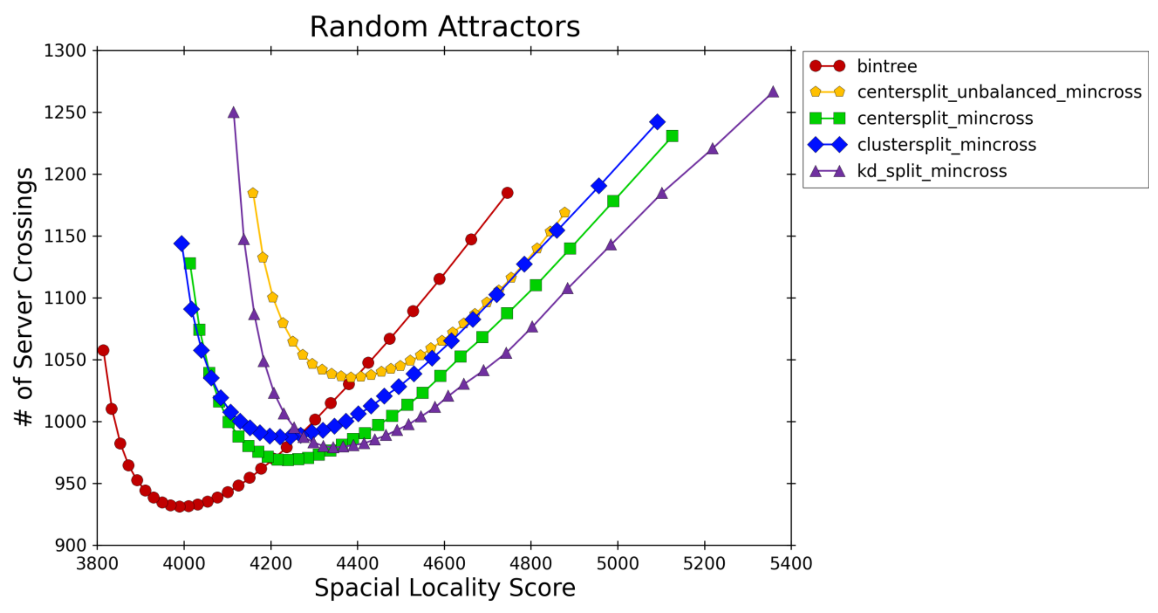


Figure B.34: Random attractors workload. Average number of server crossings per time interval versus average special locality score. Better performance is towards the bottom left on this graph.

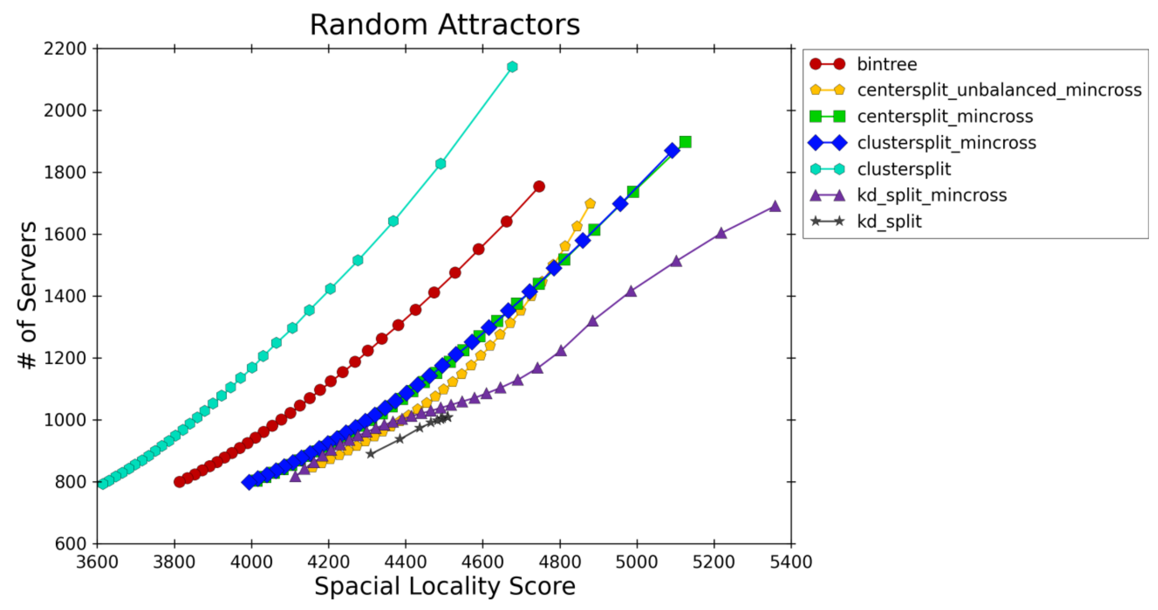


Figure B.35: Random attractors workload . Average number of servers required to manage the simulation versus average special locality score. Better performance is towards the bottom left on this graph.

Appendix C Fixed Grid Virtual World Simulation Workload Summary

Experimental results for the different workloads, using fixed square grid partitioning with between 30^2 and 188^2 servers are reported in this section. The spatial partitioning structure is set at the beginning of each experiment and remains fixed throughout the duration with no dynamic adaptation. The performance metrics reported here are described in detail in Section 4.5.

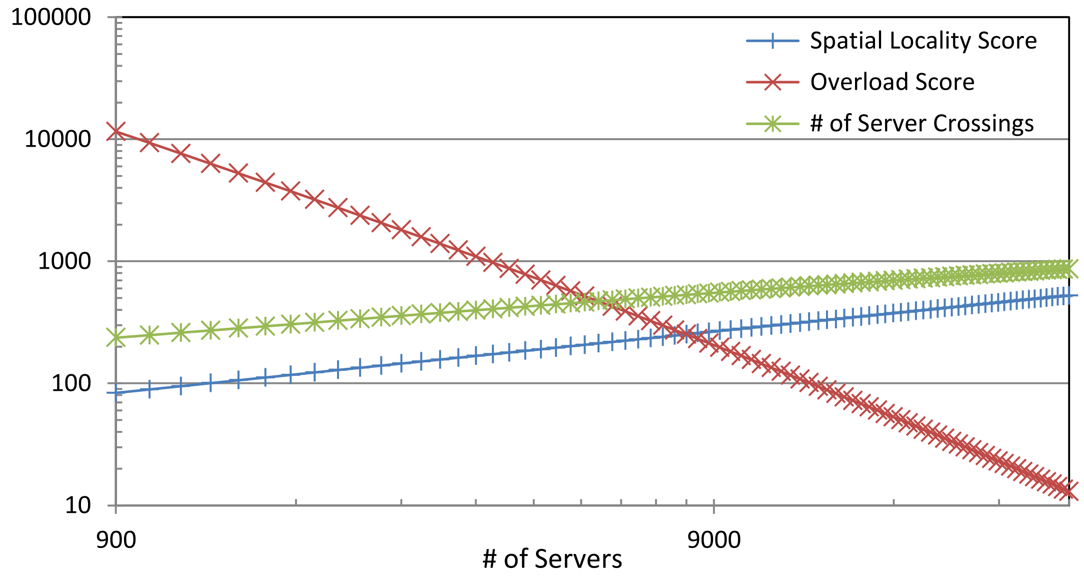


Figure C.1: No Fixed Attractor workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

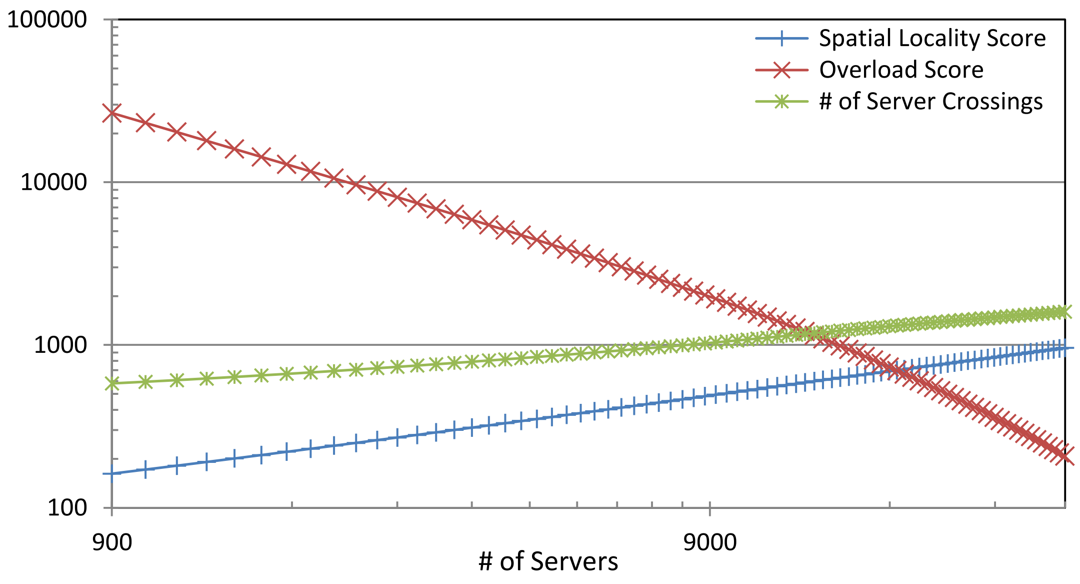


Figure C.2: Single attractor (broad initial location) workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

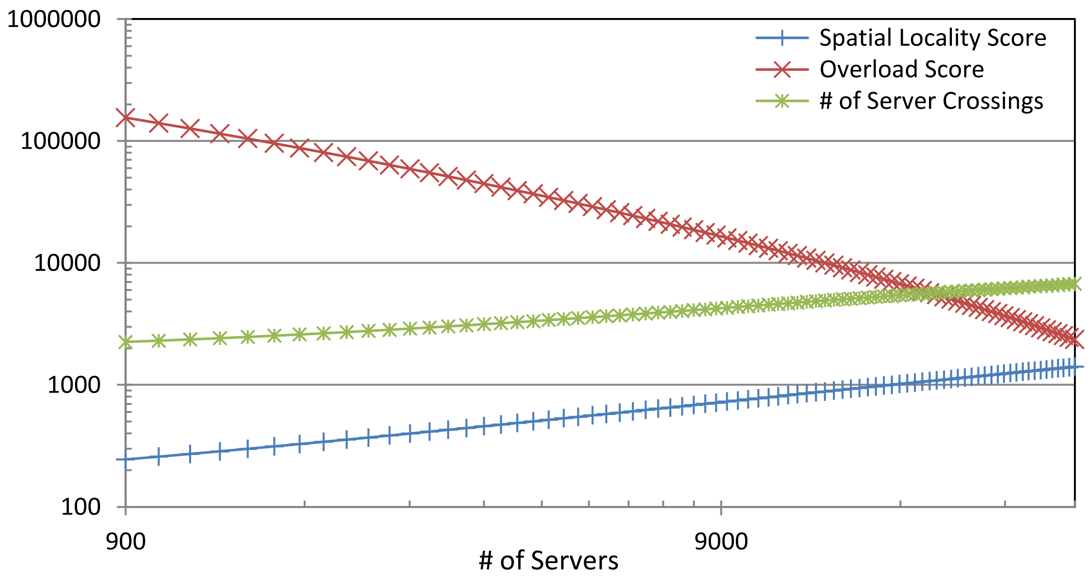


Figure C.3: Single attractor workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

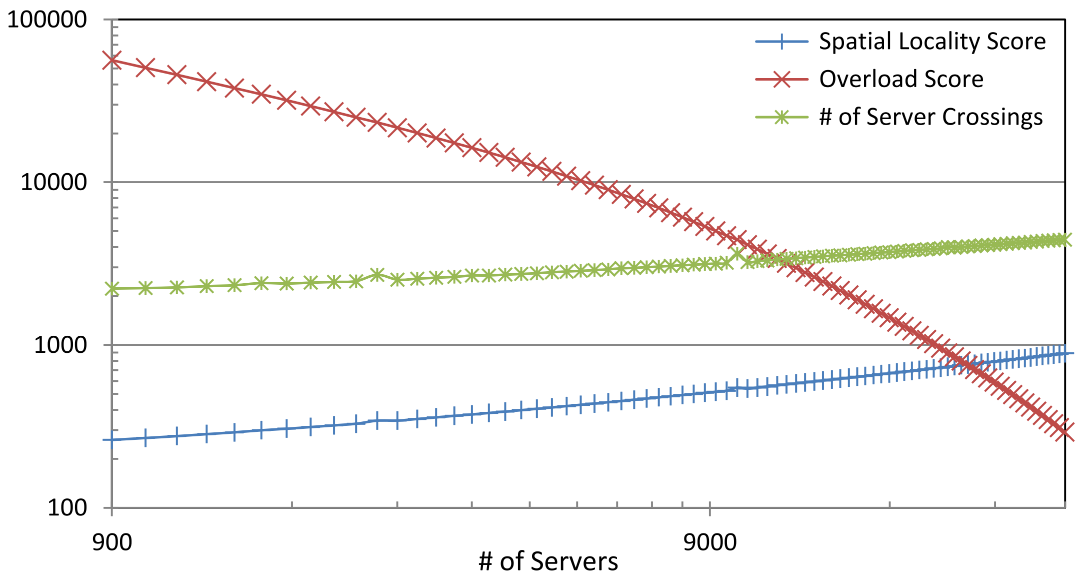


Figure C.4: Row-lined attractors workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

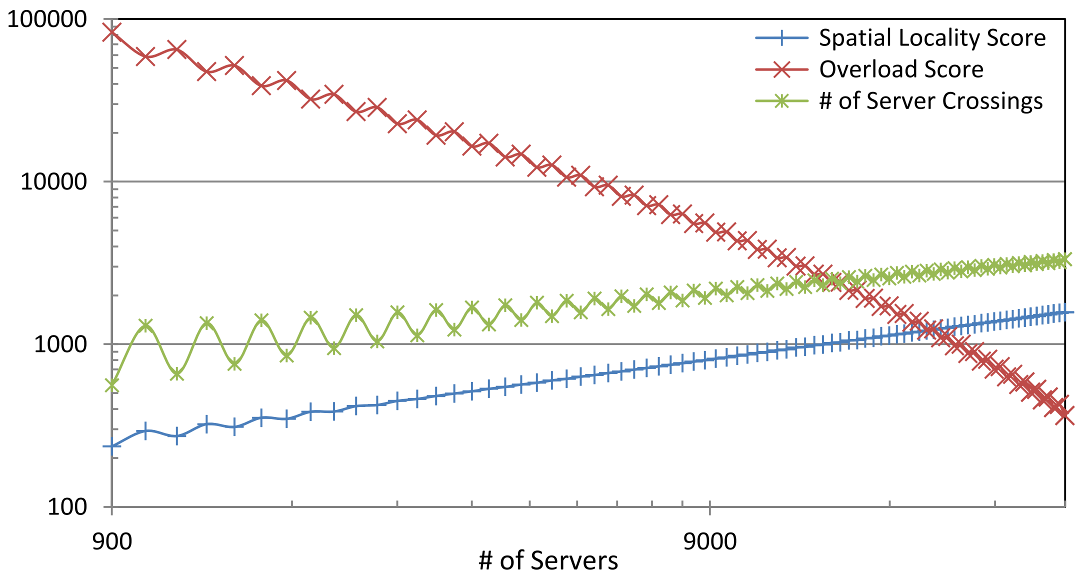


Figure C.5: 2x2 attractors workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

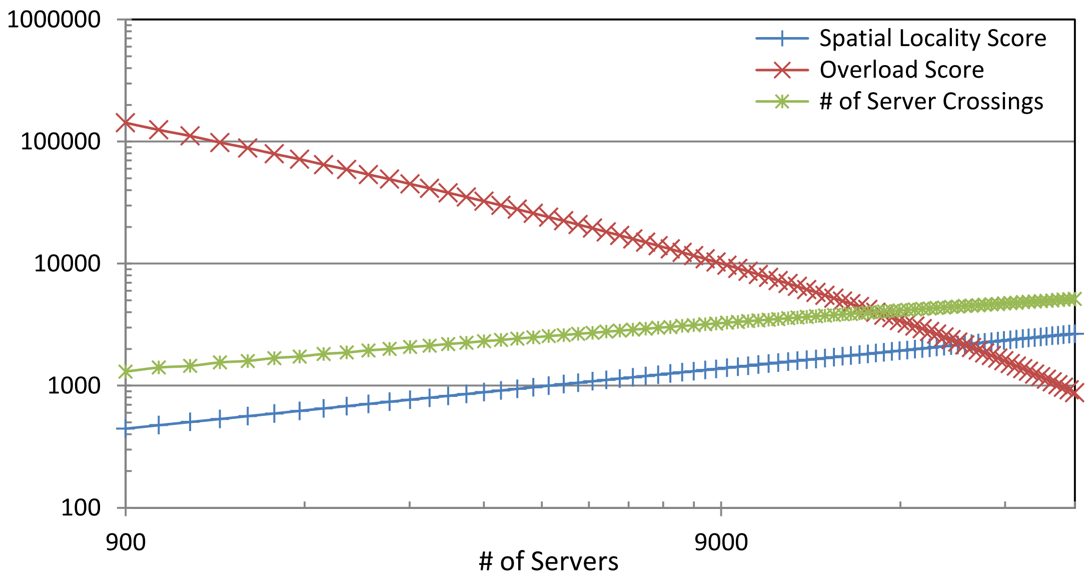


Figure C.6: Circular motion attractor workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.

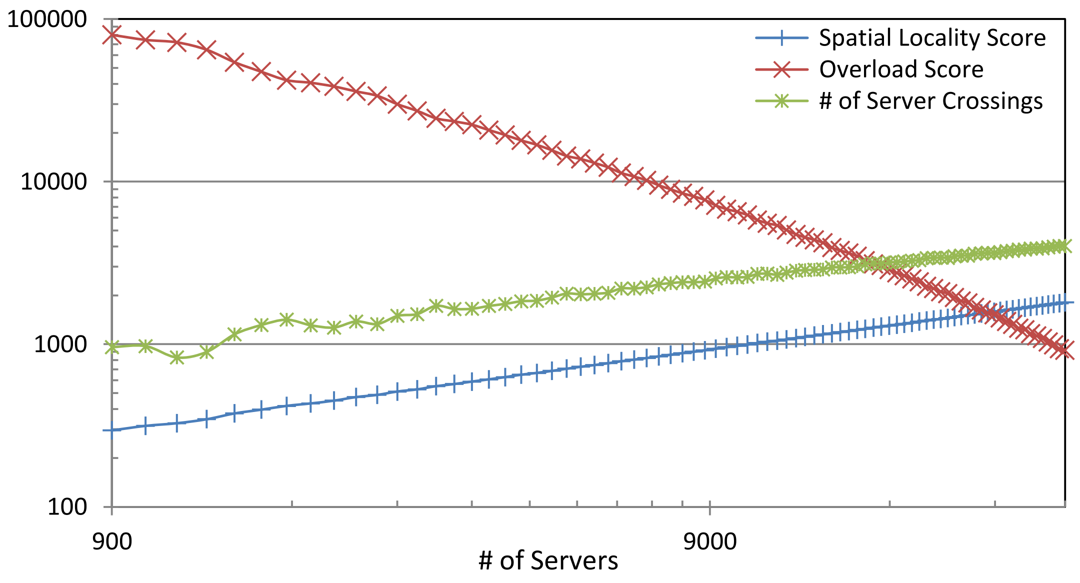


Figure C.7: Random attractors workload, using fixed square grid partitioning with between 30^2 and 188^2 servers.