SCALABLE DATA DELIVERY FOR NETWORKED SERVERS AND WIRELESS SENSOR NETWORKS

A Dissertation Presented

by

DAVID J. YATES

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2006

Department of Computer Science

SCALABLE DATA DELIVERY FOR NETWORKED SERVERS AND WIRELESS SENSOR NETWORKS

A Dissertation Presented

by

DAVID J. YATES

Approved as to style and content by:

_____

James F. Kurose, Chair

_____

Prashant Shenoy, Member

_____

Erich M. Nahum, Member

_____

J. Eliot B. Moss, Member

_____

Weibo Gong, Member

_____

W. Bruce Croft, Department Chair
Department of Computer Science

To Susan, Lauren, and Caroline

ACKNOWLEDGMENTS

This dissertation has been a long time in the making, and I could not have written it without the generous help of family, friends, and colleagues. I begin with my advisors Jim Kurose and Prashant Shenoy, who have given me the gift of excellent advice and unwavering support. I am also indebted to Erich Nahum, Eliot Moss, Don Towsley, Weibo Gong, and Deepak Ganesan who provided me with encouragement and inspired me with their own work.

Much of the research in this dissertation was funded by NSF under grant NCR-9206908, DARPA under contract number F19628-92-C-0089, and by Motorola with a University Partnership in Research Grant. I am grateful to Jim, Don, Erich, and Mike Hluchyj for working hard on these awards, and thus providing me with the resources that I needed to pursue the research that follows.

Erich Nahum deserves special mention as an outstanding friend and colleague throughout my time at the University of Massachusetts. I would also like to thank the faculty and staff in the Computer Science Department at UMass, who continue to provide a rich and fun environment in which to work. The researchers at the Intel Berkeley Research Lab also deserve thanks for sharing the sensor network traces that I used in the second half of this thesis.

Some debts are hard to put into words. My good friends Virgilio Almeida, Mike Anderson, Eugene Bernhard, Eric and Jennifer Brown, Carol Cole, Matt Cornell, Mark Crovella, Jayanta Dey, Peter Desnoyers, Amer Diwan, Jenny Fariborz, Solom Heddaya, Tony Hosking, Chris Jackson, Helene Mayer, John Morris, Carol Pineo, Jim Salehi, and Ken Virgile all know why their names are here.

Finally, I would like to express my deepest thanks to my family, Susan, Lauren, and Caroline; my parents, Eileen and Barrie; my brother, Ian; and my parents-in-law, Tom and Anne. These people collectively nurtured me with tireless love and support before, during, and since I wrote this dissertation. Susan was the most devoted spouse anyone could ask for during their graduate career. Lauren and Caroline came into our world during the time this thesis was created. Sadly, Tom departed our world before this manuscript was finished.

ABSTRACT

SCALABLE DATA DELIVERY FOR NETWORKED SERVERS AND WIRELESS SENSOR NETWORKS

FEBRUARY 2006

DAVID J. YATES

B.Sc., TUFTS UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor James F. Kurose

As Internet technology continues to evolve, network-based services will expand in number and in functionality. For example, web content is increasingly being augmented with more bandwidth-intensive audio and video. Similarly, distributed sensing applications are becoming more data-intensive. For example, the current generation of real-time weather data feeds (e.g., from NCAR) will improve by an order of magnitude in sampling frequency and resolution over the next few years. In combination, these factors will dramatically increase the performance requirements for network-based content and data servers.

Maintaining high performance while supporting a large number of connections is important to any information server. A natural way to accomplish this is by exploiting the inherent parallelism among connections. *Connection-level parallelism* associates the protocol processing required by connections with individual processes or threads. In the first part of this thesis, we investigate the performance of connection-level parallelism, and

evaluate its potential as a paradigm for communication on shared-memory multiprocessor servers. We focus on three issues in this evaluation:

- *Scalability in processors and connections.* The first part of this dissertation shows that throughput of connection-level parallel protocol stacks scales well with the number of processors. Furthermore, the maximum achievable throughput delivered is, for the most part, sustained as the number of connections increases. We also show that for moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor.

- *Fairness behavior.* The first part of this dissertation also investigates how aggregate throughput is distributed across connections. We find that in many cases throughput is *not* distributed fairly among connections, or threads. Our results show that the differences in the per-connection (as well as per-thread) throughput can be explained by the combined effects of scheduling and memory reference behavior.

- *Suitability for continuous media applications.* We then assess the suitability of connection-level parallelism (CLP) for supporting continuous media applications. We present results for a new technique, called *throughput-based scheduling*, that implements a mechanism in which threads monitor and report per-connection throughput to the scheduler. Using a network subsystem suitable for sending continuous media, we show that throughput-based scheduling is suitable for supporting continuous media applications.

We explore these issues in the context of the network subsystem, i.e., the protocol stack, examining throughput, latency, and scalability.

In wireless sensor networks, challenging issues of scale arise from the need to satisfy a variable and potentially large number of queries for sensor data. In the second part of this thesis, we explore the benefits and costs of caching sensor data in a server that is

also the gateway to a sensor network. In such systems, one or more driving applications retrieve sensor data that is disseminated from sensor fields. As data is consumed by these applications, it has an associated *quality* that is a function of two parameters: (1) the delay experienced in receiving the data, and; (2) the deviation between the value of each data item and the corresponding data in the sensor field. Acquiring sensor data usually also has a *cost* that reflects the system resources consumed in querying and delivering the data to the applications. This work examines how well several proposed caching approaches perform in terms of both cost and quality. These approaches are designed to be general since they make no assumptions about whether the overall sensor network architecture uses a structured or unstructured data model.

We assess the impact of several factors on cost and quality performance in wireless sensor networks:

- *Our caching and lookup policies*; the
- *Relative importance of data accuracy and system delay*; and the
- *Rate and magnitude of change in sensed data values in the environment.*

Our research demonstrates that different caching approaches should be used depending on the application requirements and the characteristics of the environment and sensor network. We show that caching approaches that prevent cache misses by computing and returning approximate values for sensor data can provide improved quality and reduced cost at the same time. This win-win is possible because when system delay is sufficiently important, the benefit to both cost and quality achieved by using approximate values outweighs the negative impact on quality due to the approximation. When there is a trade-off between cost and quality, this trade-off is linear. For applications that require bounded resource consumption, we identify a class of caching and lookup policies for which the sensor field query rate is bounded when servicing an arbitrary workload of user queries for sensor data. This upper bound is achieved by having multiple user queries share the cost

of a sensor field query. We also introduce an analytic model for changing the sensor field data values that captures temporal correlation, spatial correlation, and allows different rates at which the values can change. We compare our results that use this correlated model for sensor data values with results that use real-world trace data taken at the Intel Berkeley Research Lab.

Finally, we demonstrate that our results are robust with respect to the manner in which the environment being monitored changes. For our sensor network model with correlated changes to the environment, we present results that vary the rate at which the environment changes by two orders of magnitude. We present results using the real-world trace data varying the rate at which the sensor field is queried, also by two orders of magnitude.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

1.1.1 Motivation for Scalable Networked Servers

As Internet technology continues to evolve, network-based services will expand in number and in functionality. For example, web content is increasingly being augmented with more bandwidth-intensive audio and video. Examples include servers for public information (e.g., digital libraries or government information sources) and entertainment (e.g., video-on-demand). Similarly, distributed sensing applications are becoming more data-intensive. For example, the current generation of real-time weather data feeds (e.g., from NCAR) are improving by an order of magnitude in sampling frequency and resolution. Such improvements mean that meteorologists can better predict severe weather, and therefore help save lives and reduce property damage [142]. In combination, these factors will dramatically increase the performance requirements for network-based content and data servers. These servers must therefore send (or receive) information at high throughput, which must be sustained (or allowed to degrade gracefully) in the presence of large numbers of clients.

Maintaining high performance while supporting a large number of connections is important to any information server. A natural way to accomplish this is by exploiting the inherent parallelism among connections. *Connection-level parallelism* (CLP) associates the protocol processing required by connections with individual processes or threads. On a shared-memory multiprocessor (e.g., [1, 24, 48]), performance gains can be realized over multiple connections by executing these threads concurrently on different processors.

In the first part of this thesis, we investigate the performance of connection-level parallelism, and evaluate its potential as a paradigm for communication on shared-memory multiprocessor servers. We focus on three issues in this evaluation:

- *Scalability in processors and connections.* Connection-level parallelism is an approach advocated for communication on shared-memory multiprocessors [49, 116, 118]. We will provide an implementation of CLP, and experimentally demonstrate how throughput scales with the number of processors, and how throughput changes as the number of connections increases.

- *Fairness behavior.* In evaluating the scalability of connection-level parallelism, one measure of interest is the aggregate throughput of the system. Another important measure of performance, however, is the throughput seen by individual connections. We determine how aggregate throughput is distributed across connections in our implementation. We find that in many cases throughput is *not* distributed fairly among connections, or threads. We therefore investigate whether the scheduler, memory reference behavior, or both play a role in any unfairness in the system.

- *Suitability for continuous media applications.* In some applications, servers will need to "play out" continuous media data (e.g., voice and video), at a rate which closely matches the rate at which it was originally recorded. In such applications, the server protocol stack is one link in a chain of subsystems which must support a fixed rate (either relative or absolute) in terms of communication bandwidth. We will determine and evaluate what features must be included in a protocol stack and scheduler to support continuous media applications.

All of these issues will be explored using our implementation of connection-level parallelism running on a shared-memory multiprocessor. The guiding theme of this research will be to assess experimentally the suitability of connection-level parallel protocols to supporting the applications driving the evolution of the Internet.

1.1.2 Motivation for Scalable Wireless Sensor Networks

Over the next several years, wireless sensor networks will enable many new data-intensive sensing applications, ranging from environmental and infrastructure monitoring to commercial and industrial sensing. Networks of small, possibly microscopic sensors embedded in the fabric of our surroundings: in buildings, warehouses, and machinery, and even on people, will drastically enhance our ability to monitor and control our physical world. However, the realization of wireless sensor networks requires practical satisfaction of a number of real-world constraints introduced by factors such as scalability, reliability, cost, hardware, topology change, environment, and perhaps most importantly, power consumption. Examples of embedded sensor network based applications include urban structure monitoring, contaminant transport monitoring, habitat monitoring, and military surveillance.

There are many performance metrics of interest for sensor networks. We focus on two that are common to the vast majority of sensor networks:

1. The *accuracy* of the data acquired by the application from the sensor network; and

2. the total *system end-to-end delay* incurred in the sequence of operations needed for an application to obtain sensor data.

Although almost all sensor network applications have performance requirements that include accuracy and system delay, their relative importance may differ between applications. We therefore define the *quality* of the data service provided to sensor network applications to be a combination of accuracy and delay. As in most systems, improved quality comes at some *cost*. For current wireless sensor networks, the most important component of cost typically is the energy consumed in providing the requested data. In turn this is dominated by the energy required to transport messages through the sensor field. (e.g., requests, replies, updates, etc.). This cost versus quality trade-off has recently been an active area of research [15, 121, 124, 129, 138].

In the second part of this thesis, we explore wireless sensor network cost and quality performance when a cache is placed in a sensor field gateway server. To perform this research, we construct a sensor network model. We then develop novel policies for caching sensor network data values in the gateway server, and then retrieving these values via cache lookups. We also propose a new objective function for data quality that combines accuracy and delay. Finally, we use our sensor network model to assess the impact of several factors on cost and quality performance:

- Our caching and lookup policies; the
- relative importance of data accuracy and system end-to-end delay; and the
- rate and magnitude of change in sensed data values in the environment.

This assessment evaluates seven different caching and lookup policies by implementing them in a simulator based on CSIM 19 [120, 119]. We focus on four issues in our assessment:

- *When is a cost vs. quality trade-off present?* Recall that quality incorporates both accuracy and delay. The benefit and cost of different caching and lookup policies therefore varies with how quality is specified for a given sensor network application. We present results for sensor networks where user queries are generated for a set of discrete locations within each sensor field. We show that for some quality requirements, policies that prevent cache misses by computing and returning approximate values for sensor data yield a simultaneous quality improvement and cost savings.
- *When present, what is the form and magnitude of the cost vs. quality trade-off?* Within the design space of our seven caching and lookup policies, five of the seven policies age and then delete cache entries uniformly based on an age threshold parameter, T. We observe that in many system configurations these five policies expose a linear cost vs. quality trade-off. Furthermore, when this linearity is not present, the

performance differences between our policies, in terms of both cost and quality, can be small.

- *In what way does the manner in which the environment changes impact performance?* Intuition suggests that augmenting the data acquisition process for a sensor network application with a cache will work well if the phenomenon being monitored is slowly changing. On the other hand, if the phenomenon being monitored changes quickly, caching might not work well. We assess to what extent this intuition is correct by varying the frequency at which the values in the environment being monitored change relative to the query rate.

- *How does changing the age threshold parameter for cache entries affect performance?* For the caching policies that we propose and evaluate, the cache hit ratio for a given workload can be increased by increasing T. The converse is also true. The cache hit ratio can be decreased by decreasing T. We determine how cost and quality performance are impacted as T is changed by two orders of magnitude. We also compare these results with two important baseline policies: First, an empty cache that never yields a hit. We refer to this as an "all misses" baseline, which is equivalent to any of our policies where $T = 0$. Our second baseline is a cache that has an entry for every location, however, no cache entry is ever updated. This is an "all hits" baseline, which is equivalent to any of our policies for $T = \infty$. Although this second baseline provides a useful point of comparison for our results, it is clearly not a caching and lookup policy that is useful in practice without augmenting it with a mechanism to update cache entries.

1.1.3 Bringing Together Scalable Networked Servers and Wireless Sensor Networks

This dissertation investigates related research issues in two very different domains. The innovations that yield our main results are server policies and mechanisms for communicating with a large number of clients within each of these domains. In the first domain the

server is the data source. Specifically, we analyze a large-scale networked server delivering data to a large number of clients over a high-bandwidth network. In the second domain, our servers are data sinks: Embedded servers are acquiring sensor data from resource-constrained wireless sensor networks. Important applications in our first area of research include delivering web content, video, e-business services, music, etc. Emerging applications in our second area include environmental monitoring, military surveillance, and homeland security.

The common threads within these two areas of research are:

- A focus on server architecture;
- Novel techniques for performing data management within the server;
- Communication with clients that are remote with respect to the server; and
- Policies and mechanisms that directly address application requirements.

There are also important differences between these two areas of research.

Our research on scalable networked servers is important for high-performance applications. In these applications the volume of data being handled is relatively large, the data is being delivered to many clients, and the network packets being sent are large. The data sources for these applications are files that are stored on local disks. Therefore, the policies and mechanisms we propose best service a data path that begins at the disk subsystem and ends at the network interface. The data management within our scalable networked server delivers high throughput to web-based applications; and fixed rates per connection to continuous media applications.

Our research in wireless sensor networks is focused on embedded applications for monitoring and control. This means that the volume of data being handled can be large or small, and that sensor data queries and replies are sent in packets that are transmitted through the sensor field. The data sources for these applications are also remote with respect to the data

server. Since acquiring data requires communication over a resource-constrained wireless sensor network, conserving resources in the sensor field is the main goal of our data management policies and mechanisms. We therefore implement a data cache within our sensor network data server and examine its benefits and costs.

1.2 Scalable Networked Servers Background

The research issues we investigate in the first half of this thesis center around the performance of a multiprocessor information server. Such a server communicates with clients throughout a wide-area network, which is potentially global. We briefly describe how this communication is accomplished by using the Internet protocols as an example.

Figure 1.1 shows a wide-area network (WAN) where a small number clients are communicating with a multiprocessor server. Data packets carrying information between clients and the server are routed through the WAN using switches and routers. At the edge of the network, clients and servers typically connect to the WAN via a local-area network (LAN). Thus, packets sent from a client to server traverse the LAN to get to the WAN. In the reverse direction a LAN also connects the server to the WAN.

In order for a client to communicate successfully with the server, they must both use protocols (organized in a so-called *protocol stack*) which can interoperate. Over the Internet, these protocols typically include transport protocols such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Figure 1.2 shows an example of an Internet protocol stack which includes both TCP and UDP. The stack organization conceptually layers protocols on top of each other. Applications sit at the top of the protocol stack (e.g., the World-Wide Web server or Audio / Video server in Figure 1.2). Beginning at the lowest layer and moving up the stack, each protocol introduces features and services which (hopefully) match the application's communication requirements at the transport protocol layer. We now describe the features and services of the protocols below the applications shown in Figure 1.2.

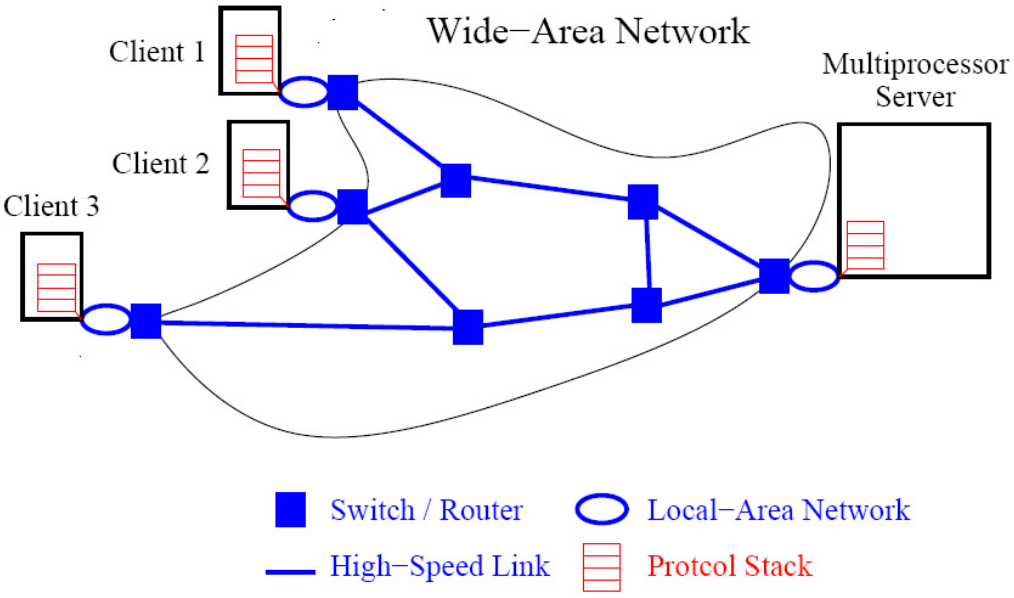

Figure 1.1. Clients and Server Communicating over Wide-Area Network.

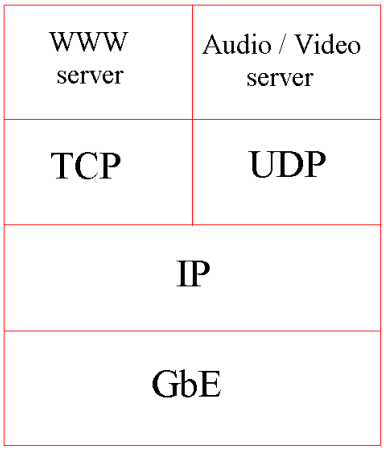

Figure 1.2. Example Protocol Stack.

UDP is a connectionless transport protocol which provides multiplexing for applications sending data and demultiplexing for applications receiving data. It provides error detection in the form a checksum over the data, but no additional reliability. Thus, data that is sent over a UDP-based protocol stack can be lost or even duplicated.

TCP is a connection-oriented protocol which also provides multiplexing and demultiplexing. TCP is a much more complex protocol than UDP. It provides reliable, in-order, data delivery with no loss, error, or duplication. Thus, it presents the abstraction of a bidirectional virtual circuit between the application at the client and server.

The Internet Protocol (IP) is mainly responsible for routing data through the network. On the client and server in Figure 1.1, it is the protocol which knows the identity of the switch or router which connects the local-area network to the wide-area network.

Gigabit Ethernet (GbE) is a standard high-speed local-area network access protocol. Its main function is to arbitrate access to the network which takes the form of a star or a bus in a GbE network.

1.2.1 Parallelism in Network Processing

Many approaches to parallelizing network protocols have been proposed and are briefly described here; additional surveys can be found in [14, 53, 118]. In general, we attempt to classify approaches by the unit of concurrency, or what it is that *processing elements* do in parallel. Here a processing element is a locus of execution for protocol processing, and can be a dedicated processor, a heavyweight process, or lightweight thread. Figure 1.3 illustrates the various approaches to concurrency in host protocol processing. The dashed ovals represent processing elements. The large rectangles represent protocols. Messages marked with a number indicate the connection with which the message is associated.

In *functional parallelism*, a protocol stack's functions are the unit of concurrency. Functions (e.g., checksum and ACK generation, or send and receive processing) are decomposed, and each assigned to a processing element. Performance gains in throughput, are

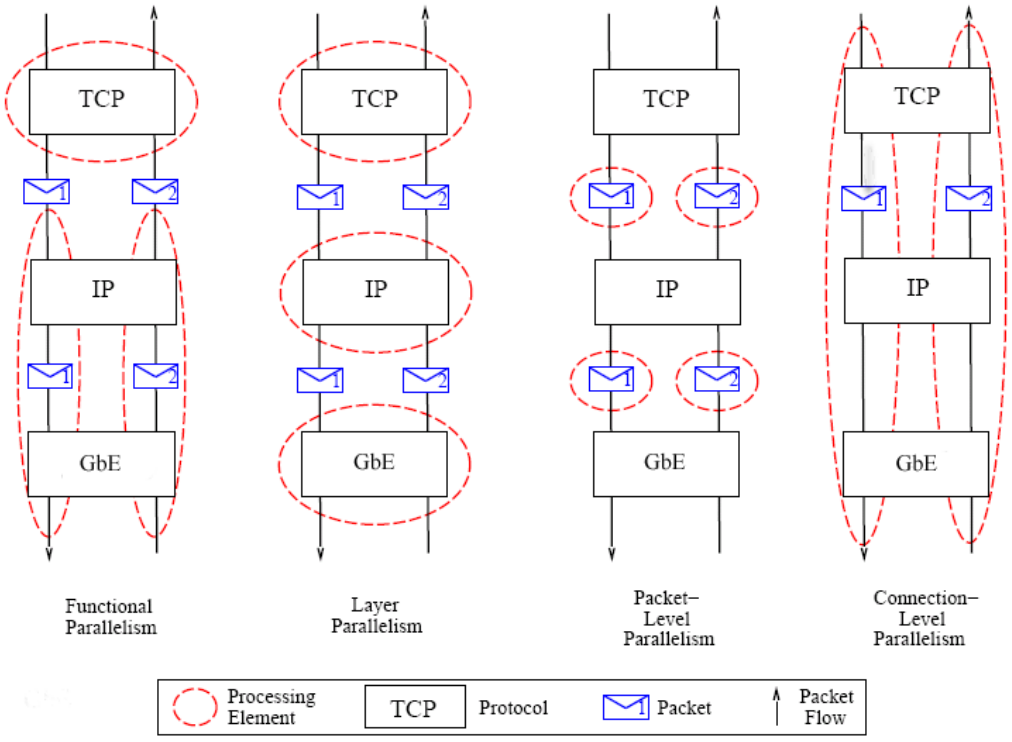

Figure 1.3. Approaches to Parallelism.

achieved through pipelining effects. However, gains in latency are possible if different functional units process the same packet in parallel (e.g., as in [78, 77]). The advantage of this approach is that it is relatively fine-grained, and therefore exhibits a high degree of concurrency. Its main disadvantage is that it requires synchronization between functions, and is inherently dependent upon the concurrency available among the functions of a protocol stack [71, 78, 77, 143]. To address this disadvantage, researchers have focused on combining and splitting functions to reduce synchronization and balance the work performed by different processing elements [18, 20, 55, 75, 78, 77, 96, 143].

A special case of functional parallelism, which we call *layer parallelism*, makes each protocol layer a unit of concurrency [50, 69]. Protocol layers are assigned to processing elements, and messages are passed between layers through interprocess communication. The main advantage of layered parallelism is that it is simple and defines a clean separation between protocol boundaries. The disadvantages are that concurrency is limited to the number of layers in the stack, and performance on a shared-memory processor is hampered by context-switching and synchronization overhead incurred while crossing protocol boundaries [118]. Since concurrent processing of packets by different layers is often not possible, performance gains for layer parallelism are limited to throughput.

In *packet-level parallelism*, the packet is the unit of concurrency. Sometimes referred to as thread-per-packet or processor-per-message, packet-level parallelism assigns each packet or message to a processing element. The advantage of this approach is that packets can be dispatched to any processing element, regardless of their connection or where they are in the protocol stack, achieving speedup both with multiple connections and within a single connection. The disadvantage is that it requires locking shared state, most notably the protocol state at each layer. Systems using this approach include [14, 53, 65, 67, 95].

A set of connections forms the unit of concurrency in *connection-level parallelism* [49, 110, 116, 118]. Speedup is achieved using multiple connections, which can potentially be processed in parallel. The advantage of this approach is that it exploits the natural

concurrency between connections. For each set of connections which form a unit of concurrency, all or part of the protocol stack is treated as a single critical section requiring one synchronization variable. Thus, locking is kept to a minimum along the "fast path" of data transfer. The disadvantage of connection-level parallelism is that concurrency within a single connection can only be achieved by introducing locks (and therefore overhead) at asynchronous interfaces in the protocol stack [49].

The relative merits of one approach over another depend on many factors, including the host architecture, the cost of primitives such as locking and context switching, the workload and number of connections, the thread scheduling policies employed, and whether the implementations are in hardware or software.

The most comprehensive study to date comparing different approaches to parallelism on a shared-memory multiprocessor is by Schmidt and Suda [118]. This work suggests that packet-level parallelism is preferable when the workload is a relatively small number of active connections, and that connection-level parallelism is preferable when the number of connections approaches the number of processors. We describe these results in more detail in Section 2.6.

1.3 Wireless Sensor Networks Background

Almost all sensor network deployments have three main components:

1. One or more sensor fields consisting of sensor field nodes that communicate with one or more base stations;

2. One or more data servers (or gateways) that accept requests for sensor data and generate replies for these requests; and

3. One or more monitoring and control centers that are connected to the appropriate sensor data servers via a backbone network.

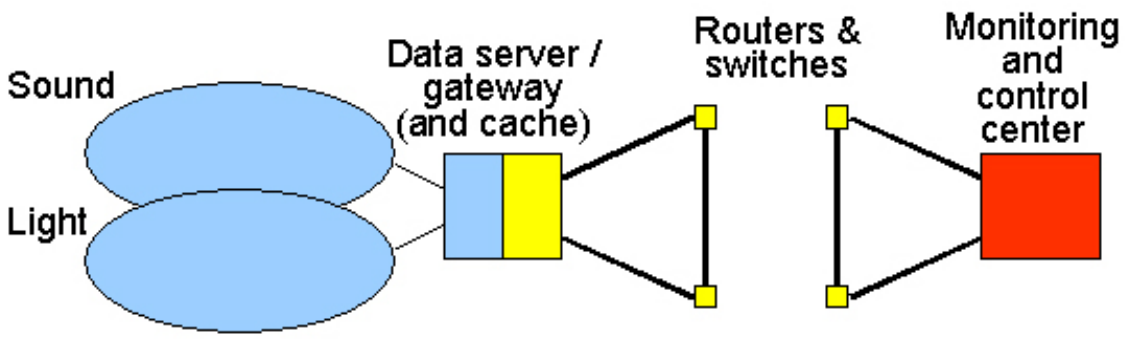

Figure 1.4. Sensor Network Deployment Example.

Figure 1.4 shows an example of such a deployment with two sensor fields, one data server, and one monitoring and control center. If the data server shown in this figure is augmented with storage, it can store and cache sensor field values. The data server receives these values either synchronously via replies to requests for sensor field data, or asynchronously via updates or alerts from sensor field nodes. In such systems, one or more driving applications in the monitoring and control centers retrieve sensor data that is disseminated from sensor fields. As data is consumed by these applications, is has an associated *quality* that is a function of two parameters: (1) the *delay* experienced in receiving the data, and; (2) the *deviation* between the *value* of each data item and the corresponding data in the sensor field. Acquiring sensor data typically also has a *cost* that reflects the system resources consumed in querying and delivering the data to the applications.

It is helpful to view the subsystems in a sensor network using a reference model. We have already done this for a wide-area network and protocol stack in Section 1.2. Figure 1.5 shows a mapping from the OSI reference model to the conceptual layering of the software and hardware for a sensor network.

In the second part of this thesis, we will explore the benefits and costs of caching in the Data Services layer shown in Figure 1.5. If a cache is effective at this layer, it has the potential to improve application performance. One way this improvement is accomplished

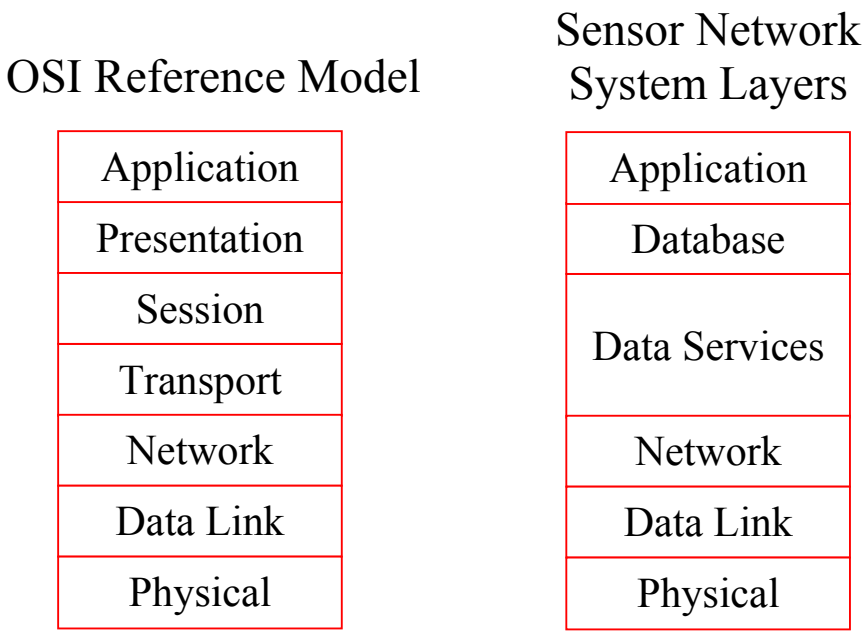

Figure 1.5. Conceptual Layering of Sensor Network System.

is by reducing the number of queries that enter the sensor field while not sacrificing too much accuracy in the replies that are returned to applications.

The caching approaches we propose are designed to be general since they make no assumptions about whether the overall sensor network architecture uses a structured or unstructured data model. In other words, our approaches are independent of the implementation of the Database layer in Figure 1.5. The database could implement a structured schema that extends a standard like the Structured Query Language (SQL). The TinyDB and Cougar systems both advocate this approach [35, 82]. However, the schema could also be modified while the system is running (e.g., IrisNet [51] uses XML and XPATH to accomplish this). The Database layer might also expose a low-level interface to the application. Directed Diffusion [64] does this by allowing applications to process attributes or attribute-value pairs directly.

1.3.1 Data Acquisition and Delivery in Sensor Networks

Figure 1.6 shows how queries and replies are processed by a sensor field gateway or data server. A query that specifies a target location l_i (i.e., requests the value of the sensor field at location l_i) first enters the sensor network query queue and is then forwarded to the sensor field itself. The sensor (or sensors) that monitor location l_i generate a reply that contains (at least) l_i, the sensor field value at this location, v_i, and the time at which v_i was recorded at l_i, t_i. The sensor field reply is then forwarded to the monitoring and control center (not shown) via the gateway reply queue. We say that the *system end-to-end delay*, S_d, for this query-reply sequence occurs between the points $Query_m$ and $Reply_m$ in this figure. Furthermore, we define *value deviation*, D_v, as the unsigned difference between the value in $Reply_m$ and the true value at l_i at the instant in time when $Reply_m$ leaves the gateway reply queue. This definition of value deviation was originally proposed by researchers at Stanford [99, 102].

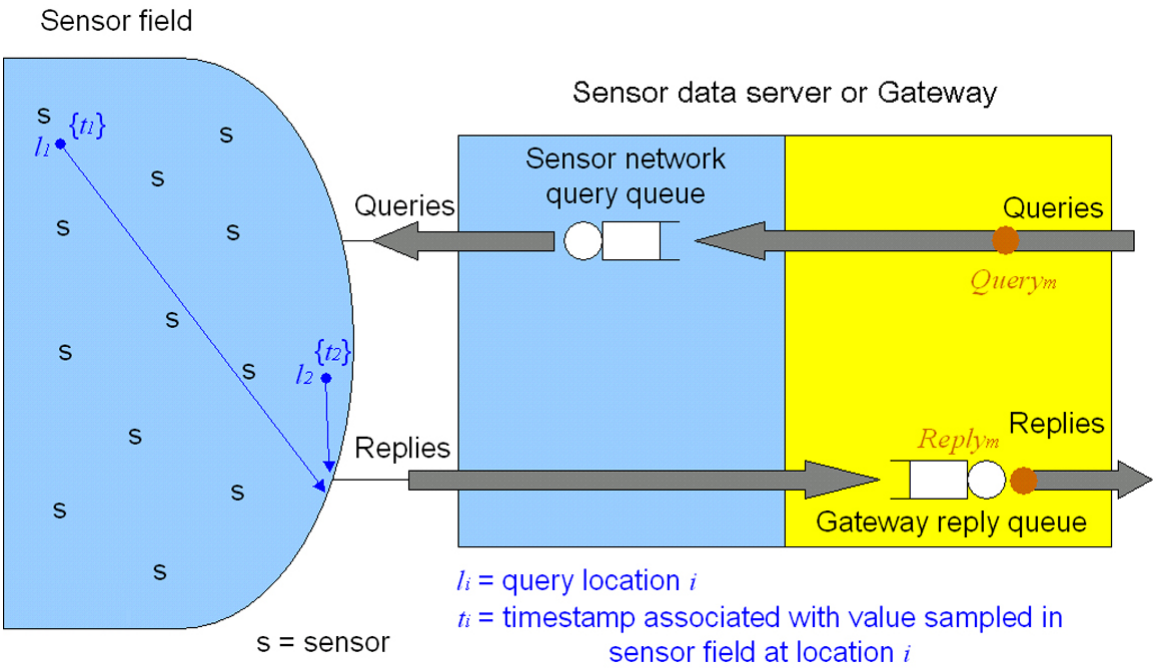

Figure 1.6. Sensor Field, Sensor Network Data Server, and Query-Reply Data Path.

Our model for the data quality delivered to sensor network-based applications captures the fact that the quality improves with lower system delay and lower value deviation (i.e., lower values of S_d and D_v). Our model for the cost of querying for sensor data values captures that fact that transporting messages in the sensor field consumes what is typically the most important resource in wireless sensor networks, namely power. Specifically, the cost to send a query or reply in the sensor field is a function of the square of the distance between a sensor and the nearest edge of the sensor field.

Consider the impact of adding a cache to the data management performed at a sensor network data server. Figure 1.7 shows such a system in which a cache is added to the internal architecture of the data server, on the "border" between the sensor field(s) and the backbone network.

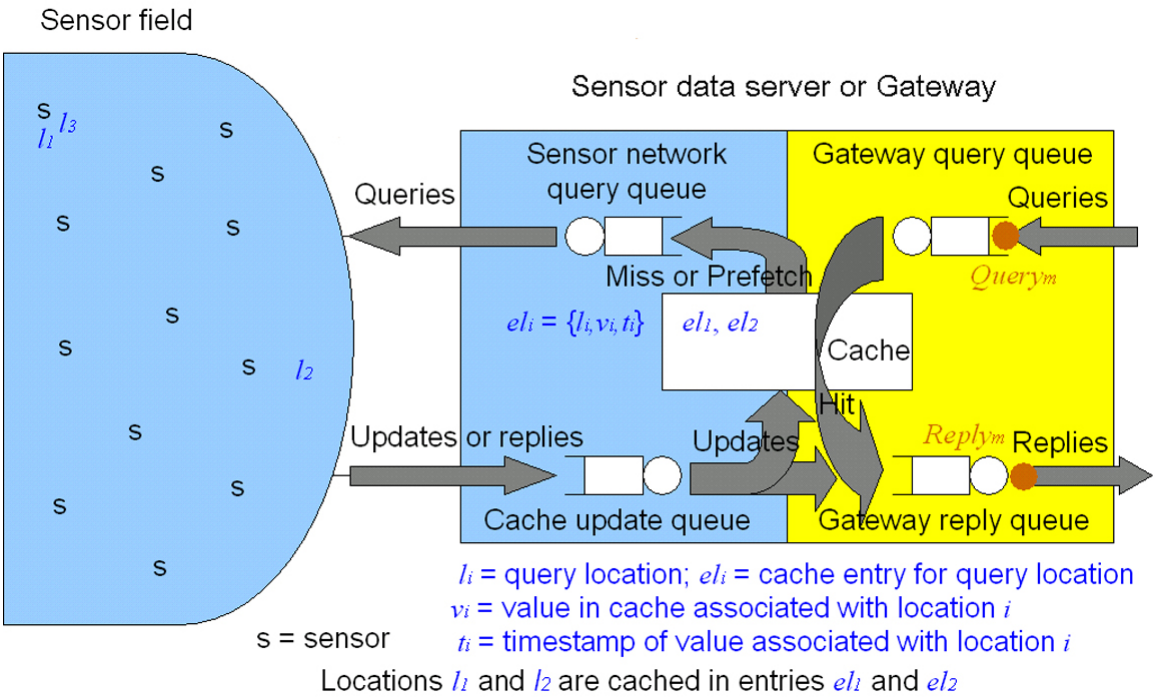

Figure 1.7. Sensor Network Data Server or Gateway with a Cache.

With the introduction of a cache in Figure 1.7, there are now two possible data paths that can be traversed in response to a query from the backbone network:

- For a cache miss, the data path is almost exactly the same as the one described for the query-reply sequence in Figure 1.6. The only additional work performed along this path is what is necessary to update the cache. To update the cache, each sensor data value v_i is copied into a cache entry. A cache entry, el_i, associates with location l_i, the most recent value observed at this location, and its timestamp into the tuple $\langle l_i, v_i, t_i \rangle$.

- The data path for a cache hit is much shorter than for a cache miss. For example, if the cache is indexed by location, and a cache entry is present for a location l_i specified in a query, a reply can be generated using only the information in the tuple that corresponds to l_i. Since the processing required to perform this cache lookup and generate a reply is relatively small, we assume that the system delay for a cache hit (S_d between $Query_m$ and $Reply_m$) and its associated cost are both zero. We also determine the value deviation for cache hits, D_v, in the same way as for cache misses: by computing the unsigned difference between the data value in $Reply_m$ and the true value at l_i when $Reply_m$ leaves the gateway reply queue.

Since each cache hit in the data server provides a cost savings and at least partially benefits quality (by reducing delay), our caching and lookup policies explore data management approaches for our cache that increase the effective cache hit ratio. Specifically, we exploit spatial locality within sensor field data, and in some caching and lookup policies allow cache "hits" in which the value at location l_i is approximated based on values $v_{i'}$ from neighboring location(s) $\{i' \in N(i)\}$. (Here $N(i)$ denotes the neighborhood of location l_i.) We develop and describe three such policies that implement what we call *approximate* lookups and queries. We compare these approximate policies with four *precise* lookup and query policies that only use information associated with location l_i to process queries that reference location l_i.

The definition of a *cache hit* (vs. a *cache miss*) for approximate lookup and query policies has both similarities and differences with the traditional definition of a cache hit (or

miss). One similarity is that the data value in the cache may be consistent with the value in the sensor field. This would always be true for a memory cache for a uniprocessor computer [23], or for a sequentially consistent multiprocessor [46]. A related similarity is that the data value in the cache may be different from the value in the sensor field. This would also be the case, some of the time, in a distributed file system that caches blocks or files [59], or a web-based application that exploits web caches [81]. These similarities are also shared with the precise lookup and query policies. An important difference from these traditional caching systems is that the cache lookup policy itself might introduce some inconsistency (e.g., value deviation). This is because our approximate policies may emulate a cache hit by approximating a data value in the sensor field (using neighboring values) rather querying the sensor field for the missing value.

Recall that all of the caching and lookup policies we propose and evaluate incorporate an age threshold parameter T that specifies how long each entry is stored in the cache. In practice, picking a good value for T depends on how rapidly the environment being monitored might change, and the utility of a cached entry as it approaches being T seconds old. If T is too small, it is easy to imagine that the cache might not be effective. If T is too large, then the cache may be helpful in terms of saving cost, but may be harmful when value deviation increases as a function of elapsed time. Clearly, larger values of T reduce the number of messages that are sent through the sensor field. We confirm this intuition, and quantify the peak query rate for our sensor field as a function of T. Since power consumption in the sensor field is a function of query rate, bounding our sensor field query rate can help bound the cost consumption for our caching and lookup policies.

We determine under what conditions the data management approaches we have summarized thus far exhibit a cost versus quality trade-off. When this trade-off is present, we characterize both its form and magnitude. Within these results, we also answer the following questions:

- Which policies provide best cost versus quality when value deviation is more important than delay?
- Which policies provide best performance when delay is more important than value deviation?
- How does the manner in which the environment changes impact cost and quality performance?
- How does varying the age threshold parameter for cache entries impact performance?

Our results show that different caching approaches should be used depending on the application requirements and the characteristics of the environment and sensor network. We show that approximate lookups and queries can provide a simultaneous improvement in both cost and quality. When there is a trade-off between cost and quality, this trade-off is linear. We also identify a class of caching and lookup policies for which the sensor field query rate is bounded when servicing an arbitrary workload of user queries. This bound is achieved by having multiple user queries for sensor data share the cost of a sensor field query.

1.4 Contributions of this Dissertation

The following are the contributions of the first part of this dissertation:

- *Scalability of connection-level parallelism.* The first part of this dissertation begins by describing our implementation of connection-level parallelism, and experimentally determining how throughput scales with the number of processors, and how throughput changes as the number of connections increases. We show that connection-level parallel protocol stacks scale well with the number of processors. Furthermore, the throughput delivered is, for the most part, sustained as the number

of connections increases. We also show that for moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor.

- *Fairness behavior of connection-level parallelism.* We then investigate how aggregate throughput is distributed across connections in our implementation of connection-level parallelism. We find that in many cases throughput is *not* distributed fairly among connections, or threads. We show that matching the number of threads to the number of physical processors, while scheduling connections assigned to each thread (or virtual processor) in a round-robin manner, yields the best fairness behavior. However, an improvement in fairness may come at the cost of aggregate throughput. Our results also show that the differences in the per-connection (as well as per-thread) throughput can be explained by the combined effects of scheduling and memory reference behavior.

- *Suitability for continuous media applications.* In the next chapter of this thesis we assess the suitability of connection-level parallelism (CLP) for supporting continuous media applications. We present results for a new technique, called *throughput-based scheduling*, that includes a mechanism in which threads monitor and report per-connection throughput to the scheduler. To support soft throughput guarantees, the scheduler balances load by reallocating work (in this case connections) among threads. To support hard throughput guarantees, the scheduler schedules threads according to their "utilization" (i.e., how much of their current capacity is consumed by the throughput they have been assigned to deliver). Using UDP as the transport protocol, we show that throughput-based scheduling is suitable for supporting continuous media applications.

The following are the contributions of the second part of this dissertation:

- *Necessary conditions for a cost vs. quality trade-off.* We measure the benefit and cost of seven different caching and lookup policies as a function of the application quality requirements. We show that for some quality requirements (i.e., when delay drives quality), policies that emulate cache hits by computing and returning approximate values for sensor data yield a simultaneous quality improvement and cost savings. This win-win is because when delay is sufficiently important, the benefit to both cost and quality achieved by using approximate values outweighs the negative impact on quality due to the approximation. In contrast, when accuracy (i.e., value deviation) drives quality, a cost vs. quality trade-off emerges. When this trade-off is present, different policies perform best depending on whether quality or cost is most important to the application.

- *Form and magnitude of the cost vs. quality trade-off.* For our seven caching and lookup policies, five of these policies age and then delete cache entries uniformly based on the age threshold parameter, T. We observe that in many system configurations these five policies expose a linear cost vs. quality trade-off. When this linearity is present, we find that the underlying cost vs. accuracy and/or cost vs. delay functions are also linear. When this linearity is not present, the performance differences between our policies, in terms of both cost and quality, can be small. When this is true, we also observe that the cache hit ratios for our policies are close in value.

- *Bounded cost for some caching and lookup policies.* For applications that require bounded resource consumption, we identify a class of policies for which the sensor field query rate can be bounded when servicing an arbitrary workload of user queries. Recall that the domain for our user queries is the set of discrete locations in the sensor field. This upper bound is a function of two variables: (1) the number of locations in each sensor field (these locations are also used to index the cache) and; (2) the age threshold parameter (T).

- *Impact of the manner in which the environment changes on cost and quality performance.* Intuition suggests that augmenting the data acquisition process for a sensor network application with a cache will work well if the phenomenon being monitored is slowly changing. On the other hand, if the phenomenon being monitored changes quickly, caching might not work well. We assess to what extent this intuition is correct by varying the frequency at which the values in the environment being monitored change relative to the query rate. Furthermore, our results characterize and quantify cost and quality performance for two different sensor fields, which each monitor environments with different characteristics. These results show that the form and magnitude of the cost and quality performance change, however, the performance trends generally remain the same. Specifically, the performance differences between policies change, but the policies that provide the best quality (and cost) performance in different system configurations are usually the same.

- *Effect of the age threshold parameter policy for cache entries on performance.* For the caching policies that we propose and evaluate, the cache hit ratio for a given workload can be increased by increasing T. The converse is also true. We determine how cost and quality performance are impacted as T is changed by two orders of magnitude. We also compare these results with "all misses" and "all hits" baseline policies. We see that for high cache hit rates (e.g., 88% and higher for the cases we consider) the cost performance of all of our five age-based policies converges toward the "all hits" baseline performance. In contrast, for low cache hit rates (32% and lower) the performance of some (but not all) of our policies is clustered near the (cost, quality) point associated with the "all misses" baseline. When this is true, we also see an interesting negative result: Unless cost is much more important than quality, some of the caching and lookup policies exhibit worse performance than having all misses. However, a cache with a positive hit ratio always provides some cost savings.

1.5 Structure of this Dissertation

The remainder of this thesis is organized as follows: Chapter 2 describes connection-level parallelism in more detail and identifies the impact of implementation alternatives on scalability. In Chapter 3, we investigate the fairness behavior of our implementation. Chapter 4 explores the protocol stack and operating system features required to support continuous media applications. Chapter 5 describes our caching and lookup policies, and investigates their cost and quality performance for wireless sensor networks. Finally, Chapter 6 summarizes the dissertation and suggests several possible avenues for future work.

CHAPTER 2

SCALABILITY OF CONNECTION-LEVEL PARALLELISM

2.1 Introduction

In this Chapter of the thesis, we experimentally evaluate connection-level parallelism in a number of previously unexamined dimensions. We focus on several issues in our performance evaluation:

- How throughput scales as processors are added.
- How throughput changes as connections are added.
- The effect of the granularity of parallelism on performance.
- The effect of the transport protocol on performance.
- The impact of processing packets in bursts.

Our evaluation of connection-level parallelism is performed for both TCP/IP and UDP/IP protocol stacks, implemented using the *x*-kernel [63]. Our implementation runs in user space on a shared-memory Silicon Graphics (SGI) Challenge multiprocessor [48].

Our main contribution here is to demonstrate that connection-level parallel protocol stacks scale well with the number of processors for both send-side and receive-side communication. Furthermore, the throughput delivered is, for the most part, sustained as the number of connections increases. We also show that for moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor.

2.2 Connection-Level Parallelism

Connection-level parallelism is essentially the synthesis of two ideas: the first is extending the notion of a connection through the entire protocol stack, even down to the device; the second is running these multiple connections in parallel.

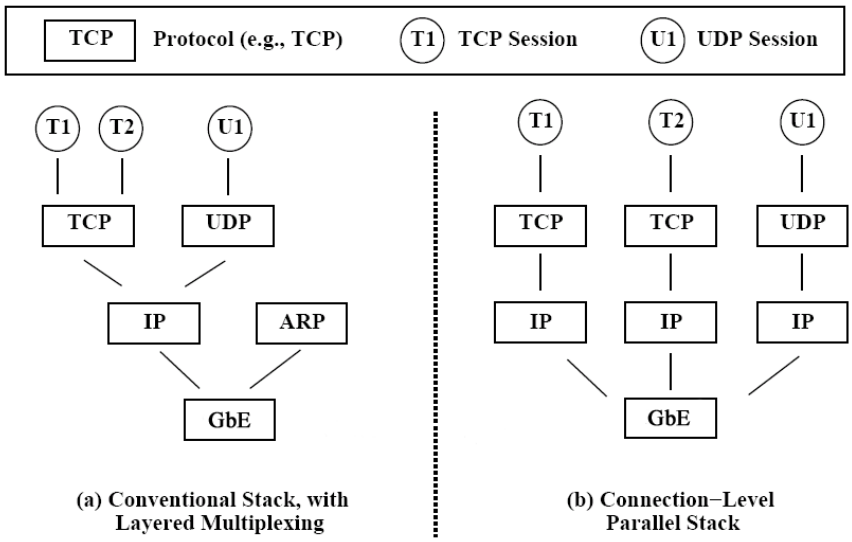

Figure 2.1. Protocol Stack Configurations.

Figure 2.1(a) shows a conventional protocol stack, where protocols and connections are multiplexed on top of other protocols and connections. In this Figure, a TCP connection and a UDP connection multiplex the IP protocol. On the receive path, IP demultiplexes incoming packets to the appropriate protocol.

Figure 2.1(b) shows a connection-level parallel protocol stack. In this example, the TCP and UDP connections extend all the way down to the device. Protocols are conceptually replicated, and multiplexing occurs at the lowest layer of the protocol stack. On the receive path, a packet is immediately demultiplexed to the appropriate connection. This demultiplexing is performed by a packet filter or classifier [9, 88, 91, 139].

Given a set of connections, threads, and processors, the assignment or mapping between them can be done in a number of different ways. Choosing a mapping defines the

granularity of a connection-level parallel implementation. Previous work on connection-level parallelism can be found in references [49, 110, 116, 118] has focused on relatively static assignments of connections to processes. One novel aspect of our implementation is that it allows us to vary the mapping between processors, connections, and threads. We introduce the abstraction of a *virtual processor*, which allows us to vary this assignment, and thus examine how structural choices affect performance.

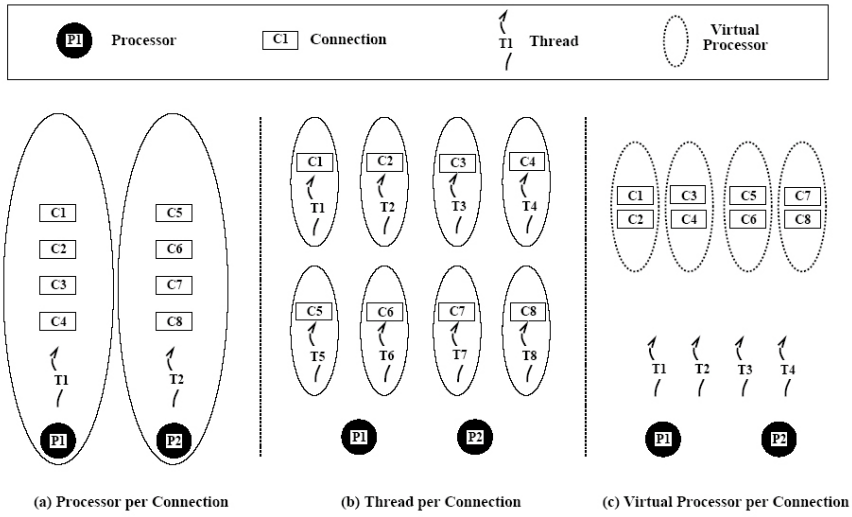

Figure 2.2. Approaches to Connection-Level Parallelism.

The simplest and most coarse-grained approach to connection-level parallelism is *processor per connection* (PPC), which associates each connection with an individual processor. Figure 2.2(a) shows an example where 8 connections are mapped to 2 threads, which are each bound to their own processor. Ovals in this Figure denote units of concurrency. Thus, only one thread at a time can perform protocol processing for each of the 2 sets of 4 connections in Figure 2.2(a). PPC has the advantage of simple implementation, but may not balance load well when some connections are more active than others.

The most fine-grained approach, *thread per connection* (TPC), makes each connection its own unit of concurrency. Figure 2.2(b) shows an example of TPC with 8 connections and 8 threads for sending data. These threads are free to execute on either of the two processors,

and perform protocol processing on behalf of any of the connections. In practice, however, each connection is typically associated with an individual thread [49, 116]. While TPC allows easy load balancing, it may not scale well with large numbers of connections, since each thread must be allocated resources (such as a thread control block and a stack).

Our implementation facilitates varying the mapping of connections to threads and processors by using virtual processors. Connections are assigned to virtual processors, rather than physical processors or threads, in what we call *virtual processor per connection* (VPPC). Instead of a connection, the notion of a virtual processor (which corresponds to one or more connections) is extended down through the protocol stack, all the way to the device. Figure 2.2(c) shows an example where 8 connections are assigned to 4 virtual processors on 2 physical processors. As with thread per connection, threads are free to execute on either of the two processors, and perform protocol processing on behalf of connections assigned to any of the four virtual processors. If the number of virtual processors is the same as the number of connections, VPPC yields the same (maximal) concurrency as thread per connection. If the number of virtual processors matches the number of physical processors, (and threads are wired to physical processors), then VPPC is equivalent to PPC.

Another way of thinking of VPPC is that it extends thread per connection to support multiplexing a set of connections onto a pool of threads, where the number of threads can be varied. This becomes important if one considers a server supporting thousands of connections. Since each thread requires resources (and increases the cost of scheduling), it is easy to imagine that TPC won't scale as well as VPPC (or even PPC) under such conditions.

At first glance, a virtual processor may seem synonymous with a thread. However, there are important differences. A virtual processor defines a unit of concurrency, not a specific thread. For example, on the send side, an application thread might be "borrowed" to run on a virtual processor in order to deliver data to the device. Conversely, on the receive side, a

thread dispatched from a lower layer in the protocol stack would run on a virtual processor to deliver data to the application.

2.3 Implementation and Experiments

In order to study the performance of connection-level parallelism (CLP), we have implemented multiprocessor versions of the core Internet protocols (TCP, UDP and IP) over Gigabit Ethernet (GbE), running in a version of the *x*-kernel which was extended to support CLP. This section describes the important features of our parallelized *x*-kernel and protocols. In the last part of this section, we describe our experimental design.

2.3.1 Connection-Level Parallel *x*-Kernel

In our implementation of connection-level parallelism, concurrency control is accomplished by having a semaphore for each virtual processor. To execute protocol code on a virtual processor, a thread must acquire the appropriate semaphore. Once running on a virtual processor, a thread runs to completion, as in the original *x*-kernel [63].

One innovative feature of our implementation is that there are no locks on the fast path through the protocol stack, on either the send or receive side (once packets have been demultiplexed to the appropriate virtual processor). This is in contrast to earlier implementations [14, 95, 110, 118] in which data structures are locked on the fast path. We accomplish this by replicating data structures on a per virtual processor basis where possible. Thus, threads which require exclusive access to an object merely look up their current virtual processor identifier, and use it to index into an array of replicated objects. Locking is not necessary since only one thread at a time can be executing on a particular virtual processor. Examples of data structures which we are able to replicate include *x*-kernel demultiplexing hash tables, TCP send buffer free lists, and IP datagram identifier counters.

It is important to point out that not all data structures can be replicated. For example, TCP connections must be uniquely instantiated. Thus, our CLP implementation requires a

packet filter mechanism to demultiplex received packets to the appropriate virtual processor for a TCP (or UDP) connection. Packet filters are becoming more popular (and efficient [9, 139]) in contemporary operating systems since early demultiplexing yields other performance gains. For example, depositing received packets directly into application buffers avoids copying data [42, 83, 128]. Our use of a packet filter, to demultiplex to the appropriate virtual processor, simply leverages this existing mechanism for an additional purpose.

2.3.2 Connection-Level Parallel Protocols

Our Internet protocols are all based on the uniprocessor implementations distributed with the December 1993 version of the *x*-kernel. In addition to modifying them for connection-level parallelism, we made two other important changes to these protocols. First, we updated the TCP code to be current with the Berkeley 4.4Lite implementation, excluding the RFC 1323 extensions [66]. We also replaced the Internet checksum code with the fastest available portable algorithm that we were aware of, which was from UCSD [72].

We also implemented a connection-level parallel version of UDP. Even though UDP is a connectionless protocol, we consider long term associations between a sender and receiver to be a "connection", or flow [32]. A UDP association is used to map connections to virtual processors, and provides a handle for demultiplexing, in the same fashion as TCP.

Since we did not have access to a device which matches the performance of our protocols, we replaced the lowest level drivers in the *x*-kernel with in-memory device drivers for both the TCP and UDP protocol stacks. The drivers emulate a GbE interface capable of operating at memory speeds (i.e., faster than 1 Gbps), and transfer packets that are slightly larger than 4 KB. This approach is similar to the those used by other researchers [14, 53, 79, 95, 118].

In our experiments, we assume that a connection always has data to send (i.e., each connection is an infinite data source). Therefore, the drivers act as peer senders or receivers, producing or consuming packets as fast as possible. This simulates the behavior of

a collection of clients sending or receiving data over simplex connections routed through an error-free network. To minimize experimental perturbation, the receive-side driver uses preconstructed packet templates. To simulate an actual sender, this driver should compute the Internet checksum and update the template (for TCP). Instead, a dummy checksum value is left in the template and the TCP or UDP receivers compute the checksum, but ignore the result.

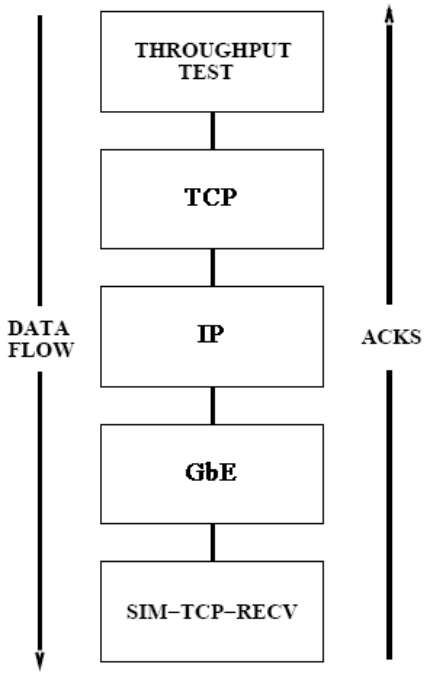

Figure 2.3. TCP Send-Side Configuration.

Figure 2.3 shows an example of a test configuration. The example is of a send-side TCP throughput test, where a simulated TCP receiver sits below the GbE layer. The simulated TCP receiver generates acknowledgments for packets sent by the actual senders. The driver acknowledges every other packet for each connection, thus mimicking the behavior of Berkeley TCP when communicating with itself as a peer. Since spawning threads is expensive in user space in IRIX, the driver "borrows" the stack of a calling thread to send an acknowledgment back up.

The TCP receive-side driver (i.e., simulated TCP sender) produces packets in-order for consumption by the actual receiver. Both simulated TCP drivers also perform their respective roles in setting up a connection.

2.3.3 Experimental Design

To evaluate our CLP implementation, we ran experiments for both TCP/IP and UDP/IP protocol stacks. In addition to the protocol stack being measured, several other parameters define an experiment: the number of physical processors, virtual processors, and connections; whether the multiprocessor is the sender or receiver; and whether or not the checksum is computed.

The throughput data from each experiment (both aggregate and per-connection) are calculated from the average of 12 runs, where a run consists of measuring the steady-state throughput of packets carrying 4KB of user data for 45 seconds, after an initial warm-up period of 45 seconds or more. For all data we present in figures, we show 90% confidence intervals. During experiments, we isolated our machine as much as possible by disallowing other user activity and removing all non-essential daemons. We now discuss some of the more subtle aspects of our experiments.

Our experiments use a thread for each virtual processor to produce and send or receive packets. The protocol processing associated with connections assigned to the same virtual processor is performed in a round-robin manner among connections, with the unit of work being the processing needed to bring a single packet (or a burst of packets) up or down the protocol stack. Between processing each unit of work, the virtual processor is yielded to allow other tasks (e.g., TCP timers) to run on that virtual processor. However, the scheduling of threads running on virtual processors is controlled by the IRIX operating system.

In some of our results, we present average per-packet latency (as well as throughput) over the 45-second sampling period, for each individual thread. We compute latency using Little's law from the measured throughput for the thread, the actual wall-clock time, and the

user time given to the thread by the operating system. The user time for a thread represents the time spent executing, and therefore processing packets, in user space. This is obtained by making a `getrusage()` system call at the beginning and end of the sampling period.

It is well known that memory reference behavior can be crucial in determining how a system performs when running on a multiprocessor. We control the memory reference behavior of our protocols in several ways. First, to capture the cost of loading packets with user data, each sending connection copies data from a statically allocated 4KB page to a page-aligned *x*-kernel communication buffer. For receiving connections, the copy is performed in the reverse direction. Other than this copy, we do not attempt to capture any computation or memory references associated with an application.

For send-side experiments, we always cache communication buffers in LIFO lists. These lists are maintained per virtual processor, and therefore require no locks. This technique has been shown to improve throughput in protocol stacks running on a uniprocessor [43]. This means that our send-side experiments measure either a cache-to-cache or a memory-to-cache copy depending on the fate of the "application" data buffer in the caches.

For receive-side experiments, it is unreasonable to assume that user data will be in the cache, so we wire threads to processors and force any user data referenced for the first time to be fetched from memory and not from processor caches. To implement this, threads allocate receive-side buffers (pages) from a per physical processor circular list which is twice as large as the second-level caches on our multiprocessor. Unfortunately, managing this list introduces locking overhead, which unfairly penalizes throughput experiments where there are more virtual than physical processors. We therefore only present receive-side results for processor per connection.

In experiments where we vary the number of processors, we use IRIX's `pset` facility, which restricts the threads in a process group to running on a subset of processors.

We report results from an evaluation of our connection-level parallel implementation along three different dimensions. First, we examine how aggregate throughput scales as

the number of processors increases. Second, we examine how throughput is affected by the number of connections. In Chapter 3, we investigate how fairly the total throughput is distributed across both moderate and large numbers of connections.

To quantify fairness, we measure the throughput seen by each connection and virtual processor, and compute percentiles between the maximum and minimum throughputs. Since our fairness experiments use a single thread per virtual processor, our results refer to per-thread (or per-connection) throughput, or throughput per thread (or connection). In contrast, we refer to aggregate (or total) throughput as just throughput.

2.4 Throughput Scalability with Respect to Processors

To evaluate how throughput scales with the number of processors we ran experiments varying the number of physical processors from 1 to 20, and bound a single connection to each processor using our processor per connection implementation.

Figure 2.4 shows throughput versus number of processors (and connections) for both TCP and UDP protocol stacks. In these experiments buffers are "written" on the send side by repeatedly copying a 128 byte (the size of a second-level cache line) memory buffer into the 4KB of user data, and "read" on the receive side by repeatedly copying portions of the user data into a 128 byte buffer. The checksum is also computed on packets. Virtual processor per connection (VPPC) and thread per connection (TPC) data are omitted from Figure 2.4 since all three schemes yield essentially the same performance in this scenario. Note that throughput is highest for UDP send-side processing and lowest for TCP receive-side processing.

Figure 2.5 shows speedup corresponding to the throughput data presented in Figure 2.4. Here speedup is throughput normalized by the corresponding single processor throughput. Note that in all four experiments speedup is linear, and in the range of 14 to 17 at 20 processors.

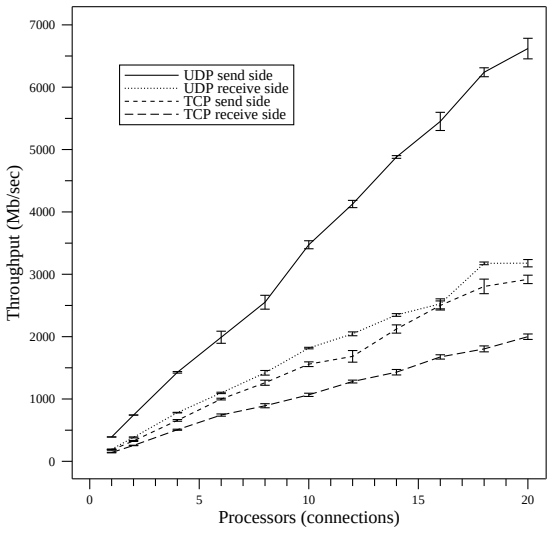

Figure 2.4. UDP and TCP Aggregate Throughput for Processor per Connection.

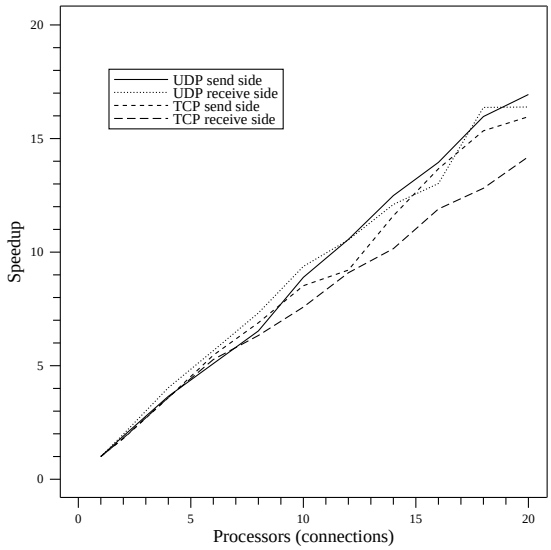

Figure 2.5. UDP and TCP Speedup for Processor per Connection.

Figures 2.4 and 2.5 give compelling evidence that connection-level parallel protocol stacks (even with TCP) can scale with the number of processors. These results are consistent with those reported in [118], which focus on the receive side for thread per connection parallelism. We extend their results in two significant ways. First, our UDP and send-side TCP results are new. Second, our receive-side results include the cost of fetching user data from memory, over the system bus, the first time it is touched.

The data presented in Figures 2.4 and 2.5 were all gathered on a 20-processor 150 MHz SGI Challenge. Unfortunately, we only had short-term access to this machine, so the throughput data presented here should not be directly compared with data elsewhere in this thesis, which were gathered on a 12-processor 100 MHz Challenge.

2.5 Throughput Sustainability with Respect to Connections

To measure the impact of adding connections on the overall throughput, we ran experiments varying the number of connections running on a 12-processor machine from 12 to 3072. We also varied the granularity of parallelism from coarse (processor per connection) to fine (thread per connection). We wanted to determine whether increasing parallelism (by

multiplexing fewer connections onto more virtual processors) is beneficial, because of increased scheduling flexibility and improved locality of memory reference, or detrimental, because of the overhead of using more virtual processors. Comparing the throughput of different implementations with the same number of connections answers this question.

In these experiments, threads are not wired to processors. Thus, the results presented here are for thread per connection (TPC) or virtual processor per connection (VPPC), even when the number of virtual processors is the same as the number of physical processors. We found that in all experiments wiring threads to processors decreased the aggregate throughput. In some configurations this decrease was as much as 50%. Furthermore, the version of IRIX we used schedules threads for cache affinity [10], which has been shown to benefit connection-level parallelism [114, 115].

Figure 2.6 shows the throughput of our multiprocessor sending data on 12 to 3072 TCP connections, where the checksum is computed. Figure 2.7 shows the corresponding results for our UDP protocol stack, without checksumming. Where VPPC data are shown, the number of virtual processors is indicated by V in the legend. We also conducted experiments using between 12 and 384 virtual processors (not shown in these figures) and found these configurations to yield throughput values which generally lie between those shown. It is worth noting that the curves for thread per connection stop at 384 connections since IRIX only allows a maximum of 512 threads in a process group to share an address space. We were unable to use every thread for sending (or receiving) data, and therefore limited the number of protocol processing threads to 384.

Several conclusions can be drawn from the data in Figures 2.6 and 2.7. The throughput for both TCP and UDP is sustained surprisingly well as the number of connections increases. For example, using 384 virtual processors, the overall throughput for TCP send-side processing only degrades from 960 Mbps to 800 Mbps as the number of connections increases from 384 to 3072. Furthermore, with more than 12 connections using the maximum number of virtual processors (i.e., using TPC or VPPC with 384 virtual processors)

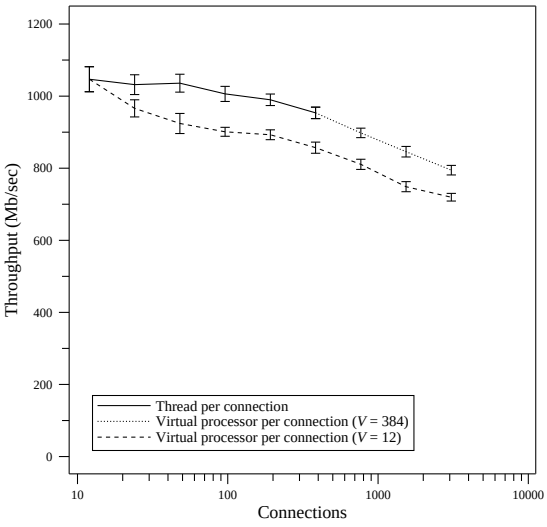

Figure 2.6. TCP Send-Side Aggregate Throughput from 12 to 3072 Connections.

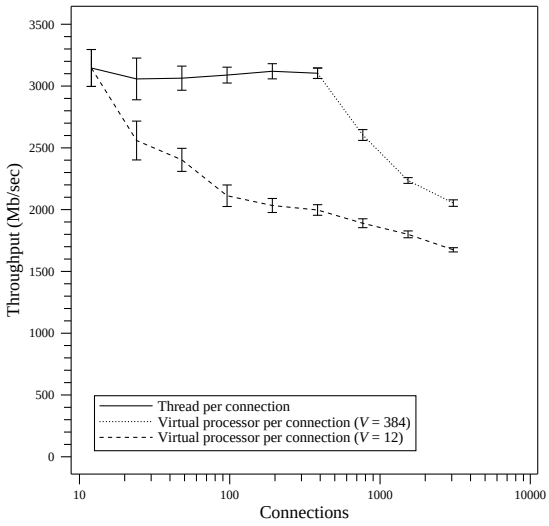

Figure 2.7. UDP Send-Side Throughput from 12 to 3072 Connections.

yields the best throughput. Using more virtual processors also has the advantage that it pushes out the point at which throughput starts to degrade. Specifically, for 12 virtual processors, the throughput starts to degrade immediately (going from 12 to 24 connections), while for TPC, the throughput does not degrade until there are 96 active connections. This is encouraging since under conditions where connection sources do not have an infinite amount of data to send (i.e., are occasionally idle), the scheduling flexibility gained by increasing the number of virtual processors widens the performance improvement over that achieved by using fewer virtual processors [115, 137].

Figures 2.6 and 2.7 also show a surprising result: the throughput of TCP is less sensitive to an increase in the number of connections than that of UDP. We suspect that this is because the code path for sending packets over UDP connections is shorter than the code path for TCP connections, making the UDP results more sensitive to conflicts in the caches on the multiprocessor. This phenomenon has been observed by others on a uniprocessor [92].

So far we have presented results only for experiments where the multiprocessor is the sender. However, for applications such as high-performance file service (which typically run on top of UDP), it is also interesting to examine the case where the multiprocessor is

the receiver. When clients write to files on the server, it is likely that consecutive packets arrive quickly enough that their processing can be amortized over a single virtual processor yield. If each receiving thread yields the virtual processor after processing two packets, the unit transfered is 8KB, a typical disk block size for networked file systems. We refer to processing multiple packets between a single virtual processor yield as processing packets in "bursts". Thus, a single packet is processed if the burst size is one, and a pair of packets are processed if the burst size is two, etc. Amortizing what are normally per-packet costs over multiple packets is a well known technique for improving networking performance.

Figure 2.8 shows processor per connection (PPC) throughput for a UDP receiver, where the burst size used by each connection is either one or two, and the checksum is not computed. Recall that the mechanism to ensure that application data is fetched from memory as it is first touched adds sufficient overhead that comparison with a greater number of virtual processors is difficult. However, for up to 96 virtual processors the results are all comparable to those shown for PPC. Comparing the dotted line in Figure 2.8 with the dashed virtual processor per connection curve in Figure 2.7 (for $V = 12$) indicates that the UDP receive side throughput degrades more gracefully than the send side, again suggesting that the results in Figure 2.7 are in part due to the impact of cache behavior on UDP's send-side code path. Figure 2.8 also shows that throughput improves up to 20% by processing received packets in pairs.

We also investigated the benefits of amortizing multiple packet transfers over a single virtual processor yield (i.e., processing packets in bursts) on the send side. This strategy is applicable in any situation where the size of data objects being transfered is large relative to the MTU for the network (e.g., image files). However, this strategy may not be applicable if the latency of sending data objects is crucial (e.g., packet voice or video). Since some simple experiments showed that this technique yields diminishing returns, we fixed the burst size at 16 packets (64KB) for our send-side experiments.

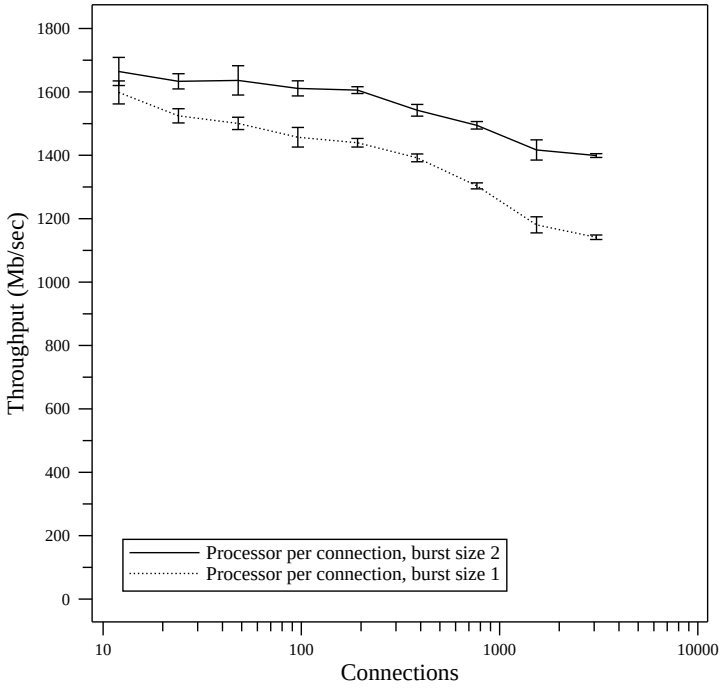

Figure 2.8. UDP Receive-Side Throughput from 12 to 3072 Connections.

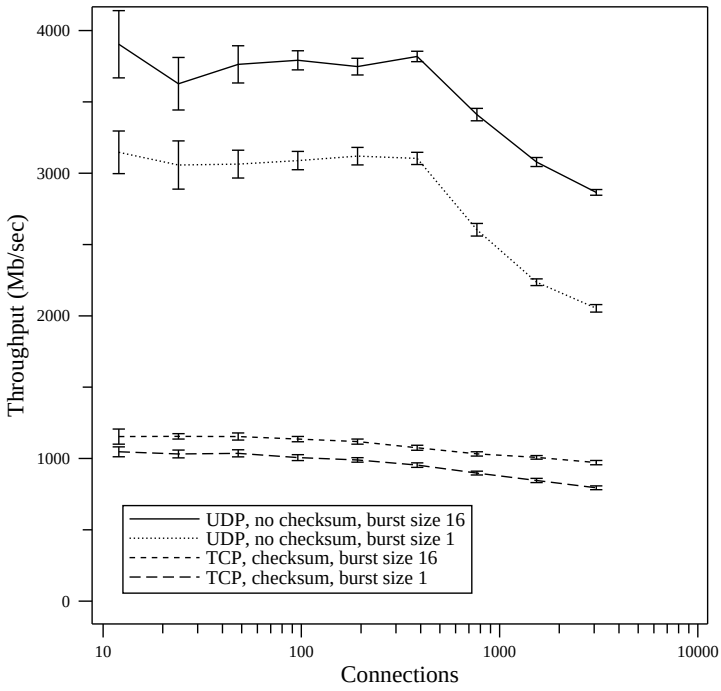

Figure 2.9. Impact of Processing Packets in Bursts on Send-Side Aggregate Throughput.

Figure 2.9 compares throughput for the maximum number of virtual processors (i.e., TPC or VPPC with 384 virtual processors) using a burst size of 16 with our previous results for a burst size of one (see Figures 2.6 and 2.7). Note that the throughput improves by 10-20% for TCP and up to 40% for UDP. For a burst size of 16, the gap between the throughput curves for 12 and 384 virtual processors in Figure 2.6 is narrowed. In fact, it becomes insignificant at 384 or more connections for TCP. This suggests that using bursts when there are a large number of active connections decreases the performance impact of the number of virtual processors in the system.

2.6 Survey of Related Work

Previous work on connection-level parallelism can be found in [49, 110, 116, 114, 115, 118]. The first performance study is given in [116], in which the authors measure the performance of 5 connections sending data over a TCP/IP protocol stack. Their results show that for multiple connections, using an asynchronous interface between protocol layers (to potentially achieve speedup within a single connection) yields inferior performance to executing the entire protocol stack as a single critical section. Thus, this work is the first to advocate thread per connection (as described in Section 2.2).

The most complete experimental study to date is [118], which examines a connection-level parallel protocol stack running on a 20-processor Sun SPARCCenter 2000 [24]. The authors show that using thread per connection, throughput scales well for 20 TCP connections, all receiving data. Their results show that speedup is linear up to 12 processors when presentation layer processing (e.g., XML or XDR) is included, and is 10 at 20 processors. Without presentation layer processing, speedup is linear up to 20 processors, where it is about 15. The authors conjecture that the speedup with XDR is clipped at 12 processors because the system bus becomes the bottleneck at this point.

The most complete simulation study to date is [114, 115], which examines the performance of packet-level parallelism and connection-level parallelism for UDP connections.

The arrival process for each connection is given by a packet train model. In this work, Salehi et al. demonstrate that processor per connection yields higher aggregate throughput than packet-level parallelism whereas packet-level parallelism provides lower latency. Furthermore, they show that packet processing time for thread per connection is lower than for processor per connection.

During and since the publication of our work there are a number of other researchers that have worked on parallelized protocols as a way of scaling the networking capacity of a server. These efforts include work on multi-gigabit applications [47, 90], parallelism at the network layer [28], and off-board processing [73]. Perhaps more similar to the work in this thesis are other techniques for implementing connection-level parallelism [57, 60], and aggregation of connections into even coarser units of parallelism [3].

Because of both scalability and cost benefits, many researchers have investigated the use of connection-level parallelism in cluster architectures [2, 4, 5, 8, 11, 22, 29, 30, 34, 39, 62, 61, 104, 103, 140]. Clusters are more scalable than a shared-memory multiprocessor because the memory, backplane and disk bandwidth all scale with the number of nodes. For large configurations, clusters are also typically much less expensive than SMP's with the same processing power. Clustering technology for web-based applications has also evolved in synergy with important enabling technologies such as Layer-4 and Layer-7 switches [16]. This evolution has led to many commercial products and solutions, including those offered by both IBM [85] and Microsoft.

2.7 Conclusions

In this Chapter we described our implementation of connection-level parallelism (CLP), and experimentally determined how throughput scales with the number of processors (from 1 to 20), and how throughput changes as the number of connections increases (from 12 to 3072).

We also investigated three other issues:

- The effect of the granularity of parallelism on performance.
- The effect of the transport protocol on performance.
- The impact of processing packets in bursts.

We found that connection-level parallel protocol stacks scale well with the number of processors for UDP and TCP, performing both send-side and receive-side communication. Furthermore, the throughput delivered is, for the most part, sustained as the number of connections increases. We also showed that for moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor.

Our results showed that thread per connection parallelism yields the best aggregate throughput. However, thread per connection may not be feasible if the number of connections exceeds the number of threads that can be reasonably supported by a particular multiprocessor. In this case, the best throughput is obtained by using virtual processor per connection and maximizing the number of threads in the system.

We also showed that if an application needs maximal throughput from a multiprocessor protocol stack, amortizing per-packet costs over multiple packets is worthwhile.

CHAPTER 3

FAIRNESS BEHAVIOR OF CONNECTION-LEVEL PARALLELISM

3.1 Introduction

In the results we have discussed in Chapter 2, the throughput measure of interest has been the overall, *aggregate* throughput of all of the connections. Another important measure of performance, however, is the throughput seen by *individual connections*. Recall that our experiments use a single thread for each virtual processor to send or receive packets. Furthermore, processing of packets for connections assigned to the same virtual processor is performed in a round-robin manner. Thus, any difference in throughput for connections assigned to the same virtual processor is at most one packet's worth of data over the sampling interval (e.g., 45 seconds for our implementation). However, we present both per-connection and per-thread throughput results for the sake of completeness. In describing these results, we will use the term "thread" to mean the thread performing protocol processing on a virtual processor.

Clearly, the extent to which a thread receives its fair share of the total throughput is influenced by the manner in which threads are scheduled by IRIX. With N threads, one might be tempted to assume that each thread receives approximately $1/N$ (i.e., roughly its "fair share") of this aggregate throughput, particularly when the per-thread throughput is measured over a sufficiently long period of time. As we will see, however, our results show there can be significant differences in the per-thread (as well as per-connection) throughput.

We quantify the degree to which our protocol implementations distribute aggregate throughput fairly by measuring the throughput seen by each connection (and thread) and

computing percentiles of the throughputs, between the maximum and minimum values. We report here on results from four configurations. In the next section we consider UDP send-side processing without checksumming, with both a moderate and large number of connections. In Section 3.3, we consider TCP send-side processing with checksumming, again varying the number of connections.

3.2 UDP Fairness

Figure 3.1 shows the difference in throughput that 384 connections see when they are serviced by different numbers of threads. The curves plot percentiles of per-connection throughput as a function of the number of threads. Figure 3.2 shows the corresponding per-thread throughput. The throughputs in Figure 3.2 are normalized by the average throughput seen by all threads. The dotted line in Figure 3.2, for example, shows the normalized throughput such that 75% of the threads receive a throughput of less than or equal to the y-axis value. The spread between the curves is thus a rough indication of the distribution of per-thread throughput seen by the number of threads indicated by the x-axis.

Note that in both Figures 3.1 and 3.2 there is significant range in the distribution of per-connection (and per-thread) throughputs for different numbers of threads. This is true for both small (12-24) and large (up to 384) numbers of threads. Furthermore, since connections assigned to threads are serviced round-robin, the curves in Figures 3.1 and 3.2 match in terms shape and slope. Given the similar nature of Figures 3.1 and 3.2, we turn our attention to Figure 3.2 since we are also able to obtain scheduling and latency measurements on a per-thread basis, but not a per-connection basis.

Figure 3.2 illustrates several interesting aspects of the server's behavior. First, we note that there can be a significant difference in the throughput seen by individual threads. For example, with 96 threads, the thread receiving the highest throughput receives approximately 10 times the throughput received by the thread with the lowest throughput. We found this quite surprising since with 96 threads, 1500 scheduling time quanta (in a 45-

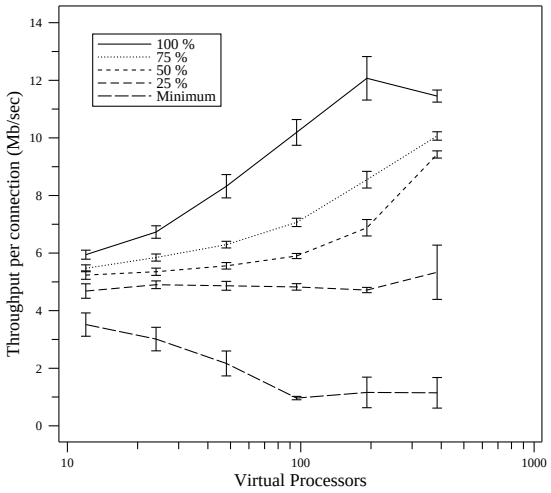

Figure 3.1. UDP Send-Side Throughput per Connection for 384 Connections.

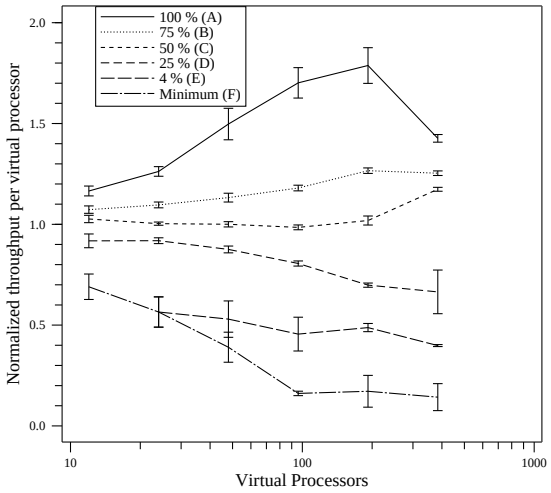

Figure 3.2. UDP Send-Side Throughput per Thread for 384 Connections.

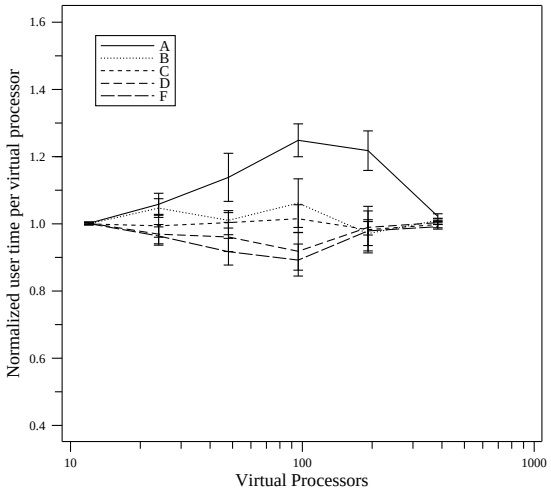

Figure 3.3. UDP Send-Side CPU Time for 384 Connections.

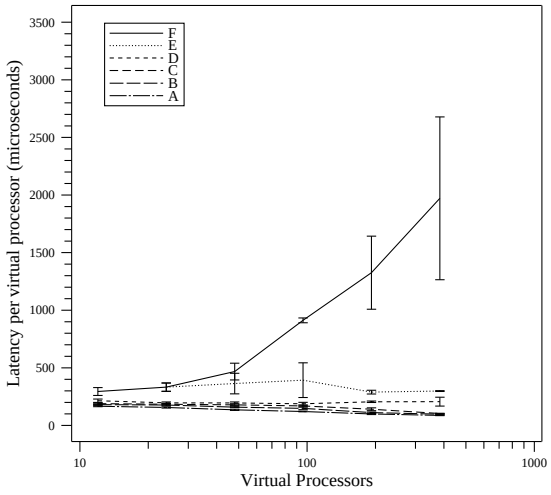

Figure 3.4. UDP Send-Side Latency for 384 Connections.

second interval), and 12 processors, each thread should have received approximately 190 time quanta – long enough to have averaged out the performance differences one might expect in the throughputs during the individual 30 ms time quanta.

Figure 3.2 also illustrates that the differences in throughput per thread are minimized when there are only 12 threads. Thus, from a fairness standpoint, a smaller number of threads is to be preferred. Recall from our discussion of Figure 2.7, however, that a smaller number of threads results in lower throughput. For example, with 384 connections, the aggregate throughput (as indicated in Figure 2.7) is 2.0 Gbps and 3.1 Gbps for 12 and 384 threads, respectively. Thus, while 12 threads provide a fairer allocation of throughput among threads, the aggregate throughput is lower.

We were surprised by the magnitude of the unfairness shown in Figure 3.2, and wanted to understand its cause. As we will see, the user time given to threads, and the latency processing packets, both appear to play a role in the unfair distribution of aggregate throughput to individual threads.

Figure 3.3 shows the user time given to the threads corresponding to the quartiles in Figure 3.2 (labeled A through D, and F). The user time shown is normalized by the average user time given to all threads. We stress that Figure 3.3 is *not* a quartile plot. Hence, curves in this Figure can cross, as seen for threads B and D at 192 threads. The similarity in shape and slope of the curves in Figures 3.2 and 3.3, at least up to 96 threads, suggests that the scheduler has some determining effect on fairness in this region. However, this is no longer the case with more than 96 threads, since the user time for threads B, C, D, and F are all similar in this region, yet deliver dramatically different throughput (as shown in Figure 3.2).

Figure 3.4 shows the average per-packet latency seen by the threads in Figure 3.2 (labeled A through F). Again, note that Figure 3.4 is not a percentile plot. The most noticeable feature of Figure 3.4 is the dramatic difference in latency between threads, when there are a moderate or large number of them. For example, thread E (which corresponds to the

4th percentile in throughput) and F (which corresponds to the minimum throughput), show differences of a factor of 2.3 to 6.3, at 96 threads and above. This suggests that the distribution of per-packet latency seen by the threads has a heavy tail in this region. One possible explanation for this is the memory reference behavior during protocol processing. These differences in latency have a dramatic affect on the per-thread throughput, especially when there are more than 96 threads.

We were also interested in examining the fairness behavior of our protocol stack as the number of connections increased from a moderate to large size. To compare our results with those described above, we used the same protocol stack (i.e., UDP send-side, without checksumming), the same range of threads (i.e., 12 to 384), and again measured values over a 45-second interval. Figures 3.6 through 3.8 show the fairness behavior of our server sending data over 3072 UDP connections. We discuss these figures in order, as before.

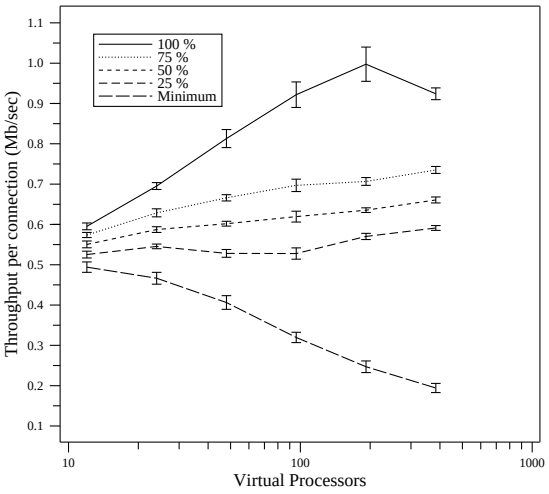

Figure 3.5. UDP Send-Side Throughput per Connection for 3072 Connections.

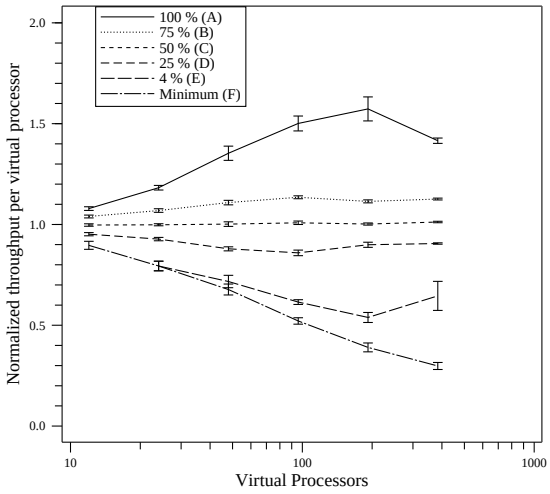

Figure 3.6. UDP Send-Side Throughput per Thread for 3072 Connections.

Figures 3.5 and 3.6 show percentiles of normalized per-connection and per-thread throughputs as a function of the number of threads for 3072 UDP connections. In Figure 3.6, note that the differences in per-thread throughput in this Figure are still significant (i.e., are up to a factor of 5), but are less than those in Figure 3.2, which exceed a factor of 10. In common with our results for fewer connections, Figure 3.6 also illustrates that the differ-

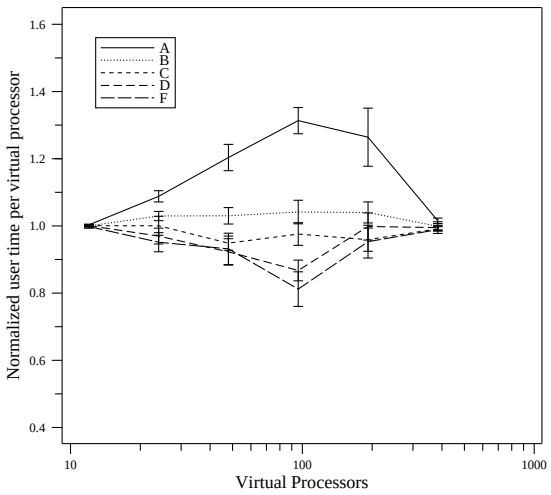

Figure 3.7. UDP Send-Side CPU Time for 3072 Connections.

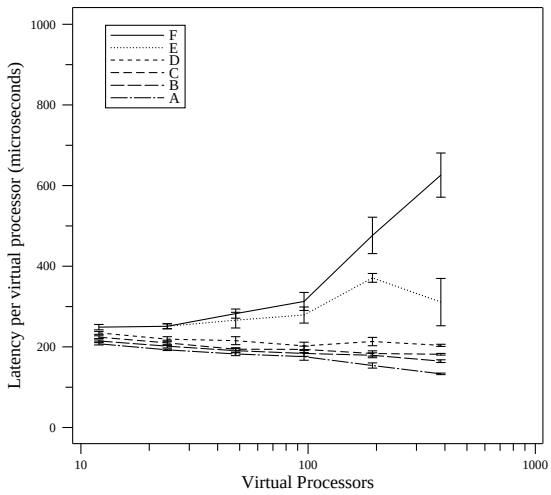

Figure 3.8. UDP Send-Side Latency for 3072 Connections.

ences in per-thread throughput are minimized when there are only 12 threads. However, this improvement in fairness behavior again comes at the cost of aggregate throughput. For example, when using 384 threads (at the extreme right of Figure 3.6) the aggregate throughput from Figure 2.7 is 2.1 Gbps. However, for 12 threads (at the far left of Figure 3.6) the aggregate throughput is 1.7 Gbps.

Figures 3.7 and 3.8 again show the effect of user time given to threads, and average per-packet latency, on the per-thread throughput. Figure 3.7 shows the normalized user time given to the threads corresponding to the quartiles in Figure 3.6 (labeled A through D, and F). The similarity in shape and slope of the curves in Figures 3.6 and 3.7, at least up to 96 threads, again suggests that the scheduler has some determining effect on fairness in this region. However, for more than 96 threads, the scheduling behavior again has less impact on fairness. Figure 3.8 shows the average per-packet latency seen by the threads in Figure 3.6 (labeled A through F). The most interesting feature of Figure 3.8 is the drop in average per-packet latency for the thread which delivers the lowest throughput (F), when compared with the corresponding curve in Figure 3.4. For example, with 384 connections and the same number of threads, the average per-packet latency for thread F is roughly 2000 microseconds (see Figure 3.4). For 3072 connections and 384 threads, the

per-packet latency for thread F is only 625 microseconds. In fact, over the range of threads we examined, the gap between the latencies of the thread delivering the 25th percentile throughput (D) and minimum throughput (F), is narrowed considerably as the number of connections is increased. Thus, while increasing the number of connections assigned to the same number of threads reduces aggregate throughput, it can improve the worst-case latency behavior. Although we can not specifically say why this is the case, it is easy to see how an increase in the number of connections could narrow the range of the distribution of the per-thread (and per-connection) throughputs. Consider per-thread throughputs when there are 384 threads (the point at which aggregate throughput is greatest). In Figure 3.2 there is only one connection per thread in this configuration. Thus, any effects memory reference behavior seen by a thread effect its connection throughput. On the other hand, in Figure 3.6 there are 8 connections per thread when there are 384 threads, and these connections are serviced round-robin one packet at a time. This means that any impact on throughput due to favorable (or unfavorable) effects of memory references are distributed across eight connections. This amortization of differences in per-connection throughput across multiple connections would tend to narrow the range of the throughputs, as seen in going from 384 to 3072 connections.

In addition to understanding the effect of increasing the number of connections at our server, we also wanted to understand the effect on fairness behavior of using TCP as the transport layer protocol instead of UDP. The next sections examines this issue.

3.3 TCP Fairness

Considering the fairness behavior of TCP in addition to UDP is important since much of the continuous media delivery over the World-Wide Web today still uses TCP. Although providing throughput guarantees for a protocol that retransmits data may not make sense, improving fairness to reduce the variance in throughput delivered to a connection does. Reducing variance in throughput provides the benefit of smoothing traffic as it is delivered

to the network, and reducing the buffer space required at the client as the media is displayed to the user.

This section investigates the fairness behavior of TCP, and compares it to the fairness behavior of UDP presented in the previous section. Before we begin to look at TCP, it is important to point out the important differences between UDP and TCP. First, TCP is a more complex protocol than UDP - it provides reliable delivery of data without loss, duplication, reordering, or corruption. Even though our experiments don't include dealing with such errors, the TCP code that is required to detect and recover from errors is still executed for every TCP packet sent or received. Part of the error detection that is performed for every packet is the checksum computation.

Figure 3.9 shows the per-connection throughput for 384 TCP connections all sending data on our 12 processor server. For each packet sent, the TCP checksum is computed. The curves plot per-connection throughput as a function of the number of threads. The dotted line in Figure 3.9, for example, shows the throughput such that 75% of the connections receive a throughput of less than or equal to the y-axis value. The spread between the curves is thus a rough indication of the distribution of per-connection throughput seen by the number of threads.

Figure 3.10 shows the per-thread throughput for the same TCP connections as in Figure 3.9. Note that Figures 3.9 and 3.10 exhibit the same general features, just like the corresponding per-connection and per-thread throughput curves for UDP (see Figures 3.1 and 3.2). The differences in per-connection and per-thread throughput are also smallest for TCP when there are only 12 threads. In addition, there is still a significant difference in the throughput seen by individual threads for TCP (see Figure 3.10), although not as great as seen for UDP (see Figure 3.2). For example, with 96 threads the thread receiving the highest throughput receives approximately 3 times the throughput received by the thread with the lowest throughput. Recall that for UDP this ratio was higher, namely a factor of 10.

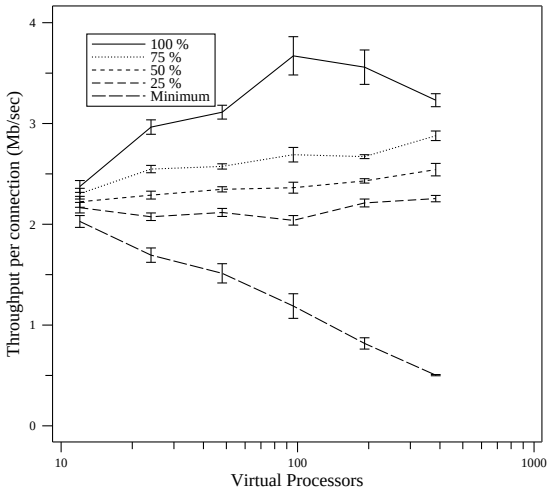

Figure 3.9. TCP Send-Side Throughput per Connection for 384 Connections.

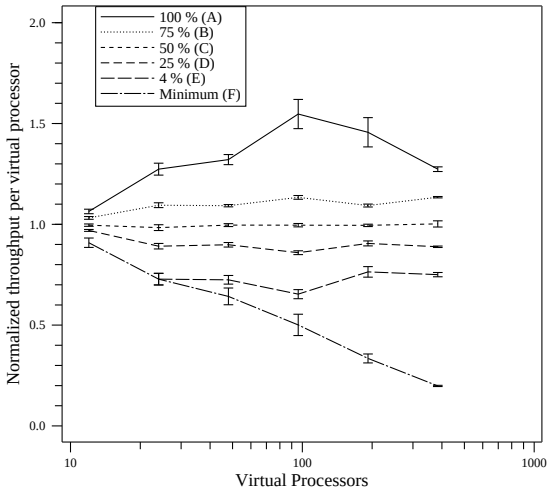

Figure 3.10. TCP Send-Side Throughput per Thread for 384 Connections.

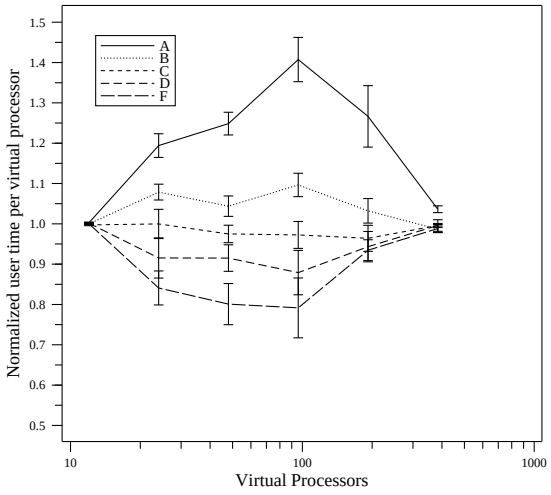

Figure 3.11. TCP Send-Side CPU Time for 384 Connections.

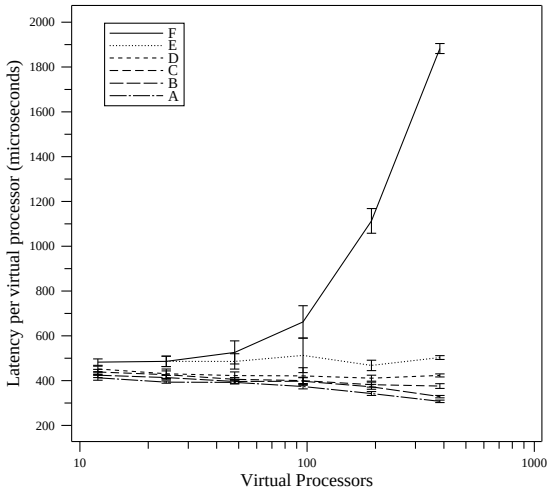

Figure 3.12. TCP Send-Side Latency for 384 Connections.

Figure 3.10 shows that when considering fairness, in terms of how throughput is distributed among TCP connections, having a single thread per processor is best. Unfortunately this is in number of threads for which aggregate throughput is lowest. For example, with 384 connections, the aggregate throughput (as indicated in Figure 2.6) is 950 Mbps and 850 Mbps for 12 and 384 threads, respectively. Thus, while 12 threads provide a fairer allocation of throughput among both connections and threads, the aggregate throughput is lower.

One interesting difference between the per-thread throughput results for TCP and UDP is difference in the way the worst-case per-thread throughput degrades. As shown in Figure 3.2, for UDP the worst case throughput drops off rapidly, but levels off at 96 threads and above, In Figure 3.10, we see that for TCP the worst case throughput starts of much higher (relative to the 25th percentile, and the other percentiles) but falls off steadily all the way from 12 to 384 threads. In fact, for TCP the largest difference between the maximum and minimum per-thread throughputs occurs when there are 384 threads each servicing one connection. At this point the ratio of the maximum to minimum per-thread throughput is 5. Although we can not definitively says why this is the case, it is worth pointing out that our TCP experiments have an additional risk in their memory reference behavior that is not present in our UDP experiments. Because TCP computes a checksum for every packet, there is an entire page that is referenced an additional time after it is loaded. This means that if the both the data source buffer and communication buffer experience unfavorable memory reference behavior, TCP will see these effects compound three times: once as the data source buffer is read, once as the communication buffer is written (i.e., loaded from the source buffer), and again as the communication buffer is checksummed. For UDP, only the first two page references are made (to load the communication buffer), and the checksum computation is not performed.

Figure 3.11 shows the user time given to threads corresponding to the quartiles in Figure 3.10 (labeled A through D, and F). The user time shown is normalized by the average

user time given to all threads. Figure 3.11 is not a quartile plot, which means that curves in this Figure could cross (but obviously don't). The similarity in shape and slope of the curves in Figures 3.10 and 3.11, at least up to 96 threads, suggests that the scheduler has some determining effect on fairness in this region. However, this is not the case with more than 96 threads. The user time for threads C, D and F are all similar in this region, yet deliver quite different throughputs (as shown in Figure 3.10).

It is interesting to compare the user time given to threads servicing TCP connections (see Figure 3.11) to the user time given to threads servicing UDP connections (see Figure 3.3). These graphs are similar in that they exhibit roughly the same worst-case behavior (at 96 threads), and best-case behavior (at 12 and 384 threads). We were somewhat surprised by these results since our protocol stack with TCP requires an additional thread per physical processor when compared with our UDP stack. Furthermore, 12 of these threads (one per physical processor) are scheduled every 100-200 milliseconds. These threads manage the timers associated with TCP's error control. We expected the scheduling of these timer threads to interfere with the scheduling of the threads sending data, and thus make the range of user times assigned to threads much worse for TCP.

Figure 3.12 shows the average per-packet latency seen by the threads in Figure 3.10 (labeled A through F). As for UDP, the most dramatic feature of Figure 3.12 is the large difference in latency between threads when there are a moderate or large number of them. For example, threads E (which corresponds to the 4th percentile in throughput) and F (which corresponds to the minimum throughput) show differences of a factor of 2 to 6 at 96 threads and above.

As with the CPU time associated with each thread, the per-packet latency results for TCP in Figure 3.12 are very similar to the results for UDP (see Figure 3.4). Figure 3.12 shows the same heavy tail in the region where there are 96 or more threads servicing 384 connections. Again, this is most likely due to memory reference behavior during protocol processing. It is interesting to note that the variance of the throughput received by threads

E and F is lower for TCP than UDP, as indicated by the narrower confidence intervals. This is most likely due to the fact that computing the checksum in our TCP protocol stack reduces variability in memory reference behavior. This is consistent with previous results which study the cache behavior of network protocols on a uniprocessor. The differences in latency when there are moderate or large numbers of threads have a significant effect on per-thread throughput, as shown in Figure 3.10.

To complete our fairness results, we examined the fairness behavior of TCP sending data over 3072 connections. We used the same parameters for our experiments as was used to obtain the TCP results for 384 connections. Figures 3.13 through 3.16 present these results. As before, the number of threads servicing the connections is varied from 12 to 384 in each figure. Figures 3.13 and 3.14 show per-connection and per-thread throughputs, respectively. Figure 3.15 presents the relative CPU time given to the threads in Figure 3.14, and Figure 3.16 shows the average per-packet latency for the same threads. We describe the results in these graphs by comparing their significant features with both their UDP (3072 connection) and TCP (384 connection) counterparts.

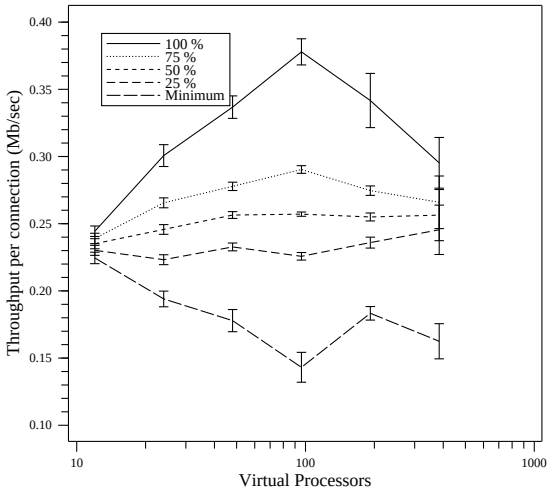

Figure 3.13. TCP Send-Side Throughput per Connection for 3072 Connections.

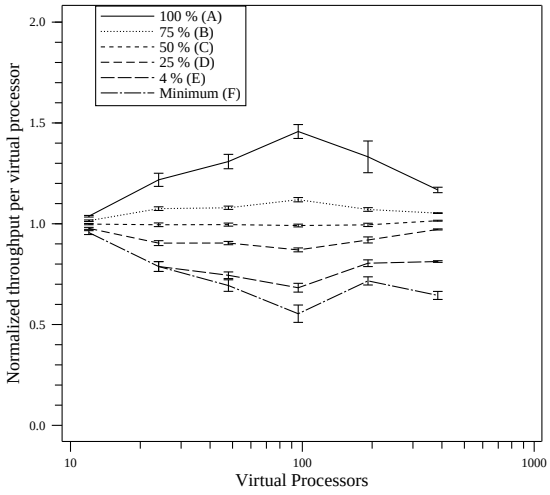

Figure 3.14. TCP Send-Side Throughput per Thread for 3072 Connections.

The per-thread throughput for 3072 TCP connections (see Figure 3.14) behaves as expected. In changing protocols to TCP (from UDP), fairness improves in general. This was

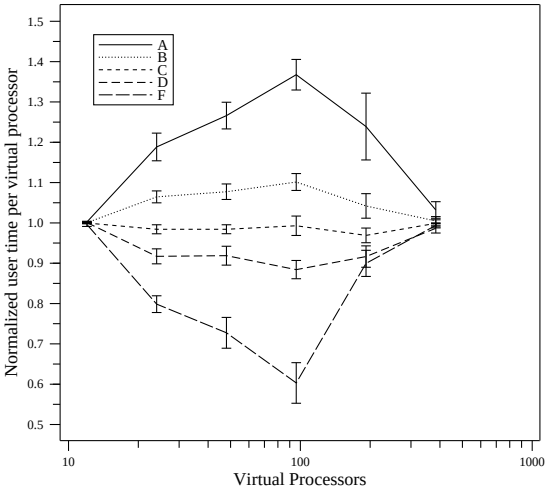

Figure 3.15. TCP Send-Side CPU Time for 3072 Connections.

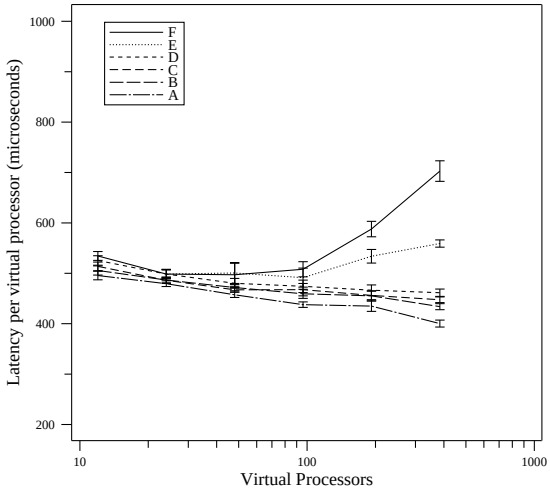

Figure 3.16. TCP Send-Side Latency for 3072 Connections.

also the case for 384 connections (see Figures 3.10 and 3.2). In increasing the number of connections to 3072 (from 384), fairness also improves. This was also the case for UDP (see Figures 3.6 and 3.2). There is one important exception to this expected behavior. The thread that sees the lowest normalized throughput in Figure 3.14 (thread F) has its minimum at 96 threads, and then increases for 192 threads, and subsequently decreases again. Note also that the per-connection curves in Figure 3.13 have the same characteristic shape as the per-thread curves in Figure 3.14. This was also true in our other three experimental configurations.

The results showing how CPU time is allocated to threads (see Figure 3.15) conforms to what one would predict, given the results from the other experimental configurations. The range of relative CPU time allocations is greater for 3072 TCP connections than for 3072 UDP connections (see Figure 3.8). Furthermore, the general form of the curves for both TCP CPU time allocations (for 384 connections in Figure 3.11, and 3072 connections) is the same.

Our results for average per-packet latency for 3072 TCP connections (see Figure 3.16) also conforms to what one would predict, given the results from the other experimental configurations. The thread that delivers the worse throughput to its connection(s) also sees

the greatest per-packet latency, and the magnitude of the tail of this latency (at 192 and 384 threads) drops by a factor of three as connections are added. We suspect that this is due to the fact that any unfavorable memory reference behavior experienced by a small number of connections is amortized over more connections (when there are 3072 of them, instead of 384 as in Figure 3.12). Furthermore, comparing Figures 3.16 and 3.8, the results for 3072 TCP and UDP connections are quite similar. As before, the most significant difference is the reduction in variance in the TCP results, presumably due the checksum computation introduced for our TCP experiments. It is also surprising that the nominal values of the TCP and UDP per-packet latencies are so similar when a large number of threads are used to serve the same number of connections. Specifically, the largest per-packet latency is about 2 milliseconds in both Figures 3.12 and 3.4, and 700 microseconds in Figures 3.16 and 3.8. These results again suggest that memory reference behavior plays an important role in determining per-thread throughput when there are a large number of threads sending data through the protocol stack.

Most of the recent work on fairness properties of network communication has evolved from earlier work on cluster architectures. The common theme within this research area is developing effective techniques for supporting performance isolation, even in the presence of over-subscribed resources. Some notable recent projects are those at Rice [5], Hewlett-Packard [31], and the University of Massachusetts [25, 26, 27].

3.4 Conclusions

In this Chapter we studied how aggregate throughput is distributed across connections in our implementation of connection-level parallelism (CLP). We find that in many cases throughput is *not* distributed fairly among connections, or threads.

We quantified whether our protocol implementations were distributing aggregate throughput fairly by measuring the throughput seen by each connection and thread and computing percentiles of the throughputs, between the maximum and minimum values. We presented

results for both UDP and TCP, servicing both a moderate and large number of connections. Our results show that matching the number of threads to the number of physical processors, while scheduling connections assigned to each thread (or virtual processor) in a round-robin manner, yields the best fairness behavior. However, an improvement in fairness may come at the cost of aggregate throughput.

We also found that there can be significant differences in the per-connection (as well as per-thread) throughput. We demonstrated that scheduling and memory reference behavior both play a role in the unfair distribution of aggregate throughput. Furthermore, fairness problems tend to increase as more threads are used in the protocol stack, and this increase is due to greater differences in memory reference behavior seen by the threads.

CHAPTER 4

SUPPORT FOR CONTINUOUS MEDIA

4.1 Introduction

One obvious trend in the Internet is that text and image data (e.g., as seen today on the World-Wide Web) will become increasingly enriched with continuous media, such as voice, video and sensor data. This trend will be driven by the desire to use the Internet for commerce, education, entertainment, monitoring and control, where multimedia presentation has tangible benefits. In such applications, large scale servers will need to support high bandwidth communication, some of which will be real-time (e.g., for voice and video). Furthermore, since some of these applications will be highly interactive (e.g., shopping online, browsing audio/video libraries, and virtual reality), the opportunity to buffer data at the client may be limited to a few milliseconds worth of data. Thus, servers will need to "play out" real-time data, at a rate which closely matches the rate at which it was originally recorded. Providing some form of quality-of-service guarantee to a set of connections, each of which begins at the disk subsystem on the server and ends at an output device on a client, is therefore a crucial requirement for the operating system, and communication subsystem therein.

We consider two classes of quality-of-service continuous media here. The first is for applications where the playout rate is required to exactly match an original fixed recording rate (we refer to these as constant bit rate, or CBR, applications). In this case, the protocol stack and operating system (as well as other system components) should provide service where throughput delivered to connections is constant and guaranteed. We refer to this service class as service with hard throughput guarantees. Other applications are

more forgiving. These applications (sometimes referred to as “rate adaptive”) can adjust their playout rate as they detect that the available bandwidth being delivered to connections changes. In this case the protocol stack and operating system need to distribute the available bandwidth equitably among all connections. We refer to this service class as service with soft throughput guarantees.

Experimentally demonstrating how a connection-level parallel protocol stack can provide throughput guarantees (both soft and hard) to a number of real-time connections is the subject of this Chapter. In the previous Chapter, we showed that a collection of threads sending data on a moderate or large number of connections exhibit unfair behavior. In other words, some connections obtain substantially greater throughput than others, when their throughput is measured over reasonably long periods of time (e.g., 45 seconds). Furthermore, we demonstrated that scheduling behavior and memory reference behavior both seem to play a role in this unfairness.

Clearly the unfairness we have observed is an obstacle to delivering throughput guarantees (either hard or soft) to connections. The remainder of this Chapter describes and presents results for a technique, called *throughput-based scheduling*, that provides a mechanism whereby threads monitor and report their per-connection throughput to the scheduler. To support soft throughput guarantees, the scheduler balances load by reallocating work (in this case connections) among threads. We refer to this technique as *soft throughput-based scheduling*. This is quite different from traditional scheduling mechanisms, which might allocate more CPU time to the threads with more work to do (i.e., more connections to service). To support hard throughput guarantees, the scheduler schedules threads according to their “utilization” (i.e., how much of their current capacity is consumed by the throughput they have been assigned to deliver). We refer to this technique as *hard throughput-based scheduling*. Such monitoring of thread utilization has the additional potential of allowing servers providing hard throughput guarantees to reject requests for service when all threads are highly utilized.

We introduce the notion of a *scheduling signature* in this Chapter which shows how CPU time is distributed among threads. In examining baseline results (i.e., results from experiments with no throughput-based scheduling), the scheduling signature for a particular experiment is an illustrative way of visualizing what impact the scheduler has on *per-thread* throughputs. Thus we can draw conclusions about the relative contribution of the scheduler versus memory references to the unfairness in throughput delivered by different threads. For experiments using either hard or soft throughput-based scheduling, the signature has the additional advantage of showing any perturbation that the throughput-based scheduling mechanisms have on the CPU time allocated to threads performing protocol processing.

The results presented in this Chapter evaluate how well throughput-based scheduling performs in providing throughput guarantees to all active connections on a shared-memory multiprocessor server. In this evaluation, we assume that each connection's quality of service requirements are the same. However, we note the techniques that we study can be generalized to support different service requirements for different connections.

4.2 Service with Soft Throughput Guarantees

A necessary step in providing soft throughput guarantees is eliminating any unfairness in the throughput delivered to connections sending continuous media data which have the same quality of service requirements. Clearly, the extent to which a connection receives its fair share of the total throughput is influenced by the manner in which connection protocol processing is scheduled. For the results in the previous Chapters, the processing of packets for connections assigned to the same virtual processor is performed in a round-robin manner. However, the scheduling of the threads running on virtual processors is controlled by the IRIX operating system, which uses a traditional UNIX time-share scheduler by default, with a 30 ms time quantum.

We have shown that the scheduler and memory reference behavior are obstacles to delivering soft throughput guarantees to connections. If memory reference behavior was the

same for all connections, then allocating CPU time to threads in proportion the the number of connections they are servicing would be sufficient to provide soft throughput guarantees. Techniques for scheduling a resource (such as CPU time) to support proportional resource sharing are well understood. However, the differences in throughput due to differences in memory reference behavior, will still persist even if threads are given proportional CPU allocation.

In the remainder of this Chapter, we investigate techniques for eliminating differences in per-connection throughput. In this section we focus the goal of doing this with the goal of providing *soft* throughput guarantees to connections. In the next section our goal is to provide *hard* throughput guarantees to connections.

Our implementation of throughput-based scheduling, which provides both soft and hard guarantees, uses a common component which allows the scheduler to monitor the per-connection throughput for each thread, and make scheduling decisions accordingly. In designing and experimenting with different implementations, we found it was important to capture the most recent "service" that a thread receives (from the IRIX scheduler and the memory system), as well as capture the long-term "average" service. Therefore, in our implementation of throughput-based scheduling, thread throughputs are monitored every 100 milliseconds, and used to update exponential moving averages. Our scheduler then uses the throughputs measured by the threads to make the appropriate resource management decisions, depending on whether soft or hard throughput guarantees are being provided.

4.2.1 Soft Throughput-Based Scheduling

To provide soft throughput guarantees, throughput-based scheduling migrates connections between threads in an attempt to equalize average per-connection throughput. Figure 4.1 shows an example of how this works using the throughput monitoring mechanism described earlier. In Figure 4.1, a single connection is migrated from virtual processor i to virtual processor j because the maximum per-connection throughput was measured

on *i*, and the minimum per-connection throughput was found on *j*. So that the throughput estimates are reasonable, load balancing between virtual processors (i.e., connection migration) is performed every 200 milliseconds, half as often as throughput sampling.

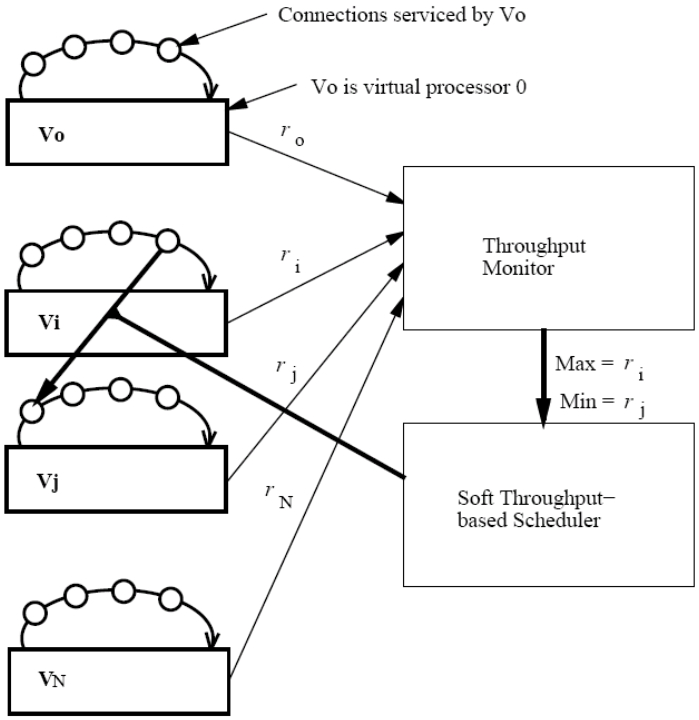

Figure 4.1. Example of Soft Throughput-Based Scheduling.

The throughput monitor shown in Figure 4.1 runs every 100 milliseconds and updates an exponential moving average of each thread's throughput. Recall that this averaging method captures both short-term and long-term aspects of the service that a particular thread receives from the operating system and the memory system. The short-term effects include how much time a thread ran in the last 100 ms (if at all) and whether memory references were favorable or not. The long-term effects include any bias that the IRIX scheduler has toward allocating more or less than a thread's fair share of the overall CPU time over intervals measured in seconds. We saw evidence for such bias in the previous Chapter that is supported by the scheduling signatures we'll see in this Chapter.

There are two other design decisions that we made in our implementation of soft throughput-based scheduling. The first was the gain for the exponential moving average (α). The second was the policy for choosing which connection to migrate (e.g., from thread i to thread j in Figure 4.1). We experimented with several combinations for both of these. We varied α by factors of 2 between 1/2 and 1/128, and tried FIFO, LIFO, and random policies for picking which connection to migrate. We found that using a value of $\alpha = 0.875$ to measure throughputs, and FIFO to migrate connections performed best.

As we will see, soft throughput-based scheduling has the desired effect of moving connections from threads experiencing unfavorable memory reference behavior to threads experiencing favorable memory reference behavior. We also find that wiring virtual processors to physical processors improves how equitably aggregate throughput is distributed among connections when using soft throughput-based scheduling.

4.2.2 Results for Soft Throughput Guarantees

We evaluate how effectively throughput-based scheduling performs in terms providing soft performance guarantees to connections in two configurations. Our performance metric of interest is the difference between the connection that receives the maximum throughput and the connection that receives the minimum. In the first configuration we evaluate soft throughput-based scheduling using the same experimental setup described in the previous Chapter (where threads are not wired to processors). In the second configuration we wire threads running on virtual processor to physical processors. This has the effect of suppressing IRIX's affinity-based scheduling. As we will see, wiring threads to processors improves how fairly CPU time is distributed among threads as well as how fairly aggregate throughput is distributed among connections.

Figure 4.2 and Figure 4.3 show the per-connection throughput for UDP send-side experiments with 384 connections where the number of threads servicing the connections is varied. Figure 4.3 shows throughput where connections are migrated between threads,

but in Figure 4.2 connections remain bound to threads. Note that soft throughput-based scheduling (accomplished by periodically migrating connections between threads) is effective in significantly reducing the gap between the maximum and minimum per-connection throughput, especially when the number of threads is small (i.e., less than 96). Soft throughput-based scheduling is less effective with moderate to large numbers of threads for several reasons. First, given the way the cache affinity scheduling algorithm in IRIX works, it is more likely that a thread moves from one processor to another between periods that it runs when there are more threads. Specifically, IRIX degrades a thread's affinity for the last processor it ran on in proportion to the amount of time it spends waiting to run again. Thus, the more threads there are running on the multiprocessor, the less likely a thread is to return to the processor it ran on the last time it was scheduled. Since moving from one processor to another makes memory reference behavior less predictable, it is more difficult for a thread to accurately measure and predict a stable throughput value. Furthermore, because there are fewer connections per-thread with more threads, the impact that a single connection which is migrated has on the source and destination thread is much greater. Again, this makes the algorithm monitoring throughput less likely to accurately measure and predict throughput. Thus, soft throughput-based scheduling is effective when the number of threads is either small or moderate (i.e., in the range of 12-48 in these experiments). Furthermore, when the number of threads is the same as the number of processors, soft throughput-based scheduling is very effective. This is shown by the dramatic reduction in the range of per-connection throughputs using 12 threads from Figure 4.2 to Figure 4.3. Specifically, the difference between the maximum and minimum throughputs is 40% in Figure 4.2, but negligible in Figure 4.3.

Figure 4.4 and Figure 4.5 show the *per-thread* throughput for the same UDP send-side experiments as Figures 4.2 and 4.3. It is interesting that in the region where soft throughput-based scheduling narrows the gap between the maximum and minimum per-connection throughput, the range of per-thread throughputs is also reduced. This suggests that in this

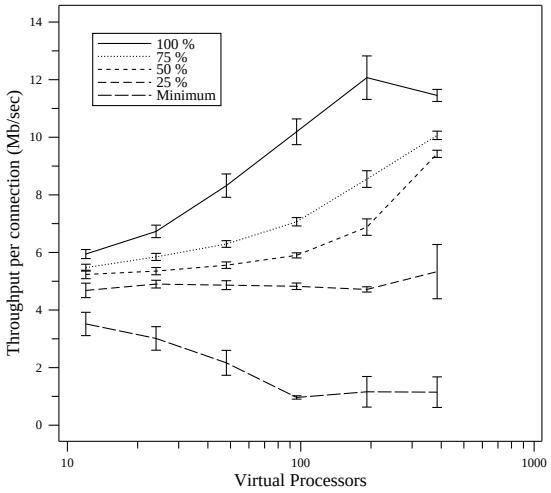

Figure 4.2. UDP Send-Side Throughput per Connection for 384 Connections.

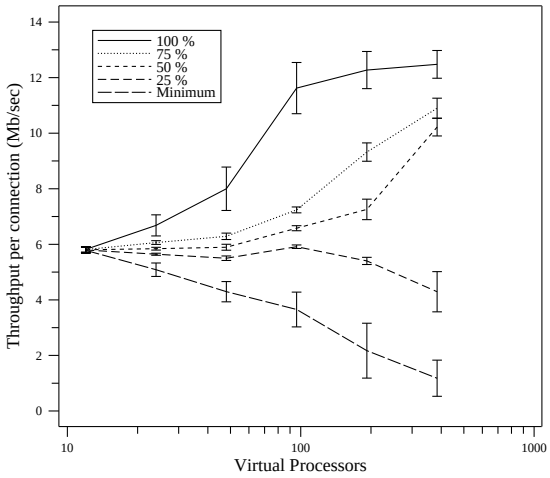

Figure 4.3. UDP Send-Side Throughput per Connection for 384 Connections, Soft T-based Scheduling.

region (where there are less than 96 threads), throughput-based scheduling is effective at distributing the negative effect of poor memory reference behavior among the threads. On the other hand, with 192 threads connection migration actually increases the range of per-thread throughputs. This is interesting given that soft throughput-based scheduling is ineffective, but not harmful, with respect to changing per-connection throughput with 192 threads (going from Figure 4.2 to Figure 4.3).

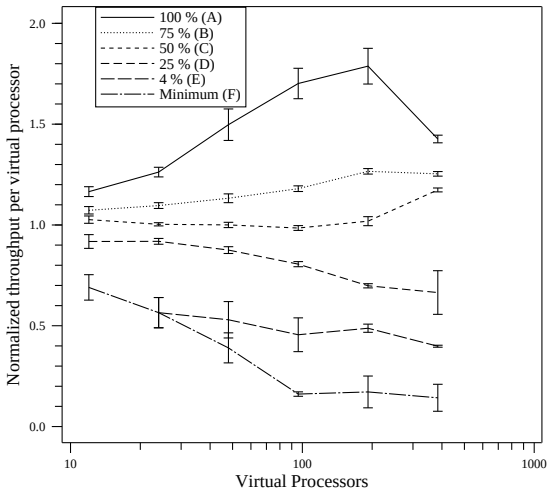

Figure 4.4. UDP Send-Side Throughput per Thread for 384 Connections.

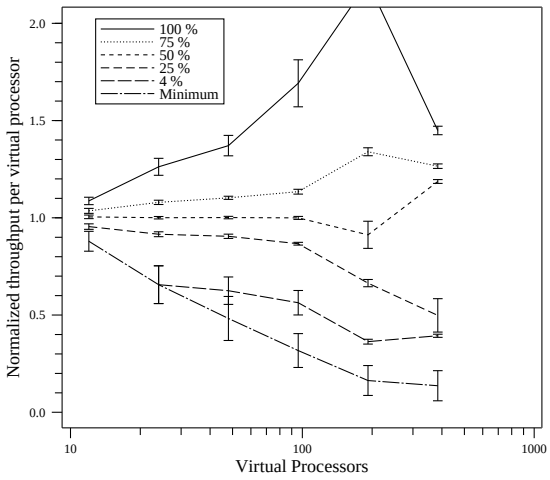

Figure 4.5. UDP Send-Side Throughput per Thread for 384 Connections, Soft T-based Scheduling.

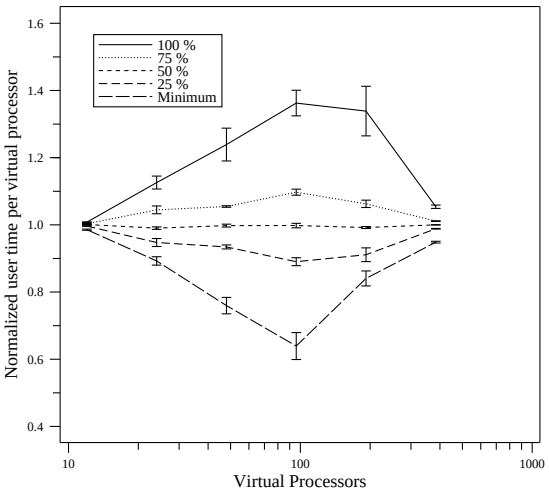

Figure 4.6. UDP Send-Side CPU Time for 384 Connections.

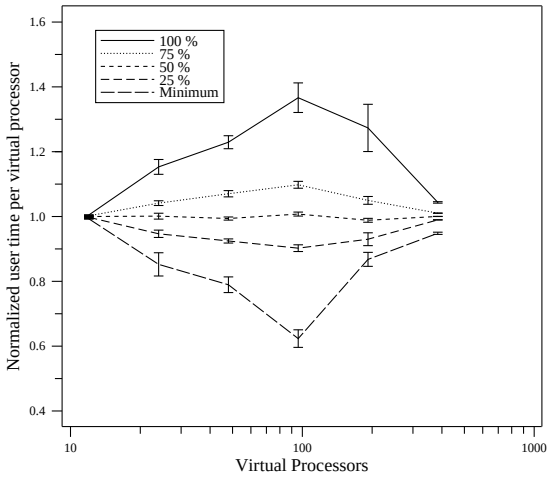

Figure 4.7. UDP Send-Side CPU Time for 384 Connections, Soft T-based Scheduling.

Figure 4.6 and Figure 4.7 show the scheduling signatures for the experiments with and without soft throughput-based scheduling. These Figures show the differences in CPU time that threads see over the duration of our experiments (i.e., 45 seconds). The curves plot quartiles of CPU time as a function of the number of threads. The CPU times are normalized by the average CPU time seen by all threads. Note that Figure 4.6 is different from Figure 3.3 in that Figure 3.3 gives the CPU times for 6 specific threads out of the number indicated by the x-axis whereas the spread between the curves in Figure 4.6 is truly an indication of the distribution of per-thread CPU time seen by the threads.

Perhaps the most remarkable result in comparing Figures 4.6 and 4.7 is how little impact throughput-based scheduling has on the scheduling signature. Figures 4.6 and 4.7 are amazingly similar! In both Figures the best case fairness behavior of the scheduler occurs when when there is only one thread per processor. Likewise, the worst case behavior of the scheduler occurs with a moderate number of threads (between 96 and 192). In this region the threads that receive the most CPU time receive as much as a factor of three more CPU time than the threads that receive the least CPU time.

It is also interesting to see how scheduling and memory referencing effects contribute differently to per-thread throughput. Consider Figures 4.4 and 4.6. Figure 4.4 shows normalized per-thread throughput whereas Figure 4.6 shows normalized per-thread CPU time. Thus, memory reference behavior is responsible for the difference in the spread between the curves going from Figure 4.6 to Figure 4.4. The same is also true going from Figure 4.7 to Figure 4.5. However, Figures 4.5. and 4.7 have the complication that the set of connections that each thread services changes as connections are migrated. These results are consistent with those in Chapter 3 in suggesting that using more threads to service the same number of connections causes memory reference behavior to have an increasingly negative effect on how well throughput is distributed among threads (and connections).

Thus far, we have seen that both scheduling and memory reference behavior are obstacles to delivering throughput guarantees to connections in the presence of IRIX's affinity-based scheduling. In addition, we have seen that we can deliver soft throughput guarantees when the number of threads in our connection-level parallel protocol stack is the same as, or close to, the number of processors. We also wanted to evaluate the performance of soft throughput-based scheduling in the absence of affinity-based scheduling. We accomplished this by wiring threads to processors and reproducing the experiments that generated the results in Figures 4.2 through 4.7. As we'll see from the scheduling signatures, this significantly reduces the range of CPU times allocated to threads during the experiments.

Figures 4.8 and 4.9 show the per-connection throughput for UDP send-side experiments with 384 connections, where the number of threads servicing the connections is varied. As before, Figure 4.9 shows throughput where connections are migrated between threads, but in Figure 4.8, connections remain bound to threads. Comparing Figures 4.8 and 4.9, it is clear that throughput-based scheduling is quite effective at delivering soft throughput guarantees to connections by evenly distributing aggregate throughput across connections. Again, soft throughput-based scheduling performs best when the number of threads matches the number of processors. However, Figure 4.9 also shows that it is robust up to 96

threads. Furthermore, we observe a significant improvement in the per-connection throughput delivered to the connection that receives the minimum throughput (labeled “Minimum” in these figures). This suggests that in the region where soft throughput-based scheduling is effective, most of it’s benefit is due to eliminating the poor memory reference behavior seen by the connection(s) that receive the smallest allocation of throughput.

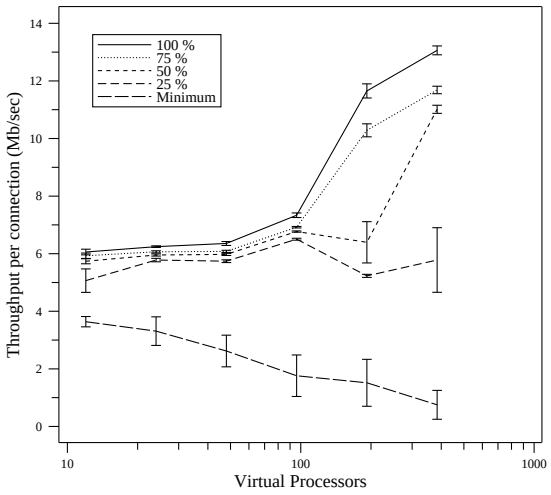

Figure 4.8. UDP Send-Side Throughput per Connection for 384 Connections, Threads Wired.

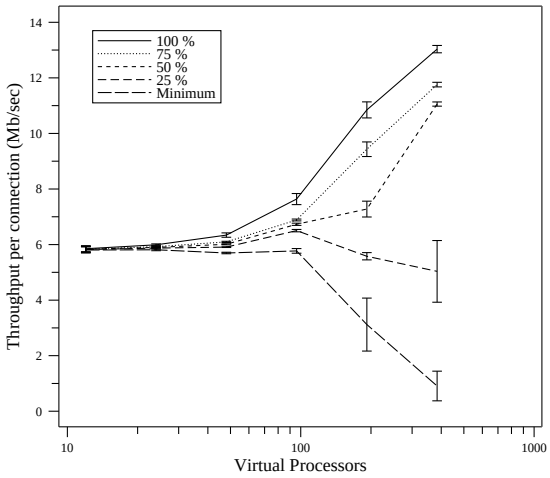

Figure 4.9. UDP Send-Side Throughput per Connection for 384 Connections, Threads Wired, Soft T-based Scheduling.

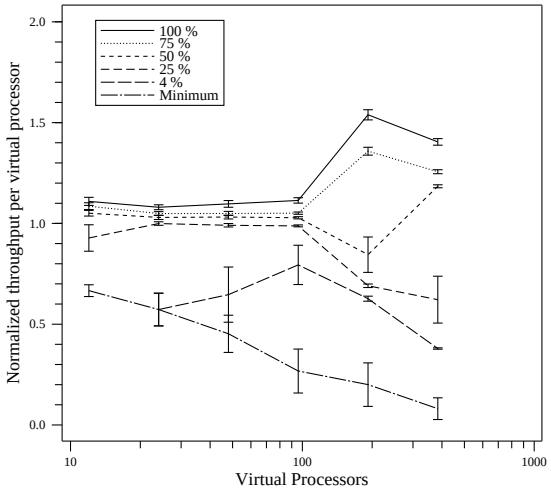

Figure 4.10. UDP Send-Side Throughput per Thread for 384 Connections, Threads Wired.

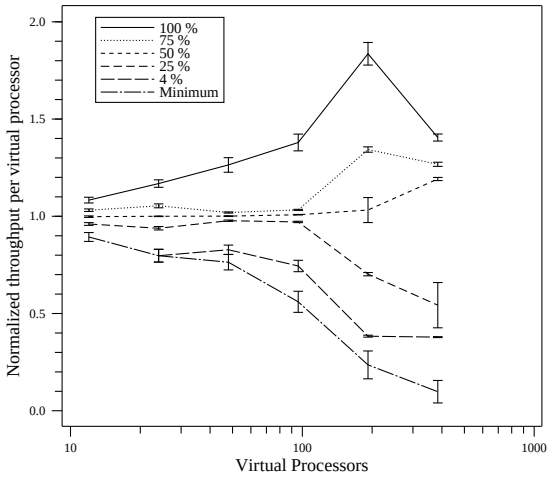

Figure 4.11. UDP Send-Side Throughput per Thread for 384 Connections, Threads Wired, Soft T-based Scheduling.

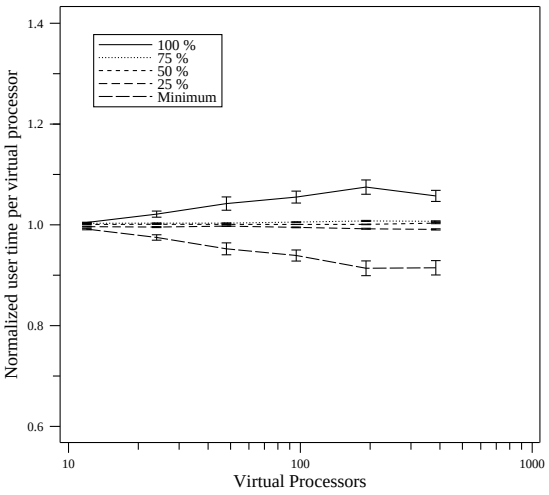

Figure 4.12. UDP Send-Side CPU Time for 384 Connections, Threads Wired.

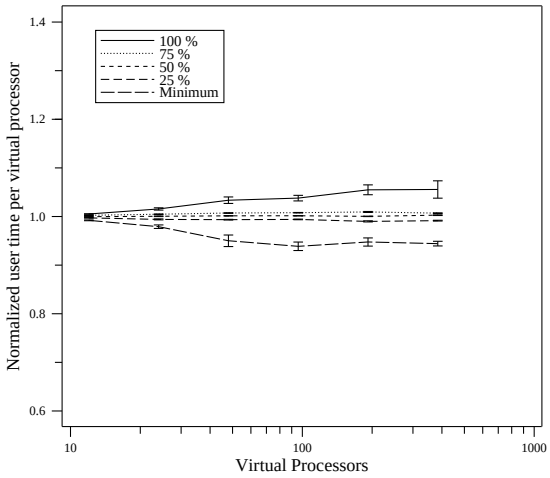

Figure 4.13. UDP Send-Side CPU Time for 384 Connections, Threads Wired, Soft T-based Scheduling.

These results also show that wiring threads to processors improves the effectiveness of soft throughput-based scheduling. Compare the per-connection throughput distributions shown in Figures 4.3 and 4.9. We see that in all experiments where connections are actually migrated (from 12 to 192 threads), the range of the per-connection throughputs is reduced, and the worst-case throughput is greater.

Figures 4.10 and 4.11 show the per-thread throughput for the same experiments as Figures 4.8 and 4.9. These figures give more evidence suggesting that most of the benefit of soft throughput-based scheduling is achieved by counteracting poor memory reference behavior. Consider the quartiles in Figures 4.10 and 4.11, focusing on the region where soft throughput-based scheduling is effective (from 12 to 96 threads). Without throughput-based scheduling, the threads that are in the top 75% in terms of their throughput are fairly close in value (the gap between the 100% and 25% curves in Figure 4.10 is less than 20%). In other words, most of the range of per-connection throughput distribution lies in the lowest quartile (where there is between a 25% and 60% gap). With soft throughput-based scheduling, the range of per-thread throughputs is narrowed, and the worst-case per-thread

throughput is significantly greater when there are between 12 to 96 threads. However, this comes at the expense of widening the range of per-thread throughputs in the top 75%.

Figure 4.12 and Figure 4.13 show the scheduling signatures for the experiments with and without soft throughput-based scheduling, but with threads wired to processors. These Figures show a significant reduction in the differences in CPU time that threads see. Figures 4.12 and 4.13 show at most a 20% difference in amount of CPU time given to different threads compared with the factor of 3 difference that we saw in Figures 4.6 and 4.7.

In Figures 4.12 and 4.13 we again see that throughput-based scheduling does not perturb the scheduling signature in any meaningful way. In both Figures the best case fairness behavior of the scheduler occurs when when there is only one thread per processor. Likewise, the worst case behavior of the scheduler occurs with a large number of threads (192 or 384).

When threads are wired to processors it is also the case that scheduling and memory reference behavior affect per-thread throughput differently. Consider Figures 4.10 and 4.12. Figure 4.10 shows normalized per-thread throughput whereas Figure 4.12 shows normalized per-thread CPU time. Thus, memory reference behavior is responsible for the difference in the spread between the curves going from Figure 4.12 to Figure 4.10. Note that this difference is especially large when there is a large number of threads. The same is also true going from Figure 4.13 to Figure 4.11.

We now consider UDP experiments where the number of connections is increased so that the per-connection application data buffers completely cover the second-level cache in our experiments. Recall that the purpose of this is to show the effect of a more heavily loaded server, where the pressure in the memory system is greater because of the larger number of connections.

Figure 4.14 shows the per-connection throughput for 3072 connections where the number of threads is varied from 12 to 384. In comparing Figure 4.14 with Figure 4.2, we see one important difference: The overall spread of the per-connection throughput is narrowed.

For example, using 384 threads, the ratio of the maximum to minimum per-connection throughput is 10:1 in Figure 4.2, but 5:1 in Figure 4.14. Although we cannot directly say why this is the case, it is easy to see how this could be explained by the increase in the number of connections serviced by each thread. In Figure 4.2, there is only one connection per thread for the 384 threads. In this case, any effects of memory reference behavior seen by a thread effect its connections throughput. In Figure 4.14 there are 8 connections per thread for the 384 threads, and these connections are serviced round-robin one packet at a time. This means that impact on throughput due to favorable (or unfavorable) effects of memory references are distributed across eight connections. This amortization of differences in per-connection throughput across multiple connections would tend to narrow the spread of the throughputs, as seen in going from 384 to 3072 connections.

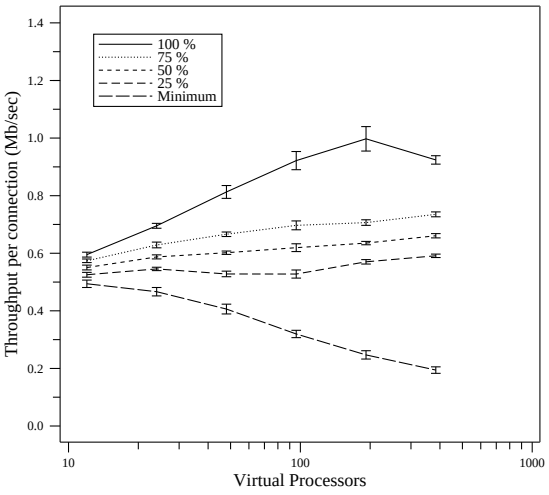

Figure 4.14. UDP Send-Side Throughput per Connection for 3072 Connections.

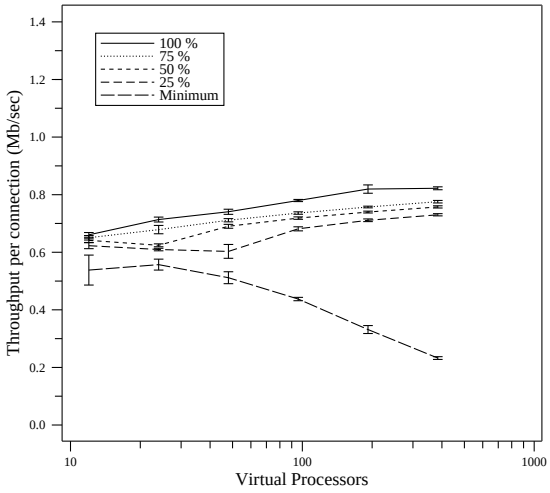

Figure 4.15. UDP Send-Side Throughput per Connection for 3072 Connections, Threads Wired.

Figure 4.15 shows the per-connection throughput for 3072 connections where the number of threads is varied from 12 to 384, and threads are wired to processors. In comparing these results with those in Figure 4.14, we see that wiring threads to processors further reduces spread of the distribution of per-connection throughputs. This is consistent with our earlier results for 384 connections (see Figures 4.2 and 4.8), with one important differ-

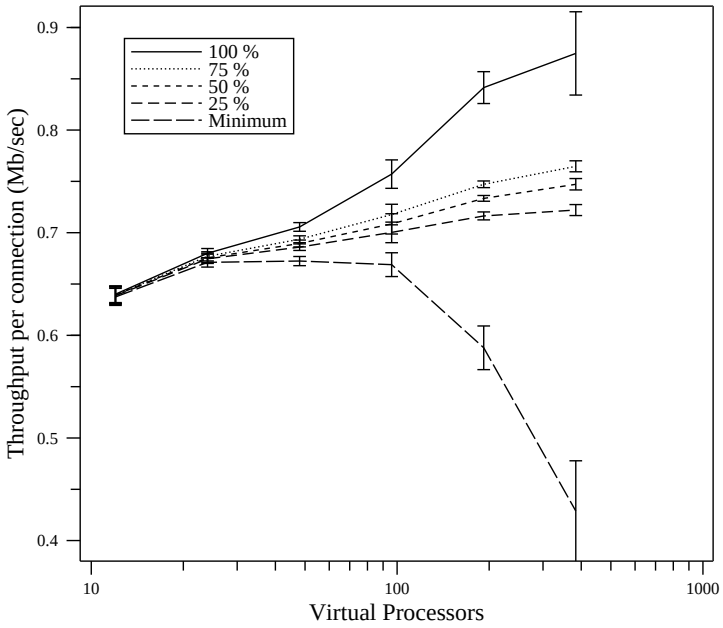

Figure 4.16. UDP Send-Side Throughput per Connection for 3072 Connections, Threads Wired, Soft T-based Scheduling.

ence. In Figure 4.8, wiring threads narrows the spread of throughputs in the range of 12 to 96 threads, but actually widens the spread for more threads. In Figure 4.15, the range of the distribution is narrowed across the board. Again, the most likely explanation for the differences between Figure 4.8 and Figure 4.15 is the amortization of favorable or unfavorable effects of memory references across more connections. Furthermore, the scheduling signatures for the configurations in Figures 4.14 and 4.15 are almost identical to those in Figures 4.7 and 4.13. Thus, the number of connections serviced by the threads has no significant impact on the CPU time given to each thread, as one would expect.

To determine the effectiveness of soft throughput-based scheduling with large numbers of connections, we ran experiments for 3072 connections using soft throughput-based scheduling with threads wired to physical processors. Figure 4.16 shows the results for these experiments. Recall that Figure 4.9 gave the corresponding results for 384 connections. In comparing these Figures it is clear that throughput-based scheduling is also effective at delivering soft throughput guarantees to a large number of connections. As for fewer

connections, soft throughput-based scheduling is robust in the region from where there is one thread per processor, up to 96 threads. Furthermore, we again observe a significant improvement in the per-connection throughput delivered to the connection that receives the minimum throughput, presumably due to eliminating the poor memory reference behavior seen by the connection that receives the smallest allocation of throughput.

4.3 Service with Hard Throughput Guarantees

Thus far, we have only looked at per-thread and per-connection throughputs over long time-scales (i.e., tens of seconds). However, for applications such as playing continuous media streams over the network, throughput over short time scales is also important. We wanted to demonstrate that hard throughput guarantees could be made to continuous media connections (using UDP). Furthermore, we wanted such guarantees to be robust over both short and long time scales, and insensitive to unfairness caused by memory reference behavior effects.

Our implementation of hard throughput-based scheduling uses the same mechanism to monitor per-thread throughput as was used in soft throughput-based scheduling. Recall that in this implementation thread throughputs are sampled every 100 milliseconds, and are used to update a per-thread exponential moving average. The throughput-based scheduler then uses the these averaged throughputs to make scheduling decisions in order to provide hard throughput guarantees. We use the same 100 ms timer used by the throughput monitor to sample per-thread throughputs.

4.3.1 Hard Throughput-Based Scheduling

To provide hard throughput guarantees, throughput-based scheduling adjusts the utilization of each thread to ensure that the threads are delivering their required throughput. Figure 4.17 shows a model of how this works using the throughput monitoring mechanism described above. In Figure 4.17, the measured rate r_i is compared with the throughput

corresponding to the sum of the throughputs specified by the connections assigned to virtual processor i. If the measured throughput r_i is less than the specified throughput, the idle period p_i is reduced. On the other hand, if the measured throughput is greater than the specified throughput, the idle period is increased. Unfortunately, most operating systems do not support primitives necessary to adjust p_i with resolution any finer than 10 milliseconds. We therefore perform a functionally equivalent computation in our implementation of hard throughput-based scheduling. Each thread computes how many packets it should send between $k \times 10$ millisecond intervals (where k is a non-negative integer), in order to deliver the correct throughput. The idle period p_i for virtual processor i is therefore changed by minimizing k and then computing the number of packets to send between $k \times 10$ millisecond pauses. We also successfully used the same gain for the exponential moving average ($\alpha = 0.875$) as was used in soft throughput-based scheduling.

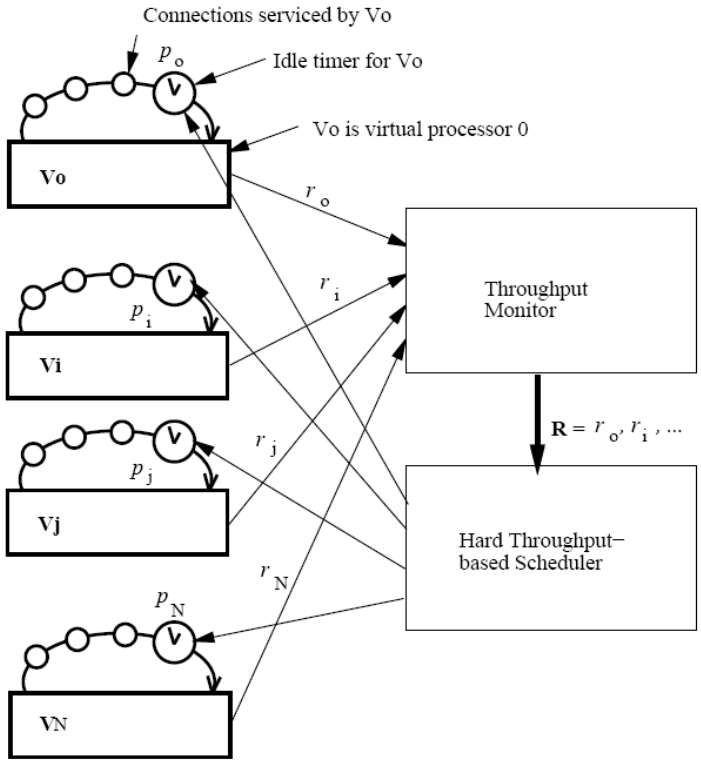

Figure 4.17. Model of Hard Throughput-Based Scheduling.

As we will see, hard throughput-based scheduling has the desired effect of delivering the correct throughput to all connections. However, unlike soft throughput-based scheduling, wiring threads to processors does not make a difference in the configuration we examined.

4.3.2 Results for Hard Throughput Guarantees

To determine the throughput performance of our protocols over short time-scales, we instrumented a UDP protocol stack to sample per-thread throughputs every 100 milliseconds. It was necessary to sample per-thread throughputs instead of per-connection throughputs since the volume of data logged to memory (and then disk) was manageable given the duration of our experiments, the size of memory in our multiprocessor, and the size of disks available for archiving data. This would not have been the case if we had implemented per-connection sampling every 100 msec.

Figure 4.18 shows per-thread throughputs for 384 UDP connections serviced by 12 threads. From the per-thread throughputs sampled every 100 msec, we compute the per-thread throughputs over longer time scales by using cumulative statistics over the appropriate number of 100 msec intervals. Thus, the x-axis varies from 100 msec to about 12 seconds. Note that the per-thread throughput distribution over long time scales closely matches the behavior of our earlier experiments. This can be seen by matching the distribution seen above 10 seconds in Figure 4.18 with the 12 thread results in Figure 4.4.

Figure 4.19 shows per-thread throughputs for 384 connections serviced by 24 threads. Comparing this Figure with Figure 4.18, it is clear that with more threads than processors, the distribution of per-thread throughput is wider for shorter time scales. For example, at 100 msec, the range of per-thread throughputs is 20% greater using 24 threads instead of 12. As one would expect, using 48 or more threads yields an even greater difference between the maximum and minimum per-thread throughput.

Figures 4.20 and 4.21 show 12 and 24 thread results, except that the threads are wired to processors. Wiring threads to processors seems to make only a slight difference, narrowing the range of per-thread throughputs over all time scales, except on the scale of seconds with 24 threads.

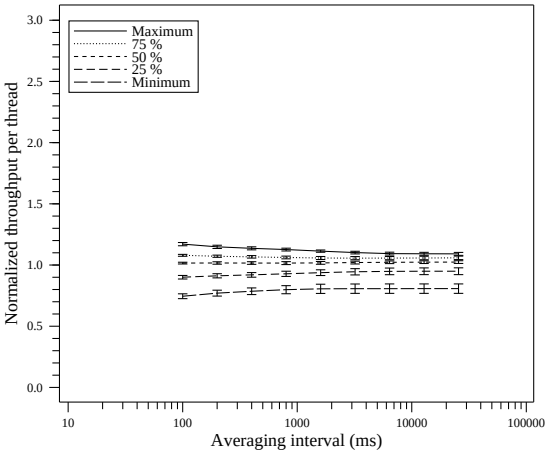

Figure 4.18. UDP Send-Side Throughput per Thread for 384 Connections, 12 Threads.

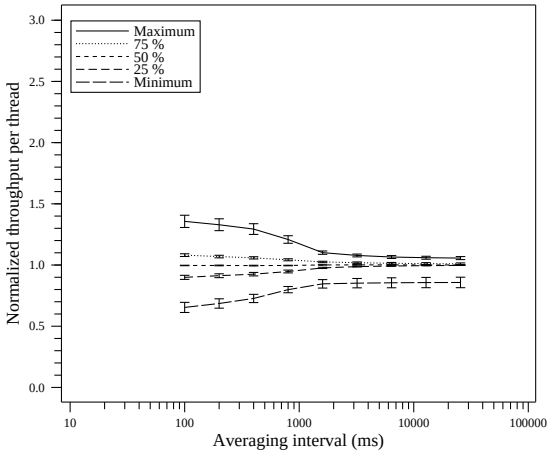

Figure 4.19. UDP Send-Side Throughput per Thread for 384 Connections, 24 Threads.

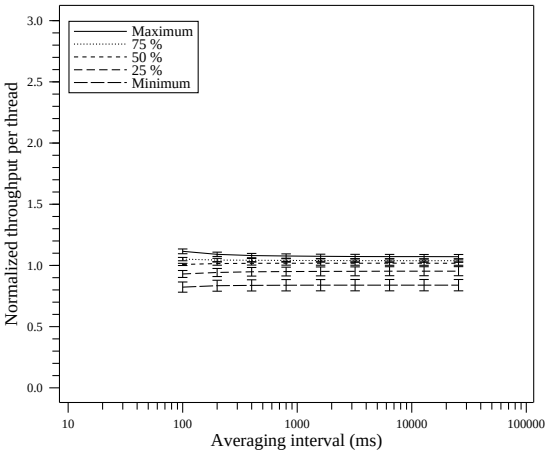

Figure 4.20. UDP Send-Side Throughput per Thread for 384 Connections, 12 Threads, Wired.

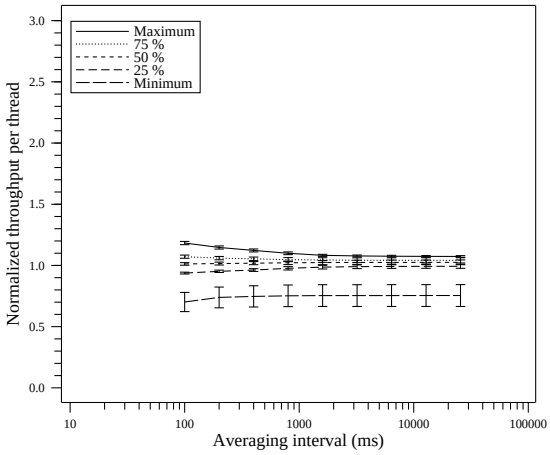

Figure 4.21. UDP Send-Side Throughput per Thread for 384 Connections, 24 Threads, Wired.

To determine how our algorithm for making hard throughput guarantees to connections performed, we ran experiments where the desired throughput per-connection was specified

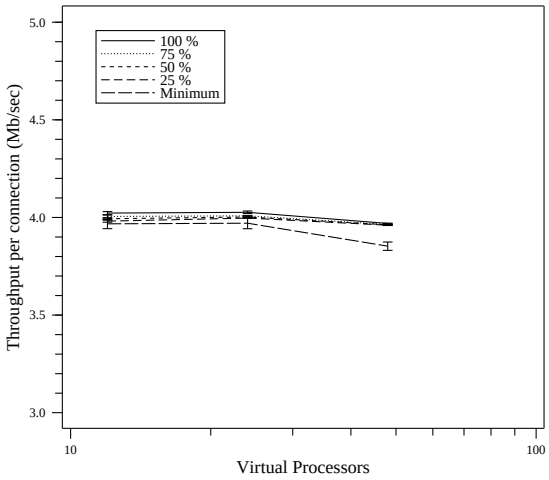

Figure 4.22. UDP Send-Side Throughput per Connection for 384 Connections, Hard T-based Scheduling.

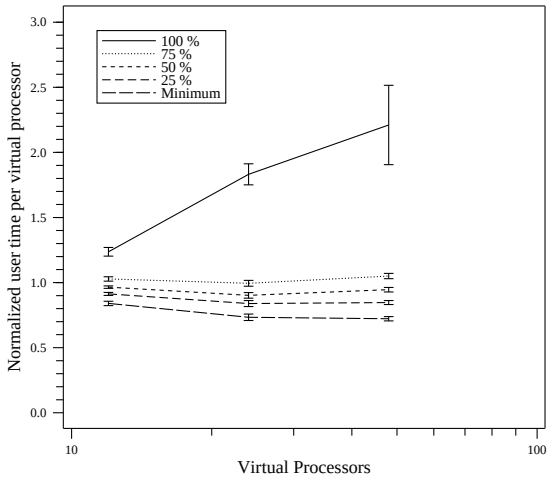

Figure 4.23. UDP Send-Side CPU Time Distribution for 384 Connections, Hard T-based Scheduling.

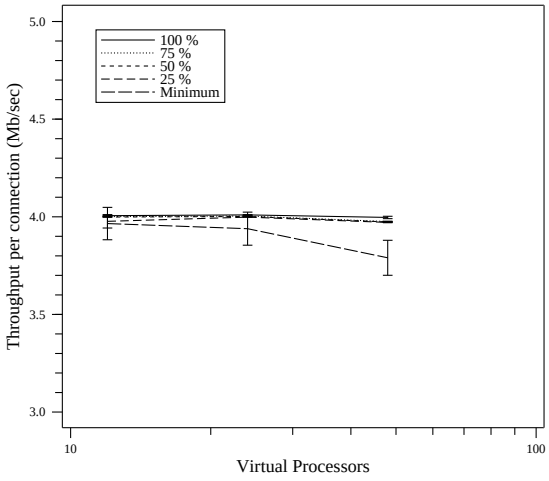

Figure 4.24. UDP Send-Side Throughput per Connection for 384 Connections, Threads Wired, Hard T-based Scheduling.

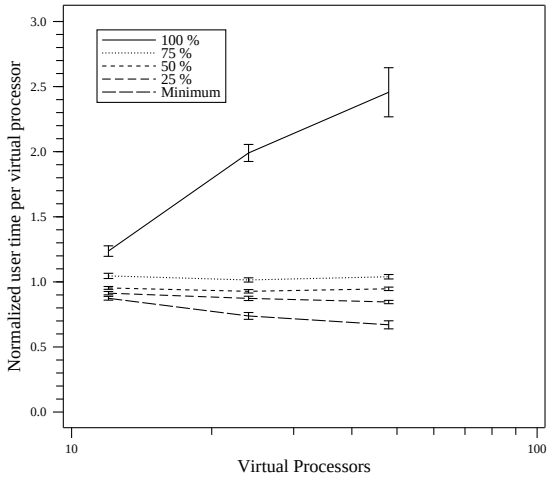

Figure 4.25. UDP Send-Side CPU Time Distribution for 384 Connections, Threads Wired, Hard T-based Scheduling.

at 4 Mbps. This represents a utilization of about 80%. Figure 4.22 shows results for these experiments using different numbers of threads to again service 384 UDP connections. Figure 4.22 shows that providing throughput guarantees to connections using the algorithm described above is effective with a small number of threads (either 12 or 24). However, with 48 threads the throughput guarantees begin to be difficult to sustain in the absence of tighter control of the scheduler. To understand the difference in memory reference behavior seen by the threads in these experiments, we plotted the scheduling signature. Figure 4.23 shows these results. Note that the hard throughput-based scheduler needs to allocate CPU time to threads that differs by more than 50% in order to achieve the goal of making throughput guarantees to connections. These results are consistent with those earlier in this Chapter, and in Chapter 3, where we saw that even though the IRIX scheduler was able to give roughly equal time to 12 threads running on 12 processors, the throughput delivered by threads differed by about 50%.

Figures 4.24 and 4.25 show corresponding results for hard throughput-based scheduling when threads are wired to processors. Again, we see that wiring threads to processors has no significant impact on the results.

4.4 Survey of Related Work

Many people have studied supporting multimedia applications on both uniprocessor and multiprocessor hosts. Most of this work has focused on the disk subsystem. Common to most of this work is the idea that the disk subsystem and communication subsystem will be decoupled by an intermediate buffer. For a given client connection, discontinuities in playback will be caused by overflow or underflow in this buffer. Avoiding this behavior requires matching the per-connection rates in the disk and communication subsystem, thus motivating the work described in this Chapter.

Proportional-share resource management (where resources are consumed by active tasks in proportion to their relative allocations) has been studied in both the networking and op-

erating systems communities. Proportional-share link scheduling has been proposed and studied by many researchers [36, 54, 106, 107, 105], and is expected to be deployed in commercial switches in the near future. Proportional-share processor scheduling has been implemented by Waldspurger and Weihl [133, 134] and appeared the SGI operating system [10] subsequent to our work. In their first paper [133], Waldspurger and Weihl present an example where processes playing MPEG video on a client receive proportional-share access to the processor. However, they observe differences of a few percent in their actual frame play-out rates.

Throughput-based scheduling requires the scheduler to obtain information about a thread's forward progress, which is how much data it has transmitted per unit time. Our implementation depends on using a call-back mechanism to accomplish this (calls are made by the scheduler to code that reports the throughput for a specific thread). This requires some form of cross-domain IPC in a traditional UNIX operating system. However, the cost of cross-domain IPC is being reduced to a procedure call in research operating systems, which are making the operating system itself extensible, and therefore customizable. We therefore consider it reasonable to use a call-back in our implementation without loss of generality.

An important component of throughput-based scheduling, is the idea of adjusting resource allocation to even out differences in memory reference behavior. This is a new idea, however, the notion that memory reference behavior can effect code execution performance is well known. In fact, there are tools for characterizing and changing program memory reference behavior to improve performance. Such tools were pioneered by research in the early 1990s (e.g., [52, 87]), and were developed for several processors of that era [33, 125, 132]. In the context of memory reference behavior of network protocols, Mosberger et al. [93] and Nahum et al. [94] have proposed and evaluated techniques for reducing packet latency. Also, Salehi et al. [114, 115] have shown the benefit scheduling network protocol processing for cache affinity,

Most of the recent work on server support for continuous media has also evolved from earlier work on cluster architectures [6, 44, 122, 123, 131]. As in our work, these approaches all explicitly manage resources to support application requirements. The policies and mechanisms in some of this work are also based on one of the underlying principles of throughput-based scheduling: on-line monitoring of server performance in order to ensure performance guarantees [6, 44, 131].

4.5 Conclusions

In this Chapter we assessed the suitability of connection-level parallelism (CLP) for supporting continuous media applications. In such applications, the server protocol stack is one link in a chain of subsystems which must provide throughput guarantees to connections. We consider two types of throughput guarantees. The first is for applications where the playout rate is required to exactly match an original fixed recording rate (i.e., CBR applications). In this case, the protocol stack and operating system (as well as other system components) should provide service where throughput delivered to connections is constant and guaranteed. We refer to this service class as service with hard throughput guarantees. The second is for applications that can adjust their playout rate as they detect that the available bandwidth being delivered to connections changes. In this case the protocol stack and operating system need to distribute the available bandwidth equitably among all connections. We refer to this service class as service with soft throughput guarantees.

We presented results for a new technique, called *throughput-based scheduling*, that provides a mechanism whereby threads monitor and report their per-connection throughput to the scheduler. To support soft throughput guarantees, the scheduler balances load by reallocating work (in this case connections) among threads. We refer to this technique as *soft throughput-based scheduling*. To support hard throughput guarantees, the scheduler schedules threads according to their "utilization" (i.e., how much of their current capacity is

consumed by the throughput they have been assigned to deliver). We refer to this technique as *hard throughput-based scheduling*.

Using UDP as the transport protocol, we showed that throughput-based scheduling is suitable for supporting continuous media applications. Soft throughput-based scheduling performed best with fewer threads in the system and without competing scheduling mechanisms (i.e., IRIX's cache affinity-based scheduling). Hard throughput-based scheduling also performed best with fewer threads performing protocol processing. Furthermore, when the number of threads used is the same as the number of processors, the throughputs delivered are accurate over both short (i.e., down to 100 milliseconds) and long time scales.

CHAPTER 5

COST AND QUALITY PERFORMANCE IN SENSOR NETWORKS

5.1 Introduction

Wireless sensor networks will enable many new data-intensive sensing applications, ranging from environmental and infrastructure monitoring to commercial and industrial sensing. Examples of sensor network-based applications include urban structure monitoring, contaminant transport monitoring, habitat monitoring, and military surveillance.

There are many performance metrics of interest for such sensor networks. We focus on two that are common to the vast majority of sensor networks:

1. The *accuracy* of the data acquired by the application from the sensor network; and
2. the total *system end-to-end delay* incurred in the sequence of operations needed for an application to obtain sensor data.

Although almost all sensor network applications have performance requirements that include accuracy and delay, their relative importance differs between applications. For example, a rainfall data acquisition and analysis application might care about accuracy but not delay. In contrast, an air traffic monitoring system might care about delay more than accuracy. On the other hand, a sensor network designed to aid in military target tracking would care about both accuracy and delay. We therefore define the *quality* of the data provided to sensor network applications as a combination of accuracy and delay. As in most systems, improved quality usually comes at some *cost*. For current wireless sensor networks, the most important component of cost is arguably the energy consumed in processing a task. This cost, in turn, is dominated by the energy required to transport messages through the

sensor field. This cost versus quality trade-off has recently been an active area of research [15, 121, 124, 129, 138].

This chapter explores sensor network cost and quality performance when a cache is placed in a sensor field gateway server. We begin our exploration by describing a sensor network model, and developing policies for caching sensor network data values and then retrieving these values via cache lookups. We then propose a new objective function for data quality that combines accuracy and delay. We use our sensor network model to evaluate and compare the cost and quality performance of our caching and lookup policies. Finally, we present results for two specific sensor networks in which user queries are generated for data values at a set of discrete locations within each sensor sensor field.

5.1.1 Sensor Network Model

Figure 5.1 shows a model of a sensor network configuration in which a cache is part of the internal architecture of the data server. Because of the cache in Figure 5.1, there are two possible data paths that can be traversed in response to a query from the backbone network:

- For a cache miss, the data path includes a sensor field query that is sent into the sensor field as the result of a user query. In addition, the value that appears in each reply from the sensor field is used to update the cache. Thus, a cache entry, el_i, associates a location l_i, the most recent value observed at this location, v_i, and its timestamp into the tuple $\langle l_i, v_i, t_i \rangle$.

- The data path for a cache hit is much shorter than that of a cache miss. For example, if the cache is indexed by location, and a cache entry is present for a location l_i specified in a query, a reply can be generated using only the information in the tuple that corresponds to l_i. Since the processing required to perform this cache lookup and generate a reply is relatively small, we assume that the system delay for a cache hit (S_d between $Query_m$ and $Reply_m$) and its associated cost are both zero.

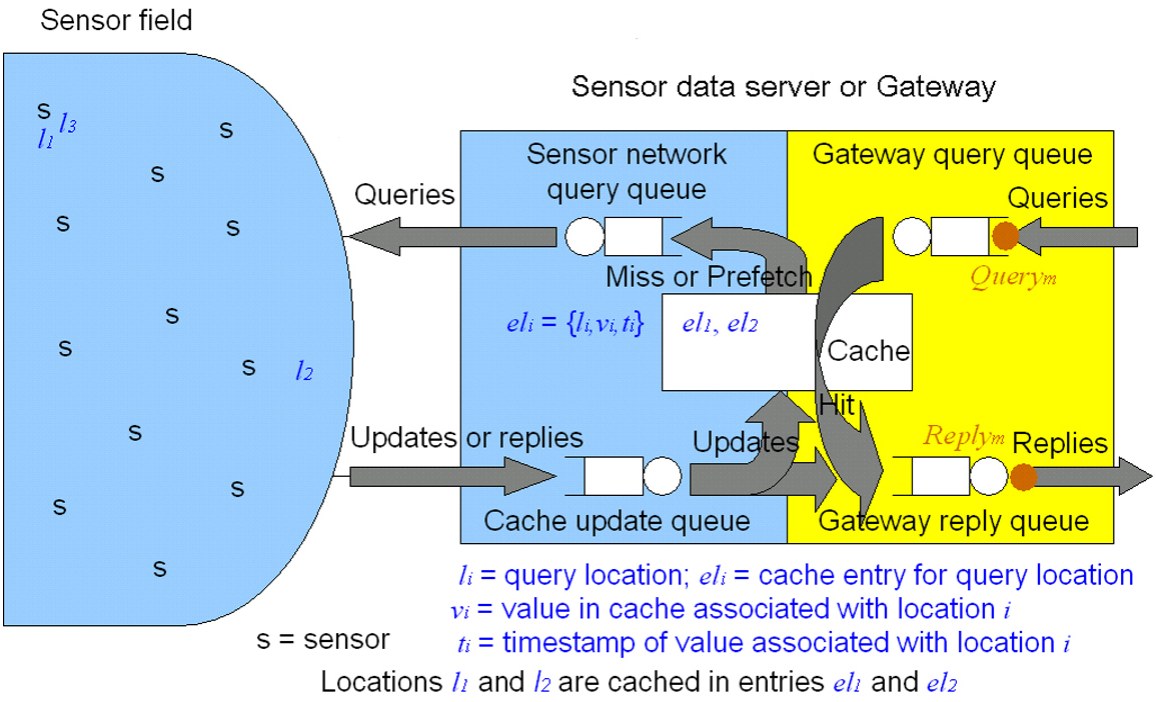

Figure 5.1. Sensor Network Model with a Data Server.

As with system end-to-end delay, we determine the accuracy of a reply by quantifying the *value deviation* for cache misses and hits in a consistent manner. Value deviation is the unsigned difference between the value v_i in $Reply_m$ and the true value at l_i when $Reply_m$ leaves the gateway reply queue.

5.1.2 Caching and Lookup Policies

Our caching and lookup policies are designed to explore alternative techniques for increasing the effective cache hit ratio, and thus conserving system resources. We will see that for applications that value delay more than accuracy, some of our caching and lookup policies provide a simultaneous quality improvement and cost savings by approximating sensor field values using cached values. We will also show that some of our policies provide an upper bound on the sensor field query rate. The key feature of these policies it that they piggyback multiple user queries on individual sensor field queries that are pending.

All of the caching and lookup policies we propose and evaluate incorporate an age threshold parameter T that specifies how long each entry is stored in the cache. In practice, picking a good value for T depends on how rapidly the environment being monitored might change, and the utility of a cached entry as it approaches being T seconds old. If T is too small, it is easy to imagine that the cache might not be effective. If T is too large, then the cache may be helpful in terms of saving cost, but may be harmful if value deviation increases as a function of elapsed time. Clearly, larger values of T reduce the number of messages that are sent through the sensor field. We will confirm this intuition, and quantify the maximum query rate for our sensor field models as a function of T. Since power consumption in a wireless sensor field is a function of query rate, bounding our sensor field query rate can be useful in bounding the resource consumption of our caching and lookup policies.

In addition to increasing T, we can improve the effective cache hit ratio by exploiting the spatial locality within sensor field data. We will examine cache "hits" in which the value at location l_i is approximated based on values $v_{i'}$ from neighboring location(s) $\{i' \in N(i)\}$. (Here $N(i)$ denotes the neighborhood of location l_i.) Three of our policies below implement such *approximate* lookups and queries. We will compare these approximate policies with four *precise* lookup and query policies that use only information associated with location l_i to process queries that reference location l_i. We now describe all seven of these caching and lookup policies. *All hits*, *all misses*, *simple lookups* and *piggybacked queries* implement precise lookups and queries. On the other hand, *greedy age lookups*, *greedy distance lookups*, and *median-of-3 lookups* implement approximate lookups and queries.

- **All hits** (age threshold parameter $T = \infty$): In this policy cache entries are loaded into the cache but are never deleted, updated, or replaced.

- **All misses** (age parameter $T = 0$): In this policy entries are not stored in the cache.

- **Simple lookups** (T): This caching policy results in a cache hit or a cache miss based on a lookup at the location specified in the user query. Once an entry is loaded into the cache, it is stored for T seconds and then deleted.

- **Piggybacked queries** (T): A cache hit or miss is determined only by a lookup at the location specified in the user query. If a query has already been issued to fill the cache at a particular location, subsequent queries block in a queue behind the original query and leverage the pending reply to fulfill multiple queries.

- **Greedy age lookups** (T): A cache hit or miss is determined by a lookup first at the location specified in the query, and second by lookups at all neighboring locations. If there is more than one neighboring cache entry, the freshest (newest) cache entry is selected. As for piggybacked queries, if a query has already been issued to fill the cache at any of these locations, subsequent queries block in a queue behind the original query and leverage the pending reply to fulfill multiple queries. This is also true for the last two policies: greedy distance lookups and median-of-3 lookups.

- **Greedy distance lookups** (T): A cache hit or miss is determined by a lookup first at the location specified in the query, and second by lookups at neighboring locations. If there is more than one neighboring cache entry, the nearest cache entry is selected.

- **Median-of-3 lookups** (T). A cache hit or miss is determined by a lookup first at the location specified in the query, and second by lookups at all neighboring locations. If there are at least three neighboring cache entries, the median of three randomly selected entries is selected as the value returned with a cache hit. If there are one or two neighboring cache entries, a randomly selected entry provides a cache hit. Otherwise, the query is treated as a miss.

By implementing blocking behind pending sensor field queries, four of these seven policies have an upper bound on the sensor field query rate, R_f. Specifically,

$$\max(R_f) \quad = \quad \frac{|\mathbf{N}|}{T}. \tag{5.1}$$

The four policies are piggybacked queries, median-of-3 lookups, and the two approximate greedy policies. In Equation (5.1), $|\mathbf{N}|$ is the number of distinct locations that can be specified in queries for sensor data. For the 1000-node sensor field model described in Section 5.2.1, $|\mathbf{N}| = 1000$, so $\max(R_f) = 112.5$ queries per second when $T = 8.8\overline{8}$ seconds. Without a cache shielding the sensor field from queries (e.g., as in Figure 1.6), the sensor field query rate for our workload has no upper bound.

The definition of a cache hit (vs. a cache miss) for our three approximate lookup and query policies (median-of-3 lookups and the two greedy policies) has both similarities and differences with the traditional definition of a cache hit (or miss). One similarity is that the data value in the cache may be consistent with the value in the sensor field. This would always be true for a memory cache for a uniprocessor computer [23], or for a sequentially consistent multiprocessor [46]. A related similarity is that the data value in the cache may be different from the value in the sensor field. This would also be the case, some of the time, in a distributed file system that caches files or blocks [59], or a web-based application that exploits web caches [81]. These similarities are also shared with the precise lookup and query policies. An important difference from these traditional caching systems is that the cache lookup policy itself might introduce some inconsistency (e.g., value deviation). This is because our approximate policies may emulate a cache hit by approximating a data value in the sensor field (using neighboring values) rather querying the sensor field for the missing value.

The caching approaches we propose are designed to be general since they make no assumptions about whether the overall sensor network architecture uses a structured or unstructured data model. In other words, our approaches are independent of the implementation of the Database layer in Figure 1.5. Our sensor network database could implement a structured schema that extends a standard like the Structured Query Language (SQL)

[35, 82]. The schema could also be modified while the system is running (e.g., as in IrisNet [51]). The Database layer might also expose a low-level interface to the application, as in Directed Diffusion [64].

5.1.3 Quality of Sensor Network Data

As data is acquired by wireless sensor network applications, it has an associated *quality* that is a function of at least two parameters: (1) the delay experienced in receiving the data, and; (2) the deviation between the value of each data item and the corresponding data in the sensor field. Acquiring sensor data typically also has a *cost* that reflects the system resources consumed in querying and delivering the data to the applications. Cost is determined by the transmission technology used in the sensor field, and the physical location of the sensors and base stations. We define cost for our sensor network model in Section 5.2.1.

We normalize the quality of sensor network data in order to compare quality measurements from different sensor networks, as well as for different system parameters (e.g., number of sensors, distance between sensors, etc.) We define quality as a linear combination of normalized *system delay* and normalized *value deviation* using a parameter A, which is the relative importance of delay when compared with value deviation. The expression that defines quality, denoted Q_n, is:

$$Q_n = A \frac{1}{\left(1+e^{-b}\right)} + (1-A) \frac{1}{\left(1+e^{-c}\right)} \tag{5.2}$$

where $-b$ and $-c$ are the exponents used to perform *softmax normalization* on delays and value deviations, and $0 \leq A \leq 1$. The exponents in Equation (5.2) are the z scores of their respective values and are therefore defined as follows:

$$-b = -\frac{S_d - \text{mean}(S_d)}{\text{stddev}(S_d)}, \text{ and} \tag{5.3}$$

$$-c \quad = \quad -\frac{D_v - \text{mean}(D_v)}{\text{stddev}(D_v)}. \tag{5.4}$$

where S_d is the system delay for a query to be processed by the data server and potentially to be forwarded to the sensor field, and D_v is the deviation of the value used in a query reply. Since small values of system delay and value deviation are both desirable, smaller values of Q_n, e.g., $0 < Q_n \ll 0.5$ imply better data quality, and larger values of Q_n correspond to worse quality.

We chose to use softmax normalization for the S_d and D_v values (which together define quality) for several reasons:

- Softmax normalization works well on populations with a large dynamic range (e.g., some of our D_v values); and
- it is easy to use in practice since it requires that we know only the mean and standard deviation of our S_d and D_v populations.

Softmax normalization also yields transformed values that lie in the range $[0,1]$. Because of this property, and because of our definition of A, $0 \leq Q_n \leq 1$ and $Q_n = 0.5$ when the system delay and value deviation are simultaneously at their respective means. This type of normalization has been used by others in neural networks and data mining for pattern recognition and data classification [13, 21, 56]. It has two other interesting properties that make it convenient for normalizing our system delays and value deviations:

- For finite values, softmax normalization reaches "softly" toward its maximum and minimum values of 0 and 1, never quite getting there [111]; and
- its transformed values are more or less linear in the middle range, and have a non-linearity at both ends that make it well suited for data values with distributions that have long tails [111].

We considered four normalization methods for our system delays and value deviations:

- *min-max* normalization;
- *z-score* normalization;
- *sigmoidal* normalization; and
- *softmax* normalization.

We also thought it was important to have an expression for quality that is bounded for both our sensor network data and for data values from other sensor networks. Sigmoidal normalization and softmax normalization both have this property whereas min-max normalization and *z*-score normalization do not. This requirement makes median-based or percentile-based normalization methods undesirable as well. We ultimately chose softmax normalization over sigmoidal normalization because we found it convenient to conceptualize and graph non-negative values of quality. To provide some intuition for how softmax normalization varies between 0 and 1, Table 5.1 shows softmax-normalized values for ± 6 standard deviations from the mean.

Table 5.1. *Z* Scores and Softmax-normalized Values.

z score of data value	softmax-normalized value
-6	0.0025
-5	0.0067
-4	0.0180
-3	0.0474
-2	0.1192
-1	0.2689
0	0.5
1	0.7311
2	0.8808
3	0.9526
4	0.9820
5	0.9933
6	0.9975

5.2 Model Details, System Variables and Parameters

5.2.1 Sensor Field Models

We use two different sensor field models in our research in order to generalize our results. The first model uses random variables to simulate how the environment changes for 1000 sensor locations. This model gives us the flexibility to vary how the environment changes. The second model uses real-world trace data to drive how the environment changes. This trace data was taken from 54 light, temperature, and humidity sensors deployed in the Intel Berkeley Research lab over a five-week period [37]. We describe the two models in more detail in the remainder of this section.

For the simulated changes to environment, the sensor field is a 3-dimensional field with rectangular planes on six faces. There is an 8-unit spacing between 10 sensors in the X-dimension, a 6-unit spacing for 10 sensors in the Y-dimension, and a 4-unit spacing for 10 sensors in the Z-dimension.

In our simulations of this 3-D sensor field, the average number of sensor values that change in a unit of time (one second) is 9, 90 or 900. Change events in our first sensor field model occur at discrete locations, in sequence, at a fixed period of between $1/9$ and $1/900$ seconds. These rates of change also imply that between 0.9% and 90% of our sensor field values change each second.

Four base stations are placed on the X-Y plane. These four base stations are then connected to the gateway server that has the common cache. The sensors always communicate with their closest base station, and the properties of each one-way communication to and from location l are as follows:

$$Cost_l \quad = \quad p\, r_{b'}^2 \quad | \quad \min(Cost_l) = 1 \text{ unit} \tag{5.5}$$

where $r_{b'}$ is the distance between location l and its nearest base station b', and p is the normalization constant for the set of costs. In addition,

$$Delay_l = q r_{b'} \quad | \quad \max(Delay_l) = 1 \text{ second} \tag{5.6}$$

where q is the normalization constant for the set of delays. We assume that all four base stations communicate with the gateway server containing the cache at zero cost, with zero delay, and using infinite bandwidth. Thus, the minimum cost to query a location in the sensor field has been normalized to 2 units (1 for the query + 1 for the reply), and the maximum delay to query a location in the sensor field is 2 seconds (not including queuing delay). The maximum queuing delay at the sensor network query queue is set to 2 seconds, making the maximum total system delay, $\max(S_d)$, just over 4 seconds.

Each base station is connected to the sensor field with an access link with a capacity of 25 queries per second.

For the trace-driven changes to the environment, our second sensor field model has more than an order of magnitude fewer locations (54 instead of 1000). The sensors are arranged in a 2-dimensional field at the numbered locations in Figure 5.2, which is taken from [37]. Each entry in the trace is from a Mica2Dot sensor, which senses humidity, temperature, light, and battery voltage. The trace contains over 2.2 million entries taken over more than five weeks in early 2004. This means that one location reads and records new sensor field values an average of about once every 1.33 seconds.

We wanted to use the most dynamically changing of the sensor field values in our model to maximize the error in query accuracy. We therefore chose the value with the largest standard deviation as a function of time. This was light intensity, which is reported in Lux. A value of 1 Lux corresponds to moonlight, 400 Lux to a bright office, and 100,000 Lux to full sunlight.

Four base stations that share a gateway server and cache are placed in the corners of the sensor field in our simulations. Again, we assume that sensors always communicate with their closest base station, and the cost and delay of each one-way communication follow the models captured in Equations (5.5) and (5.6), respectively. The minimum cost to query

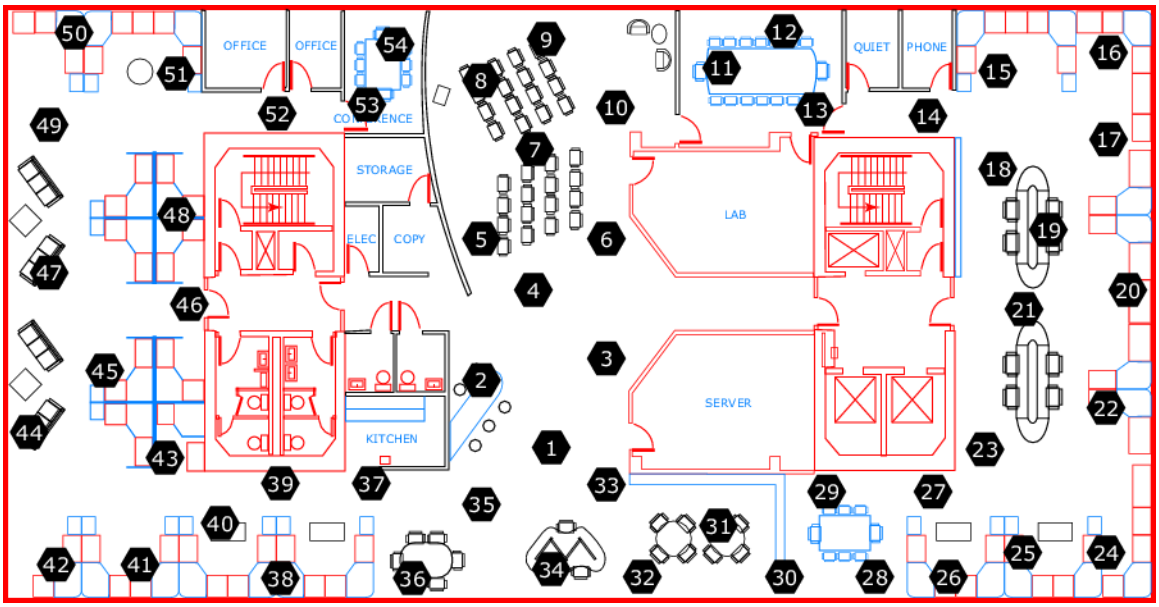

Figure 5.2. Sensor Field at Intel Berkeley Research Lab.

a location in the sensor field is again 2 units, and the maximum delay to query a location in the sensor field is again taken to be 2 seconds.

The trace data specifies how the environment changes in the Intel Berkeley lab. Therefore to vary the rate at which the environment changes relative to the query rate, we are forced to vary the query rate. As we vary the query rate, we also scale the total access link capacity and maximum queuing delay according to Table 5.2.

Table 5.2. Query Rates and Scaled Parameters for Trace-driven Sensor Field Model.

Average Query Rate (queries/second)	Access Link Capacity (queries/second)	Max Queuing Delay (seconds)	Max System Delay (seconds)
0.9	1	7.5	12
9	10	3.75	6
90	100	2	4.025

5.2.2 Query Workload Model

We use a query workload model that is well suited for sensor network applications that include monitoring and control functions. Many of these applications have a workload that includes a periodic arrival process of queries as well as a random arrival process. There are examples of query workloads that capture both of these components in the literature [64, 68, 135]. On the other hand, other researchers assume that queries either have exclusively periodic interarrival times [80, 82] or random (usually exponential) interarrival times [35, 141]. We assume that the query workload for our applications consists of the superposition of two query processes: a polling component that slowly scans the sensor field at a fixed rate, and a random component that consists of queries to different locations in the sensor field. Within this random component it is equally likely that each location in the sensor field will be sampled. This workload model is similar to models used by others in [64, 68, 135]. Specifically, we assume that our workload is characterized by two parameters:

- τ = the period of the polling component of the query workload ($\tau > 0$); and
- λ = the average query arrival rate of a process that represents the random component of our workload.

For simulated changes to the environment, λ and τ are initially fixed: $\lambda = 81$ queries per second is used as the rate parameter to generate queries with exponentially distributed interarrival times with mean $1/\lambda$. The parameter τ is initially set to $111.1\overline{1}$ seconds so that the arrival rate for polling queries is 9 queries per second. When $\lambda = 81$ and $\tau = 111.1\overline{1}$, the aggregate arrival rate for queries is $81 + 9 = 90$ queries per second. Since the total capacity of the sensor field access links is $4 \times 25 = 100$ queries per second, their average link utilization is 0.90 for "all miss" runs, and less for runs that include some cache hits.

For trace-driven changes to the environment, λ and τ vary with the average query rate according to Table 5.3.

Table 5.3. Query Rates, and λ and τ Values for Trace-driven Sensor Field Model.

Average Query Rate (queries/second)	λ (queries/second)	τ (seconds)
0.9	0.81	600
9	8.1	60
90	81	6

5.2.3 Delay and Value Deviation

We wanted to use our sensor network model to understand the impact of different application requirements for delay and value deviation on our results. In order to do this we ran four different sets of baseline experiments. Each set of experiments reflects different choices from two different dimensions as follows:

1. Different values of A chosen from $\{0.1, 0.9\}$, where *A* is the relative importance of system delay when compared with value deviation. Recall that *A* appears in Equation (5.2), which defines quality, Q_n.

2. Two different models for how the environment changes, namely using random variables and using real-world trace data. Section 5.2.1 describes both models for how the environment changes in detail.

5.3 Discussion of Experiments

Our experiments assess the impact of several factors on the cost and quality performance in wireless sensor networks:

- Caching and lookup policies; the
- relative importance of accuracy and system delay; and the
- the manner in which the environment changes.

This assessment evaluates all seven caching and lookup policies by implementing them in a simulator based on CSIM 19 [120, 119]. We focus on four issues in our assessment:

- *When is a cost vs. quality trade-off present?* Recall that quality incorporates both accuracy and delay. The benefit and cost of different caching and lookup policies therefore varies with how quality is specified for a given sensor network application. We present results for sensor networks in which user queries are generated for a set of discrete locations within each sensor field. We show that for some quality requirements, policies that avoid cache misses by computing and returning approximate values for sensor data can yield a simultaneous quality improvement and cost savings.

- *When present, what is the form and magnitude of the cost vs. quality trade-off?* Within the design space of our seven caching and lookup policies, five of the seven policies age and then delete cache entries uniformly based on the age threshold parameter, T. We observe that in many system configurations these five policies expose a linear cost vs. quality trade-off.

- *In what way does the manner in which the environment changes impact performance?* Intuition suggests that augmenting the data acquisition process for a sensor network application with a cache will work well if the phenomenon being monitored is slowly changing. On the other hand, if the phenomenon being monitored changes quickly, caching might not work well. We assess to what extent this intuition is correct by varying the frequency at which the values in the environment being monitored change relative to the query rate.

- *How does changing the age threshold parameter for cache entries affect performance?* For the caching policies that we propose and evaluate, the cache hit ratio for a given workload can be increased by increasing T. The converse is also true. The cache hit ratio can be decreased by decreasing T. We determine how cost and quality performance are impacted as T is changed by two orders of magnitude.

Each data value presented in our results is derived by averaging 20 simulation runs initialized with different seeds. Each run consists of up to 5,000,000 queries and replies. In order to make sure that the system is in steady state while sampling query and reply data, we allow up to 1,000,000 queries to pass through the system and discard the data from these queries. In the following sections, we will omit showing confidence intervals for our results. We computed confidence intervals for all cost and quality values presented. We found that the 90% confidence intervals for quality were within 1.8% of the point value, and for cost were within 3.4% of the point value, as discussed shortly.

Section 5.4 discusses the results for the experiments described above. We then examine performance trends for different application quality requirements in Sections 5.5 and 5.6. Finally, we vary the age threshold parameter of our caching and lookup policies in Section 5.7.

5.4 Discussion of Results

5.4.1 Independent and Trace-driven Changes to the Environment

The results in this section compare how the cost and quality performance differ between two models for how the environment changes. In the first model, each location in the environment changes independently and identically according to a normal distribution with truncated tails. This distribution has mean $\mu = 0$ and has a standard deviation $\sigma = 0.407514$, and tails that are truncated at minimum / maximum values of $\sigma - 6\mu$ / $\sigma + 6\mu$. This standard deviation is the same as the standard deviation of the system end-to-end delays during a set of 20 runs without a cache for our 1000-node sensor network model. For this network, changes in sensor field values occur at a fixed periodic rate of 90 locations per second. In the second model, changes in the environment are specified by the timestamped light intensity readings from the 54-node sensor network deployed in the Intel Berkeley lab [37].

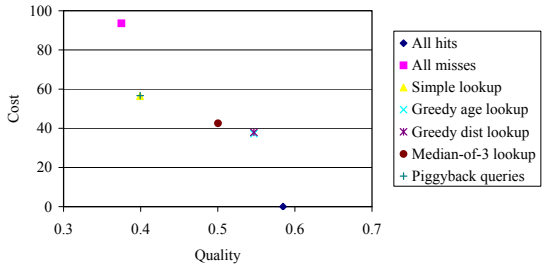

Figure 5.3. Cost vs. Quality for $A = 0.1$ and Independent changes over 1000 locations.

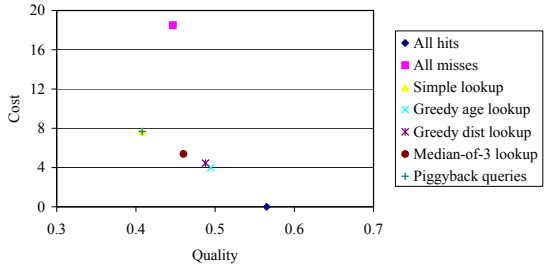

Figure 5.4. Cost vs. Quality for $A = 0.1$ and Trace-driven changes over 54 locations.

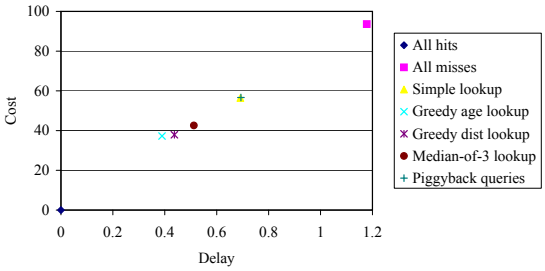

Figure 5.5. Cost vs. Delay for $A = 0.1$ and Independent changes over 1000 locations.

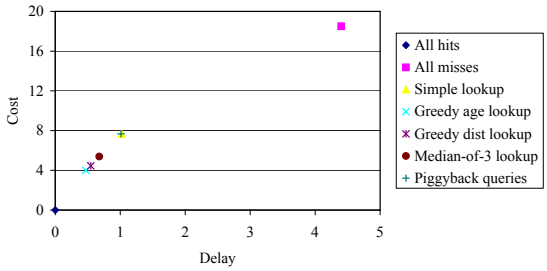

Figure 5.6. Cost vs. Delay for $A = 0.1$ and Trace-driven changes over 54 locations.

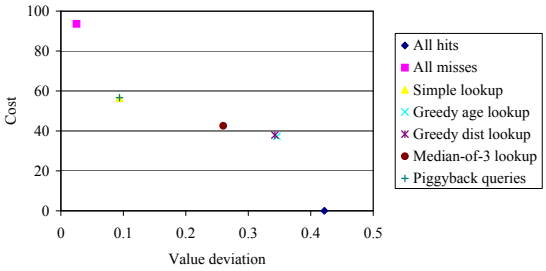

Figure 5.7. Cost vs. Value deviation for $A = 0.1$ and Independent changes over 1000 locations.

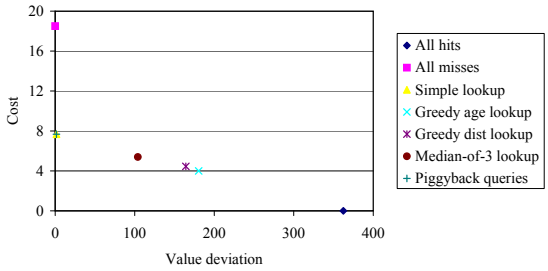

Figure 5.8. Cost vs. Value deviation for $A = 0.1$ and Trace-driven changes over 54 locations.

We first explore cost and quality performance when value deviation is significantly more important to quality than delay (as evidenced by a value of $A = 0.1$), in Figures 5.3 through 5.8. Figure 5.3 shows cost vs. quality for independent changes to the environment drawn from the normal distribution described above. Comparing this figure with Figure 5.4, which shows cost vs. quality for trace-driven changes to the environment, we find results are surprisingly similar in both cases. We see that there is a cost vs. quality trade-off when A is sufficiently small (again, $A = 0.1$ in these figures). Also, two of our precise lookup and query policies (simple lookups and piggybacked queries) provide the best (or close to best) quality performance when $A = 0.1$. However, if cost is more important than quality, the two greedy policies provide the flexibility to trade decreased quality for a cost savings of up to 50% (see Figure 5.4). It is easy to understand the characteristic shape and form of Figures 5.3 and 5.4 by understanding how A combines the underlying delay and value deviation distributions to yield quality. Consider the results when changes to the environment are driven by independent random variables: Figure 5.3 is a normalized combination of Figure 5.5 (with weight 0.1), and Figure 5.7 (with weight 0.9). Since the dispersion of delay values in Figure 5.5 is fairly small, Figure 5.3 has the similar trends and characteristics to Figure 5.7. Similar results hold for trace-driven changes to the environment: Figure 5.6 and Figure 5.8 combine to yield the main results in Figure 5.4. However, because $A = 0.1$ and $1 - A = 0.9$ in Equation (5.2), Figure 5.4 has shape and form that are similar to Figure 5.8.

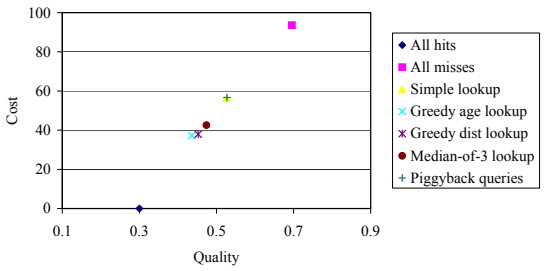

Figure 5.9. Cost vs. Quality for $A = 0.9$ and Independent changes over 1000 locations.

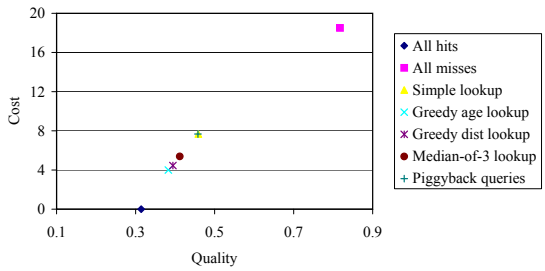

Figure 5.10. Cost vs. Quality for $A = 0.9$ and Trace-driven changes over 54 locations.

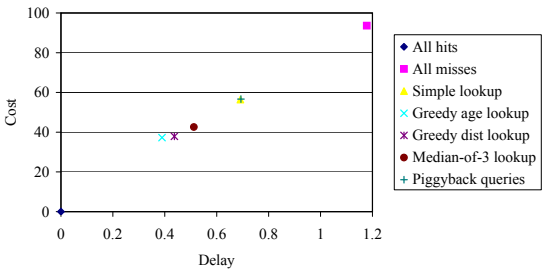

Figure 5.11. Cost vs. Delay for $A = 0.9$ and Independent changes over 1000 locations.

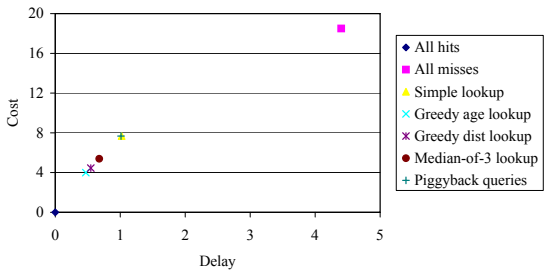

Figure 5.12. Cost vs. Delay for $A = 0.9$ and Trace-driven changes over 54 locations.

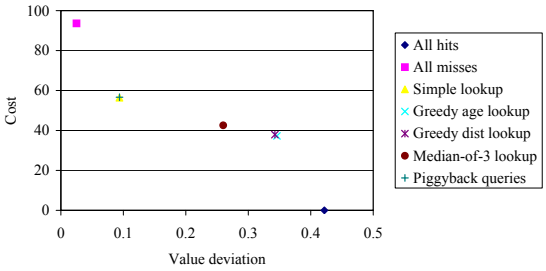

Figure 5.13. Cost vs. Value deviation for $A = 0.9$ and Independent changes over 1000 locations.

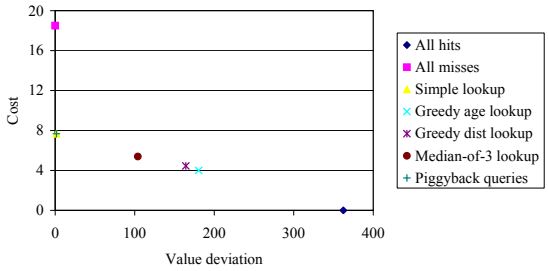

Figure 5.14. Cost vs. Value deviation for $A = 0.9$ and Trace-driven changes over 54 locations.

Because application requirements for accuracy and delay can be quite different, we now consider cost and quality performance when delay is significantly more important to quality than delay ($A = 0.9$), as shown in Figures 5.9 through 5.14. The even-numbered figures, Figures 5.10, 5.12, and 5.14, show results for light intensity in Lux measured over time in the Intel Berkeley lab data set. These results are for $T = 90$ seconds, and 0.9 queries per second. The odd-numbered figures, Figures 5.9, 5.11, and 5.13, are for independent changes to the environment with both the age parameter, T, and the average query rate scaled for the more rapidly changing environment. Specifically, $T = 8.8\overline{8}$ seconds and the average user query rate is 90 queries per second. The most remarkable result in these figures is that there is no trade-off between cost and quality when we significantly prioritize delay over value deviation (e.g., when $A = 0.9$). The two greedy caching and lookup policies have the best cost performance and the best quality performance for both models of changing the environment (see Figure 5.9 and Figure 5.10). Even though the "all hits" policy has

the best absolute performance in these figures, we don't consider this a practical policy since it never updates the cache. For trace-driven changes to the environment, even though the dispersion of value deviations in Figure 5.14 is greater than the dispersion of delays in Figure 5.12, delay still drives quality. In other words, the trends and characteristics of cost and quality performance in Figure 5.10 are driven by the delays shows in Figure 5.12.

Similar results hold for independent changes to the environment. The cost and quality performance in Figure 5.9 is driven by the system delays shown in Figure 5.11.

5.4.2 Correlated and Trace-driven Changes to the Environment

We wanted to build on the results in the previous section by capturing the fact that sensor field readings are correlated in both space and time. This feature is missing when we model how the environment changes using independent random variables, as in the previous section. We therefore revised our model with changes that are i.i.d. to make changes in the environment correlated in both space and time. At time $t+1$, the value at each location l is drawn from a normal distribution with mean

$$\mu_{l,t+1} = \frac{1}{3}\mu + \frac{1}{3}\mu_{l,t} + \frac{1}{3}\mu_{N(l),t}$$

and the same standard deviation (σ) as in Section 5.4.1. The long-term mean of this distribution is μ. $N(l)$ denotes the neighbors of location l, and each neighbor l' of l contributes to $\mu_{N(l),t}$ in proportion to $\mu_{l',t}/r_{l'}$, where $r_{l'}$ is the distance between locations l and l'. This model for a changing environment is based on the model for correlated sensor network data developed by Jindal and Psounis [70].

We can draw at least three conclusions from our experiments using the correlated and trace-driven models for how the environment changes. Results from these experiments appear in Figures 5.15 through 5.30.

1. There is a cost vs. quality trade-off for some quality requirements but not others. For example, consider the results shown in Figures 5.15 and 5.16. Figure 5.16 shows cost

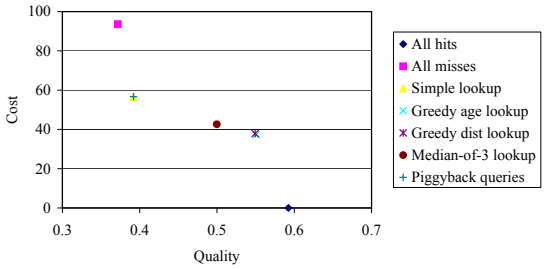

Figure 5.15. Cost vs. Quality for $A = 0.1$ and Correlated changes over 1000 locations.

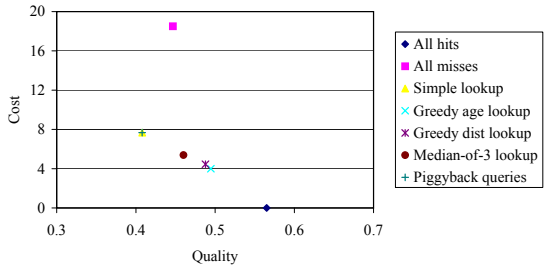

Figure 5.16. Cost vs. Quality for $A = 0.1$ and Trace-driven changes over 54 locations.

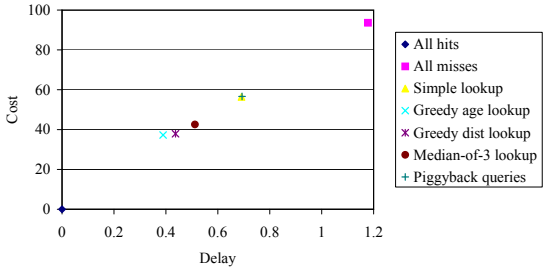

Figure 5.17. Cost vs. Delay for $A = 0.1$ and Correlated changes over 1000 locations.

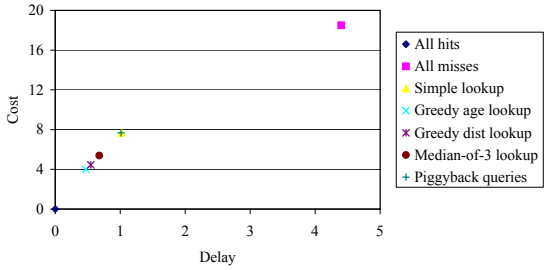

Figure 5.18. Cost vs. Delay for $A = 0.1$ and Trace-driven changes over 54 locations.

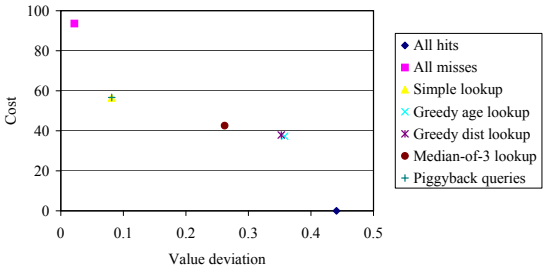

Figure 5.19. Cost vs. Value deviation for $A = 0.1$ and Correlated changes over 1000 locations.

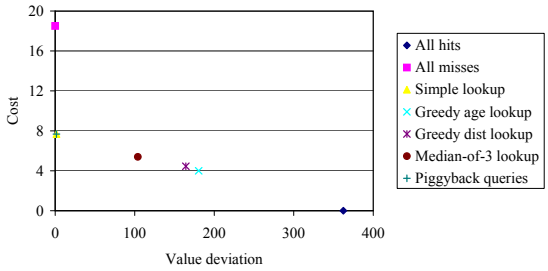

Figure 5.20. Cost vs. Value deviation for $A = 0.1$ and Trace-driven changes over 54 locations.

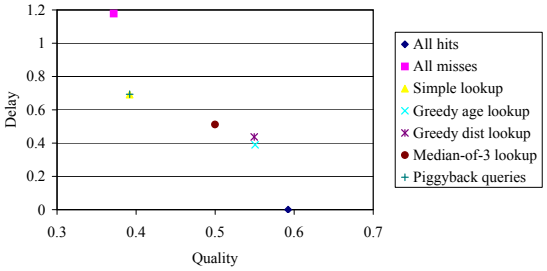

Figure 5.21. Delay vs. Quality for $A = 0.1$ and Correlated changes over 1000 locations.

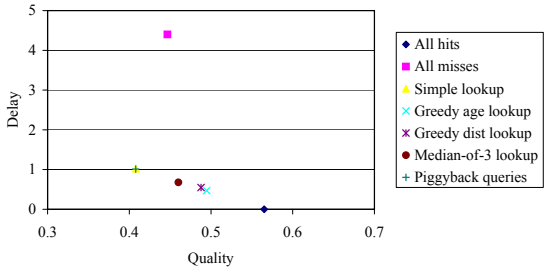

Figure 5.22. Delay vs. Quality for $A = 0.1$ and Trace-driven changes over 54 locations.

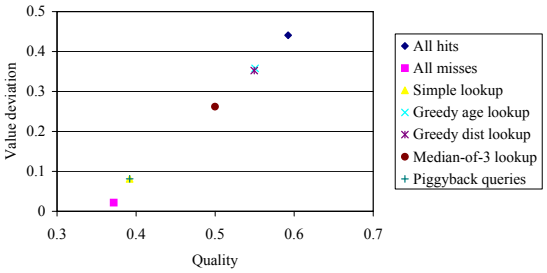

Figure 5.23. Value deviation vs. Quality for $A = 0.1$ and Correlated changes over 1000 locations.

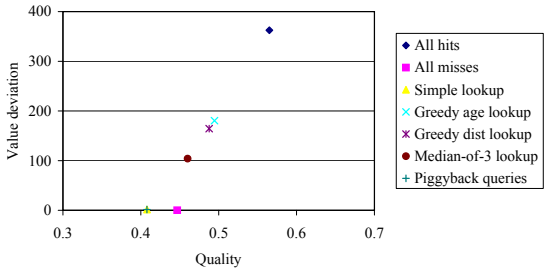

Figure 5.24. Value deviation vs. Quality for $A = 0.1$ and Trace-driven changes over 54 locations.

Table 5.4. Hit ratios, Costs, and Delays for $T = 8.8\overline{8}$, 90 Queries per second, and Correlated changes over 1000 locations.

Policies	Hit ratio	Cost	Delay
All hits	1	0	0
All misses	0	94	1.18
Simple lookup	0.40	56	0.69
Greedy age lookup	0.62	37	0.39
Greedy distance lookup	0.60	38	0.44
Median-of-3 lookup	0.55	43	0.51
Piggyback queries	0.40	57	0.69

Table 5.5. Hit ratios, Costs, and Delays for $T = 90$, 0.9 Queries per second, and Trace-driven changes over 54 locations.

Policies	Hit ratio	Cost	Delay
All hits	1	0	0
All misses	0	19	4.4
Simple lookup	0.59	7.7	1.0
Greedy age lookup	0.78	4.0	0.47
Greedy distance lookup	0.76	4.4	0.55
Median-of-3 lookup	0.71	5.4	0.68
Piggyback queries	0.59	7.7	1.0

versus quality for all seven caching and lookup policies, where $A = 0.1$ and the values at each location are changed according to the Intel Berkeley lab trace [37]. At the smallest cost, we have a 100% cache hit ratio (labeled "All hits") that provides a quality of below 0.6 for zero cost. For the largest cost, we see that a 0% cache hit ratio (labeled "All misses") provides the third-best quality at a cost of approximately 19 units. Recall that our cost units are normalized units, and are proportional to the square of the distance to query a particular location in the sensor field (see Section 5.2.1 for more detail). Recall also that for quality, smaller values indicate better quality. The remaining five caching and lookup policies provide a linear trade-off between cost and quality. Figure 5.15 also shows a trade-off between cost and quality for the same value of A and the same seven caching and lookup policies, but with changes to the environment now modeled by a series of values correlated in space and time. There are two observations worth noting when comparing these first two figures. First, the cost values in Figure 5.16 are less than in Figure 5.15 because the distances within the sensor field are smaller. Second, the trends are similar between these two figures, with the exception of the increase in quality of the "all misses" policy between Figure 5.15 and Figure 5.16. This worse "all misses" quality is due entirely to an increase in the normalized delay term in the left hand side of Equation (5.2). This can be verified by comparing the relative differences in delays between the policies, shown on the horizontal axes in Figures 5.17 and 5.18.

2. Our choice of A is important because A determines to what extent delay or value deviation is the most important component of quality. A value of $A = 0.1$ is used in Figures 5.15 through 5.24. This value of A makes value deviation more important than system delay. This is demonstrated in Figures 5.21–5.24. Figure 5.24 shows a mostly positive correlation between value deviation and quality, whereas Figure 5.22 shows a mostly negative correlation between delay and quality. These figures are both for trace-driven changes to the environment. Figures 5.21 and 5.23 show that the same results hold for correlated changes to the environment: There is a positive correlation between value deviation and quality on the one hand, but a negative correlation between delay and quality on the other.

In studying Figures 5.15 through 5.30 it is interesting to understand which system variables depend on which system parameters. For example, cost, delay, and hit ratio values in these simulation results each depend on the following three variables:

- The caching and lookup policies themselves (including the value of T); the
- physical configuration of the sensor field; and the
- query arrival process.

Thus, a cost vs. delay or a cost vs. hit ratio graph should be the same for different experiments in which these three variables are held constant. Figure 5.17 shows the characteristic cost vs. delay results for our 1000-location network with caching and lookup policies parameterized with $T = 8.8\overline{8}$ seconds. Figure 5.18 shows the characteristic cost vs. delay results for the smaller-scale, 54-location network with caching and lookup policies parameterized with $T = 90$. In other words, these figures show cost vs. delay for any legal value of A, a fixed value of T, and the respective model for changing the environment. Note that the trends are the same in both of these figures. The most significant difference between the two figures is the increase in delay for the "all misses" case in Figure 5.18 when compared with Figure 5.17. This is due to the increase in the queuing delay at the sensor network query queue caused by the lower-capacity access link. To see how cost and delay both in-

crease with lower cache hit ratios, Table 5.4 shows the cache hit ratio for each of the (cost, delay) points in Figure 5.17. Similarly, Table 5.5 shows the hit ratio for each of the (cost, delay) points in Figure 5.18.

Value deviation depends on the same parameters listed above, and additionally on the manner in which the environment changes. Thus, cost vs. value deviation graphs are the same when the policies, sensor field, query arrival process, and method for changing the environment are all identical. Figure 5.19 and Figure 5.20 show cost vs. value deviation results for correlated changes and trace-driven changes to the environment, respectively. The most interesting difference between the two figures is the overall increase the dispersion of the value deviations in Figure 5.20 when compared those in Figure 5.19. This is because the variation in sensor field values is much greater in the Intel Berkeley trace data than values that are drawn from our normal distribution with a time-dependent mean.

We now examine system configurations for which delay is the more important component of quality in more detail. Figures 5.25 through 5.30 show such configurations for a value of $A = 0.9$. Figure 5.28 shows a strong positive correlation between delay and raw quality whereas Figure 5.30 shows a negative correlation between value deviation and quality. These figures are both for trace-driven changes to the environment sensed by 54 nodes. However, Figure 5.27 and Figure 5.29 show that the same delay and value deviation results hold for correlated changes to the environment with 1000 locations.

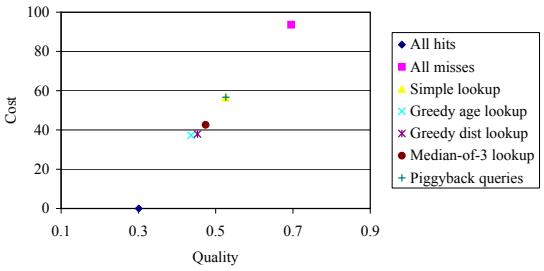

Figure 5.25. Cost vs. Quality for $A = 0.9$ and Correlated changes over 1000 locations.

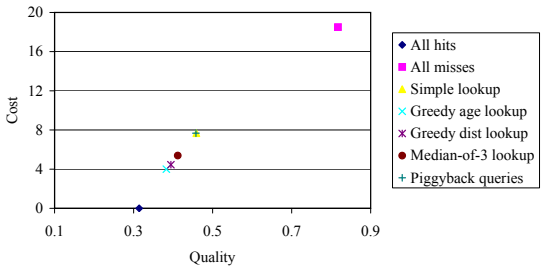

Figure 5.26. Cost vs. Quality for $A = 0.9$ and Trace-driven changes over 54 locations.

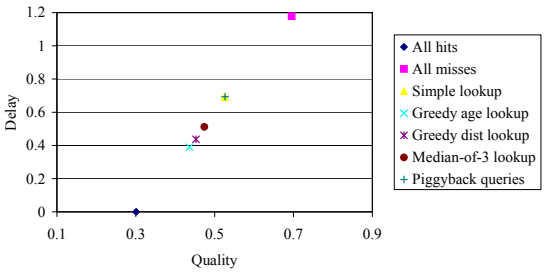

Figure 5.27. Delay vs. Quality for $A = 0.9$ and Correlated changes over 1000 locations.

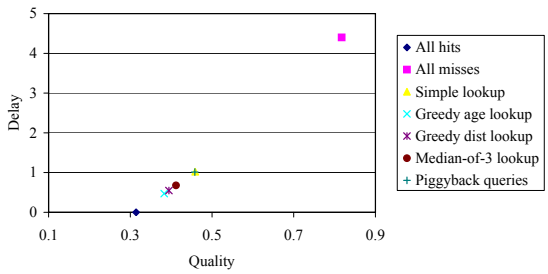

Figure 5.28. Delay vs. Quality for $A = 0.9$ and Trace-driven changes over 54 locations.

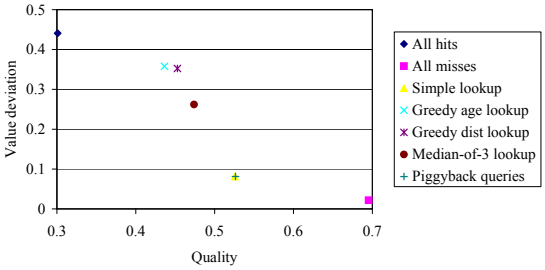

Figure 5.29. Value deviation vs. Quality for $A = 0.9$ and Correlated changes over 1000 locations.

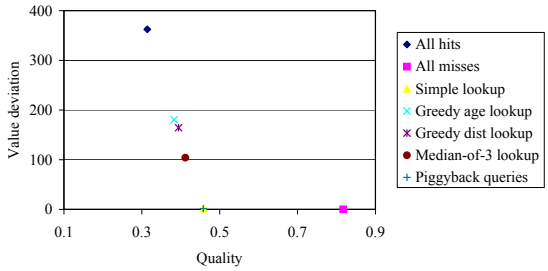

Figure 5.30. Value deviation vs. Quality for $A = 0.9$ and Trace-driven changes over 54 locations.

3. Different lookup policies perform best depending on whether delay or value deviation is most important to the application. Recall that the first of our alternative age-based policies (simple lookups, piggybacked queries, greedy age lookups, greedy distance lookups, and median-of-3 lookups) performs "simple lookups" in the sensor data cache to determine whether or not we have a cached value for a particular location. Recall also that four other caching policies add features that are designed to improve the effective cache hit ratio and thus conserve system resources. These policies are piggybacked queries, greedy age lookups, greedy distance lookups, and median-of-3 lookups.

If quality is more important to the application than cost, and value deviation is more important than delay, simple lookups and piggybacked queries provide the best performance. This can be seen in Figures 5.15 and 5.16. In both of these figures, simple lookups and piggybacked queries yield the best quality, other than the "all misses" policy for correlated changes. When value deviation is most important, the expense of taking a cache miss

(by not computing an approximate value from neighboring values for these two policies) is worthwhile, since value deviation is deemed most important. If cost is at a premium compared with quality, using greedy age lookups or greedy distance lookups is preferred. These two policies have the most favorable cost performance in both sensor field models, other than the "all hits" case in which the sensor field is never queried.

If delay is more important to quality than value deviation, Figures 5.25 and 5.26 show that performing greedy age lookups or doing greedy distance lookups yields the best performance. This is true regardless of whether cost or quality is more important to the application. We again assume that the "all hits" case is not useful to realistic applications. For these policies, getting the fast response time of a cache "hit" (which might be approximated from values at one or more neighboring locations) is worthwhile, since low delay is more important than a more accurate value.

The fact that different lookup policies perform best for different application requirements can be explained by examining the underlying delays and value deviations of the policies themselves. For example, consider the case where $A = 0.1$ and changes to the environment are driven by the Intel Berkeley lab traces. Figure 5.22 shows that the delay for the simple lookup and piggyback query policies is greater than that of the median-of-3 and greedy lookup policies. However, Figure 5.24 shows that the value deviation for the simple lookup and piggyback query policies is significantly less than that of the median-of-3 and greedy lookup policies. A value of $A = 0.1$ biases quality toward value deviation performance rather than delay performance. Figure 5.16 therefore shows that the quality supported by the simple lookup and piggyback query policies is superior to the quality supported by the greedy and median-of-3 lookup policies. Now consider the case where $A = 0.9$ and changes to the environment are again driven by the lab trace data. Figure 5.30 shows that the value deviation for the greedy policies is greater than for the simple and piggyback policies, and the median-of-3 policy. However, Figure 5.28 shows that the delay for the greedy policies is less than that of the other policies. A value of $A = 0.9$ biases quality

toward delay performance rather than value deviation performance. Figure 5.26 therefore shows that the cost incurred for doing greedy age lookups or greedy distance lookups is superior to (i.e., less than) the cost incurred by the other policies for quality that is also better.

We observed our widest confidence intervals in the results presented in this section. Specifically, for our simulated sensor field (with 1000 nodes), the 90% confidence interval surrounding the All misses cost was $\pm 3.4\%$. We also observed our widest confidence interval for quality. For our sensor network model of the Intel Berkeley lab, the 90% confidence interval surrounding the All misses quality was $\pm 1.8\%$, when $A = 0.9$.

It is helpful to summarize the cost and quality performance results presented above as follows:

- When value deviation is more important to quality than delay, there is a linear cost vs. quality trade-off. We obtain the best cost performance by implementing policies that approximate sensor values by using cached values from nearby locations. The best quality performance is achieved by policies that always query and cache the sensor field location specified in the user query.

- When delay is more important than value deviation, policies that approximate values using cached values from nearby locations provide the best cost performance as well as the best quality performance.

- These results hold for both simulated changes to the environment and trace-driven changes to the environment.

5.5 Performance Trends when Value Deviation is Most Important

Our baseline results provide a thorough understanding of cost and quality performance in wireless sensor networks for two models of how the environment changes. We next

investigate performance trends that emerge as the query rate increases or decreases relative to the rate at which the environment changes.

Because our simulator fully models an environment with correlated changes, we explicitly vary the rate at which the environment changes. To do this we increase the number of locations changed per second from 9 to 90 to 900 while maintaining a query workload with an average rate of 90 queries per second. The Intel Berkeley lab traces specify how the environment changes for our trace-driven experiments. In this case, we vary the relative rate at which the environment changes by decreasing the average query rate from 90 to 9 to 0.9 queries per second.

We begin this investigation by considering cost and quality performance when value deviation is more important in determining query and reply quality than system end-to-end delay. Results for these simulations appear in this section. In the following section, we explore sensor network applications for which system delay is more important than value deviation.

The same three high-level conclusions presented in the last section hold true when the relative rate at which the environment changes is increased by two orders of magnitude. However, some of the underlying results differ either qualitatively or quantitatively. Figures 5.31 through 5.36 show these results for correlated changes to the environment. Figures 5.37 through 5.42 show the corresponding results for trace-driven changes to the environment.

If value deviation is more important than delay, simple lookups and piggybacked queries again provide the best performance when quality is more important than cost. This can be seen in Figures 5.31, 5.32 and 5.33. If cost is at a premium compared with quality, greedy age lookups or greedy distance lookups trade worse quality for lower cost. In fact, these policies have the most favorable cost performance, other than the “all hits” case. These figures also show that our results are robust with respect to how rapidly our correlated environment changes. For example, the trade-off between cost and quality is linear in Fig-

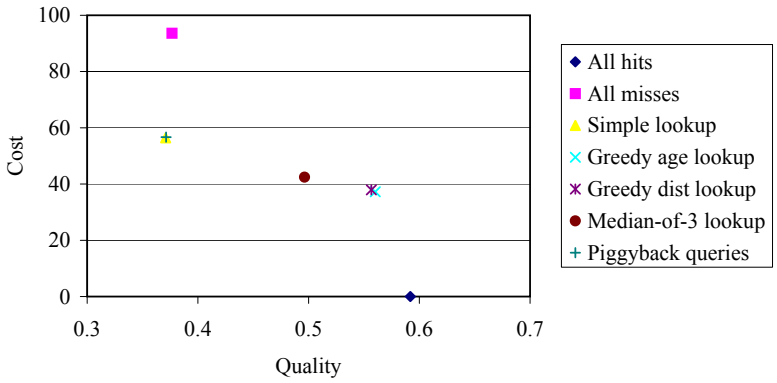

Figure 5.31. Cost vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 9 of 1000 Correlated changes / second.

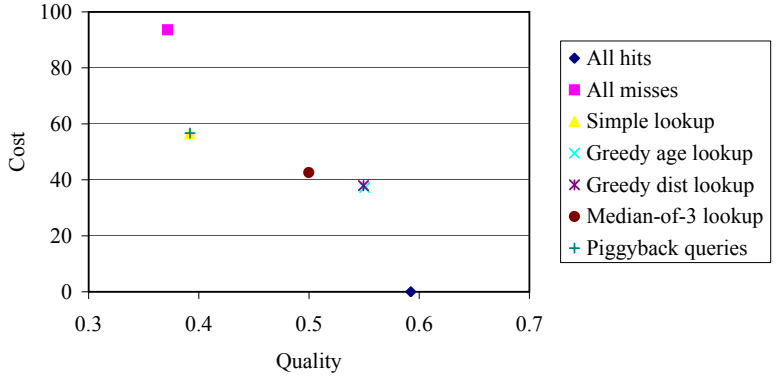

Figure 5.32. Cost vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 90 of 1000 Correlated changes / second.

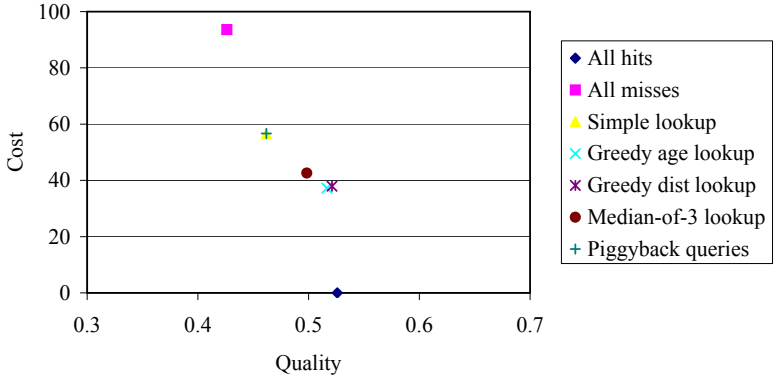

Figure 5.33. Cost vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 900 of 1000 Correlated changes / second.

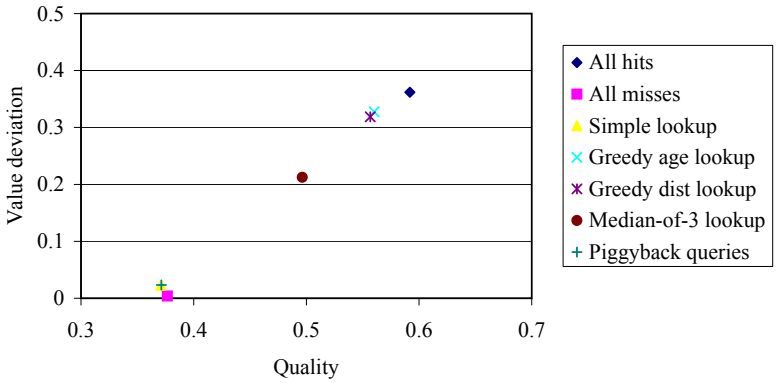

Figure 5.34. Value deviation vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 9 of 1000 Correlated changes / second.

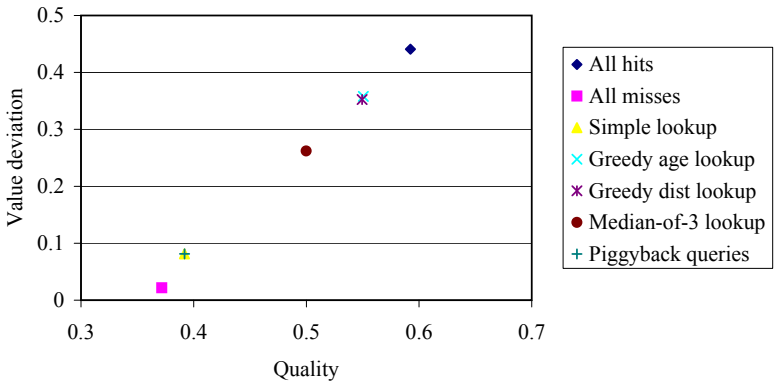

Figure 5.35. Value deviation vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 90 of 1000 Correlated changes / second.

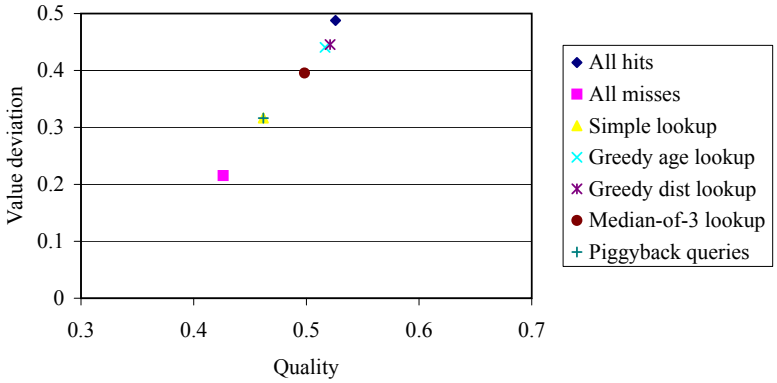

Figure 5.36. Value deviation vs. Quality for $A = 0.1$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 900 of 1000 Correlated changes / second.

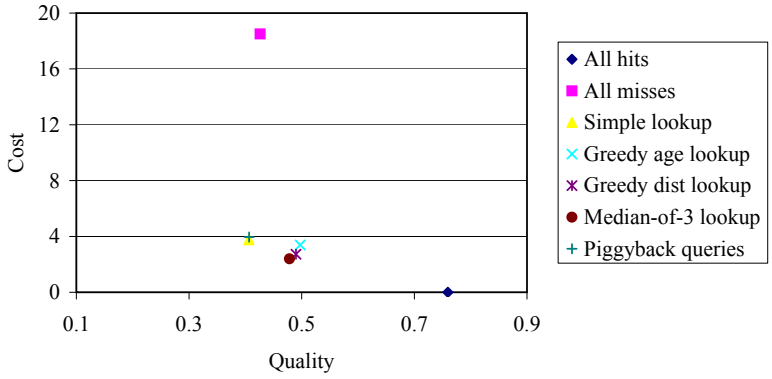

Figure 5.37. Cost vs. Quality for A = 0.1, T = 9 seconds, 90 Queries / second and Trace-driven changes over 54 locations.

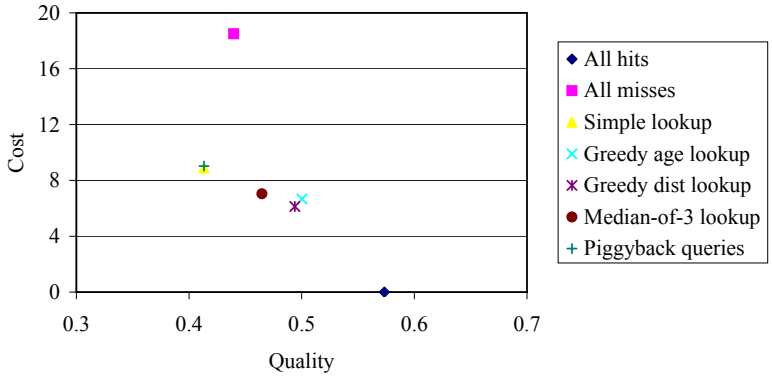

Figure 5.38. Cost vs. Quality for A = 0.1, T = 9 seconds, 9 Queries / second and Trace-driven changes over 54 locations.

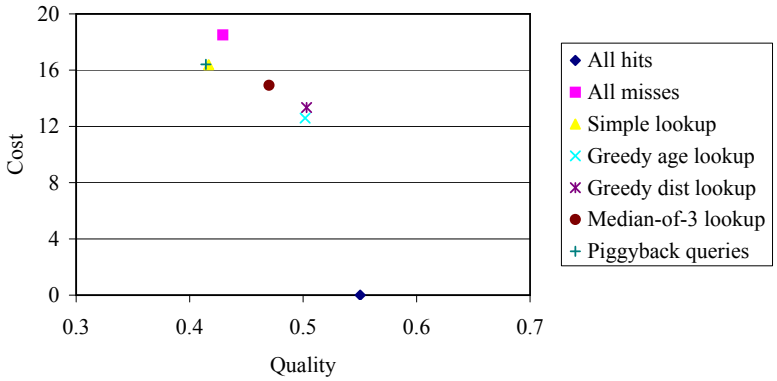

Figure 5.39. Cost vs. Quality for A = 0.1, T = 9 seconds, 0.9 Queries / second and Trace-driven changes over 54 locations.

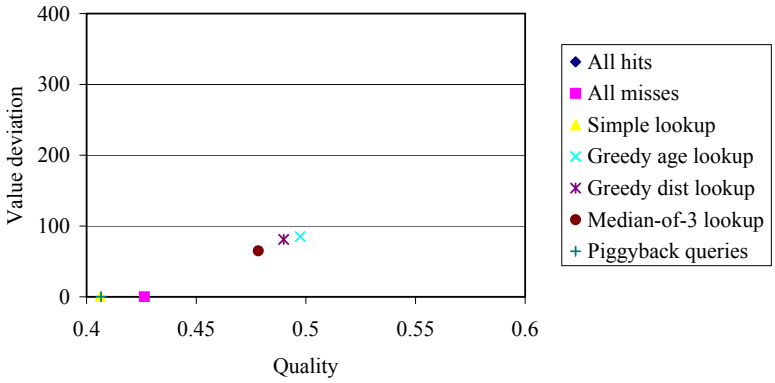

Figure 5.40. Value deviation vs. Quality for A = 0.1, $T = 9$ seconds, 90 Queries / second and Trace-driven changes over 54 locations.

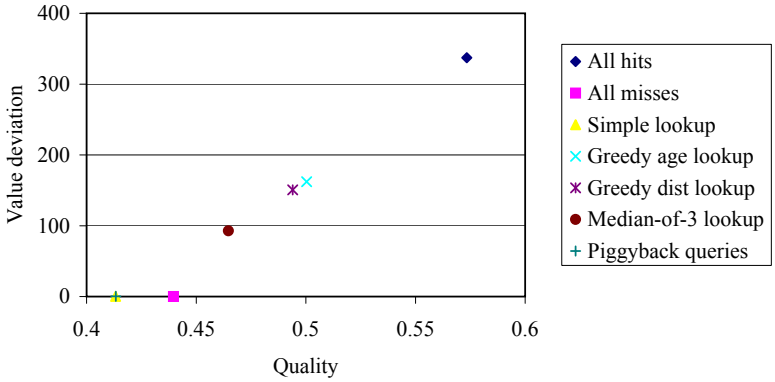

Figure 5.41. Value deviation vs. Quality for A = 0.1, $T = 9$ seconds, 9 Queries / second and Trace-driven changes over 54 locations.

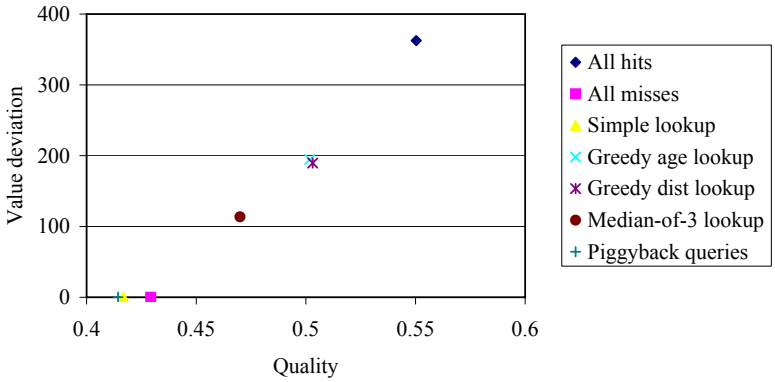

Figure 5.42. Value deviation vs. Quality for A = 0.1, $T = 9$ seconds, 0.9 Queries / second and Trace-driven changes over 54 locations.

ures 5.31, 5.32, and 5.33. Excluding the "all hits" case again, the range of quality values decreases as the rate at which the environment changes increases. In spite of these differences in quality performance, the relative cost and quality performance of most of the policies remain the same. Figures 5.34, 5.35, and 5.36 confirm a strong positive correlation between value deviation and quality when $A = 0.1$. Examining these figures in increasing order illustrates how value deviation increases as the rate at which the environment changes increases.

For trace-driven changes to the environment, simple lookups and piggybacked queries also provide the best performance when quality is more important than cost. This can be seen in Figures 5.37, 5.38, and 5.39. The trade-off between cost and quality is also linear in these Figures. Although the costs increase from Figure 5.37 to Figure 5.38 to Figure 5.39, the range of quality is approximately the same (again, excluding the "all hits" case).

Figures 5.40, 5.41, and 5.42 confirm a positive correlation between value deviation and quality when $A = 0.1$ and changes to the environment are driven by real-world trace data. Examining these figures in increasing order illustrates how cost increases as the average query rate (and thus the cache hit ratio) decreases.

We have shown that when value deviation is more important to quality than system delay, we consistently observe a cost vs. quality trade-off which can be characterized as follows:

- *Approximate* query and lookup policies, which approximate sensor field values by using values from one or more nearby cached locations, provide the best cost performance.

- *Precise* query and lookup policies, which always query the specified location in the sensor field, provide the best quality performance.

- For our sensor network model with correlated changes to the environment, this trade-off occurs even as we vary the rate that the environment changes by two orders of magnitude.

- For our smaller sensor network model that uses real-world trace data, these results hold as we vary the rate at which the sensor field is queried, also by two orders of magnitude.

5.6 Performance Trends when End-to-End Delay is Most Important

In the previous section we presented performance results that are important to sensor network applications for which high accuracy (i.e., low value deviation) is the most important factor in the quality of their data. In this section we consider how performance varies as the query rate increases or decreases relative to the rate at which the environment changes when system end-to-end delay is most important. For correlated changes to the environment, we again vary the rate at which the environment changes from 9 to 90 to 900 locations per second while maintaining a query workload with an average rate of 90 queries per second. For the Intel Berkeley lab traces, we again vary the relative rate at which the environment changes by decreasing the average query rate from 90 to 9 to 0.9 queries per second.

We examine system configurations for which delay is a more important component of quality than value deviation. Figures 5.43 through 5.48 show such configurations for correlated changes to the environment. For correlated changes to the environment, the cost remains constant for policies as the rate of change for sensor values in the environment increases from 9 locations/s to 900 locations/s. At the same time, the quality performance for all policies is very similar, but not identical. Figures 5.49 through 5.54 show configurations for trace-driven changes to the environment. In these system configurations, the cost increases as the average query rate and the cache hit ratio both decrease.

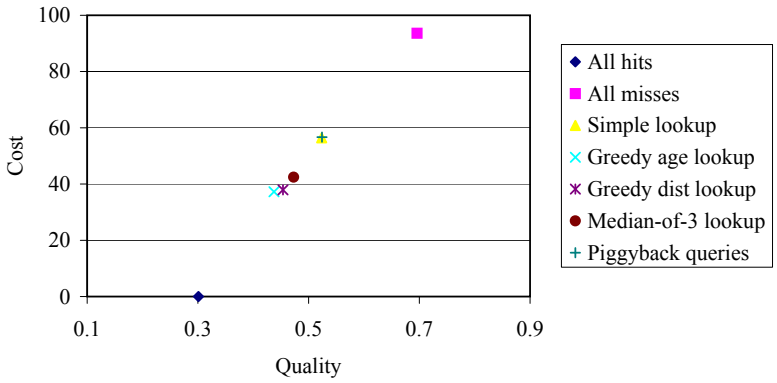

Figure 5.43. Cost vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 9 of 1000 Correlated changes / second.

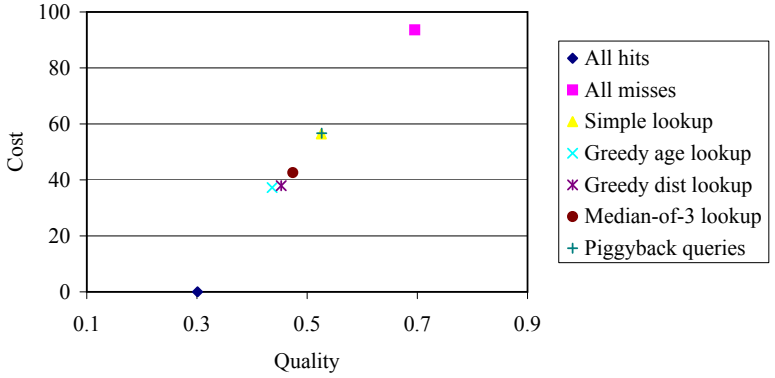

Figure 5.44. Cost vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 90 of 1000 Correlated changes / second.

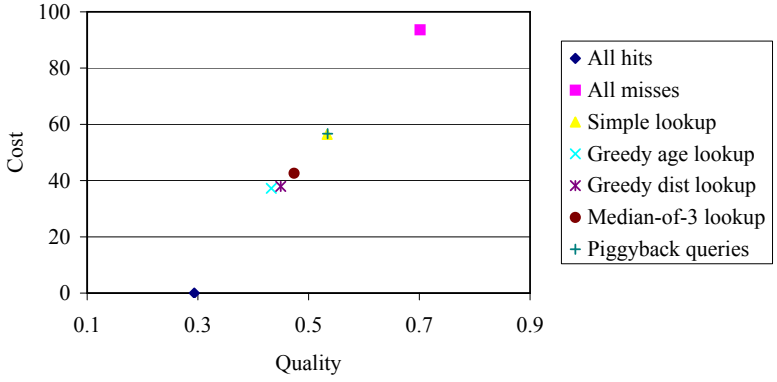

Figure 5.45. Cost vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 900 of 1000 Correlated changes / second.

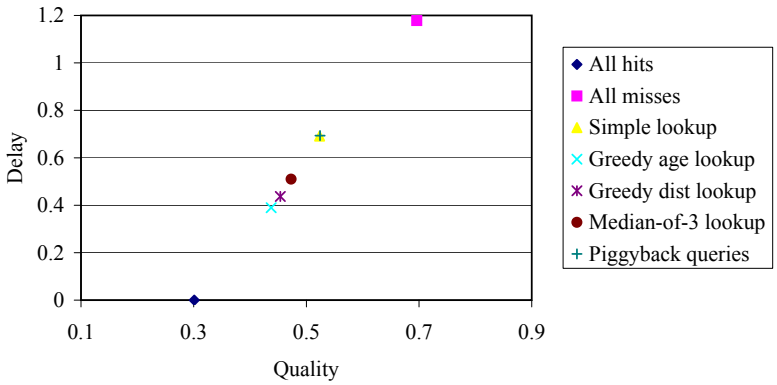

Figure 5.46. Delay vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 9 of 1000 Correlated changes / second.

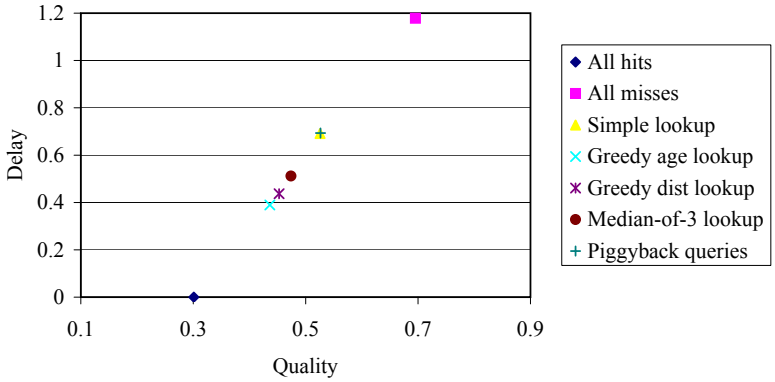

Figure 5.47. Delay vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 90 of 1000 Correlated changes / second.

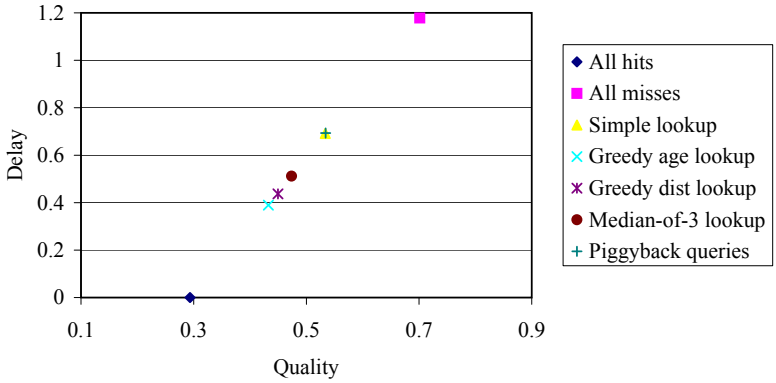

Figure 5.48. Delay vs. Quality for $A = 0.9$, $T = 8.8\overline{8}$ seconds, 90 Queries / second, and 900 of 1000 Correlated changes / second.

A value of $A = 0.9$ is used in Figures 5.43 through 5.54. This choice of A makes delay significantly more important to quality than value deviation. This is demonstrated, for example, by Figures 5.46, 5.47, and 5.48. These figures all show a strong positive and linear correlation between delay and quality for correlated changes to the environment. Examining these figures also illustrates that delays for specific policies are independent of the rate at which the environment changes, and thus are constant for these system configurations since the query workloads are the same.

For correlated changes to the environment, Figures 5.43, 5.44, and 5.45 show that performing greedy age lookups or greedy distance lookups yields the best cost performance, and the best quality performance, excluding the "all hits" case. These figures also show that our results are robust with respect to how rapidly our correlated sensor field values change, and that the relative performance differences among our policies remain the same. Similar results hold for trace-driven changes to the environment. These can be seen in Figures 5.49, 5.50, and 5.51. However, it appears that as higher cache hit ratios (e.g., in Figure 5.49) cause a performance convergence among the caching policies, the median-of-3 policy exhibits better cost and quality performance than doing greedy lookups.

We have seen that for correlated changes to the environment, one or more of the greedy policies provide the best cost and quality performance among the realistic policies. For trace-driven changes to the environment, greedy policies (including the median-of-3 policy) also provide the best cost performance and the best quality performance. This can be seen in Figures 5.49, 5.50, and 5.51, in which there is no trade-off between cost and quality. As most costs increase from Figure 5.49 to Figure 5.50 to Figure 5.51, the range of quality values also increases. However, these quality values remain in the continuum between the "All hits" and "All misses" quality values.

Figures 5.52, 5.53, and 5.54 illustrate the positive correlation between delay and quality when $A = 0.9$ and changes to the environment are driven by the lab trace data. Examining

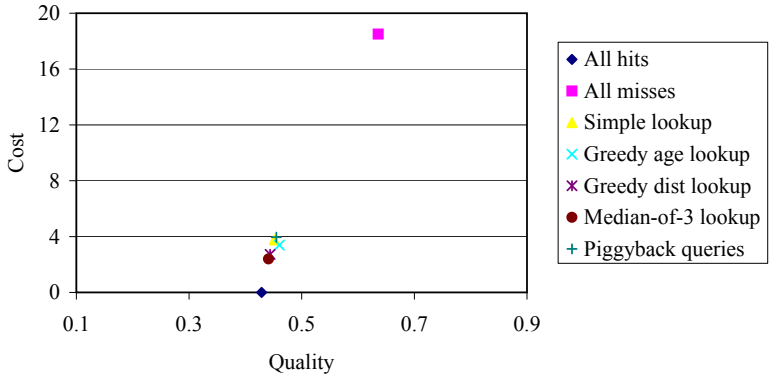

Figure 5.49. Cost vs. Quality for $A = 0.9$, $T = 9$ seconds, 90 Queries / second and Trace-driven changes over 54 locations.

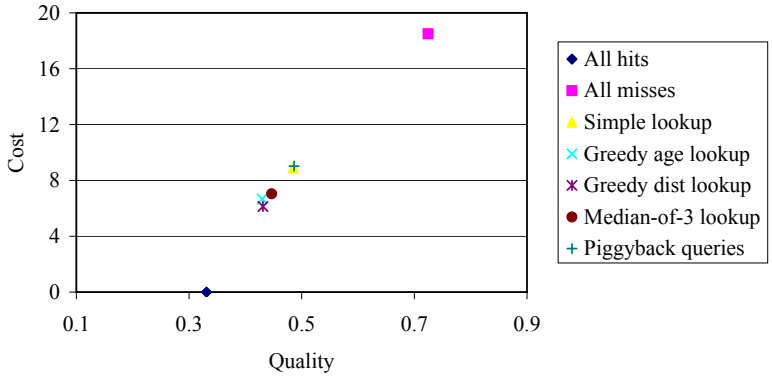

Figure 5.50. Cost vs. Quality for $A = 0.9$, $T = 9$ seconds, 9 Queries / second and Trace-driven changes over 54 locations.

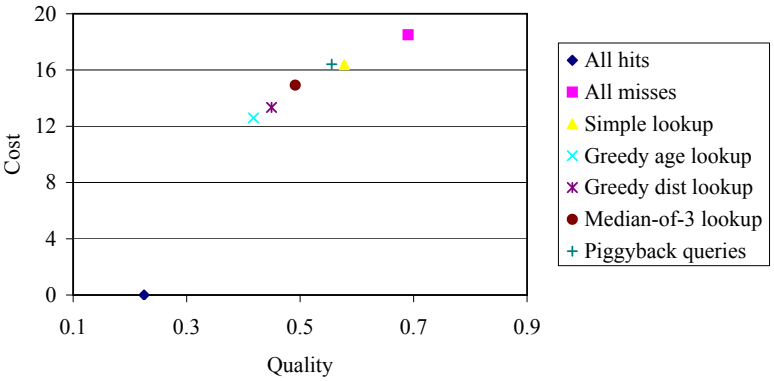

Figure 5.51. Cost vs. Quality for $A = 0.9$, $T = 9$ seconds, 0.9 Queries / second and Trace-driven changes over 54 locations.

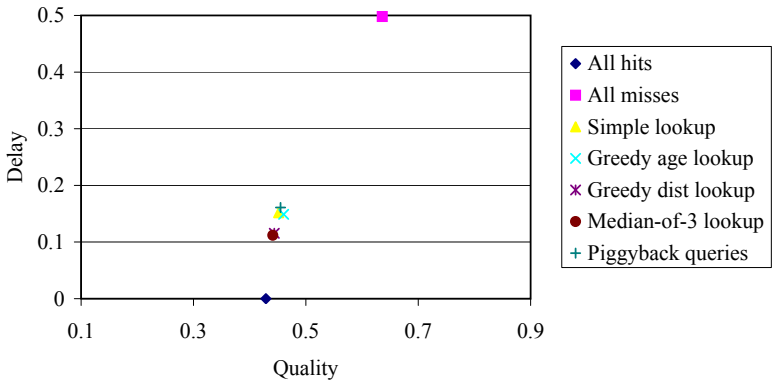

Figure 5.52. Delay vs. Quality for $A = 0.9$, $T = 9$ seconds, 90 Queries / second and Trace-driven changes over 54 locations.

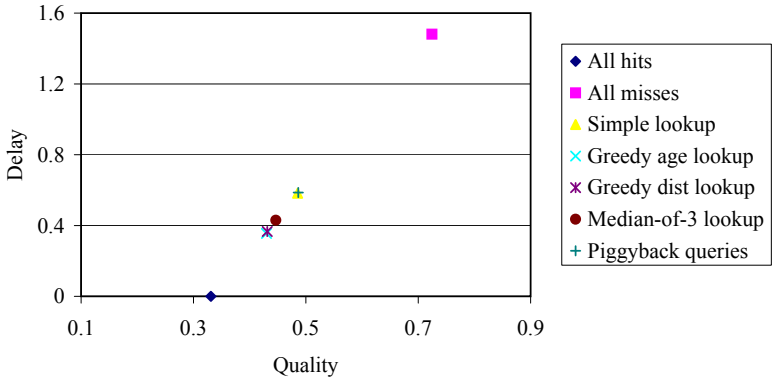

Figure 5.53. Delay vs. Quality for $A = 0.9$, $T = 9$ seconds, 9 Queries / second and Trace-driven changes over 54 locations.

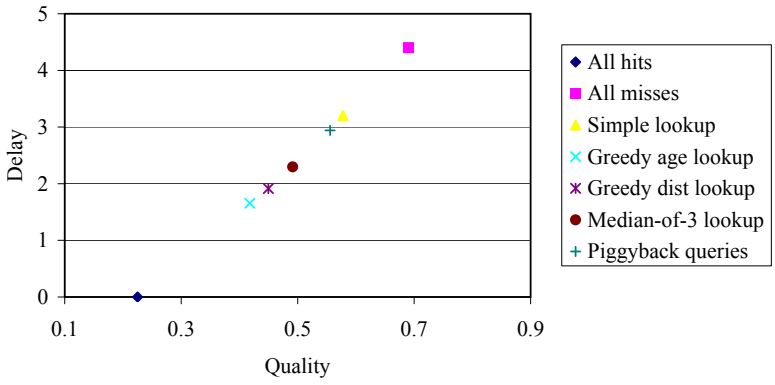

Figure 5.54. Delay vs. Quality for $A = 0.9$, $T = 9$ seconds, 0.9 Queries / second and Trace-driven changes over 54 locations.

these figures in increasing order confirms that the trajectory of increases in cost (as the average query rate decreases) is independent of A.

In summary, we have shown that caching approaches that prevent cache misses by computing and returning approximate values for sensor data can avoid a trade-off between cost and quality. This is because when delay is more important than value deviation, the benefit to both cost and quality achieved by using approximate values outweighs the negative impact on quality due to the approximation. Furthermore, we have demonstrated that these results are robust with respect to the manner in which the environment changes for two different sensor field models.

5.7 Performance Impact of Cache Entry Age Threshold (T)

5.7.1 Larger Values of T

Since power is typically a critical resource in wireless sensor networks, we wanted to understand the effects of decreasing resource consumption by reducing the volume of sensor field query traffic. We accomplished this by increasing the value of the age threshold parameters associated with cache entries (our values of T), and thus boosting the cache hit ratio for our age-based policies. For simple lookups, piggybacked queries, the median-of-3 policy, and both greedy policies, age thresholds of $T = 88.8\overline{8}$ and $T = 900$ seconds were used for correlated and trace-driven changes, respectively. Theses values are an order of magnitude or more larger than the values of T used for our earlier results. For our larger environment (with 1000 locations), the sensor field was queried an average of 90 times per second and changed at a rate of 90 locations per second. For our smaller environment (with 54 locations), the sensor field was queried at a rate of 0.9 queries per second and changed at a rate of approximately once every 1.33 seconds (per the trace data). Our results for larger values of T appear in Figures 5.55 through 5.66.

The most noticeable difference between the results in Figures 5.55 through 5.66 when compared with our earlier results is that the performance differences between the five age-

based policies are quite small. This similar performance of our age-based policies is most pronounced for performance metrics that are independent of A (e.g., cost, delay, and hit ratio). For example, Figure 5.57 and Figure 5.58 show cost vs. delay performance for all non-trivial policies (from simple lookups through piggybacked queries) that is almost the same for each model of how the environment changes. The corresponding hit ratios for these policies are also virtually identical within their respective models (see Table 5.6 and Table 5.7). In fact, the hit ratio for all of these policies falls between 88% and 90% for correlated changes, and between 94% and 95% for trace-driven changes. These hit ratios are much greater than our earlier results in which the hit ratio ranges were 40% to 62% in Table 5.4, and 59% to 78% in Table 5.5. Even the characteristic cost vs. value deviation graphs, Figure 5.59 and Figure 5.60, show much less difference in value deviation than our earlier results. In all of our results for larger values of T, the much higher hit ratios (88% or greater) make the cost performance of our policies fairly close to the case where the hit ratio is 100% (i.e., the "all hits" case). Furthermore, the quality performance is close to the quality performance of the all hits case when either changes are correlated (see Figures 5.55 and 5.61) or delay is deemed more important than value deviation (e.g., when $A = 0.9$ in Figure 5.62).

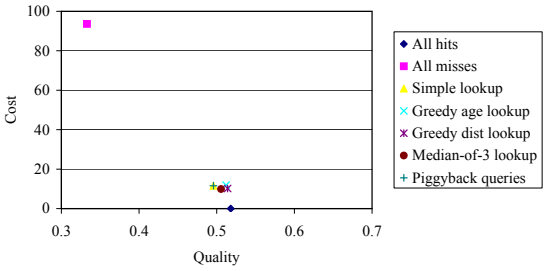

Figure 5.55. Cost vs. Quality for $A = 0.1$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

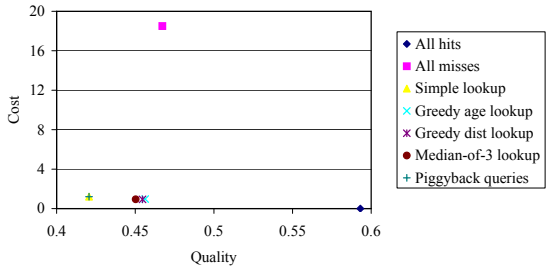

Figure 5.56. Cost vs. Quality for $A = 0.1$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

We were surprised that in spite of significant changes in our cache entry aging parameter, some of our important earlier results still hold. These results can be summarized as follows:

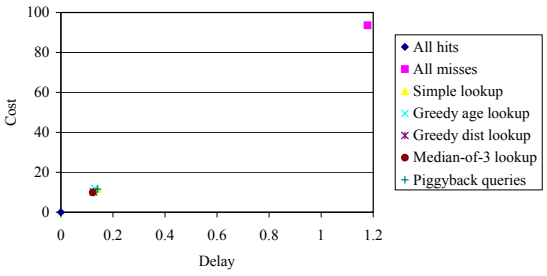

Figure 5.57. Cost vs. Delay for $A = 0.1$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

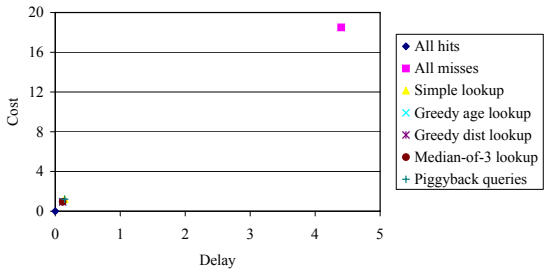

Figure 5.58. Cost vs. Delay for $A = 0.1$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

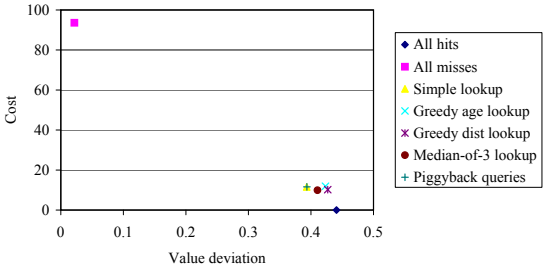

Figure 5.59. Cost vs. Value deviation for $A = 0.1$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

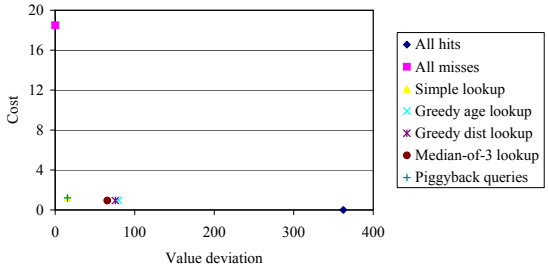

Figure 5.60. Cost vs. Value deviation for $A = 0.1$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

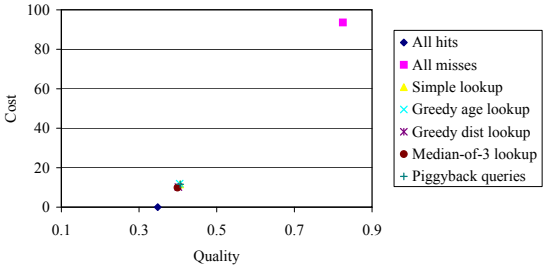

Figure 5.61. Cost vs. Quality for $A = 0.9$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

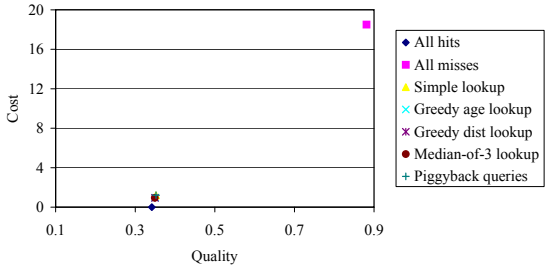

Figure 5.62. Cost vs. Quality for $A = 0.9$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

Table 5.6. Hit ratios, Costs, and Delays for $T = 88.8\overline{8}$, 90 Queries per second, and Correlated changes over 1000 locations.

Policies	Hit ratio	Cost	Delay
Simple lookup	0.88	12	0.14
Greedy age lookup	0.88	12	0.13
Greedy distance lookup	0.89	10	0.13
Median-of-3 lookup	0.90	10	0.12
Piggyback queries	0.88	12	0.14

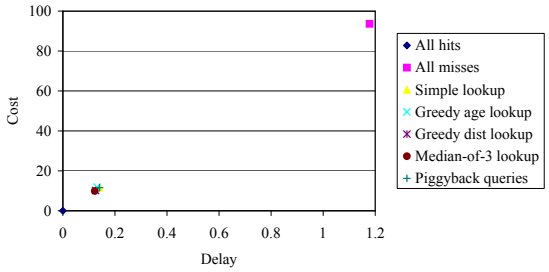

Figure 5.63. Cost vs. Delay for $A = 0.9$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

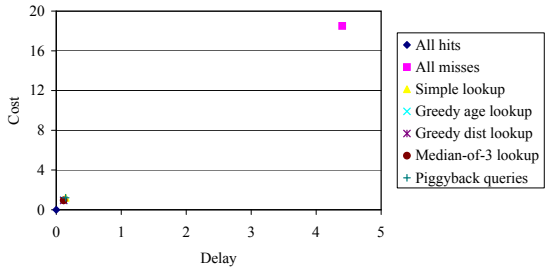

Figure 5.64. Cost vs. Delay for $A = 0.9$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

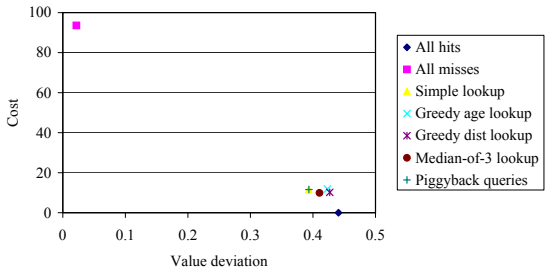

Figure 5.65. Cost vs. Value deviation for $A = 0.9$, $T = 88.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

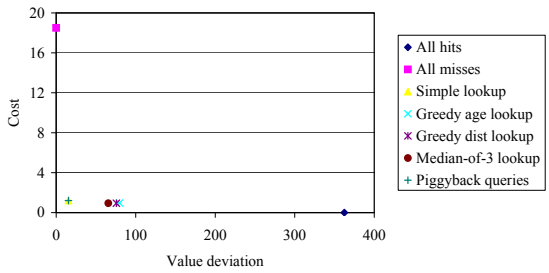

Figure 5.66. Cost vs. Value deviation for $A = 0.9$, $T = 900$ seconds, and Trace-driven changes over 54 nodes.

Table 5.7. Hit ratios, Costs, and Delays for $T = 900$, 0.9 Queries per second, and Trace-driven changes over 54 locations.

Policies	Hit ratio	Cost	Delay
Simple lookup	0.94	1.2	0.14
Greedy age lookup	0.95	0.97	0.11
Greedy distance lookup	0.95	0.94	0.11
Median-of-3 lookup	0.95	0.94	0.11
Piggyback queries	0.94	1.2	0.14

- For $A = 0.1$, value deviation still mostly drives the trade-off between quality and cost. For example, Figure 5.56 shows the same trends as its value deviation counterpart, Figure 5.60, except in the "all misses" case. In this case, the poor delay performance of all misses shown in Figure 5.58 hurts the corresponding all misses quality performance in Figure 5.56.

- For $A = 0.9$, system delay drives quality, and there is no trade-off between quality and cost. Thus for correlated changes, Figure 5.61 exhibits the same trends as its delay counterpart, Figure 5.63. Likewise for trace-driven changes, Figure 5.62 shows the same trends as Figure 5.64.

5.7.2 Smaller Values of T

In the previous section we examined results for high cache ratios, which were induced by increasing the age threshold, T. We also wanted to study system configurations with low cache hit ratios. To obtain results for these configurations, we decreased our values of T by an order of magnitude when compared with the values of T used to produce the baseline results (see Section 5.4). For our five age-based policies, an age parameter of $T = 9$ seconds was used for our trace-driven experiments, and $T = 0.8\overline{8}$ seconds was used with correlated changes. Note also that these values are two orders of magnitude smaller than the respective values of T used in Section 5.7.1. The results for smaller values of T appear in this section in Figures 5.67 through 5.74.

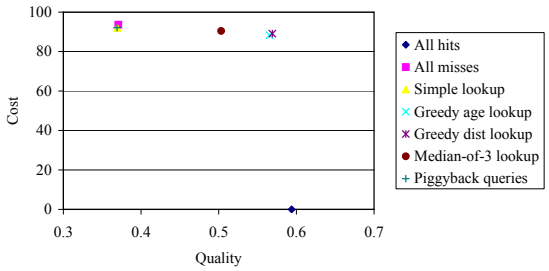

Figure 5.67. Cost vs. Quality for $A = 0.1$, $T = 0.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

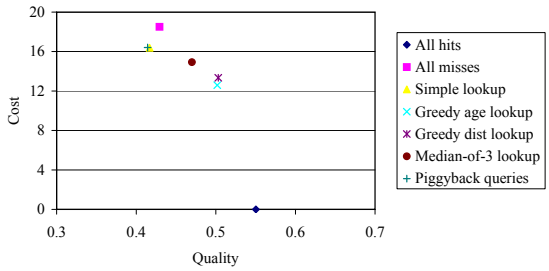

Figure 5.68. Cost vs. Quality for $A = 0.1$, $T = 9$ seconds, and Trace-driven changes over 54 nodes.

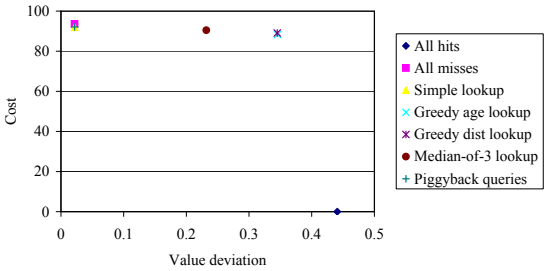

Figure 5.69. Cost vs. Value deviation for $A = 0.1$, $T = 0.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

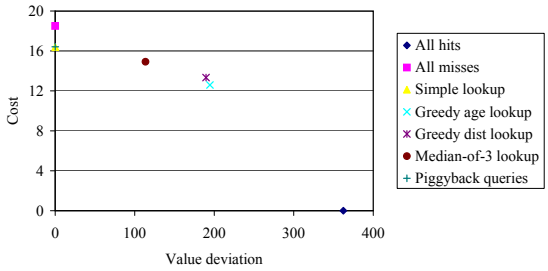

Figure 5.70. Cost vs. Value deviation for $A = 0.1$, $T = 9$ seconds, and Trace-driven changes over 54 nodes.

The most interesting results in Figures 5.67 through 5.74 are the negative results. Figure 5.67 shows that a cache can, in some operating regions, provide only marginal benefit when compared to having no cache (illustrated by the "All misses" results). Furthermore, by using an inappropriate caching and lookup policy, the presence of a cache for a sensor network can even hurt quality performance. For example, Figure 5.67 shows that the greedy lookup policies yield a cost savings of about 4% while incurring a significant penalty in quality. What renders these policies perhaps not worthwhile (because of their worse quality performance) is the significant value deviation that is introduced by a cache that is almost ineffective. These value deviations are shown in Figure 5.69 and Figure 5.70, in which the greedy policies and the median-of-3 lookup policy show significantly worse value deviation for a negligible or small amount of cost savings.

There are also interesting positive results for both sensor networks. The cost vs. quality trade-off favors using simple lookups or piggybacked queries when value deviation drives quality (e.g., when $A = 0.1$). For correlated changes, Figure 5.67 shows that using simple lookups or piggybacked queries provides about the same quality performance as all misses, while yielding a small cost savings of 1-2%. The cost savings and quality performance are both more compelling for trace-driven changes. Figure 5.68 shows that these policies provide better cost and quality performance for the 54-node network model. Specifically, the cost savings increases to a savings of 12-15% while quality also improves by about 5%.

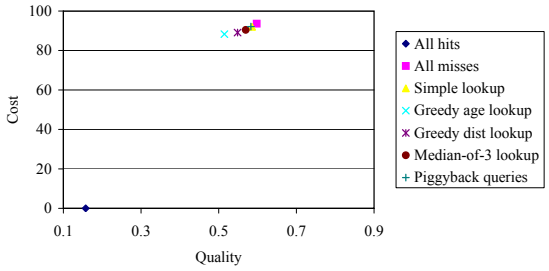

Figure 5.71. Cost vs. Quality for $A = 0.9$, $T = 0.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

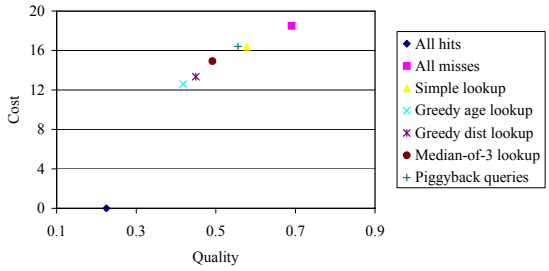

Figure 5.72. Cost vs. Quality for $A = 0.9$, $T = 9$ seconds, and Trace-driven changes over 54 nodes.

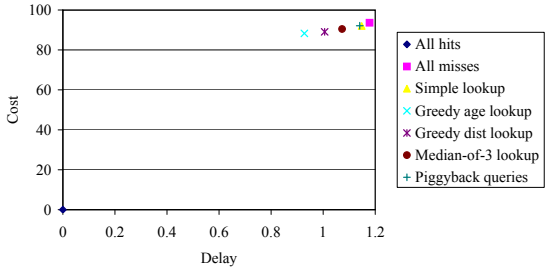

Figure 5.73. Cost vs. Delay for $A = 0.9$, $T = 0.8\overline{8}$ seconds, and Correlated changes over 1000 nodes.

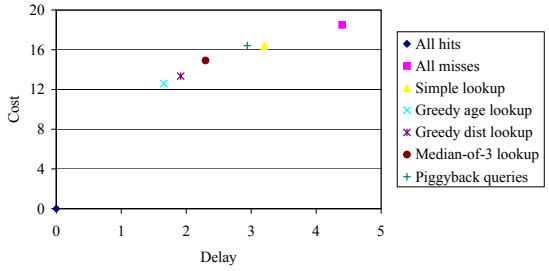

Figure 5.74. Cost vs. Delay for $A = 0.9$, $T = 9$ seconds, and Trace-driven changes over 54 nodes.

Table 5.8. Hit ratios, Costs, and Delays for $T = 0.8\overline{8}$, 90 Queries per second, and Correlated changes over 1000 locations.

Policies	Hit ratio	Cost	Delay
Simple lookup	0.02	92	1.15
Greedy age lookup	0.08	88	0.93
Greedy distance lookup	0.06	89	1.01
Median-of-3 lookup	0.05	90	1.07
Piggyback queries	0.02	92	1.14

Table 5.9. Hit ratios, Costs, and Delays for $T = 9$, 0.9 Queries per second, and Trace-driven changes over 54 locations.

Policies	Hit ratio	Cost	Delay
Simple lookup	0.12	16	3.2
Greedy age lookup	0.32	13	1.7
Greedy distance lookup	0.27	13	1.9
Median-of-3 lookup	0.20	15	2.3
Piggyback queries	0.12	16	2.9

Now consider the case where delay is more important than value deviation in Equation (5.2). For example, when $A = 0.9$, Figure 5.71 shows that the cost and quality performance of the greedy policies and the median-of-3 lookup policy is better than the performance of not using a cache. Specifically, a cost savings of up to 4% is obtained with a simultaneous improvement in quality of up to 20%. Figure 5.72 shows even greater cost savings and quality improvements for trace-driven changes to the environment. This figure shows cost savings of up to 33% and quality improvements of up to 40%.

The characteristic cost vs. delay figures for both sensor field models, Figure 5.73 and Figure 5.74, are nearly linear and have *x*-axis values that are more dispersed than those in the corresponding cost vs. quality graphs (Figures 5.71 and 5.72). However, the value deviations introduced by our age-based policies make the differences in quality smaller than the differences in delay. Thus, the quality provided by these policies is more clustered toward the "All misses" points near the top of Figures 5.71 and 5.72 than the delays for the "All misses" case in Figures 5.73 and 5.74. Figures 5.71 and 5.72 also both show that the greedy policies provide the best performance in terms of both quality and cost. However, the gains are more compelling when the underlying cache hit ratios are higher, since higher cache hit ratios mean greater cost savings. Table 5.9 shows cache hit ratios that range between 12% and 32% for our 54-node sensor network model. The relative cost and quality gains for this network model are superior to those of the 1000-node network model, for which cache hit ratios are between 2% and 8%. Table 5.8 shows the cache hit ratios for our 1000-node sensor network model.

Thus far, we have seen that the presence of a cache provides greater performance benefits when delay drives quality. Specifically when $A = 0.9$, we have observed cost savings of up to 33% and quality improvements of up to 40%. In contrast, Figures 5.67 and 5.69 make a weaker case for using a cache when its hit ratio is quite low (32% or less for all of our age-based policies, as shown in Table 5.8 and Table 5.9). Our results are also more compelling for trace-driven changes. For example, Figure 5.68 shows that simple lookups or

piggybacked queries can yield a performance gain in both cost and quality when compared to the "all misses" policy.

In this section and the previous section, we studied the impact of manipulating the cache hit ratio by varying the cache age threshold parameter, T:

- We achieve a significant cost savings (up to 95% in our configurations) by increasing T, and thus increasing the cache hit ratio.
- We still achieve a small cost savings (from 2% to 33%) when T is small, but greater than zero.
- In both of these cases, and for both of our sensor field models, these cost savings occur with a simultaneous quality improvement when delay is more important to quality than value deviation. However, when value deviation is more important to quality, we observe quality that is usually worse (by up to 50% in our configurations), and sometimes better (by up to 5%).

5.8 Survey of Related Work

Trading communication cost for data quality has been investigated by others [101, 100, 102, 99, 136]. Work at Stanford [101, 100, 102, 99] and UIC [136] both explore this idea, but in ways quite different from our research. The UIC work investigates the trade-off in wireless sensor networks from a theoretical point of view. However, their proposed algorithms focus on caching within the sensor field itself rather than at the sensor field gateway. The Stanford work focuses on a complementary problem: Supporting precision bounds on a set of queries over streamed data. These bounds are provided on a best-effort basis, and their cost functions are too simple to translate their results to wireless sensor networks. Specifically, their work assumes unit cost to deliver each data item to an application.

Data quality has many definitions in the UIC and Stanford work. These papers consider four different methods for quantifying sensor data quality:

1. Change in value;
2. Rate of change in value;
3. Version of a data item; and
4. Rate of update of a data item.

We adopt the first of these alternatives and use the same term to describe the accuracy component of quality, namely value deviation.

Work at Virginia [12] and work at Berkeley / Carnegie Mellon / Intel [38] has proposed different distributed caching schemes for wireless and wired sensor networks, respectively. However, these results consider only the benefits of caching sensor data without considering its cost.

From the work cited above, there are three important references that are related to this thesis that are worth describing in detail:

- C. Olston, J. Jiang, and J. Widom. *Adaptive Filters for Continuous Queries over Distributed Data Streams* [99];
- C. Olston and J. Widom. *Best-Effort Cache Synchronization with Source Cooperation* [102]; and
- A. Deshpande, S. Nath, P. Gibbons, S. Seshan. *Cache-and-Query for Wide Area Sensor Databases* [38].

We discuss these papers in this order.

Olston et al. [99], *Adaptive Filters for Continuous Queries over Distributed Data Streams.*

This paper presents an adaptive filtering algorithm where data sources periodically make their associated filter less restrictive, making their bound on the source more precise. At the same time a central query processing engine selectively chooses sources that

may use more restrictive filters, making their bound on the source less precise, without violating the precision bounds for any of the outstanding queries. The sources are assumed to be sufficiently homogeneous that for the purposes of adaptation a single adjustment period and a single shrink fraction work well for all sources.

The proposed algorithm was evaluated for a network monitoring application on a small LAN (10 hosts). Communication was assumed to have uniform unit cost. The performance metric of interest was communication cost per second of running the algorithm for a given answer precision.

Interesting aspects of the results in this paper were:

- The adaptive filtering algorithm always outperforms uniform bound allocation in the results shown. This performance difference is significant (50-90%) for multiple queries and in wide-area environments.
- Uniform bound allocation always outperforms using no filtering.
- As query precision constraints become tighter (using filters that are less restrictive), the behavior and cost of the adaptive filtering algorithm, uniform bound allocation, and using no filtering converge.
- Finally, inconsistency can occur as source updates are delayed or lost en route to the central query processing engine. This was investigated only for a single query in the LAN environment. It was show that the data at the central query processing engine was consistent 99.997% of the time.

Olston et al. [102], *Best-Effort Cache Synchronization with Source Cooperation.*

This paper presents an adaptive scheduling algorithm where data sources periodically decrease the bandwidth used for refreshing a master cache. At the same time the cache selectively chooses data sources that may use more bandwidth for refreshing the cache. Local scheduling of refresh messages is implemented as a priority queue wherein the priority of

refresh messages is a function of the divergence of the data source value from the cached value, and a system-dependent weight (e.g., importance $\times$ popularity). The sources are assumed to be sufficiently homogeneous that for the purposes of adaptation a single factor for increasing the bandwidth and a single factor for decreasing the bandwidth work well for all sources.

The proposed algorithm was evaluated for sources that increase or decrease in value by one unit according to a Poisson process. Available bandwidth for refresh messages was either fixed or varied according to a randomly selected sine wave. Message transmission was assumed to consume a uniform unit of bandwidth. The performance metric of interest was the divergence between the cache and the data sources (i.e., a measure of the cache consistency). The results show that as the average theoretically attainable divergence increases (due to low bandwidth and/or many rapidly diverging objects), the adaptive algorithm attains divergence nearly as good as the ideal case (within 30%). In the less interesting case when divergence is small, the magnitude of the divergence achieved by the proposed algorithm and that of the idealized case is within a factor of four.

The adaptive algorithm was also evaluated in trace-driven simulation using real-world data from 40 weather buoys deployed in January 2000 by the Pacific Marine Environmental Laboratory. When the available bandwidth was either very small or very large, the adaptive algorithm closely matched the ideal case. At moderate bandwidths (10's of packets/second), the adaptive algorithm yielded average divergence that was within a factor of two of the ideal case.

Deshpande et al. [38], *Cache-and-Query for Wide Area Sensor Databases.*

The IrisNet system maintains the logical view of the data as a single XML document, while physically the data is distributed across any number of host nodes. For scalability, sensor data is stored close to the sensors, but can be cached elsewhere as dictated by the queries. The contributions of this paper are:

- It presents a scalable system for executing XPATH queries on wide-area sensor databases. The system uses a logical hierarchy of sites dictated by the XML document; these logical sites are mapped to a smaller collection of physical sites, as dictated by the system administrator or the queries themselves.

- It proposes a technique for self-starting distributed queries, which jump directly to the lowest common ancestor (LCA) of the query result, by using DNS site names (made up of query node IDs) extracted from the query itself. DNS lookups are then used to determine the IP addresses of the desired sites, and DNS entries are updated when document nodes are remapped.

- It shows how general XPATH queries can be evaluated on a single XML document when the document itself is fragmented across machines, and the fragmentation is constantly changing. It also proposes a novel query-evaluate-gather (QEG) technique for detecting (1) which data in the database fragment at a site is part of the query result, and (2) how to gather the missing parts. Because XPATH is insufficiently powerful, the authors actually use XSLT to query the database, evaluate what is there, and send subqueries to gather the missing parts of the answer.

- It presents a scheme for caching query results at sites as dictated by the queries. Owned data and cached data are stored in the same site database, with different tags, unifying the query processing at a site.

- Finally, this paper presents experimental results demonstrating the effectiveness of these techniques in dramatically increasing update and query throughputs and decreasing query response times in wide area sensor databases. The results show that distributed querying (using DNS to self-start) can improve performance by up to a factor of three over using centralized queries.

5.9 Conclusions

We explored the benefits and costs of caching for sensor network-based applications in this Chapter. In such systems, one or more driving applications retrieve sensor data that is disseminated from sensor fields. Issues of scale arise from the need to satisfy a variable and potentially large number of concurrent queries for sensor data.

Our work examined the cost and quality performance of several proposed approaches to querying for, and then caching data in a sensor field data server. These approaches are designed to be general since they make no assumptions about whether the higher layers in a sensor network architecture use a structured or unstructured data model.

We showed that for some application quality requirements (i.e., when system delay drives quality), policies that emulate cache hits by computing and returning approximate values for sensor data yield a simultaneous quality improvement and cost savings. This is because when system delay is sufficiently important, the benefit to both cost and quality achieved by using approximate values outweighs the negative impact on quality due to the approximation. In contrast, when accuracy (i.e., value deviation) drives quality, a linear trade-off between cost and quality emerged:

- We obtained the best cost performance by implementing policies that approximate sensor readings by using cached values from nearby locations.
- The best quality performance was achieved by policies that always query and cache the sensor field location specified in the user query.

We introduced an analytic model for changing the sensor field data values that captures temporal correlation, spatial correlation, and allows different rates at which the values can change. We compared our results that used this correlated model for sensor data values with results that used real-world trace data, namely light intensity readings from an Intel Berkeley lab trace described in [37] via simulation.

We identified a class of caching and lookup policies for which the sensor field query rate is bounded when servicing an arbitrary workload of user queries for sensor data. This upper bound is achieved by having multiple user queries share the cost of a sensor field query. Furthermore, this bound is a simple function of the number of locations in the sensor field and an age threshold parameter that is associated with our caching and lookup policies.

We also studied the performance impact of manipulating the sensor data cache hit ratio by varying this cache age threshold parameter (T):

- We achieved a significant cost savings (up to 95% in our configurations) by increasing T, and thus increasing the cache hit ratio.
- We still achieved a small cost savings (from 2% to 33%) when T was small, but greater than zero.
- When T was too small and value deviation was most important to quality, we observed quality that was usually worse (by up to 50%), and sometimes better (by up to 5%).

Finally, we showed that our results were robust with respect to the manner in which the environment being monitored changed. Specifically, for our sensor network model with correlated changes to the environment, we varied the rate at which the environment changed by two orders of magnitude. When we used the real-world light intensity data, we varied the rate at which the sensor field is queried, also by two orders of magnitude.

CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Summary of Scalable Networked Servers Research

As Internet technology continues to evolve, network-based services will expand in number and in functionality. For example, web content is increasingly being augmented with more bandwidth-intensive audio and video. Similarly, distributed sensing applications are becoming more data-intensive. For example, the current generation of real-time weather data feeds (e.g., from NCAR) are improving by an order of magnitude in sampling frequency and resolution. In combination, these factors will dramatically increase the performance requirements for network-based content and data servers. This first part of this thesis evaluates the suitability of *connection-level parallelism* for supporting emerging applications that will run on these servers.

In Chapter 2 we described our own implementation of connection-level parallelism (CLP), and experimentally determined how throughput scales with the number of processors, and how throughput changes as the number of connections increases. We also investigated three other issues:

- The effect of the granularity of parallelism on performance.
- The effect of the transport protocol on performance.
- The impact of processing packets in bursts.

We found that connection-level parallel protocol stacks scale well with the number of processors for UDP and TCP, performing both send-side and receive-side communication.

Furthermore, the throughput delivered is, for the most part, sustained as the number of connections increases. We also show that for moderate to large numbers of connections, the number of threads in the system and the number of connections assigned to each thread are the key factors in determining performance on a multiprocessor.

Our results showed that thread per connection parallelism yields the best aggregate throughput. However, thread per connection may not be feasible if the number of connections exceeds the number of threads that can be reasonably supported by a particular multiprocessor. In this case, the best throughput is obtained by using virtual processor per connection and maximizing the number of threads in the system.

We also showed that if an application needs maximal throughput from a multiprocessor protocol stack, amortizing per-packet costs over multiple packets is worthwhile.

In Chapter 3 we studied how aggregate throughput is distributed across connections in our implementation of connection-level parallelism (CLP). We find that in many cases throughput is *not* distributed fairly among connections, or threads.

We quantified whether our protocol implementations were distributing aggregate throughput fairly by measuring the throughput seen by each connection and thread and computing percentiles of the throughputs, between the maximum and minimum values. We presented results for both UDP and TCP, servicing both a moderate and large number of connections. Our results show that matching the number of threads to the number of physical processors, while scheduling connections assigned to each thread (or virtual processor) in a round-robin manner, yields the best fairness behavior. However, an improvement in fairness may come at the cost of aggregate throughput.

We also found that there can be significant differences in the per-connection (as well as per-thread) throughput. We demonstrated that scheduling and memory reference behavior both play a role in the unfair distribution of aggregate throughput. Furthermore, fairness problems tend to increase as more threads are used in the protocol stack, and this increase is due to greater differences in memory reference behavior seen by the threads.

In Chapter 4 we assessed the suitability of connection-level parallelism (CLP) for supporting continuous media applications. In such applications, the server protocol stack is one link in a chain of subsystems which must provide throughput guarantees to connections. We consider two types of throughput guarantees. The first is for applications where the playout rate is required to exactly match an original fixed recording rate (i.e., CBR applications). In this case, the protocol stack and operating system (as well as other system components) should provide service where throughput delivered to connections is constant and guaranteed. We refer to this service class as service with hard throughput guarantees. The second is for applications that can adjust their playout rate as they detect that the available bandwidth being delivered to connections changes. In this case the protocol stack and operating system need to distribute the available bandwidth equitably among all connections. We refer to this service class as service with soft throughput guarantees.

We presented results for a new technique, called *throughput-based scheduling*, that provides a mechanism whereby threads monitor and report their per-connection throughput to the scheduler. To support soft throughput guarantees, the scheduler balances load by reallocating work (in this case connections) among threads. We refer to this technique as *soft throughput-based scheduling*. To support hard throughput guarantees, the scheduler schedules threads according to their "utilization" (i.e., how much of their current capacity is consumed by the throughput they have been assigned to deliver). We refer to this technique as *hard throughput-based scheduling*.

Using UDP as the transport protocol, we showed that throughput-based scheduling is suitable for supporting continuous media applications. Soft throughput-based scheduling performed best with fewer threads in the system and without competing scheduling mechanisms (i.e., IRIX's cache affinity-based scheduling). Hard throughput-based scheduling also performed best with fewer threads performing protocol processing. Furthermore, when the number of threads used is the same as the number of processors, the throughputs delivered are accurate over both short (i.e., down to 100 milliseconds) and long time scales.

6.2 Summary of Wireless Sensor Networks Research

Over the next several years wireless sensor networks will enable many new data-intensive sensing applications, ranging from environmental and infrastructure monitoring to commercial and industrial sensing. Networks of small, possibly microscopic sensors embedded in the fabric of our surroundings: in buildings, warehouses, and machinery, and even on people, will drastically enhance our ability to monitor and control our physical world. However, realization of wireless sensor networks requires practical satisfaction of a number of real-world constraints introduced by factors such as scalability, reliability, cost, hardware, topology change, environment, and perhaps most importantly, power consumption. Examples of wireless embedded sensor network-based applications include urban structure monitoring, contaminant transport monitoring, habitat monitoring, and military surveillance.

In the second part of this thesis, we explored the benefits and costs of caching for sensor network-based applications. In such systems, one or more driving applications retrieve sensor data that is disseminated from sensor fields. In Section 1.3.1, we saw how caching data in the sensor field gateway has the potential to help application performance. In Sections 5.1 through 5.3, we formulated models for a sensor network, its sensor fields, and its application workload. These models introduce an analytic model for changing the sensor field data values that captures temporal correlation, spatial correlation, and allows different rates at which the sensor field values can change. We compared our results that use this correlated model for sensor readings with results that use real-world trace data from the Intel Berkeley Research Lab via simulation.

Sections 5.4 through 5.7 examined how well different data caching approaches perform in terms of both *cost* and *quality*. We presented results for seven proposed data caching approaches:

- *All hits*;
- *All misses*;

- *Simple lookups*;
- *Piggybacked queries*;
- *Greedy age lookups*;
- *Greedy distance lookups*; and
- *Median-of-3 lookups*.

These approaches are designed to be general since they make no assumptions about whether the higher layers in a sensor network architecture use a structured or unstructured data model.

We demonstrated that different approaches should be used depending on the application requirements and the characteristics of the environment and sensor network. The results in Sections 5.4 through 5.7 can be summarized as follows:

- *Necessary conditions to avoid a cost vs. quality trade-off.* We measured the benefit and cost of our seven proposed caching and lookup policies as a function of different application quality requirements. We showed that for some quality requirements (i.e., when *system delay* drives quality), policies that prevent cache misses by computing and returning approximate values for sensor data yield a simultaneous quality improvement and cost savings. This win-win is because when system delay is sufficiently important, the benefit to both cost and quality achieved by using approximate values outweighs the negative impact on quality due to the approximation. In contrast, when accuracy (i.e., *value deviation*) drives quality, a trade-off between cost and quality emerged.

- *Form and magnitude of the cost vs. quality trade-off.* Five of our caching and lookup policies aged and then deleted cache entries uniformly based on an age threshold parameter, T. We observed that in many system configurations these five policies exposed a linear cost vs. quality trade-off:

  - We obtained the best cost performance by implementing policies that approximate sensor readings by using cached values from nearby locations.
  - The best quality performance was achieved by policies that always query and cache the sensor field location specified in the user query.

  When this linear trade-off was present, we found that the underlying cost vs. value deviation and/or cost vs. system delay functions were also linear.

- *Bounded cost for specific caching and lookup policies.* For applications that require bounded resource consumption, we identified a class of caching and lookup policies for which the sensor field query rate is bounded when servicing an arbitrary workload of user queries for sensor data. This upper bound is achieved by having multiple user queries share the cost of a sensor field query. Furthermore, this bound is a simple function of the number of locations in the sensor field and T.

- *Impact of the manner in which the environment changes on cost and quality performance.* For our sensor network model with correlated changes to the environment, we varied the rate at which the environment changed by two orders of magnitude. When we used the real-world trace data, we varied the rate at which the sensor field was queried, also by two orders of magnitude. For both methods for changing the environment, our results showed that the magnitude of the cost and quality performance changed, however, the trends for the most part remained the same. Specifically, relative performance differences between policies varied within our results, however, the policies that provided the best cost performance (i.e., the greedy policies) remained the same. Similarly, the policies that provided the best (or near-best) quality performance were the same, namely simple lookups and piggybacked queries.

- *Effect of changing the age threshold parameter for cache entries on performance.* We varied the cache entry age threshold (T) as a way to manipulate the cache hit ratio:

– We achieved a significant cost savings (up to 95% in our configurations) by increasing T, and thus increasing the cache hit ratio.

– We still achieved a small cost savings (from 2% to 33%) when T was small, but greater than zero.

We saw that for high cache hit ratios (i.e., 88% and higher) the performance of all of our five age-based policies converged toward the "All hits" baseline performance. In contrast, for low cache hit rates (32% and lower) the performance of some (but not all) of our policies was clustered near the (cost, quality) point associated with the "All misses" baseline. When this was true, we also observed an interesting negative result: Unless cost is much more important than quality, some of the caching and lookup policies exhibit worse performance than having all misses. However, a cache with a positive hit ratio always provides some cost savings.

6.3 Suggestions for Future Work

There are many interesting directions for future work that would build on the foundation laid in this thesis.

With respect to our networked servers contributions, it would be interesting to investigate additional support for continuous media. One direction to proceed would be to investigate techniques for supporting heterogeneous qualities of service (recall that our throughput-based scheduling results focused on connections with homogeneous requirements). Furthermore, it would be a challenge to integrate a policy that provides throughput guarantees with another policy for performing call admission. This line of research would be a natural extension to our results for hard throughput-based scheduling, but could also complement soft throughput-based scheduling.

One advantage of connection-level parallelism not highlighted in this thesis is its portability and applicability to other architectures. This is in contrast to other forms of parallelism for network protocols that require more synchronization to process each packet. An-

other direction for future work therefore would be to study the research issues in evaluating connection-level parallelism on platforms such as distributed shared-memory machines and clusters of workstations (or PC's).

With respect to our sensor networks research, it would be interesting to explore alternative models for the query workload and how the environment changes. For a large set of end users, it is reasonable to assume that the workload arrival process will be a combination of a deterministic process and a Poisson process. This assumption, however, is context dependent. In a scenario when one or more significant events are occurring, this workload model would likely be inadequate. It would be important to experiment with models for both the environment and the workload that capture the dynamic behavior of a sensor network in the presence of a significant event (or events).

It would also be interesting to explore techniques for distributed caching within the sensor field itself. These techniques could work in cooperation with (or independently from) the caching approaches described in this thesis. Here is a list of possible issues that would need to be addressed in distributed caching for sensor networks:

- When to store data in a received message;
- Which data to store once it is extracted from a received message and aggregated with other data that has been received or sent;
- When to relay, relay and update, relay and piggyback, or just discard a received message;
- When to send an synchronous or asynchronous message; and
- To which node(s) or location(s) should an outbound message be addressed.

Given the dependencies in this list of possible issues, it is likely that a distributed caching approach would need to be aware of the routing policy for the sensor field, and the transmission scheduling policy at sensor nodes.

BIBLIOGRAPHY

[1] Allison, Brian. DEC 7000/10000 Model 600 AXP multiprocessor server. In *Proceedings IEEE COMPCON* (San Francisco, CA, Feb. 1993), pp. 456–464.

[2] Anderson, Darrell, Chase, Jeff, and Vahdat, Amin. Interposed request routing for scalable network storage. *ACM Transactions on Computer Systems 20*, 1 (Feb. 2002).

[3] Andresen, D., and Yang, T. Multiprocessor scheduling with client resources to improve the response time of WWW applications. In *SIGARCH International Conference on Supercomputing* (Vienna, Austria, July 1997).

[4] Aron, Mohit, Druschel, Peter, and Zwaenepoel, Willy. Efficient support for p-http in cluster-based web servers. In *USENIX Annual Technical Conference* (Monterey, California, June 1999).

[5] Aron, Mohit, Druschel, Peter, and Zwaenepoel, Willy. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (Santa Clara, California, June 2000).

[6] Aron, Mohit, Iyer, Sitaram, and Druschel, Peter. A resource management framework for predictable quality of service in web servers. Department of Computer Science, Rice University, 2003.

[7] Asthana, Abhaya, Delph, Catherine, Jagadish, H. V., and Kryzyzanowski, Paul. Towards a gigabit IP router. *Journal of High Speed Networks 1* (1992), 218–288.

[8] Aversa, Luis, and Bestavros, Azer. Load balancing a cluster of web servers using distributed packet rewriting. In *IEEE International Performance, Computing, and Communication Conference* (Phoenix, Arizona, Feb. 2000).

[9] Bailey, Mary L., Gopal, Burra, Pagels, Michael A., Peterson, Larry L., and Sarkar, Prasenjit. PathFinder: a pattern-based packet classifier. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, Nov. 1994), pp. 115–123.

[10] Barton, James M., and Bitar, Nawaf. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing* (Santa Barbara, CA, Apr. 1995), pp. 24–40.

[11] Bestavros, A., Crovella, M. E., Liu, J., and Martin, D. Distributed packet rewriting and its application to scalable server architectures. In *International Conference on Network Protocols (ICNP)* (Oct. 1998).

[12] Bhattacharya, Sagnik, Kim, Hyung, Prabh, Shashi, and Abdelzaher, Tarek. Energy-conserving data placement and asynchronous multicast in wireless sensor networks. In *International Conference On Mobile Systems, Applications And Services* (San Francisco, CA, USA, 2003), pp. 173–185.

[13] Bishop, Christopher M. *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, 1995.

[14] Björkman, Mats, and Gunningberg, Per. Locking effects in multiprocessor implementations of protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols* (San Francisco, CA, Sept. 1993), ACM, pp. 74–83.

[15] Boulis, Athanassios, Ganeriwal, Saurabh, and Srivastava, Mani B. Aggregation in sensor networks: an energy-accuracy tradeoff. In *IEEE Workshop on Sensor Network Protocols and Applications (SNPA)* (Piscataway, NJ, USA, May 2003), IEEE Press.

[16] Bourke, Tony. *Server Load Balancing*. O'Reilly & Associates, Sebastopol, California, 2001.

[17] Boykin, Joseph, and Langerman, Alan. The parallelization of Mach/4.3BSD: Design philosophy and performance analysis. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS I)* (Ft. Lauderdale, Fl., Oct. 1989), pp. 105–125.

[18] Braun, Torsten, and Schmidt, Claudia. Implementation of a parallel transport subsystem on a multiprocessor architecture. In *2nd International Symposium on High-Performance Distributed Computing* (Spokane, Washington, July 1993).

[19] Braun, Torsten, and Zitterbart, Martina. High performance internetworking protocol. In *15th IEEE Conference on Local Computer Networks* (Minneapolis, Minnesota, Sept. 1990).

[20] Braun, Torsten, and Zitterbart, Martina. Parallel transport system design. *Fourth IFIP WG 6.4 Conference on High Performance Networking* (Dec. 1992), 397–412.

[21] Bridle, John S. Probabilistic interpretation of feed-forward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing: Algorithms, Architecture and Applications 6* (1990).

[22] Bunt, Richard B., Eager, Derek L., Oster, Gregory M., and Williamson, Carey L. Achieving load balance and effective caching in clustered web servers. In *Fourth International Web Caching Workshop* (San Diego, California, Apr. 1999), pp. 159–169.

[23] Burroughs Corporation. *Burroughs B5500 Information Processing Systems Reference Manual*. Burroughs Corporation, Detroit, Michigan, 1964.

[24] Cekleov, Michel et al. SPARCCenter 2000: Multiprocessing for the 90's! In *Proceedings IEEE COMPCON* (San Francisco, CA, Feb. 1993), pp. 345–353.

[25] Chandra, Abhishek, Gong, Weibo, and Shenoy, Prashant. An online optimization-based technique for dynamic resource allocation in GPS servers. Tech. Rep. UM-CS-2002-030, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, July 2002.

[26] Chandra, Abhishek, Gong, Weibo, and Shenoy, Prashant. Dynamic resource allocation for shared data centers using online measurements. In *ACM/IEEE International Workshop on Quality of Service (IWQoS)* (Monterey, California, June 2003).

[27] Chandra, Abhishek, Goyal, Pawan, and Shenoy, Prashant. Quantifying the benefits of resource multiplexing in on-demand data centers. In *ACM Workshop on Algorithms and Architectures for Self-Managing Systems* (San Diego, California, June 2003).

[28] Chen, Benjie, and Morris, Robert. Flexible control of parallelism in a multiprocessor PC router. In *USENIX Annual Technical Conference* (Boston, Massachusetts, June 2001).

[29] Cherkasova, L., DeSouza, M., and Ponnekanti, S. Performance analysis of a content-aware load balancing strategy FLEX: Two case studies. In *Thirty-Fourth Hawaii International Conference on System Sciences (HICSS-34)* (Jan. 2001).

[30] Cherkasova, L., and Karlsson, M. Scalable web server cluster design with WARD. In *Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)* (San Jose, California, June 2001), pp. 212–221.

[31] Cherkasova, L., and Phaal, P. Session based admission control: a mechanism for peak load management of commercial web sites. *IEEE Transactions on Computers 51*, 6 (June 2002).

[32] Claffy, Kimberly C., Braun, Hans-Werner, and Polyzos, George C. Internet traffic flow profiling. Tech. Rep. UCSD Report CS93-328, SDSC Report GA-A21526, Department of Computer Science, University of California, San Diego, March 1994.

[33] Cmelik, Robert, and Keppel, David. Shade: A fast instruction set simulator for execution profiling. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (Nashville, Tenn., May 1994), pp. 128–137.

[34] Damani, O. P., Chung, P. E., Huang, Y., Kintala, C., and Wang, Y.-M. ONE-IP: techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems 29*, 8–13 (1997), 1019–1027.

[35] Demers, Alan, Gehrke, Johannes, Rajaraman, Rajmohan, Trigoni, Niki, and Yao, Yong. The Cougar project: a work-in-progress report. *ACM SIGMOD Record 32,* 4 (Dec. 2003), 53–59.

[36] Demers, Alan, Keshav, Srinivasan, and Shenker, Scott. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience 1,* 1 (Jan. 1990), 3–26.

[37] Deshpande, Amol, Guestrin, Carlos, Madden, Samuel R., Hellerstein, Joseph M., and Hong, Wei. Model-driven data acquisition in sensor networks. In *International Conference on Very Large Data Bases (VLDB)* (Toronto, Aug. 2004), Morgan Kaufmann, pp. 588–599.

[38] Deshpande, Amol, Nath, Suman, Gibbons, Phillip B., and Seshan, Srinivasan. Cache-and-query for wide area sensor databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), ACM Press, pp. 503–514.

[39] Dias, D.M., Kish, W., Mukherjee, R., and Tewari, R. A scalable and highly available web server. In *Forty-first IEEE Computer Society International Conference* (San Jose, California, Feb. 1996).

[40] Diot, Christophe, and Dang, Michel Ng X. Using transputer in the design of high performance architectures dedicated to the implementation of OSI transport protocols. In *Transputer Research and Applications 3* (Sunnyvale, California, Apr. 1990), pp. 17–25.

[41] Dove, Ken. A high capacity TCP/IP in parallel streams. In *Proceedings of the United Kingdom UNIX Users Group* (Jan. 1990).

[42] Druschel, Peter, Peterson, Larry, and Davie, Bruce. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM Symposium on Communications Architectures and Protocols* (London, England, Aug. 1994), ACM, pp. 2–13.

[43] Druschel, Peter, and Peterson, Larry L. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, NC, Dec. 1993), pp. 189–202.

[44] Elnikety, Sameh, Nahum, Erich M., Tracey, John, and Zwaenepoel, Willy. A method for transparent admission control and request scheduling in dynamic e-commerce web sites. In *International Conference on the World Wide Web* (New York, New York, USA, May 2004), pp. 276–286.

[45] Eykholt, J. R., Kleiman, S. R., Barton, S., Faulkner, R., Stein, D., Smith, M., Shivalingiah, A., Voll, J., Weeks, M., and Williams, D. Beyond multiprocessing: Multithreading the SunOS kernel. In *USENIX Summer 1992* (San Antonio, Texas, June 1992).

[46] Fuller, S. H., and Harbison, S. P. The c.mmp multiprocessor. Technical Report CMU-CS-78-146, Department of Computer Science, Carnegie-Mellon University, Oct. 1978.

[47] Gallatin, Andrew, Chase, Jeff, and Yocum, Ken. Trapeze/ip: Tcp/ip at near-gigabit speeds. In *USENIX Annual Technical Conference (Freenix Track)* (Monterey, California, June 1999).

[48] Galles, Mile, and Williams, Eric. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Tech. rep., Silicon Graphics Inc., Mt. View, CA, May 1994.

[49] Garg, Arun. Parallel STREAMS: a multi-processor implementation. In *Proceedings of the Winter 1990 USENIX Conference* (Washington, D.C., Jan. 1990).

[50] Giarrizzo, Dario, Kaiserswerth, Matthias, Wicki, Thomas, and Williamson, Robin C. High-speed parallel protocol implementation. *First IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (May 1989), 165–180.

[51] Gibbons, Phillip B., Karp, Brad, Ke, Yan, Nath, Suman, and Seshan, Srinivasan. IrisNet: An architecture for a world-wide sensor web. *IEEE Pervasive Computing 2*, 4 (Oct. 2003), 22–33.

[52] Goldberg, A. J., and Hennessy, J. MTOOL: A method for isolating memory bottlenecks in shared memory multiprocessing programs. In *International Conference on Parallel Processing* (Aug. 1991), pp. 251–257.

[53] Goldberg, Murray W., Neufeld, Gerald W., and Ito, Mabo R. A parallel approach to OSI connection-oriented protocols. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (May 1992), 219–232.

[54] Greenberg, Albert G., and Madras, Neal. Comparison of a fair queueing discipline to processor sharing. In *Performance '90: 14th International Symposium on Computer Performance Modelling, Measurement and Evaluation* (Edinburgh, Scotland, Sept. 1990), P. J. B. King, I. Mitrani, and R. J. Pooley, Eds., IFIP WG7.3, North-Holland, Amsterdam, Holland, pp. 193–207.

[55] Haas, Zygmunt. A communication architecture for high-speed networking. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (San Francisco, CA, June 1990), pp. 433–441.

[56] Han, Jiawei, and Kamber, Micheline. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, USA, 2000.

[57] Hansen, Jorgen Svaerke, and Jul, Eric. A scheduling scheme for network saturated nt multiprocessors. In *USENIX Windows NT Workshop* (Seattle, Washington, Aug. 1997).

[58] Heavens, Ian. Experiences in parallelisation of streams-based communications drivers. *OpenForum Conference on Distributed Systems* (Nov. 1992).

[59] Howard, John H., Kazar, Michael L., Menees, Sherri G., Nichols, David A., Satyanarayanan, Mahadev, Sidebotham, Robert N., and West, Michael J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[60] Hu, James C., Pyaraliy, Irfan, and Schmidt, Douglas C. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Global Internet mini-conference held in conjunction with IEEE GLOBECOM* (Phoenix, Arizona, Nov. 1997).

[61] Hunt, G. D., Goldszmidt, G., King, R., and Mukherjee, R. Network dispatcher: A connection router for scalable internet services. *Computer Networks and ISDN Systems 30*, 7 (Apr. 1998), 347–357.

[62] Hunt, Guerney, Nahum, Erich, and Tracey, John. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, May 1997.

[63] Hutchinson, Norman C., and Peterson, Larry L. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

[64] Intanagonwiwat, Chalermek, Govindan, Ramesh, Estrin, Deborah, Heidemann, John, and Silva, Fabio. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking 11*, 1 (Feb. 2003), 2–16.

[65] Ito, Mabo, Takeuchi, Len, and Neufeld, Gerald. A multiprocessing approach for meeting the processing requirements for OSI. *IEEE Journal on Selected Areas in Communications SAC-11*, 2 (Feb. 1993), 220–227.

[66] Jacobson, Van, Braden, Robert, and Borman, David. TCP extensions for high performance. In *Network Information Center RFC 1323* (Menlo Park, CA, May 1992), SRI International, pp. 1–37.

[67] Jain, Niraj, Schwartz, Mischa, and Bashkow, Theordore R. Transport protocol processing at Gbps rates. In *SIGCOMM Symposium on Communications Architectures and Protocols* (Philadelphia, PA, Sept. 1990), ACM, pp. 188–199.

[68] Jamieson, Kyle, Balakrishnan, Hari, and Tay, Y. C. Sift: a MAC protocol for event-driven wireless sensor networks. Technical Report 894, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, May 2003.

[69] Jensen, M. N., and Skov, M. Multi-processor based high-speed communication systems. In *6th European Fibre Optic Communications and Local Area Networks Exposition* (Amsterdam, Netherlands, June 1988), pp. 430–434.

[70] Jindal, Apoorva, and Psounis, Konstantinos. Modeling spatially-correlated sensor network data. In *IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON)* (Piscataway, NJ, USA, Oct. 2004), IEEE Press, pp. 162–171.

[71] Kaiserwerth, Matthias. The parallel protocol engine. *IEEE/ACM Transactions on Networking 1*, 6 (Dec. 1993), 650–663.

[72] Kay, Jonathan, and Pasquale, Joseph. Measurement, analysis, and improvement of UDP/IP throughput for the DECStation 5000. In *Proceedings of the Winter 1993 USENIX Conference* (San Diego, CA, 1993), pp. 249–258.

[73] Kim, Hyong-youb, Pai, Vijay S., and Rixner, Scott. Exploiting task-level concurrency in a programmable network interface. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, June 2003).

[74] Kleinman, Steve et al. Symmetric multiprocessing in solaris 2.0. In *IEEE Spring COMPCON* (1992).

[75] Koufopavlou, O. G., Tantawy, A. N., and Zitterbart, Martina. Analysis of TCP/IP for high performance parallel implementations. In *Proceedings of the 17th Conference on Local Computer Networks* (Minneapolis, Minnesota, Sept. 1992), pp. 576–585.

[76] Koufopavlou, Odysseas G., and Zitterbart, Martina. Parallel TCP for high performance communication subsystems. In *Proceedings of the Global Telecommunications Conference (GLOBECOM)* (1992), pp. 1395–1399.

[77] La Porta, Thomas F., and Schwartz, Mischa. Performance analysis of MSP: A feature-rich high-speed transport protocol. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (San Francisco, CA, Mar. 1993), IEEE, pp. 513–520.

[78] La Porta, Tom F., and Schwartz, Mischa. A high-speed protocol parallel implementation: Design and analysis. *Fourth IFIP TC6.1/WG6.4 International Conference on High Performance Networking* (Dec. 1992), 135–150.

[79] Lindgren, Bert, Krupczak, Bobby, Ammar, Mostafa, and Schwan, Karsten. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols* (San Francisco, CA, Mar. 1993), pp. 234–242.

[80] Lu, Chenyang, Blum, Brian M., Abdelzaher, Tarek F., Stankovic, John A., and He, Tian. RAP: a real-time communication architecture for large-scale wireless sensor networks. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (Washington, DC, USA, Sept. 2002), IEEE Computer Society, pp. 55–66.

[81] Luotonen, Ari, and Altis, Kevin. World-wide web proxies. In *Selected papers of the First Conference on the World-Wide Web* (Amsterdam, The Netherlands, May 1994), Elsevier Science Publishers B.V., pp. 147–154.

[82] Madden, Samuel, Franklin, Michael J., Hellerstein, Joseph M., and Hong, Wei. The design of an acquisitional query processor for sensor networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), ACM Press, pp. 491–502.

[83] Maeda, Chris, and Bershad, Brian N. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Dec. 1993), pp. 244–255.

[84] Maly, K., Khanna, S., Mukkamala, R., Overstreet, C. M., Yerraballi, R., Foudriat, E. C., and Madan, B. Parallel TCP/IP for multiprocessor workstations. *Fourth IFIP TC6.1/WG6.4 International Conference on High Performance Networking* (Dec. 1992), 135–150.

[85] Manoel, Edson, Carlane, Chrisanthy, Ferreira, Luis, Hill, Steve, Leitko, David, and Zutenis, Peter. Linux clustering with CSM and GPFS. Tech. rep., IBM Redbook Series, Dec. 2002.

[86] Marimuthu, Peram, Viniotis, Ioannis, and Sheu, Tsang-Ling. A parallel router architecture for high speed LAN internetworking. In *17th IEEE Conference on Local Computer Networks* (Minneapolis, Minnesota, Sept. 1993), IEEE, pp. 335–344.

[87] Martonosi, Margaret, Gupta, Anoop, and Anderson, Thomas. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (Newport, RI, June 1992), pp. 1–12.

[88] McCanne, Steve, and Jacobson, Van. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference* (Jan. 1993), pp. 259–269.

[89] McCutcheon, Mark J., Ito, Mabo Robert, and Neufeld, Gerald W. Interfacing a multiprocessor protocol engine to an ATM network. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (May 1993), 155–170.

[90] Melvin, Stephen, and Patt, Yale. Handling of packet dependencies: A critical issue for highly parallel network processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (Grenoble, France, Oct. 2002).

[91] Mogul, J., Rashid, R., and Accetta, M. The packet filter: An efficient mechanism for user-level network code. In *Proceedings 11th Symposium on Operating System Principles* (Austin, TX, November 1987), pp. 39–51.

[92] Montz, A. B., Mosberger, D., O'Malley, S. W., Peterson, L. L., Proebsting, T. A., and Hartman, J. H. Scout: A communications-oriented operating system. Technical Report TR 94-20, University of Arizona, Tuscon, AZ, June 1994.

[93] Mosberger, David, Peterson, Larry, and O'Malley, Sean. Protocol latency: MIPS and reality. Tech. Rep. TR95-02, Department of Computer Science, University of Arizona, Tucson, AZ, May 1995.

[94] Nahum, Erich, Yates, David, Kurose, Jim, and Towsley, Don. Cache behavior of network protocols. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (New York, New York, USA, 1997), ACM Press, pp. 169–180.

[95] Nahum, Erich M., Yates, David J., Kurose, James F., and Towsley, Don. Performance issues in parallelized network protocols. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, Nov. 1994), pp. 125–137.

[96] Netravali, Arun N., Roome, W. D., and Sabnani, K. Design and implementation of a high-speed transport protocol. *IEEE Transactions on Communications 38*, 11 (Nov. 1990), 2010–2024.

[97] Neufeld, Gerald W., Ito, Mabo Robert, Goldberg, Murray W., McCutcheon, Mark J., and Ritchie, Stuart. Parallel host interface for an ATM network. *IEEE Network* (July 1993), 24–34.

[98] Nuckolls, Neal. Multithreading your STREAMS device driver in SunOS 5.0. Internet Engineering, Sun Microsystems, Dec. 1991.

[99] Olston, Chris, Jiang, Jing, and Widom, Jennifer. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2003), ACM Press, pp. 563–574.

[100] Olston, Chris, Loo, Boon T., and Widom, Jennifer. Adaptive precision setting for cached approximate values. *ACM SIGMOD Record 30*, 2 (2001), 355–366.

[101] Olston, Chris, and Widom, Jennifer. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *International Conference on Very Large Data Bases (VLDB)* (San Francisco, CA, USA, 2000), pp. 144–155.

[102] Olston, Chris, and Widom, Jennifer. Best-effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2002), ACM Press, pp. 73–84.

[103] Pai, Vivek S. *Cache Management in Scalable Network Servers*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, Nov. 1999.

[104] Pai, Vivek S., Aron, Mohit, Banga, Gaurav, Svendsen, Michael, Druschel, Peter, Zwaenepoel, Willy, and Nahum, Erich M. Locality-aware request distribution in cluster-based network servers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, California, Oct. 1998), pp. 205–216.

[105] Parekh, Abhay K., and Gallager, Robert G. A generalized processor sharing approach to flow control in integrated services networks — the multiple node case. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (San Francisco, CA, Mar. 1993), vol. 2, IEEE, pp. 521–530 (5A.1).

[106] Parekh, Abhay K., and Gallager, Robert G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking 1*, 3 (June 1993), 344–357.

[107] Parekh, Abhay Kumar J. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Feb. 1992.

[108] Peacock, J. Kent, Saxena, Sunil, Thomas, Dean, Yang, Fred, and Yu, Wilfred. Experiences from multithreading system V release 4. In *Proceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)* (Newport Beach, CA, Mar. 1992), USENIX, pp. 77–91.

[109] Peel, Roger. TCP/IP networking using transputers. In *Transputer Research and Applications 3* (Sunnyvale, California, Apr. 1990), pp. 27–38.

[110] Presotto, David. Multiprocessor streams for Plan 9. In *Proceedings of the United Kingdom UNIX Users Group* (Jan. 1993).

[111] Rodriguez, Caroline. A computational environment for data preprocessing in supervised classifications. Master's thesis, University of Puerto Rico, Mayaguez, July 2004.

[112] Rütsche, Erich, and Kaiserwerth, Mattias. TCP/IP on the parallel protocol engine. *Fourth IFIP TC6.1/WG6.4 International Conference on High Performance Networking* (Dec. 1992), 119–134.

[113] Sabnani, Krishan, and Netravali, Arun. A high speed transport protocol for datagram/virtual circuit networks. In *SIGCOMM Symposium on Communications Architectures and Protocols* (Austin, TX, Sept. 1989), ACM, pp. 146–157.

[114] Salehi, James D., Kurose, James F., and Towsley, Don. The performance impact of scheduling for cache affinity in parallel network processing. In *International Symposium on High Performance Distributed Computing (HPDC-4)* (Pentagon City, VA, Aug. 1995).

[115] Salehi, James D., Kurose, James F., and Towsley, Don. The effectiveness of affinity-based scheduling in multiprocessor networking. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (San Francisco, CA, Mar. 1996), pp. 215–223.

[116] Saxena, Sunil, Peacock, J. Kent, Yang, Fred, Verma, Vijaya, and Krishnan, Mohan. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. In *Proceedings of the Winter 1993 USENIX Conference* (San Diego, CA, Jan. 1993), pp. 85–96.

[117] Schmidt, Douglas C., and Suda, Tatsuya. Measuring the impact of alternative parallel process architectures on communication subsystem performance. *Fourth IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (Aug. 1994).

[118] Schmidt, Douglas C., and Suda, Tatsuya. Measuring the performance of parallel message-based process architectures. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (Boston, MA, Apr. 1995), pp. 624–633.

[119] Schwetman, Herbert D. Introduction to process-oriented simulation and csim. In *Proceedings of the Winter Simulation Conference* (New York, NY, USA, Dec. 1990), ACM Press, pp. 154–157.

[120] Schwetman, Herbert D. CSIM 19: A powerful tool for building systems models. In *Proceedings of the Winter Simulation Conference* (New York, NY, USA, Dec. 2001), ACM Press.

[121] Sharaf, Mohamed A., Beaver, Jonathan, Labrinidis, Alexandros, and Chrysanthis, Panos K. Balancing energy efficiency and quality of aggregate data in sensor networks. *The VLDB Journal 13*, 4 (2004), 384–403.

[122] Shen, Kai, Tang, Hong, Yang, Tao, and Chu, Lingkun. Integrated resource management for cluster-based internet services. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, Massachusetts, Dec. 2002), pp. 225–238.

[123] Shen, Kai, Yang, Tao, and Chu, Lingkun. Clustering support and replication management for scalable network services. *IEEE Transactions on Parallel and Distributed Systems – Special Issue on Middleware* (Nov. 2003).

[124] Son, Sung-Hyun, Chiang, Mung, Kulkarni, Sanjeev R., and Schwartz, Stuart C. The value of clustering in distributed estimation for sensor networks. In *Proceedings of the IEEE International Conference on Wireless Networks, Communications, and Mobile Computing (WirelessCom)* (Piscataway, NJ, USA, June 2005), IEEE Press.

[125] Srivastava, Amitabh, and Eustace, Alan. ATOM: A system for building customized program analysis tools. Tech. Rep. 94/2, Digital Western Research Laboratory, Palo Alto, CA, Mar. 1994.

[126] Tantawy, Ahmed, and Zitterbart, Martina. Multiprocessing in high performance IP routers. *Third IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (May 1992), 235–254.

[127] Thekkath, C., Eager, D., Lazowska, E., and Levy, H. A performance analysis of network I/O in shared memory multiprocessors. Technical report, Department of Computer Science and Engineering FR-35 University of Washington, Seattle, WA, July 1992.

[128] Thekkath, Chandramohan A., Nguyen, Thu D., Moy, Evelyn, and Lazowska, Edward D. Implementing network protocols at user level. In *SIGCOMM Symposium on Communications Architectures and Protocols* (San Francisco, CA, Sept. 1993), ACM, pp. 64–73.

[129] Tilak, Sameer, Abu-Ghazaleh, Nael B., and Heinzelman, Wendi. Infrastructure tradeoffs for sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications* (New York, NY, USA, Sept. 2002), ACM Press, pp. 49–58.

[130] Unix System Laboratories. Design of the streams subsystem for SVR4 ES/MP. USL Proprietary: Distribution Subject to Signed Agreement, June 1992.

[131] Urgaonkar, Bhuvan, Shenoy, Prashant, and Roscoe, Timothy. Resource overbooking and application profiling in shared hosting platforms. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, Massachusetts, Dec. 2002), pp. 239–254.

[132] Veenstra, Jack E., and Fowler, Robert J. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Durham, NC, January 1994).

[133] Waldspurger, Carl A., and Weihl, William E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (1994), pp. 1–11.

[134] Waldspurger, Carl A., and Weihl, William E. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report Technical Memo MIT/LCS/TM-528, Massachusetts Institute of Technlogy, Cambridge, MA, June 1995.

[135] Wan, Chieh-Yih, Eisenman, Shane B., and Campbell, Andrew T. Coda: congestion detection and avoidance in sensor networks. In *ACM SenSys Conference on Embedded Networked Sensor Systems* (New York, NY, USA, Nov. 2003), ACM Press, pp. 266–279.

[136] Xu, Bo, Wolfson, Ouri, Chamberlain, Sam, and Rishe, Naphtali. Cost-based data dissemination in satellite networks. *Mobile Networks and Applications 7*, 1 (2002), 49–66.

[137] Yates, David J., Nahum, Erich M., Kurose, James F., and Towsley, Don. Networking support for large scale multiprocessor servers. In *Proceedings of the ACM Sigmetrics*

Conference on Measurement and Modeling of Computer Systems (Philadelphia, PA, May 1996), ACM, pp. 116–125.

[138] Yu, Yang, Krishnamachari, Bhaskar, and Prasanna, Viktor K. Energy-latency tradeoffs for data gathering in wireless sensor networks. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)* (New York, NY, USA, Mar. 2004), IEEE Communications Society.

[139] Yuhara, M., Bershad, B. N., Maeda, C., and Moss, J. E. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference* (Jan. 1994).

[140] Zhang, Xiaolan, Barrientos, Michael, Chen, J. Bradley, and Seltzer, Margo. HACC: an architecture for cluster-based web servers. In *USENIX Windows NT Symposium* (Seattle, Washington, July 1999).

[141] Zhao, Jerry, and Govindan, Ramesh. Understanding packet delivery performance in dense wireless sensor networks. In *ACM SenSys Conference on Embedded Networked Sensor Systems* (New York, NY, USA, Nov. 2003), ACM Press, pp. 1–13.

[142] Zink, Michael, Westbrook, David, Abdallah, Sherief, Horling, Bryan, Lakamraju, Vijay, Lyons, Eric, Manfredi, Victoria, Kurose, Jim, and Hondl, Kurt. Meteorological command and control: an end-to-end architecture for a hazardous weather detection sensor network. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services* (Berkeley, CA, USA, June 2005), USENIX Association, pp. 37–42.

[143] Zitterbart, Martina. High-speed protocol implementation based on a multiprocessor architecture. *First IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks* (May 1989), 151–163.