Applications of Randomness
in
System Performance Measurement

A thesis presented by

Trevor Leslie Blackwell

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

May 1998

Abstract

This thesis presents and analyzes a simple principle for building systems: that there should be a random component in all arbitrary decisions. If no randomness is used, system performance can vary widely and unpredictably due to small changes in the system workload or configuration. This makes measurements hard to reproduce and less meaningful as predictors of performance that could be expected in similar situations.

To measure the sensitivity of non-randomized systems to slight configuration changes, we measured the variation in performance of both TCP/IP and workstation memory systems as a result of “small” configuration perturbations. By “small,” we mean within the range over which things may change unintentionally due to other modifications being evaluated, or within the range of accuracy that an independent researcher could reasonably achieve.

For TCP/IP, changes of a few percent in link propagation delays and other parameters caused order of magnitude shifts in bandwidth allocation between competing connections. For memory systems, changes in the essentially arbitrary order in which functions were arranged in memory caused changes in runtime of tens of percent for single benchmarks, and of a few percent when averaged across a suite of benchmarks. In both applications the measured variability is larger than performance increases often reported for new improved designs, suggesting that many published measurements of the benefits of new schemes may be erroneous or at least irreproducible.

To make TCP/IP and memory systems measurable enough to make benchmark results meaningful and convincing, randomness must be added. Methods for adding randomness to conventional program linkers, to linkers which try to optimize memory system performance by avoiding cache conflicts, and to TCP/IP are presented and analyzed. In all of the systems, various amounts of randomness can be added in many different places. We show how to choose reasonable amounts of randomness based on measuring configuration sensitivity, and propose specific recipes for randomizing TCP/IP and memory systems. Substantial reductions in the configuration sensitivity are demonstrated, making measurements much more robust and meaningful. The accuracy of the results increases with the number of runs and thus is limited only by the available computing resources.

When the overall performance of a system is strongly influenced by the worst case behavior, reducing the sensitivity of the system can also make it perform better. Using average waiting time as a metric, TCP/IP performance is shown to improve significantly when randomization is added to the sending host's congestion window calculations. Although the improvements are less than those achieved by previously proposed schemes using randomized packet discard algorithms inside the network, the proposed modifications can be implemented entirely in the sending host and so can be deployed more easily.

Acknowledgments

The two times one gets to write a formal comment about one’s life and the people who have been a part of it are in one’s thesis, and on one’s tombstone. I don’t know what conclusion to draw from that.

Many thanks to my advisor, H.T. Kung, and my other readers, Margo Seltzer and Mike Smith. All have helped make this thesis much better than it would have been without their advice and suggestions.

Thanks are due to many other folks at Harvard who were sources of inspiration or ideas. Most noteworthy are Robert Morris, who could usually suggest a way to make my ideas simpler and better, and Brad Karp, who in his capacity as graduate school cruise director made life at Harvard much more fun.

I owe also a great debt to my family, who has been very supportive, and always ready to lend their voices to ever-more-deafening choruses of “Finish Your Thesis!”

My wife Laurie has been my constant support throughout this process, the fun and painful parts alike. Thank you for being such a great companion in this adventure, and in many adventures yet to come.

Table of Contents

Chapter 1
Introduction

Arbitrary decisions should be made randomly. Applying this simple, fundamental principle when designing systems makes performance measurements more robust and reproducible, and can result in improved overall performance.

Of course, some decisions are more arbitrary than others. Randomizing completely arbitrary decisions is straightforward: just choose randomly among the alternatives. Decisions for which there exists some basis for preferring one alternative over another should have some randomness added so that preferred alternatives are chosen with higher probability, but not to the complete exclusion of other good alternatives. For instance, if a heuristic function estimates an approximate cost of 1000 for alternative A and 1001 for alternative B, A should be chosen with slightly higher probability than B. The probability should be based on the difference in cost relative to the accuracy of the approximation.

Making it possible to make robust and reproducible performance measurements is not merely of academic interest. When building a system of any complexity there are design trade-offs which affect system performance and that cannot be made on a purely theoretical basis. The only way to maximize system performance is through tuning: iteratively trying different design alternatives and choosing the one that performs best. When performance is highly sensitive to aspects of the system which can change unintentionally due to some other modification being evaluated, performance results will be misleading, wrong choices will be made, and the system will not perform as well as it could. Thus, the accuracy with which performance can be measured is as important as high performance itself.

When arbitrary decisions are made deterministically based on small-scale properties of the system, the overall performance of the system is likely to be highly sensitive to slight changes in configuration, initial conditions, or the results of internal computations of the system itself. For such a system, the graph of performance as a function of one or more of its configuration parameters looks jagged, and a measurement with any given set of configuration parameters may not be representative. If past decisions can affect future decisions, the system may be *chaotic*, meaning that an arbitrarily small change in initial conditions can cause an arbitrarily large change in the behavior of the system.

Highly sensitive deterministic systems have two disadvantages. First, it is hard to make robust and reproducible measurements of them. These terms will be described formally below, but intuitively, if the system performs radically better or worse due to small changes in some aspect of the system, then any single measurement is a poor predictor of the performance that could be expected in general.

The second disadvantage of highly sensitive deterministic systems is that they are likely to fall into behavior patterns which repeat regularly on the small scale and result in poor overall performance on the large scale. For example, a memoryless resource allocator must decide to grant its resource to one user or another based only on the current set of requests. If there is no random input to its decision making process, it may persistently favor some users and starve others, resulting in poor overall performance.

To avoid extreme sensitivity, arbitrary decisions should be randomized. When randomness is added to highly sensitive systems and performance is reported as the distribution across a large number of individual measurements, we show that performance does not change radically with small changes in the initial conditions.

The following hypothetical example illustrates how extreme sensitivity leads to misleading results and incorrect design decisions. Consider the problem of deciding whether or not a particular compiler optimization improves performance (i.e. reduces the runtime of the compiled program.) Suppose that the optimization eliminates some redundant instructions, thereby reducing the number of instruction execution cycles by 2%. The eliminated instructions reduce the sizes of some procedures, and (as will be shown in detail in chapter 4,) cache effects cause runtime to be highly sensitive to procedure sizes. Although in general the optimization would be expected to reduce memory system costs, suppose that changing cache conflicts cause it to increase memory system costs by 10% on the particular benchmark being used by the compiler developer. Thus a measurement would show an 8% performance decrease leading to a decision not to include the optimization in the compiler. Note that if another optimization is added which also affects procedure sizes, the memory system effects might be quite different.

Adding randomness to a system destroys the property that with identical initial conditions, two runs of the system should give identical behavior. This can be a disadvantage in some applications, and must be weighed against the benefit of better performance measurements.

1.1 Meaningful Measurements

The essence of a meaningful performance measurement is its ability to predict the performance of similar systems. A common result of a research paper is of the form, “We took system *X* and modified it to create system *Y* and on the set of tests we ran, Y performed Z% better.” Assuming a system of sufficient complexity that the conditions cannot be reproduced exactly, this result is of practical use only if it implies a likelihood that system Y would perform better under most similar sets of conditions. Measurements of highly sensitive systems can be misleading, because any particular choice of input or configuration parameters may lead to measurements which are quite different from the performance which can be expected of very similar systems.

More technically, a meaningful measurement of system performance should be:

- **Robust**. Small changes in configuration or initial conditions should not cause drastic changes in results. Small unintentional changes to some part of the system as a result of an intentional change to another part of the system should not affect performance very much.
- **Reproducible**. An independent researcher should be able to build a similar system (to the level of detail reported in a good conference paper) which performs similarly.
- **Predictive**. Qualitative performance comparisons should continue to hold for systems built along similar principles.

There are two ways to get the above properties, both of which are used in this thesis. If the system is not highly sensitive (perhaps because of randomization,) then a single measurement is robust, reproducible, and predictive. If the system is highly sensitive, then no single measurement can accurately represent its performance, and performance must be reported as a distribution in order to be meaningful. The distribution of performance measurements across a large number of perturbed configurations is reproducible and predictive, and the median of the distribution may be fairly robust.

1.2 Thesis Contributions

This thesis presents studies of congestion control algorithms and of computer memory systems to show that the performance of these kinds of systems is extremely sensitive to factors that cannot be effectively controlled, and therefore that performance results may not be very robust or reproducible. Various mechanisms underlying this effect are explored, and a metric is proposed to quantify the meaningfulness of performance results.

Then, new techniques for building systems that do not suffer from extreme sensitivity are proposed and analyzed. Building randomization into the core decision making processes of complex systems is shown to make them much easier to measure and compare. When randomization is added, the performance of the system on a given input becomes not a single number, but a probability distribution that can be sampled. It is demonstrated that the center of the distribution characterizing the randomized system is less sensitive than the single measurement characterizing the deterministic system, and that the way the shape and position of the distribution changes as a function of the amount of randomization added can provide additional insight into system behaviour.

Two applications are analyzed in detail. First, performance of programs on typical workstation CPUs is shown to vary substantially as a result of small changes to the program or compiler because of changing cache conflicts. Results presented show a potential systematic error of 10% in measurements of the effectiveness of new optimizations. Even when averaging measurements across a substantial benchmark suite, the error may be larger than the real improvement of many important optimizations.

Second, straightforward measurements of the performance of TCP/IP congestion control schemes are shown to be susceptible to errors of up to an order of magnitude. Well-known randomized packet dropping algorithms reduce the sensitivity to reasonable levels. New ways of randomizing protocols are also proposed and shown to be effective.

Extreme sensitivity can result in low average performance when performance is maximized by making smooth and steady progress. Thus, when randomness is added to congestion control algorithms, average transfer time as well as fairness is shown to improve significantly. This thesis proposes some techniques for improving TCP/IP performance involving only modifications to the sending host, which may be easier to deploy than previously proposed schemes for adding randomness in the network.

1.3 Outline of Dissertation

Chapter 2 discusses related work and provides some minimal background on sensitivity analysis, TCP/IP, and compilers.

Chapter 3 develops the theoretical basis for adding randomization to systems. It outlines the principles for estimating the sensitivity of complex systems, making reproducible measurements of systems in spite of their sensitivity by reporting distributions rather than single measurements, and for adding randomization to systems to make them more amenable to making robust and reproducible measurements.

Chapter 4 applies the theory to measuring the performance of CPU instruction caches and of optimization algorithms designed to improve cache performance by relocating related blocks of code to non-conflicting addresses. It is demonstrated that without randomization, systematic errors of several percent are likely when measuring program performance changes due to new optimization algorithms. It is shown that by adding randomization to a profile-driven program layout system, much more accurate performance measurements are possible.

Chapter 5 applies the theory to measuring the performance of TCP/IP's congestion control algorithms. It is demonstrated that in the absence of any techniques to combat it, measurements of the fairness of competing connections can be misleading by multiple orders of magnitude. Some previously known techniques are analyzed, and shown to be somewhat effective in achieving valid results. Modifications to TCP are described that enable reliable measurements to be made without any special techniques.

Chapter 6 develops several new ways of adding randomness to TCP/IP congestion control systems, and analyses the performance impact of each. Random drop gateways, previously proposed in the literature, are shown to yield substantial performance improvements. Some of the new proposed techniques, although they yield slightly less performance improvement, may be more practical to deploy since they can be implemented by small changes to algorithms in the sending host rather than in network routers.

Chapter 7 concludes with a summary of the results and lessons learned, and makes some suggestions for using randomness in building and measuring new systems.

Chapter 2
Background and Related Work

This thesis builds on previous work in the fields of sensitivity analysis, randomization, memory system performance optimization, and TCP/IP performance, each of which is described in turn below. At the end of this chapter, the connections between previous work and the present work are discussed more generally.

2.1 Sensitivity Analysis

A primary theme in this thesis is exposing the sensitivity of complex software systems to small changes in external parameters. Techniques for sensitivity analysis have been explored for some other kinds of systems, and are briefly reviewed here.

The sensitivity of numerical algorithms to small errors in their inputs has been discussed extensively in the numerical methods literature. A brief review of the standard results on sensitivity and stability of numerical algorithms will be given here.

For systems of linear equations, the *condition number* provides a concise and useful quantification of the sensitivity to input perturbations [27]. It provides metric for the maximum change in the output of a system, as a function of amount of change in the input. Thus, for linear systems the problem of quantifying the sensitivity can be considered solved. For more complex systems, analysis is not so simple.

2.1.1 Sensitivity Analysis of Complex Systems

In the design of analog circuits, some figure of merit of a system must be guaranteed over the range of parameter variations of a large number of components. For instance, the frequency of an oscillator may depend on dozens of component parameters, each of which can vary in several ways over the range of operating voltage and temperature and between manufacturing runs.

Because nontrivial circuits might have many dozens of parameters, it is common to use Monte-Carlo simulation to estimate the probability distribution of the relevant figure of merit [29]. The probability distribution of the figure of merit is sampled across a large number of simulations, each with values for process or component parameters randomly chosen from their respective probability distributions. The yield of acceptable parts can thus be estimated. Well-designed circuits are not usually highly sensitive to parameter variations. That is, the figure of merit usually varies smoothly and monotonically, if not linearly, due to changes within the normal range of any one parameter, and usually not by as much as an order of magnitude.

Ho and Cao [28] describe methods for approximate estimation of the sensitivity of queuing systems to small changes in the capacities of queues without running multiple simulations. The results are only accurate if events do not get reordered by the change. In TCP/IP systems, queue overflows cause large changes in the sequence of future events; thus the techniques are not applicable.

2.2 Randomization

Randomization is used extensively in cryptographic and number-theoretic algorithms, but these uses are not relevant to the present work. Here, we describe applications where randomization is used in order to achieve performance goals.

Fair Tie Breaking. In systems where two or more entities compete for a shared resource such as a network and queuing is not feasible, some requests must be rejected. A naive solution would be to accept the entities with the lowest IDs (such as network address) and reject all others. In times of heavy usage, entities with high ID numbers would experience starvation. The best known solution is to choose winners and losers randomly. The ALOHA packet radio system [24] used randomization to choose which of multiple competing stations would transmit next by making each station wait a random amount of time before starting to transmit. Ethernet [25] uses a similar scheme in the event of collisions.

Avoiding Pathological Cases. A number of important algorithms have good average-case performance, but poor worst-case performance. Randomization can be used to guarantee (with very high probability) that worst-case behavior will not occur in practice.

The most familiar example of such an algorithm is the naive implementation of Quicksort [3], which takes time $O(n \log n)$ on average, but can require $O(n^2)$ time in the worst case. Unfortunately, in a naive implementation the worst case occurs when the input is already sorted, a situation likely to occur in practice. The most appealing theoretical solution to the problem is to choose pivot elements randomly. [5 §1] Then, the worst case behavior is extremely unlikely and cannot occur regularly in any real system. In practice, more sophisticated deterministic methods of choosing pivot elements provide better results [3].

Routing on hypercube parallel interconnection networks is another application where straightforward deterministic algorithms have very bad worst-case performance. For this application the worst-case input to the obvious non-randomized algorithm happens to be a simple pattern likely to occur in practice [5 §4.2]. The solution is to route each message through a randomly chosen intermediate node, adding a large degree of randomness to message routing. Thus, worst case performance can be bounded with very high probability.

Solicitation from Large Population. In some multicast systems, the sender must adapt its sending rate according to how much bandwidth the receivers are capable of receiving. This requires feedback from the recipients. Having every receiver send a periodic status report to the sender would destroy the scaling properties of multicast, since the sender would be overloaded by status reports. Statistical Acknowledgment, proposed by Holbrook [26], solves this problem by soliciting acknowledgments from a small, randomly chosen subset of receivers.

2.3 Memory System Performance Optimization

Modern computers use memory hierarchies in which fast but small memories are placed near the CPU, and large but slow memories are more distant. Each level but the last acts as a cache for the next level. The access time to a nearby memory might be 100 times shorter than to a distant memory: thus, performance depends critically on effective cache management in order to keep frequently used data near the CPU. Sophisticated cache management schemes, such as LRU (least recently used) are infeasible to implement in fast hardware. Instead, many systems map each memory address to one possible locations in any given level of cache. Thus, some pairs of addresses conflict (cannot be held simultaneously in the cache) and some do not. For this reason, the mapping of portions of the program to memory addresses affects performance. It is better if two parts of the program which frequently alternate are placed at addresses which do not conflict in the cache. Modifying any part of the program can shift large parts of the rest of the program around in memory, and thus have a large effect on performance even if the modified function itself is never used.

Chen [6] measured memory system behaviour under variants of the Mach operating system, and showed that it is strongly affected by page mapping policy. On the machines under study, page mapping affected performance by changing the way memory locations conflict in the second level cache.

Chen's work reports simulated performance with two page mapping strategies. The "deterministic" strategy ensured that consecutive virtual pages are mapped to consecutive locations in physical memory, modulo the L2 cache size. The "random" strategy maps each virtual page independently to a physical page chosen from a free list. (Chen's terms "random" and "deterministic" do not mean the same thing as they do in the rest of this thesis.) Performance for the "deterministic" and "random" strategies was measured, but only a single experiment was made for the "random" strategy, so no distribution was presented. Results varied between the two runs by as much as 55%: sometimes the "deterministic" strategy ran faster; sometimes the "random" strategy ran faster.

Under most operating systems, page placement is determined by the history of what programs have been run before and paging activity. Memory layout often varies between versions of the operating system and amount of physical memory present, and is highly dependent on such “unimportant” factors as the order in which object modules are listed on the linker command line. Thus, Chen’s results also suggest that performance measurements may be hard to reproduce accurately by independent researchers.

By reordering memory regions at a procedure granularity rather than a page granularity, similarly large effects on performance are demonstrated in Chapter 4.

More early evidence that there might be large effects of detailed program memory layout on system performance was given by Quong [19]. He observed that cache miss rates “typically vary by 10% of their relative value” for different program layouts, and as a way of measuring the underlying loss rate of the program (independent of a particular layout,) proposed a layout-independent way of estimating cache miss rates using a statistical model of the intervals between references to blocks.

In contrast to a layout-independent cache miss rate estimate, this thesis presents efficient ways of measuring the distribution of expected cache miss rates across all layouts. I argue in chapter 4 that the distribution is more directly relevant to real-world system performance, and provides additional information useful to researchers.

2.4 TCP/IP Performance

Network congestion control systems solve the problem of allocating a limited amount of bandwidth available across a network link among all the connections which have data to send. This thesis focuses on TCP/IP in particular. In TCP/IP, a host sends packets into the network and routers forward packets from link to link until they reach the receiving host. When routers receive more than they can send or buffer, they drop packets. The receiving host returns an acknowledgment for packets received. The sender limits the amount of unacknowledged data to no more than its congestion window, which is increased when packets are successfully delivered and decreased when packets are lost.

A great deal of work has been done on tuning the TCP congestion window algorithms. As well as fundamental changes suggested by Jacobson [30], Brakmo [31], Lin [41] and Morris [42], a number of minor improvements (summarized in Stevens [32]) have been made over the years.

The sensitivity of TCP to network parameters has been remarked before, although previous papers have pointed only to small variations. Brakmo and Peterson [17] reported that fairness between competing TCPs could change by 15% due to small changes in the start times of connections, and that total throughput could change by 34% due to a change in the router buffer size of one packet. Much larger variations are demonstrated in chapter 5.

2.5 Summary

Sensitivity analysis, while considered essential in some fields, is rarely performed for compiler & network performance measurement (the author has never found a published example thereof.)

Both Chen’s [6] and Quong’s [19] results suggest that memory layout could have a large effect on system performance—larger than the improvements reported for many new optimizations. Thus, it is clear that this phenomenon should be explored and quantified. This is done in chapter 4, both for compiler systems which do not attempt to optimize memory layout and for those which do.

Likewise, earlier work by the present author and others suggested that TCP/IP could be highly sensitive to slight changes in some parameters, both in simulation and in testbeds. The degree of sensitivity to all the parameters that could be isolated is systematically analyzed in chapter 5.

A major contribution of this work is a precise and systematic analysis of the pitfalls inherent in measuring highly sensitive systems. That these pitfalls existed in TCP/IP and compiler algorithms was previously known to many researchers and could be inferred by careful reading of published works, and, at least for TCP/IP, some techniques for avoiding them were known and implemented in widely available simulators. I hope that the systematic techniques presented here for measuring and avoiding these pitfalls will make some kinds of measurement work less of a black art.

Chapter 3
Randomizing Systems

This thesis is the result of applying a simple, fundamental principle to the design of systems: that *arbitrary decisions should be made randomly*. In many current systems that do not follow this principle, performance is highly sensitive to slight changes in configuration, making it hard to make robust and reproducible measurements. Worse, when the performance of such systems is dominated by repeating patterns of decisions, overall performance is likely to be poor.

Some decisions are more arbitrary than others. Randomizing completely arbitrary decisions is straightforward. Decisions for which some basis for preferring one alternative exists should have some randomization added so that the preferred alternative is chosen more often, but not to the complete exclusion of others.

This principle and its corollaries will be developed in detail in the rest of this chapter.

3.1 Measuring Complex Systems

Performance measurements can be misleading when the system exhibits a wide range of behaviors and thus a wide range of performance levels even over a small range of input conditions. This is demonstrated both for TCP congestion control and for program layout in memory. In chapter 5, it is shown that two TCPs competing for bandwidth on a bottleneck link, fairness varies by as much as an order of magnitude due to a change of less than 1% in the propagation delay of a link. In chapter 4, it is shown that different arbitrarily chosen layouts result in 4% differences in benchmark performance.

Because both TCP and compilation/cache systems are deterministic (assuming a sufficiently controlled experimental setup), repeated experiments will yield identical results. However, these results may not be representative of performance in similar configurations. Thus, it is easy to get results with large systematic errors, even though common validity metrics such as the standard deviation of performance across ten runs of the experiment indicates low variance.

This thesis proposes making performance comparisons based on the distribution of performance results obtained by Monte Carlo sampling [9] of a small area of the space of possible configurations around each benchmark input. In its most general form, the principle can be stated thus:

P1. When measuring performance of a system, all configuration parameters that can reasonably be varied should be perturbed by a small amount. Report performance as the distribution across a large number of randomly chosen perturbed configurations.

To be more concrete, the following procedure is followed. Consider a configuration ***C*** representing a benchmark configuration in which a given system is to be tested. In a network simulation, for instance, ***C*** would specify the propagation delays of links, the size of router buffers, etc. A perturbation procedure must be devised to randomly produce values $\overset{\frown}{\boldsymbol{C}}$ that are similar to ***C***. For instance, $\overset{\frown}{\boldsymbol{C}}$ might have the propagation delays of each link multiplied by a random variable with mean 1 and small variance so that the propagation delays in all the $\overset{\frown}{\boldsymbol{C}}$ vary randomly within 5% of the delays specified in ***C***.

The perturbation procedure should vary every aspect of ***C*** that can feasibly be varied. Designing perturbation procedures is somewhat subtle. First, all the parameters must be found. While many of the parameters are obvious, some may be buried implicitly inside the system. Some may not be externally adjustable without extra implementation effort.

Second, it must be decided how much variation constitutes a small amount and what sort of distribution is reasonable. For real-valued parameters such as propagation delays, variations of +/- 5% was small enough to correspond to the range of situations that a network planner would consider equivalent, but large enough to elicit a wide range of different performance results from simple simulations. Some other situations are more complex. Perturbed parameters must continue to be physically reasonable. For instance, router buffer sizes must still be an integer number of packets after perturbation. Propagation delays must not become negative.

The distribution of performance results obtained by following P1 is much more informative than the single number produced by conventional benchmarking techniques. The distribution indicates not only the average performance, but also the range of performance values that can be expected in similar situations and hence the validity of any single measurement, the probability of observing pathologically bad cases, and evidence for potential improvement.

In this work, distributions are shown graphically as cumulative distribution functions. Figure 1 illustrates how performance distributions change as a result of improving either robustness or performance. When system performance is made more robust (for instance,

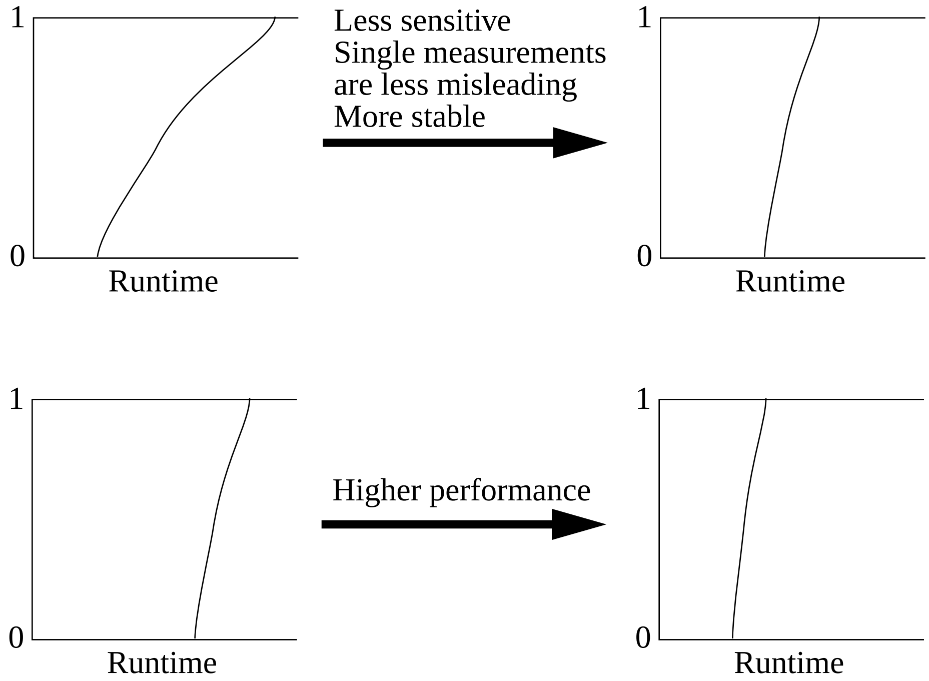


FIGURE 1. Comparing Performance Measurements. Distributions of runtime for a hypothetical benchmark across many runs with randomly perturbed configurations are shown as cumulative distribution functions. The performance variation is being reduced in the top set, and the average runtime is being reduced in the bottom set.

by adding randomness,) the cumulative distribution function becomes more vertical, spanning a narrower range of performance values. When system performance is improved, the distribution shifts to the left, corresponding to lower runtimes.

If the range of the performance distribution is broad, as it was for many of the systems analyzed in this thesis, it is strong evidence that performance results from any individual simulation of a similar system may fail to be representative of typical values. This may call previous results into question, or alternatively it may indicate that the extra computational expense of performing large numbers of randomized simulations is not needed for some kinds of situations. It will be shown for two different kinds of systems in chapters 4 and 5 that, based on the width of performance distributions, results are likely to be grossly misleading unless randomized measurement techniques are used.

Many systems have some probability of producing spectacularly bad performance. Network configurations with competing TCP sessions can sometimes have nearly complete starvation of one or more connections. The probability of observing such behavior in real world situations can be estimated by the present technique. For instance, in chapter 5 it will be shown that there is a 7% probability (over the space of configurations similar to the benchmark configuration) that one TCP session will receive only 1/1000th of its fair share of bandwidth. This is clearly an important figure of merit that is not captured by single performance numbers, or even by an average. A system that was fair on average but for 7% of specific situations was grossly unfair might well be considered unacceptable.

If the distribution of performance results includes some good results, it may suggest that with some tuning the system could be made to perform well. This is particularly the case for program memory layout, as will be shown in chapter 4. If high performance results show up with low probability, it suggests that a better algorithm might be able to achieve such good results in practice.

Following principle P1 adds random effects to measured performance numbers. Thus it is logical to ask whether it may result in less accurate results. The answer to this question must address just what is meant by accuracy of results.

There is always a trade-off between accuracy and broad applicability of results. Accuracy is often easier to obtain. For instance, a researcher could make a single measurement of the height of some particular person, and report a statement such as “The height of the subject is 71.243 in +/- 0.002 in[1]” This very accurate measurement may be perfectly correct, but it is not very useful. No other researcher would be likely to get a similar result. If another researcher measured the height of person from another city and found a different height, it would not be an adequate basis for making any sort of comparison.

On the other hand the researcher could measure the height of 1000 randomly selected Bostonians and report, “The height of Bostonians is 68.1 in with a population variance of 3.2 in.” The second statement contains more information about the range of height of people. Furthermore, another researcher should be able to do the same experiment with a different random pool of 1000 Bostonians and get a very similar result. If yet another researcher measured height of Cantabridgians and found that their height was 67.2 in with a population variance of 3.2 in, it would probably be a sound basis for concluding that people from Cambridge are shorter on average. A good discussion of some more philosophical aspects of the meaningfulness of measurements may be found in Gould [36].

In the systems analyzed in this thesis, applying principle P1 produces numbers that are less repeatable, but more reproducible and with greater predictive value. The loss of repeatability is more than compensated for by the other benefits.

[1] There are no measurements behind these examples: they are purely made up.

3.2 Building Measurable Systems

The preceding section has reviewed some of the difficulties of making good performance measurements of complex systems, and how randomized measurement techniques can be used to achieve better results as well as an estimate of the validity of the results. We now turn to the issue of designing systems that are inherently easier to measure, and argue that such systems are to be preferred as being easier to tune and perform other performance-related research on. Adding randomization to the internal decision making algorithms of complex systems is the key.

The fundamental purpose of adding randomization to systems is to avoid making decisions in a deterministic way based on insignificant factors. That is, there are some decisions that must be made one way or another, but there is no clear basis for choosing. In this case, the decision should be made randomly. For instance, a router in a congested network must drop packets. In traditional routers, the decision as to which packet is to be dropped is made based on the exact arrival pattern. The packet dropped is the one that arrives when the queue is exactly at capacity. A very small change in any one of many factors can shift the order in which packets from different connections arrive, and so can affect which connections lose packets. A random-drop router [22], however, drops a randomly chosen packet in the queue. Because the fraction of packets in the queue due to each connection changes slowly (over time periods of the same order as the queue size divided by the link rate), the system is less sensitive to small timing changes.

Lack of extreme sensitivity is fundamentally a desirable property of systems. That is, this thesis argues that given two systems that have equal average performance, the less sensitive one should be preferred for the following two reasons.

First, most high-performance systems are the result of a lot of tuning. TCP, for instance, has been improved in at least eight iterations since its first wide release in 1983 [10, 30], and many more proposed improvements exist [31]. Each improvement has been based on careful measurement of the behavior of the previous system and comparison with the new system. As is demonstrated in chapter 5, it is very difficult to measure TCP performance even in very simple configurations. Thus, we hypothesize that although TCP has many merits, its deterministic nature has made quantifying its performance so difficult that even after more than ten years, researchers are still arguing about what modifications might improve its performance further.

More basic research as to which kinds of system architectures perform better would also be facilitated by building less sensitive systems. When comparing two different flow control architectures (e.g. rate-based vs. credit-based [34,35]) it would be much easier if both systems would yield reliable, repeatable performance numbers in tests.

Another reason why less sensitive systems should be preferred is to facilitate end-user selection of the best system for a given task. For instance, someone wanting to purchase a workstation to run specific simulations could run performance tests on several competing workstations and choose the best price/performance. To the extent that program performance is strongly and deterministically dependent on ephemeral compilation conditions, as is demonstrated of real systems in chapters 4 and 5, any such measurements are likely to give the wrong answer, or at least an answer that will no longer be correct if any small changes are made to the compiler, linker, or simulation code.

Thus, this thesis proposes the following system design principles as an aid to building systems of which robust, reproducible, and predictive performance measurements may be made.

P2. For every decision, if there is no clear basis for making it one way or another, it should be made randomly rather than arbitrarily.

P3. For decisions based on a cost function, a small amount of randomness should be added to the values of the cost function before choosing the best. The amount of randomness to be added should be settable by the user.

P4. Values computed based on approximate or estimated values should be chosen randomly from a probability distribution reflecting the real accuracy of the values.

Principle P2 can be considered a special case of principle P3, where the cost function is zero for all choices.

We can summarize the difference between measuring complex randomized and deterministic systems in the following table. Although deterministic systems measured under identical conditions should give the same results, measurements under slightly different conditions may give very different results. The randomized systems described below, however, are less sensitive to their conditions. Thus, in order to reproduce a result, another researcher need not set up an identical experiment, but only a similar one. Because simulations never exactly match the real world, simulations of randomized systems are more likely to match real-world behavior than simulations of deterministic systems.

	Performance Results From	
	Deterministic System	**Randomized System**
Identical Conditions:	Same	Similar distribution
Similar Conditions:	Possibly very different	Similar distribution

TABLE 1.

It should be remembered that these conclusions apply to complex systems where performance-impacting decisions are made somewhat arbitrarily. Systems where the factors affecting performance are simple enough can be measured without special techniques. The techniques of randomized measurement presented earlier can be used to decide whether a system's performance is complex enough to warrant randomization. If a large number of measurements made under randomly perturbed conditions are all similar, then individual measurements should be sufficiently accurate.

3.3 Benchmarking Randomized Systems

Figures 2, 3, and 4 illustrate the process of benchmarking deterministic and randomized systems. True system performance can best be defined as the average performance across all possible inputs to the program, with each input weighted according to how often it will occur in practice. Even if the frequency of occurrence of each input were known, this would not lead to a feasible benchmarking system because for any interesting program the space of possible inputs is infinite.

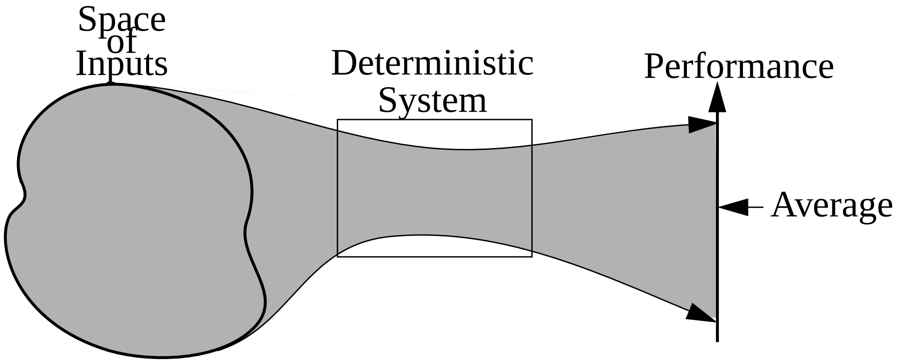


FIGURE 2. True System Performance is the average, appropriately weighted, across the entire space of inputs. The space of inputs is usually infinite, so this is not a feasible measurement.

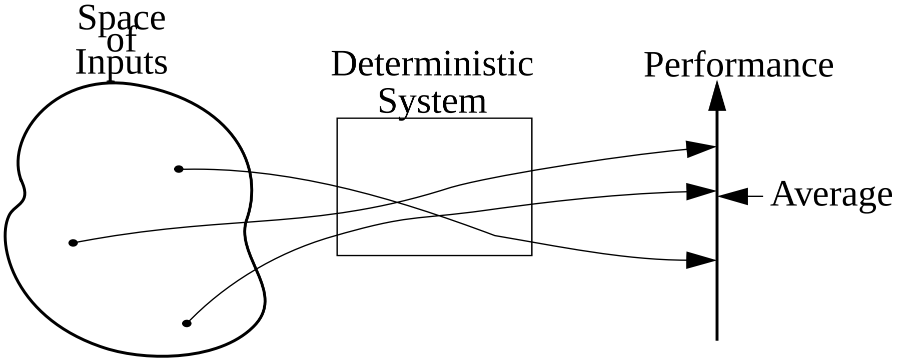


FIGURE 3. Benchmarking a system maps a few points in the space of inputs onto performance measurements. The average of the measurements is typically used to make comparisons.

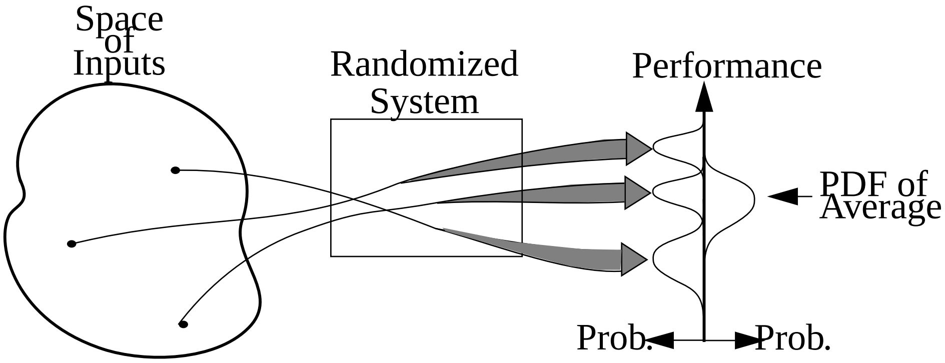


FIGURE 4. Performance of Randomized Systems is a probability distribution for each input. Running time distributions are averaged, and the resulting distribution function is used to compare different versions of a system.

3.4 System Tuning

Optimizing, or “tuning” systems for good performance is considered something of a black art in the computer science community. Superficially, system tuning is done by making small modifications to a system and comparing performance before and after the change. There are two difficult aspects to this job: conceiving of changes that are likely to improve system performance (a task best left to human experience and intuition), and making an accurate comparison between old and new systems. Conventionally, accurate comparisons of competing systems are made by averaging performance over a large suite of benchmarks. There are two limitations of this approach.

The first limitation is that a large set of test inputs must be assembled. For some applications, convenient sets of benchmarks exist, such as SPEC CPU95 for general-purpose C and Fortran compilation on POSIX-like machines. For other applications, such as congestion control algorithms, time must be spent developing benchmarks. For some other applications, such as compiler optimizations for particular operating system modules, only one benchmark may exist.

The second limitation of aggregating results from a large set of inputs is that information is lost. If an algorithm performs better on some class of inputs and worse on another, that fact is likely to be useful to the algorithm designer. However, aggregate performance tends to conceal such differences. On practical benchmark sizes, testing for large numbers of potential performance differences leads to many false positive results that are time-consuming to track down.

3.5 Meaningfulness of Results

Consider a result that is a performance comparison between two different versions of some system. Nine out of ten papers in systems conferences report such measurements [Small97]. Meaningful results are repeatable, robust, reproducible, and have broad applicability. These characteristics are discussed in detail below, in order of increasing amount of difference over which the results are still valid. Repeatable results are valid for the same system, same parameters, at a different time, whereas broadly applicable results should be valid for similar systems, similar parameters, and at different times.

Repeatability. Repeatability is the degree to which successive measurements of a system with the same parameters are close to one another. All but the very simplest systems do not have perfect repeatability.

Variation in the measured performance of a system has both exogenous and endogenous components. Exogenous performance variation is bad. It is due to imperfect controls on the experimental setup and can generally be avoided by sufficiently careful attention to detail.

Endogenous variation is due to nondeterminism inherent in the system being measured. Many standard components of systems contain significant amounts of nondeterminism, and therefore contribute to endogenous performance variation. Contention in CSMA network protocols [33], disk seek times, and interrupt timing interactions are common sources of endogenous nondeterminism.

Robustness. Robust results are not highly sensitive to the test configuration. That is, very small changes in parameter values do not often produce large variations in measured performance. The graph of performance as a function of any parameter is relatively smooth.

The robustness of a system can be determined by doing a sensitivity analysis to determine the amount by which the measured performance varies over a small range of variation in parameter values.

Robustness is important when some aspects of the system cannot be perfectly controlled. As an example, any compiler optimization which avoids unnecessary instructions is likely to change the memory layout of the program as well as reducing the instruction count, and these indirect, unwanted, and irreproducible effects can be larger than the direct effect of the optimization which the researcher wants to measure and therefore make comparison of performance with and without the optimization unreliable.

Reproducibility. Reproducible results are those for which another researcher could construct a similar (but not, of course, precisely identical) experimental setup, and produce similar results. Many results are not reproducible from their published description because details are missing, but we do not consider this human aspect of reproducibility here. We are concerned only with the reproducibility of the system itself.

Robustness is a necessary condition for reproducibility, because in general parameter values may be interpreted slightly differently by a different system. But reproducibility also requires that small changes in the system do not greatly affect the nature of the results.

Predictive Value. For results that have predictive value, applying them to a different but related system would produce a qualitatively similar effect on performance.

3.6 Discovering Significant Differences

Knowing that a proposed modification to an algorithm improves performance on some class of inputs more than on another class can be very useful. It can help researchers understand where there is room for improvement. It can lead to a hybrid algorithm that performs well on both classes. It can provide insight into what causes the algorithm to make suboptimal choices.

If a researcher can propose a partitioning of the available input set into two classes, it is usually straightforward to separately compute performance on the two classes and make a comparison. A large difference probably indicates a phenomenon worthy of exploration, whereas a small difference could be due to noise.

When only a few partitionings are to be tested, the researcher can explore all observed differences. For instance, general-purpose compiler benchmarks are conventionally partitioned into integer-intensive vs. floating-point-intensive classes, and into ALU-intensive vs. memory-intensive classes. A researcher can investigate in detail any observed differences across these two partitionings.

If many partitionings are to be tested, it is important to investigate only the ones that have statistically significant differences. The number of potentially interesting partitionings is limited only by the researcher’s imagination. For instance, it might be the case that a new instruction scheduling algorithm performs better on blocks with chained multiply-add-store instructions, but does not perform better overall. There are an enormous number of hypotheses at this level of detail. If such a proposition could be shown to be valid, it would be valuable information.

For instance, consider 1000 hypotheses of the form “The modified algorithm performs relatively better on class *A* than class *B*,” none of which are actually valid. When tested against real data, however, one would expect about 5%, or 50 of them to be accepted at a 95% confidence level. It would be a great waste of time to investigate 50 invalid hypotheses about the system. Thus, to discover interesting detailed properties about the system, it is vital to have results with high statistical significance.

If the input set is divided into two classes, it may be the case that a proposed algorithm performs relatively well on one class and relatively poorly on the other. If this dichotomy can be discovered, it may help the researcher understand the limitations of each algorithm, and guide him toward a hybrid that combines the good features of each.

The number of potential performance dichotomies is enormous. Any criterion for dividing the set of inputs into two classes could reveal a significant performance difference.

When many results must be aggregated in order to filter out variations due to extreme sensitivities, it is hard to discover such dichotomies. Randomized measurement can make it easier to discover such facts.

For instance, it commonly happens that performance of a particular compiler optimization technique is different for floating-point-intensive programs than for integer programs. Such a performance differential is reasonably likely to be discovered, because it is easy to categorize programs as integer or floating point, and because researchers know from experience to expect such differences, and are looking for them.

Consider a hypothesis about a new technique that increases performance overall, such as, “The average performance on blocks containing a multiply-add-multiply sequence decreases.” Such a state of affairs is certainly possible, and the hypothesis can be tested in a straightforward manner. However, because a combinatorially large number of similar hypotheses exist, only hypotheses with a very high confidence factor can be tested. For instance, if one million hypothesis are generated, there will be about ten thousand for which false positives can be found at a 99% confidence level.

Chapter 4
Randomness and Memory System Performance

The location of code and data in memory can have a significant impact on system performance, because some sets of memory addresses conflict in the memory cache. Traditionally, the layout of memory is determined by some simple formula, without much consideration for performance. Measurements made on such systems are strongly affected by the arbitrary choices made by the programmer, compiler, and operating system. We show that the variation in runtime due to arbitrary choices can be 20% or more, and that these ephemeral effects can fool developers into making wrong choices. Sampling many randomly chosen memory layouts yields far more robust and reproducible results.

4.1 Background

When compiling and executing code, most systems impose few limitations on where code and data objects can be placed. Code and static data objects can generally be placed in any order, with any amount of padding between them. Basic blocks (sequences of consecutive instructions with a single entry point) inside functions can be laid out in any order, although some orders are better than others. Within a function, local variables can be assigned to stack locations in any order. Virtual memory pages can be independently mapped to physical pages. Static data objects can be rearranged and padded. Each dynamically allocated data object can be independently placed in memory. All these arbitrary decisions are candidates for randomization.

The high performance of modern CPUs depends upon fast caches, on or close to the CPU, satisfying the majority of memory accesses. Each access to main memory takes dozens of instruction cycles, during which the CPU cannot accomplish much. When data is available from the cache, it can generally be accessed in one or two cycles. It is reasonable to expect this effect to become more important over time, since CPU clock speeds have a long history of increasing more rapidly than main memory access times [43].

In a cache, each memory address is mapped to a line in the cache. Each cache line can hold some fixed, small number of memory locations simultaneously. In a direct-mapped cache, the most popular design for reasons of speed and cost, each cache line holds only a single entry. If two memory locations both map to the same entry in the cache, they are said to *conflict*. If a program is laid out in memory such that two frequently accessed functions or variables are assigned to conflicting memory locations, the program will run more slowly.

Conventional compilers that do not employ any special techniques to lay out procedures in order to minimize cache miss rates must still make a decision as to where each procedure will reside in memory. Conventional C compilers for Unix systems lay out individual functions within object files adjacent to each other in the same order in which they appear in the corresponding source files. Then, the executable contents of object files are placed adjacent to one another in the executable according to the order in which the `.o` files are listed on the linker command line.

At runtime, the executable image is mapped linearly into virtual memory at some fixed starting address. Between functions within object files and again between object files, padding is generally added to align the beginning of each function to a multiple of the cache line size, such as 16 or 32 bytes. Padding is done so that no function spans more cache lines than necessary, and because some processors incur a fetch penalty when jumping to an address not aligned on a certain power of two boundary. For instance, the Alpha 21164 [38] performs best when jumping to addresses divisible by 16.

Virtual memory may or may not map linearly onto large physically indexed caches depending on the operating system. Chen describes this phenomenon in detail [6]. If the mapping is not deterministic, as is the case in the Mach 3.0 OS, separate runs of the program may produce different cache behaviour.

The ordering of procedures within source files is typically chosen for readability, or at least reflects the order in which they occurred to the author. The order in which `.o` files are listed in the `makefile` (compilation script) is similar. These choices have a strong effect on the measured “before” performance — i.e., the null hypothesis against which the performance of the new algorithm is to be compared. Furthermore, the effects are fixed. Compiling and running the program 100 times will give the same result.

4.2 Sensitivity Errors due to Cache Effects

Because program execution time depends on program memory layout, and because there is no well-defined “standard” layout, measurements of program performance with any particular memory layout can introduce a systematic error into the measurements. In general, increasing the code size increases the miss rate, and decreasing the code size decreases the miss rate. However, a program compiled with varying degrees of code expansion is subject to changing patterns of cache interference. Thus, any one particular ordering does not represent the general case, and therefore results may be misleading and irreproducible.

One time-honored way to avoid this problem is to simulate the program on a machine with an idealized memory system that never has cache misses. However, modern microprocessor performance is so dependent on cache performance that this old technique has limited validity.

It is even more important to include memory system effects when the optimization under study makes explicit trade-offs between memory size and instruction cycle count. For instance, the static correlated branch prediction algorithms described by Young [18] rearrange and make copies of basic blocks in order to reduce the number of cycles lost due to mispredicted branches. However, the size of the frequently executed code increases. Thus the processor wastes fewer cycles due to branches, but may incur more cache misses. Work by others has focussed on reducing the footprint of code to improve performance. Mosberger [37] found that removing rarely executed code from the contiguous block of memory occupied by a function significantly reduced memory system stalls. Studies of these and other optimizations need robust and reproducible measurements of memory system costs.

To estimate the magnitude of the potential for error, cache simulations were run on hundreds of versions of SPECint95 benchmark programs where the procedure orderings were as produced by the default compilation script, but where the code was expanded by factors ranging from 0.5 (compaction) to 1.5 (expansion.)

Simulations were performed for the Alpha 21064 cache architecture, with an 8 KB L1 cache and 256 KB L2 cache, both direct mapped with 32 byte lines. The 21064 has a fairly complex cache miss handling system, but for simplicity we use an approximate measure of memory system costs. The metric only include memory effects due to instruction references, since randomizing the placement of data is beyond the scope of this work. Costs are expressed as MCPI (memory cycles per instruction,) using estimated penalties of 10 cycles for a L1 cache miss, and 100 cycles for an L2 cache miss.

Figure 5 shows the results of 512 cache simulations for the `xlisp` benchmark from the SPECint95 benchmark suite. The smaller test input set was used rather than the reference input, because running a large number of cache simulations on a large benchmark is very time consuming. The figure shows graphically how for the Xlisp benchmark there is a significant probability of being misled as to the real performance impact of code expansion. Figure 5 presents similar results for the other benchmarks in the SPECint95 suite. The floating point suite was not tested.

Some large features appearing in some of the graphs deserve explanation. For instance, the large spike in miss rate for the `go` benchmark at an expansion factor of 1.4 is due to a conflict between two small procedures located about 5800 bytes apart and called frequently by the inner loop of the program. At an expansion factor of 1.4, they conflict in

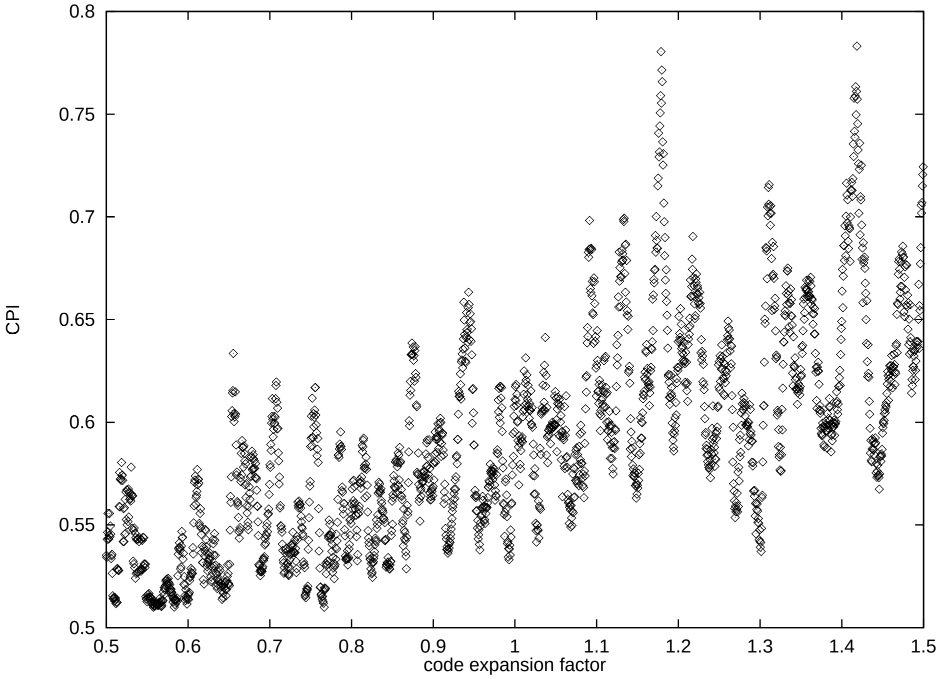


FIGURE 5. Cache Miss Rate vs Code Expansion. Benchmark is SPEC95 lisp interpreter. Upward trend is visible, but the results show extreme sensitivity.

the primary cache. The periodic spikes in `ijpeg` at intervals of about 0.1 on the X axis correspond to conflicts between two small frequently procedures located about 80 KB apart in the original program, and which therefore shift by 8 KB (the size of the primary cache) for each change of 0.1 in the expansion factor.

We can estimate the potential systematic error in the cache miss rate metric that would result from testing with only a single program layout as follows. Assume that across the space of all layouts, miss rate varies approximately linearly with expansion factor, at least within the narrow range of [0.5 .. 1.5]. There is no strong theoretical basis for this assumption — it is based on the linear appearance of the graphs. A straight line is fitted to the data using a Chi-square estimator to give the line with least squared error. An error

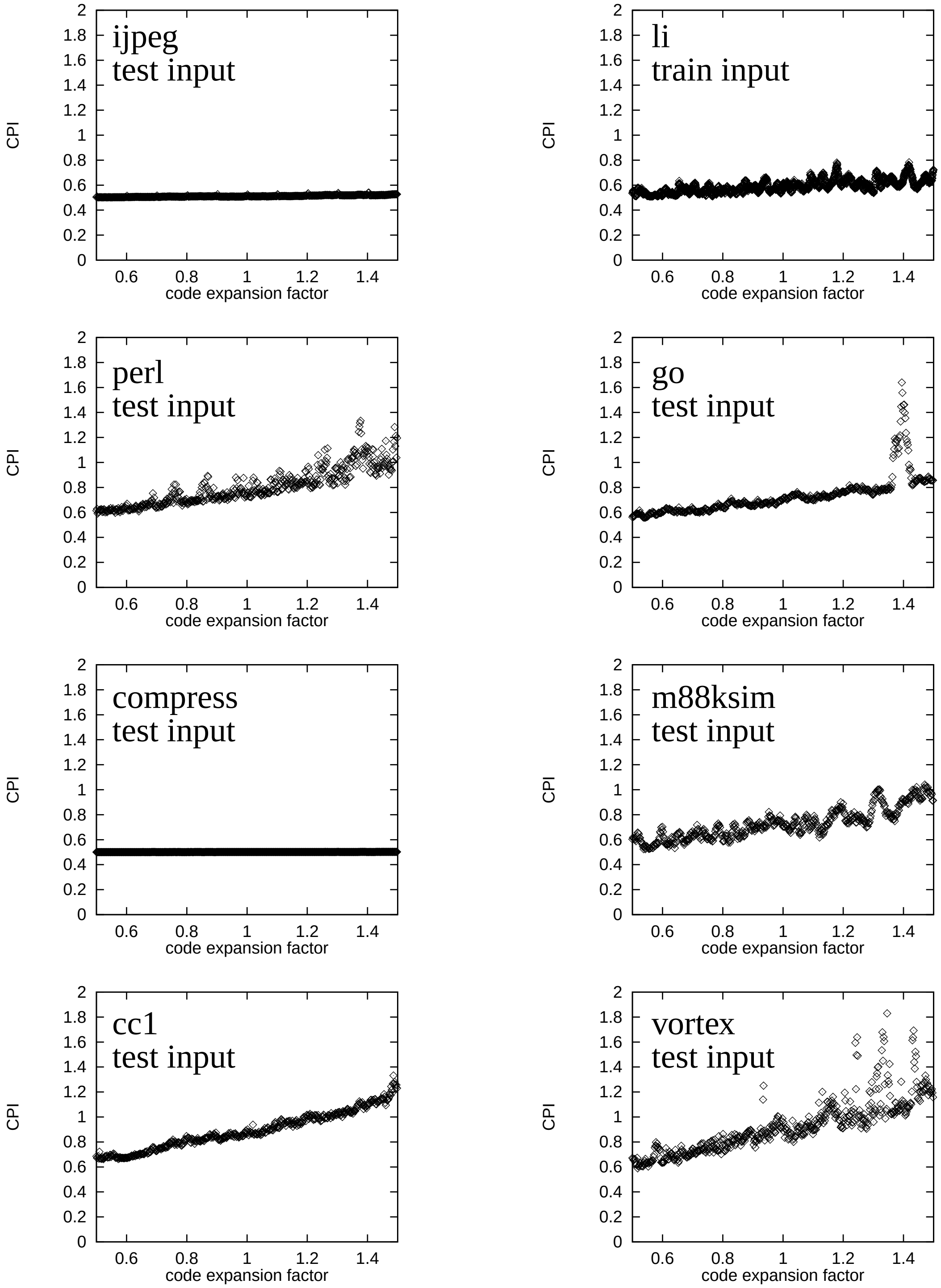


FIGURE 6. Cycles per Instruction vs Code Expansion for SPEC95 Integer Benchmarks. As in figure 5, the CPI metric is plotted for 512 values of code expansion. Most benchmarks, except ones with very small code sizes such as compress show significant sensitivities.

value is calculated for each point as the distance from the point to the fitted line, and the width of the 90% confidence interval for measurements with the given error distribution is reported.

The effect of cache misses on program runtime is highly dependent on the specific processor, and is not a linear function of the number of misses. However, we can make a rough estimate for a typical modern workstation by assuming that an instruction costs 0.5 cycles, a L1 cache miss costs 5 cycles and an L2 cache miss costs 50 cycles. Based on this CPU model, we can compute the error in runtime measurements due to error in cache miss rates. These numbers are all presented in Table 2.

The potential measurement errors are quite large. Most of the benchmarks yield error bounds for total runtime of more than 10%. The benchmarks that have narrow confidence intervals for runtime are those that are so small that they fit entirely in the cache for any layout. But all programs with significant instruction cache activity (miss rate larger than 1%) demonstrated large error potential.

The width of the 90% confidence interval when the results of all the programs are averaged together using the geometric mean is 3.7%. For a comparison between two programs, the 90% confidence interval becomes. Thus, results showing an improvement of up to 8.0% in SPECint95 performance as a result of adding some optimization may be purely due to hypersensitivity to cache effects. In the absence of the techniques presented in the next section, performance comparisons of new compiler techniques that change the memory layout of the program (it is hard to imagine optimizations that do not) should be reported with error bars in the 8% range, depending on the relative cost of cache misses.

Benchmark	Estimated CPI	CPI measurement comparison error (90% confidence interval)
ijpeg	0.512	1.1%
li	0.603	16.7%
perl	0.800	18.8%
go	0.730	15.1%
compress	0.501	0.1%
m88ksim	0.729	20.0%
cc1	0.891	7.2%
vortex	0.920	21.7%
Geometric Mean	0.691	3.7%

TABLE 2. Miss Rates and Error values. For each benchmark, the estimated true miss rate at unity expansion and the probable error are given. The width of the 90% confidence interval for the difference between two measurements is reported both as an error in MCPI and as the corresponding error in runtime for the simple CPU described in the text.

One approach to achieving good results is to average performance across a larger number of benchmark programs. Assuming a normal distribution, the variation decreases approximately as the square root of the number of programs. Thus, averaging performance across 800 programs instead of the 8 included in SPECint95 would reduce the error by a factor of 10, to a tolerable 0.8%. There are a few drawbacks to this approach. First, collecting a large suite of benchmarks may not even be possible for some problem domains. For instance, some compiler work is specifically focussed on optimizing kernel implementations of networking protocols for which there may exist only a handful of implementations.

Making experimental compilers work on a large set of programs is hard. Anecdotal evidence suggests that the extra effort required to take a research compiler that can handle most of the SPECint95 suite and make it correctly compile a large range of programs is substantial.

Having to average performance results over a large suite of programs makes it harder to identify interesting differences between programs. For instance, it might be very illuminating to discover that a particular optimization improved performance on a benchmark with many small procedures, but decreased performance on another with a few large procedures. If the large amount of error present in the measurement for each program can only be reduced to an acceptable level by averaging together all the available programs, it is impossible to reliably detect such phenomena.

More sophisticated techniques are necessary to achieve meaningful memory system performance results. The next section proposes such a technique.

4.3 Comparing Optimizations using Randomized Memory Layout

The previous section demonstrated the large systematic errors possible when measuring two slightly different versions of a program. Here, we describe a technique for reducing the sensitivity of such measurements to arbitrary cache effects by measuring the distribution of performance results across a large number of randomized memory layouts.

Two different implementations of program layout randomization were used in this work. The first uses Atom [44] to convert a binary executable into one that simulates its cache performance by adding calls to simulation code at the start of each basic block. This system was used for all measurements presented here except where noted.

The second layout randomizer randomizes the layout of the program itself to allow measurements of real (not simulated) runtime. In a prototype implementation for Digital Unix, the compiler is made to output the entire program as an assembly file, and a simple program reads in the assembly file, identifies procedure boundaries, randomly permutes the procedures, and writes out a new assembly file. The linker converts this to a running executable. Implementation is particularly easy for Digital Unix where the globally optimizing C compiler outputs a single assembly file for all the code. On a system that produces separate assembly files for each module some renaming might be necessary to isolate static variables with the same name in different files. The prototype implementation consists of less than 100 lines of code.

Figure 7 shows measured runtimes (not from a simulator, as are other results in this chapter) for Xlisp using 125 random procedure orderings. The results show similar variability to the runtime measurement comparison error reported in the previous table.

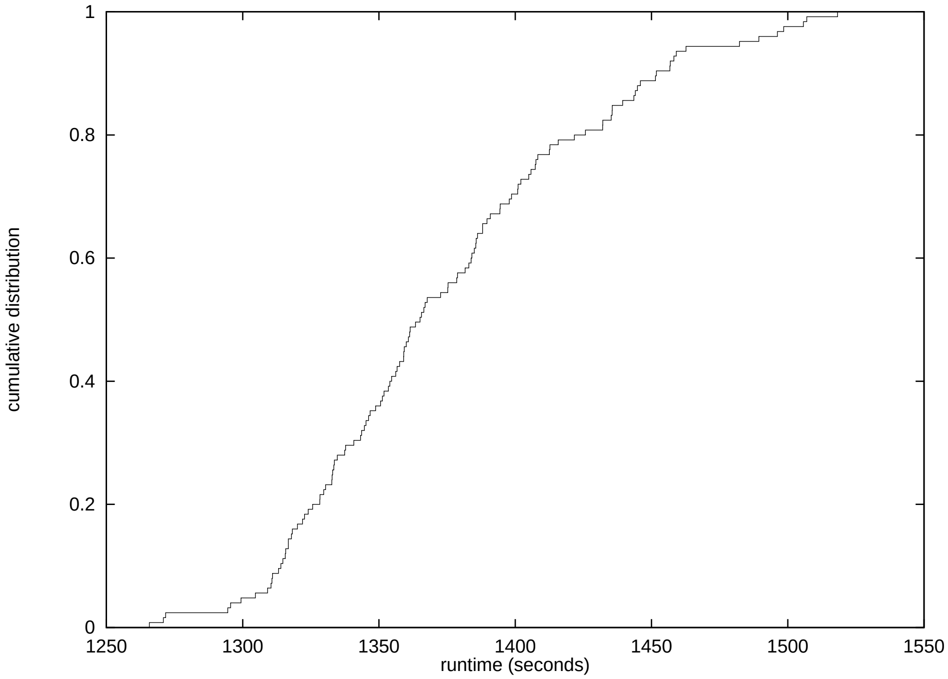


FIGURE 7. Cumulative Distribution of Measured Runtimes for Random Program Layouts for Xlisp. 125 layouts were used, and each layout was run 3 times with less than 1% variation in runtime between the 3 runs. The spread of the center 90% is about 5%, which corresponds well to the layout-to-layout variation of 11.1% reported in table 2. The program is SPEC95 `li` running on the reference data set on a DEC 3000/400.

Randomized procedure placements can be used to get a more accurate picture of the effect of code expansion on performance. Figure 8 shows the same measurement as in Figure 4, but using the median CPI value across 384 different layouts. The results no longer show extreme sensitivity.

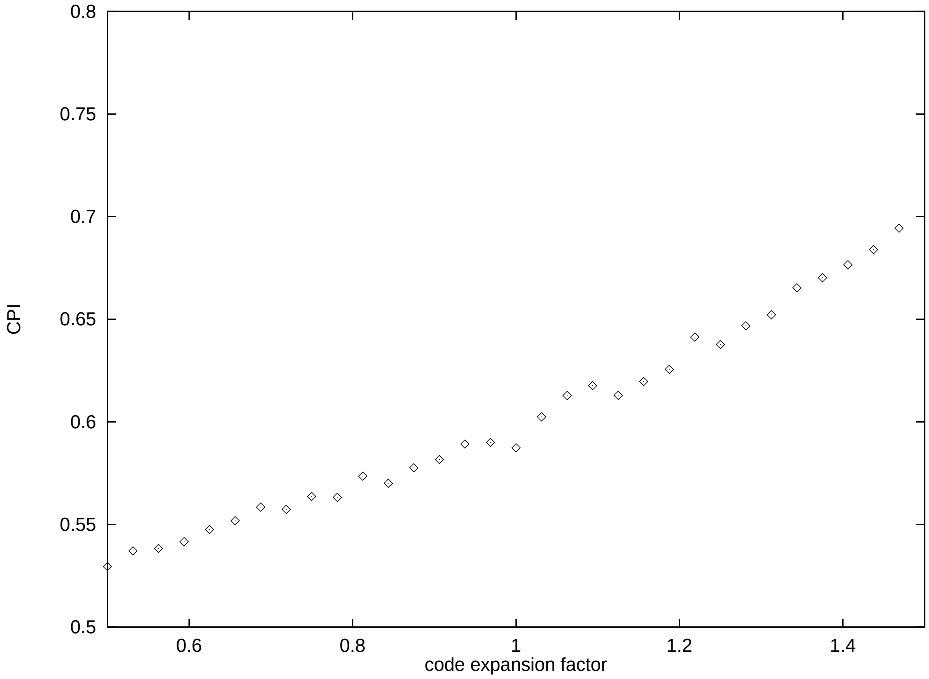


FIGURE 8. Cache Miss Rate vs Code Expansion using Random Layouts. Values for simulated Alpha 21064 with 8KB L1 cache and 256 KB L2 cache running SPEC95 xlisp on the training input. Compare to Figure 4. Each data point is the median CPI across 384 runs, each with an independent random memory layout. Using the average instead of the median gives a similar, but slightly more irregular curve. The dip at unity expansion occurs because cache line alignment of loop entry points is preserved through long procedures.

This thesis proposes that all measurements of compiler optimizations in systems that do not include layout optimizations be reported based on the median of the distributions of runtime with random procedure orderings. Otherwise, as has been demonstrated for the programs most often used for benchmarking, systematic errors more than %10 of runtime are to be expected.

4.4 Randomized Layout Optimization

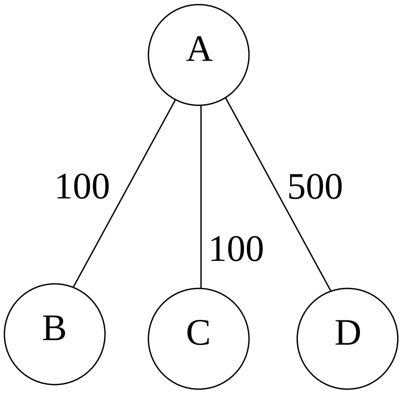
The preceding sections discussed the problem of measuring program performance when there is no reason to prefer the default program layout over any other. However, some new compiler systems include layout optimizers that use profile data to place interrelated procedures at non-conflicting addresses. In these systems, the layout optimizer produces an approximation to the best possible procedure layout or, so it is reasonable to wonder if the concepts in the preceding sections no longer apply.

We show that layout optimizers do make a large number of arbitrary or nearly arbitrary deterministic decisions that have large effects on program performance. Thus, adding randomization to the layout optimizer can facilitate performance evaluation not only of the program and the rest of the compiler, but of the layout optimizer itself.

The canonical approach to layout optimization is due to Pettis and Hansen [45]. The basic P&H procedure positioning scheme uses a call graph profile. First, the program is instrumented to record all procedure calls. Considering the procedures to be nodes of a graph, weights are assigned to edges according to how many times a call occurred between pairs of procedures. Then, the optimization algorithm tries to place pairs of procedures with high-weight edges (i.e. procedures that call each other frequently) close together. If two procedures are adjacent, and their total size is less than the size of the (direct mapped) cache, then they will not conflict. As well as reducing instruction cache misses, the technique has other benefits. On some architectures, long branches are more expensive than short branches. Also, it is likely to reduce the number of TLB misses and page faults. Here, however, we only consider the effect on the primary instruction cache.

The algorithm proceeds by finding the edge with the largest weight, and merging the two corresponding nodes. The new node represents a chain of procedures. The algorithm repeats until no edges remain in the graph.

The result of the algorithm is not in general uniquely determined. The final order of independent procedure chains is unspecified. Each individual chain can be flipped backwards. There may have been identical edge weights in the original graph — such ties are broken arbitrarily. Not only are these decisions not specified in the P&H algorithm or any improved versions thereof, but there is no obvious "right" way to do it. Given the edge weights shown below, it is clear that A and D should be adjacent. However, either B or C is an equally good choice for being the other procedure adjacent to A.



Recent work in code layout optimizations has focused on path profiling [15]. In the standard algorithm, whenever a procedure is entered, weight is added to the edge between the entered procedure and the last procedure to be active. Path profiling adds weight to edges between the entered procedure and the last several procedures to be active.

Path profiling avoids some potential procedure conflicts that are not detected by the standard algorithm. For instance, if procedure A calls both B and C, the standard algorithm does not add weight to the edge between B and C, and may place them in conflicting positions. Path profiling will add weight to the B-C edge, because B will have been recently active when C is called.

A system implementing the union of the two approaches was developed. Profiling works as follows. A queue of recently active procedures is maintained. When a procedure is entered in any way (call, return or any other mechanism such as `longjmp`,) the value a^n is added to the edge between the new procedure and the nth most recent procedure in the queue, for all $n \geq 0$. Then the new procedure is pushed onto the head of the queue. The constant *a* is a tuning parameter less than one. Setting $a = 0$ yields the standard algorithm with no path history. An efficient implementation of this algorithm drops procedures from the tail of the queue for which $a^n < 0.01$.

During initial performance measurements of the system, the improved algorithm produced a program that produced 10% more instruction cache misses than the baseline, suggesting that perhaps it is not superior after all. This was shown to be a systematic error in measurement which could be avoided by randomized tie breaking, as explained below.

The cache conflict profile is a matrix that gives an estimate for every pair of procedures of the cost of placing them in locations that conflict in the instruction cache. The matrix is used as input to a greedy layout heuristic that repeatedly finds the largest entry in the matrix for which one of the procedures has not already been placed, and tries to put those procedures in non-conflicting positions. In using the technique, a list of procedure pairs is sorted according to profile information. There are many cases where two procedure pairs have the same metric, and so the sorted order is arbitrary. When these ties are broken randomly rather than arbitrarily, several runs of the optimizer will produce programs with different performance characteristics. The table below shows the results of ten such experiments, each comparing the performance of a run with history and a run without.

Experiment #	**Change in Miss Rate**
1	-18%
2	0%
3	-17%
4	+10%
5	-18%
6	-10%
7	-38%
8	-7%
9	-20%
10	+9%

Clearly, any single measurement can seriously misrepresent the value of the technique. Making the variation visible in the experimental setup eliminates a source of systematic error, and converts it into a sampling error.

Sampling errors are much less troublesome than systematic errors. For one, they provide an estimate of the statistical significance of any comparison. Whereas many papers on compiler techniques consider a 10% change in some performance metric to be a cause for celebration, in this experiment at least a single measurement showing 10% change is virtually meaningless.

Sampling errors can also be reduced by averaging together many independent samples. To try to get a clearer picture of how the algorithm performs, ten thousand experiments were run. The results are summarized below.

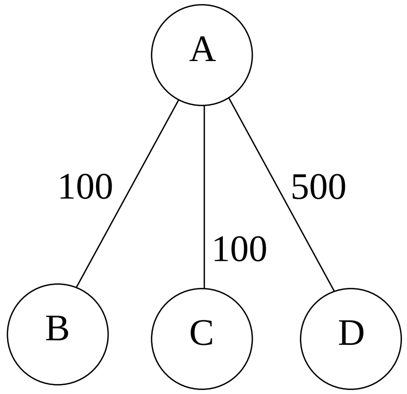
	Change in Miss Rate
Average	-17%
Maximum	+10%
Minimum	-39%

The averaged numbers may be enough to conclude that the technique does in fact improve performance.

4.5 Randomization

Two degrees of randomization are considered here in an effort to convert sources of systematic error into sources of sampling error. The first is simply to break ties randomly. That is, if the edges A-B and A-C have equal weight, the decision of which pair to make adjacent is decided randomly. The graph below shows a set of call graph edge weights for which there is a tie, and the four equally good orderings.

Call Graph Edge Weights



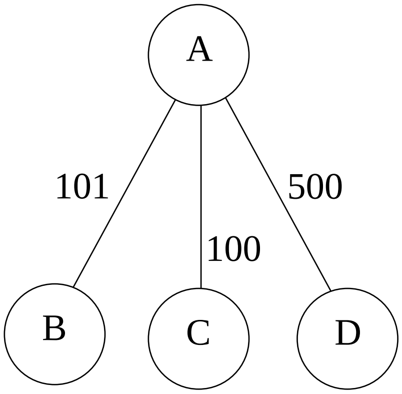
Possible P&H Orderings (all equally likely with randomized tie-breaking)

D - A - B - C
D - A - C - B
C - B - A - D
B - C - A - D

The second level of randomization is to add noise to the edge weights. The edge weights measured in a profiling run are based on a small set of inputs (often just one), whereas the goal is to optimize performance across all future inputs to the program. Thus, measured profile data constitutes a small sample out of a very large space of edge weights for all possible program inputs. There is a substantial amount of error between the measured data and the "universal" profile. By adding noise to the sampled data, the results can be made less dependent on the statistically insignificant properties of the sample.

For example, the weighed call graph below is similar to the last, but with the A-B edge increased to 101. It is doubtful that this small change in the edge weights genuinely supports the choice of D-A-B-C over D-A-C-B.

Call Graph Edge Weights



Possible P&H Orderings (all equally likely with randomized tie-breaking)

D - A - B - C

C - B - A - D

The formula used for randomizing edge weights is shown in Equation [1]. The constant *C* is used to vary the amount of randomness. The multiplicative formulation has the advantages of being self-scaling, and of not producing negative weights.

$$\widehat{W} = W \cdot e^{C\boldsymbol{Z}} \qquad \boldsymbol{Z} \text{ is normal random deviate, mean=0, variance=1} \quad \textbf{(EQ 1)}$$

When evaluating the benefit of code layout schemes (for instance, to determine whether they are worth the development effort) they must be compared against a system without any special code layout algorithms. In such systems, procedures are typically laid out in the order the linker encounters the procedures, which is dependent on the order in which object files are specified on the linker command line, and the order in which procedures appear within source files. Neither of these orderings is very fundamental. Therefore, the baseline scheme against which all results are compared uses random procedure ordering. In this scheme, every procedure is placed at an independently randomly chosen address (but overlapping is prohibited).

4.6 Results

With randomization, the results for each scheme are not a single number, but a probability distribution of performance values. These are presented as cumulative distribution functions. The results shown here are the measured distribution across 128 runs of each program, each with an independently generated procedure layout. Here, we switch from CPI to cache miss rate because with the optimization, the cache miss rate is low enough that differences are hard to discern from graphs of CPI.

Figure 9 shows results for the Xlisp program from the SPECint95 benchmark suite. A number of observations can be made from the graphs. All the cache layout algorithms produce much better performance than a purely random layout, with benefits up to more than an order of magnitude.

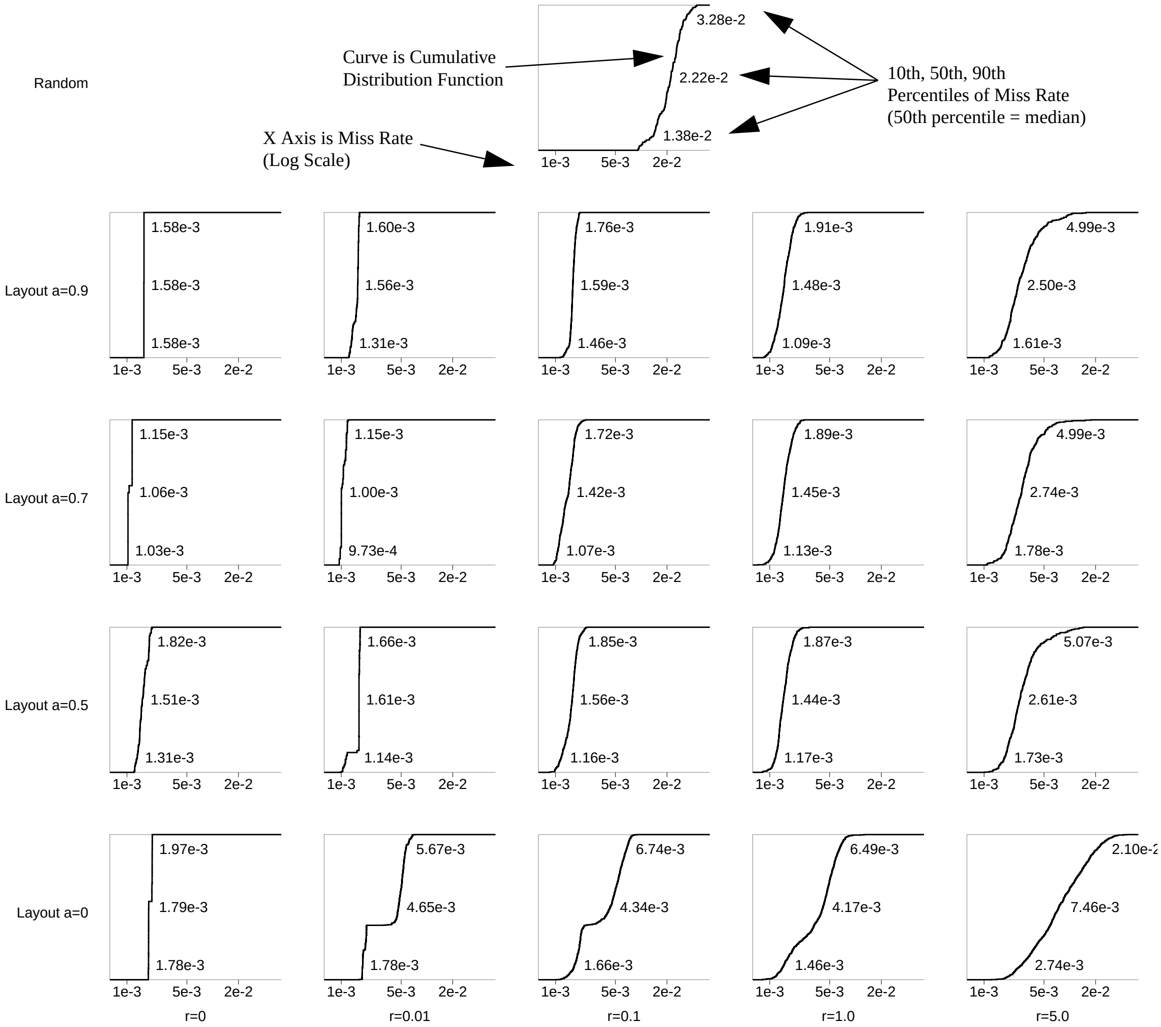


FIGURE 9. Distributions of Primary (8 KB) I-Cache Miss Rate for Xlisp across Different Layout Algorithms. Each graph shows the distribution of primary instruction cache miss rate across 128 runs with independently generated layouts. Each row corresponds to a layout algorithm. For all but the top row, the amount of randomness increases from left to right; top row shows a completely random layout. The parameter *a* controls how much weight is given to procedures which execute not consecutively but closely spaced in time. When *a*=0, only consecutive procedures are counted.

The width of the distribution of miss rate for the purely random layout is +/- 54% (geometrically[1]) at the 10th and 90th percentiles. The width of the distribution for the a=0.5 case without noise added to the edge weights (r=0) is +/- 18%. Multiplying these ratios (1.54 and 1.18) gives an estimate of the potential systematic error if randomized techniques were not used: +/- 81%. (The probability of an error this large is 2%, since both measurements must be outside the center 80% of the distribution). As is shown in Table 2, `xlisp` exhibits the highest sensitivity to layout, but most of the other benchmarks are not far behind. The least sensitive benchmark, `cc1`,

Adding 1% of random noise to the edge weights used in standard P&H layout has a dramatic effect on the primary I-cache miss rate, increasing it by more than a factor of two on average. Variants of the P&H algorithm that consider path history are much more robust, their performance deteriorating only slightly with 100% multiplicative random noise.

With a large amount of error added to the edge weights (r=5), the standard algorithm (a=0) can perform worse than a purely random layout. However, the algorithms that make use of path history still provide close to an order of magnitude improvement in miss rate even with multiplicative randomness of 500%.

The best results are achieved for a=0.7. This probably reflects how likely it is that a recently active procedure is still in the cache after N more procedures have been entered.

[1] All ranges are given geometrically. That is, +/- 54% implies that the true value is between 1.54 times the measurement and 1/1.54 times the measurement.

Considering the a=0.5 case, performance of the layout algorithm does not degrade significantly with increasing added noise until r > 2.0. Adding +/- 200% noise does not degrade performance but adding +/- 500% noise does. Similar results were obtained for the GCC and Go programs from SPECint95. Thus we conclude that edge weights need only be measured to within a factor of two, suggesting that random sampling is probably adequate for profiling.

The width of the distributions is large enough that it may be beneficial, when no expense is to be spared in order to generate the very best possible code, to generate several independent layouts (using a moderate amount of noise in the edge weights) and pick the one that performs best overall.

4.7 Conclusions

This chapter has presented results from some well-known benchmarks that demonstrate the following claims:

- Non-randomized comparisons of compiler optimizations that affect code size (as almost all optimizations do) can have substantial systematic errors. A 10% error in runtime is typical, and up to 22% error was demonstrated for one common benchmark. These errors increase in proportion to the relative importance of cache effects, which some researchers expect to increase on future CPUs.
- Using randomization reveals how large the systematic errors may be, and allows for statistically significant comparisons.
- Code layout optimizations algorithms also exhibit hypersensitivity, and should be randomized. Breaking ties randomly demonstrates some of the variability.

- Adding noise to the edge weights of a profile before using it for layout optimization exposes much more variability, and is the recommended technique for evaluating code layout optimizations.
- The code layout algorithms that consider path history perform better than the standard algorithm, and are more robust in the face of inaccurate edge weights. The standard P&H algorithm depends critically on the exact values of the edge weights, such that adding 1% noise can seriously degrade its performance. Variants that consider path history do not have the same problem.
- Systems that use statistical sampling to gather approximate call graph profiles should heed the conclusion that the standard P&H algorithm, without path history, may perform badly when there are small sampling errors in the profile data.
- Compiler optimizations that contain arbitrarily made decisions are subject to large systematic errors. Results were shown for code layout schemes that showed a one in five chance of having errors as large as a factor of two. The potential magnitude of these errors has not previously been reported.

We remark that all the problems and solutions proposed in this chapter are a result of caches being not fully associative. That is, a fully associative cache with LRU replacement would not exhibit high sensitivity to layout, and would not require the techniques described here. However, there are no serious proposals for making instruction caches fully associative.

There is reason to believe that many other compiler algorithms have similarly high sensitivity. Any algorithm that contains a step of the form “Choose an available X”, where X might be a register, execution unit, processor, memory block or other resource, is subject to large sensitivity errors if this choice is made arbitrarily but deterministically. It would be valuable to characterize the magnitude of the potential error in measuring the effectiveness of other kinds of compiler optimizations.

Many other algorithms contain a step of the form “Choose the X with the largest F(X)”, where F(X) is a heuristic function that guides the algorithm toward good choices. At a minimum, distributions should be measured with randomized tie-breaking — that is, if multiple Xs have identical values for F(X), one should be chosen randomly. To go one step further, one value of F(X) should only be always chosen over another if the difference is statistically significant. The approach used here of adding varying amounts of noise to the heuristic function (i.e. the measured call graph edge weights) not only provided valuable estimates of the error bars and isolated the results from sensitivity errors, but highlighted some undesirable behavior in the standard P&H algorithm.

Chapter 5
Randomizing TCP/IP to Improve Measurability

There is a long history of networking research that suggests that tuning congestion control protocols for good performance takes a long time. TCP, as originally proposed, suffered from a number of performance problems that were corrected over a period of many years based on extensive measurement and observation. In addition, there are several proposed new congestion control algorithms competing for inclusion in future standard implementations. The main obstacle to standardization may be the difficulty in accurately measuring the relative merits of each.

It is therefore proposed that ease of making accurate performance measurements should be considered an important attribute of any protocol. Because tuning protocols takes a long time, and much of that time is spent trying to decide whether various proposed modifications are really beneficial or not, we posit that a protocol that is easy to measure will achieve higher performance more quickly than a protocol that is harder to measure. It is therefore proposed that network protocols that are inherently amenable to robust performance measurements should be preferred over ones that are not, as they facilitate evolutionary improvement by the network research community.

We show in this chapter that it is hard to make meaningful measurements of TCP performance without randomization. The deterministic algorithms in TCP and router queue management make the system highly sensitive to slight parameter variations. This chapter will present results from systems that exhibit performance changes of more than an order of magnitude due to changes in simulation parameters of less than 1%.

Randomness can be added in four ways. First, when simulating a deterministic network, many simulations can be run with small random perturbations to the configuration parameters. Second, random traffic sources can be used to drive simulations instead of deterministic ones. Third, randomization can be added to the router implementations, for instance to drop randomly chosen packets when the queue overflows. Fourth, randomness can be added to the end-system implementations, by modifying the congestion control algorithms.

All the above techniques are shown to have a useful role in the quest for meaningful results. Randomly perturbing the network configuration gives a fairly direct measurement of how reproducible and how broadly applicable the results of measurements are. Adding random traffic sources can make the system less sensitive to slight configuration changes, and may be closer to real-world behavior. Randomized network protocols are inherently easy to measure, and less likely to show extreme behavior.

5.1 Introduction

This chapter presents measurements of the fairness between competing TCP sessions, using the network configuration shown in Figure 10. The network has two TCP connections competing for bandwidth on a single bottleneck link. In all cases, TCP's maximum window size is set to 32 KB, and packets are 552 bytes. It has been shown [40] that TCP is unfair to connections with long access links relative to ones with short access links — this configuration represents a relatively mild difference in length, but where noticeable unfairness might reasonably be expected. All simulation results in this chapter were generated by the NS [21] simulator.

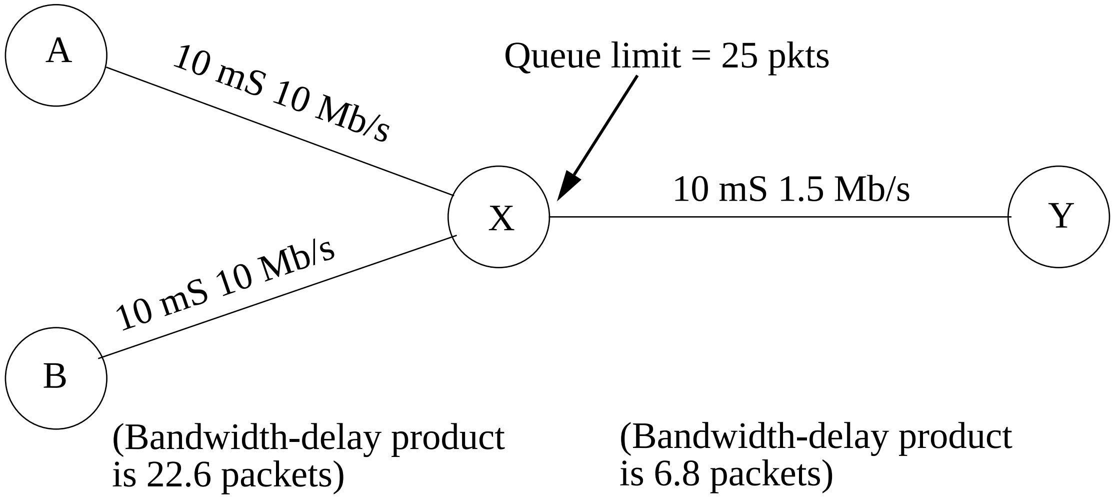

A
10 mS 10 Mb/s
Queue limit = 25 pkts
X
10 mS 1.5 Mb/s
Y
10 mS 10 Mb/s
B
(Bandwidth-delay product is 22.6 packets)
(Bandwidth-delay product is 6.8 packets)

FIGURE 10. Simple Fairness Testing Configuration. Two TCP sessions are set up: A to Y, and B to Y. Both are greedy. Baseline configuration parameters are shown. The bandwidth delay products for both kinds of links are shown in units of 552-byte packets.

Two metrics are used to report performance of TCP implementations in various network configurations: unfairness, and delay time.

Unfairness is measured as the log (base 10) of the ratio of application-level throughput between the two sessions. That is, we report $\log 10(D_{AY} \div D_{BY})$, with goodput metrics D_{AY} and D_{BY} measured as the largest ACK sequence number received by the corresponding sender. A value of +1.0 implies that the A-Y connection got ten times as much bandwidth as the B-Y connection. This metric can be generalized to more than two sessions by comparing the best and worst performing TCP.

In many places in this chapter, we will want to vary a parameter randomly “by a few percent”. The following definition makes precise how parameters are varied. Most real-valued parameters are measurements of quantities for which negative values do not make sense. Propagation delay is an example. In this chapter, “X% randomness was added to a parameter P” means that the parameter is perturbed according to EQ 2, where *C* is the amount of randomness (*X*/100) and ***Z*** is a random variable from the standard normal distribution, with mean 0 and variance 1. For small values of C, this is similar to the approximation in EQ 3.

$$\overset{\frown}{P} = P \cdot e^{C\boldsymbol{Z}} \qquad \textbf{(EQ 2)}$$

$$\overset{\frown}{P} \cong P + Pe^{C\boldsymbol{Z}} \qquad \text{(for small values of } C\text{)} \qquad \textbf{(EQ 3)}$$

5.2 Perturbing Network Configurations

This section presents the results of a number of network simulations on the same network topology, as shown in Figure 10.

In all cases presented here, total throughput and utilization of the bottleneck link was very high — above 90% of capacity. However, in many situations one TCP got most of the bandwidth, while the other got little.

The two tables below show the results of two very similar simulations. The only difference is a change of about 2% in the propagation delay of one of the links. However, the results change dramatically. In the first, the longer link is favored by about a factor of 3. In the other, the shorter link is favored by more than a factor of 3. This large level of unfairness is frustrating to network users.

Router Queue	Drop incoming when >= 25 pkts
Access Links	10 Mb/S
Congested Link	1.5 Mb/S
Tax, Tbx, Txy	10 mS, 30 mS, 10 mS
TCP	Tahoe
Sim Time	60 S
$\log 10(D_{AY} \div D_{BY})$	-0.47 (ratio of 1 : 2.9)

TABLE 3. Simulation Results. This result suggests that TCP Tahoe with drop-tail routers may be somewhat unfair. The TCP on the longer link gets more than its fair share.

Router Queue	Drop incoming when >= 25 pkts
Access Links	10 Mb/S
Congested Link	1.5 Mb/S
Tax, Tbx, Txy	10 mS, 30.7 mS, 10 mS
TCP	Tahoe
Sim Time	60 S
$\log 10(D_{AY} \div D_{BY})$	+0.51 (ratio of 3.3 : 1)

TABLE 4. Simulation Results. A very slightly different propagation delay (29.8 instead of 30.0 mS) creates vastly different results. This time, the TCP on the shorter link gets more than 3 times as much as the TCP on the shorter link.

Clearly, either of the two results in isolation would be misleading. Even considering both results is not very illuminating, except that it is clear that the system is highly sensitive to slight parameter variations.

We now consider a number of ways of adding randomness to experimental setups for measuring network performance. In each subsection below, we show the distribution of results obtained by randomizing one aspect of the experimental setup. At the end, we consider the cumulative effect of using all the techniques. The parameters preoccupied are: propagation delay, queue size, receiver window size, and relative start time of the connections. Perturbing propagation delay is the most effective technique in this instance; however since all of these parameters are arbitrary within a small range, they should all be perturbed for maximum generality.

The goal in perturbing parameters is to not change the simulation configuration very much. For instance the configurations in Tables 3 and 4 are the same for all practical purposes, but a simulation with Tbx = 100 mS would not be. Thus, we will only modify parameters by a few percent.

5.2.1 Perturbing Propagation Delay

Elaborating on the sketch results above, we show in this subsection that TCP fairness experiments exhibit extreme sensitivity to small changes in propagation delay.

For practical as well as theoretical reasons, results that are highly dependent on the exact value of the propagation delay must be considered suspect. A somewhat specious argument why propagation delays should not be fixed too accurately is as follows. The speed of signal propagation in twisted pair network links (such as 10Base-T) is variable.

The Category 5 specification allows as much as 1.2% change in propagation delay over the rated temperature range [8], so it is possible that results on the same physical network would be irreproducible during different seasons.

Figure 11 plots the fairness between two TCPs in the configuration from Table 3, varying only Tbx. The graph confirms the hypothesis that fairness changes greatly due to small variations in the propagation delay. This graph is qualitatively similar to one produced by Floyd and Jacobson [23], and corroborates their results on a similar network topology, but with different parameters.

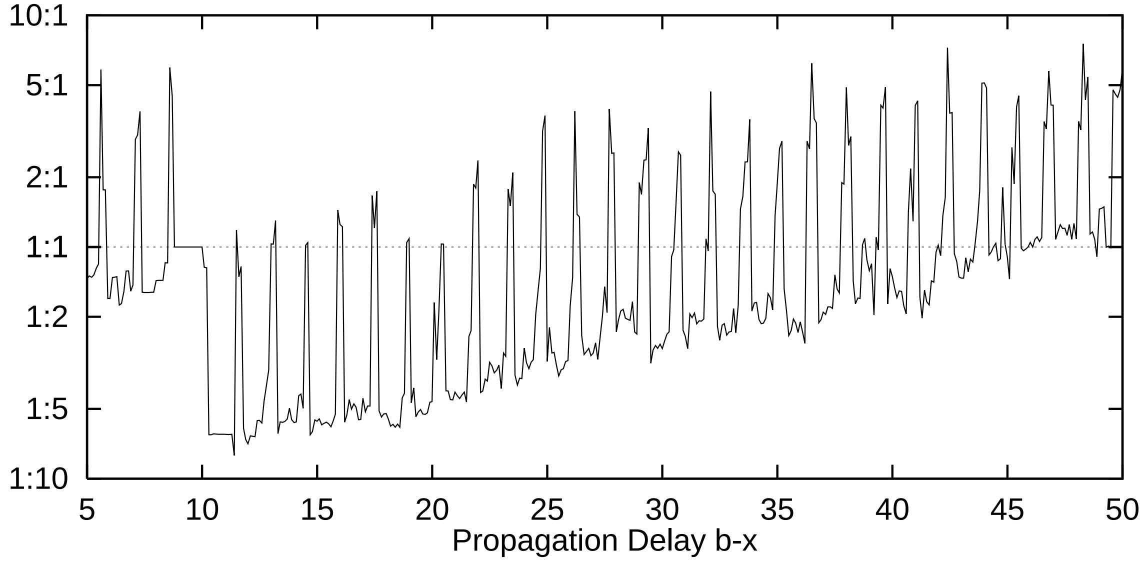


FIGURE 11. Fairness vs. Propagation Delay. Fairness changes dramatically over a range of propagation delays as small as 1.0 mS. A repeating pattern is evident, presumably due to resonances between the two TCPs at certain round trip times.

To determine the distribution of results expected across small variations in propagation delay, simulations were done with random perturbations of 5% added to the propagation delays for all three links. One thousand simulations were done with independent random perturbations; the results are summarized as cumulative distribution functions. The distribution of fairness results is shown in Figure 12.

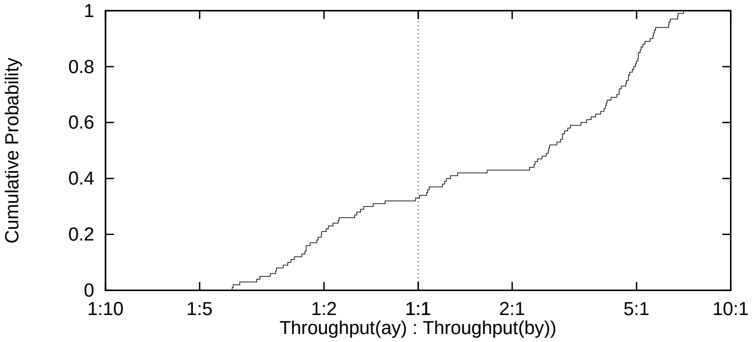


FIGURE 12. Cumulative distribution of fairness with 5% random perturbation in link propagation delays. Other parameters from Table 3..

A number of conclusions can be drawn from the distribution graphs. The worst case unfairness is about 0.85 — that is, a factor of 7:1. The system has a somewhat high chance (67%) of favoring the short connection, but the graph is not as asymmetric as might be expected from the 3:1 different in access link propagation delays.

The above observations show that it is possible to make interesting statements about TCP's fairness properties when the simulation configuration is randomized by perturbing link propagation delays slightly. It was not possible to reliably make such observations when specific values of all these parameters were used, as shown in Tables 3 and 4.

5.2.2 Perturbing Queue Sizes

The queue size at the bottleneck router is another logical candidate for randomization. Here we are considering drop-tail routers with the queue length fixed for each simulation; random dropping strategies require modifications to the congestion control algorithms themselves and are considered later.

The queue length for the configurations presented so far has been 25. If the results are substantially different with a queue length of 24 or 26, that would be worth knowing.

Even the interpretation of the queue length is not completely clear. Does a packet currently being sent get counted in the queue length? Is the overflow check made at the start of a new arriving packet, or at the end? These factors can alter the effective queue length by one or two packets.

Figure 13 shows the effect of randomizing the queue size on fairness.

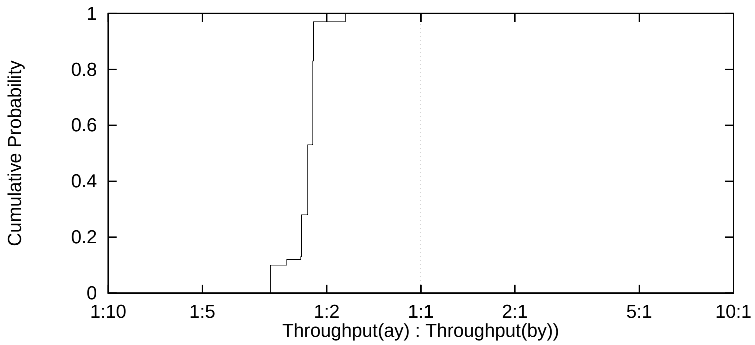


FIGURE 13. Cumulative distribution of log fairness with 5% random perturbation in router queue size. Other parameters from Table 3. The effect is much smaller than that of randomizing the links, and is not a substitute for randomizing propagation delays.

While this technique may be worthwhile in combination with the other techniques, by itself it is not enough to avoid the kind of meaningless results found in Tables 3 and 4.

5.2.3 Perturbing Window Size

TCP maximum window size is an adjustable parameter that is set (in real implementations) before the connection is opened. Hypersensitivity to window size is suggestive of irreproducible results.

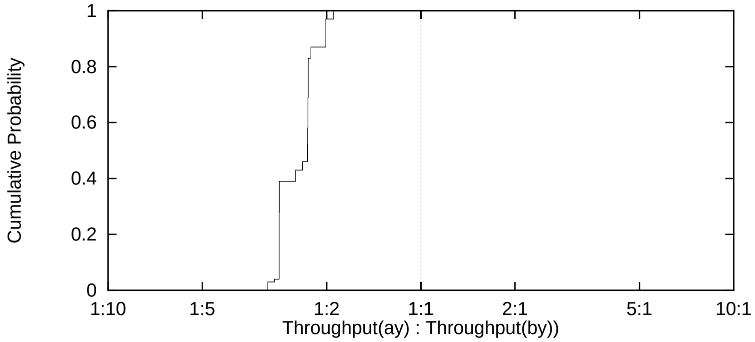


FIGURE 14. Cumulative distribution of log fairness with 5% random perturbation in window size. Window size is quantized to a multiple of the packet size (512 bytes.)

5.2.4 Perturbing Start Time

Figure 15 shows the effect of adding a uniformly distributed random variable with standard deviation 50 mS (roughly equal to one round trip time) to the start times of one of the connections. While the differing start times clearly have an effect, it is not enough to overcome the other deterministic effects. Presumably, while differing start times can affect the simulation slightly in the beginning, after a few seconds this single injection of randomness has been lost and the system reaches a stable state.

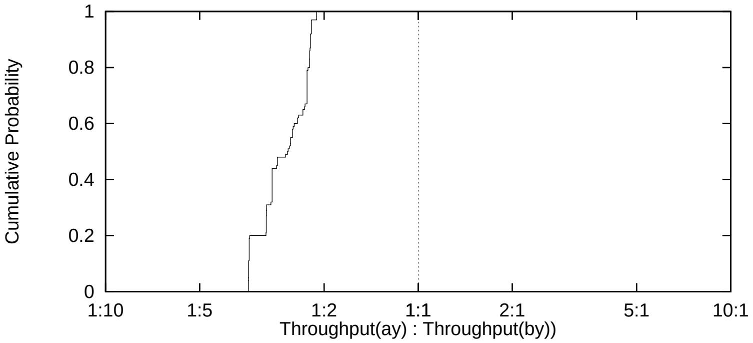


FIGURE 15. Cumulative distribution of log fairness with 50 mS random offset in start time of one connection.

5.2.5 Multiple Techniques

The techniques described above can be used in combination. Figure 16 shows the effect of varying all the configuration parameters together.

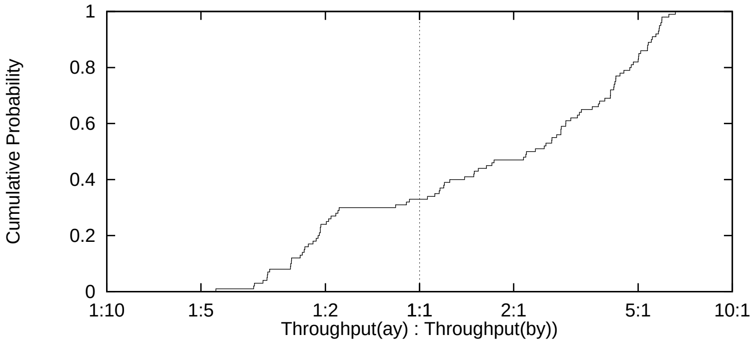


FIGURE 16. Cumulative distribution of log fairness with 5% random perturbation in all parameters. Link propagation delays, queue sizes, start time offsets, and window sizes are all randomly perturbed by 5%. Distribution is only slightly wider than that shown in Figure 12, where only propagation delays are varied.

The distribution of results obtained from varying all the configuration parameters is only slightly broader than that obtained by varying just the link propagation delays. Perturbing the queue sizes, window sizes, and start times each exposed some of the range of potential variation in results, but not nearly as much as was found by varying propagation delays.

Although it appears from the results presented here that varying propagation delays is the only worthwhile technique, this may not apply to other systems. In this system, the major cause of hypersensitivity was the precise timing relationships that caused one TCP to consistently choose a worse time to increase its window than the other, and thus the system is hypersensitive to propagation delay. However, it is possible that different systems would be sensitive to other parameters, and so it is recommended that all available configuration parameters be perturbed.

5.2.6 Summary

This section has developed techniques for discovering the range of performance results obtained across a small range of perturbations to network configurations. For simple situations such as the simple fairness configuration described here, 5% changes to configuration parameters can affect the results by more than five orders of magnitude.

We conclude that any conclusion drawn from a single simulation of such a network is essentially meaningless. Analyzing the distribution of results has shown the amount by which single simulations can be misleading, and has also shown the range of potential results that might be encountered on real networks similar to the idealized network.

5.3 Randomizing Network Activity

We have addressed some ways in which the simulation configuration can be perturbed somewhat. Now we turn our attention to ways in which the network activity itself can be randomized.

All of these techniques described here to randomize network activity have been used in published measurements. The new contribution here is to demonstrate a procedure for determining the right amount of randomization to add. It is important to be able to estimate the necessary level of random network activity correctly. Too little can lead to meaningless results; too much can change the character of the network traffic being studied.

The sensitivity analysis methodology developed above will prove useful in estimating reasonable amounts of randomization. We will be able to determine how much random network activity must be added to effectively isolate the system from spurious determinism by measuring the sensitivity of the system to small configuration changes with varying amounts of random network activity.

5.3.1 Randomizing Traffic Sources

When testing network control algorithms (TCP, router scheduling policies,) a traffic source must be supplied. This traffic source is external to the algorithm in question, and in some cases randomness can be added to the traffic model to explore a small range of possibilities.

In the network configuration described here, the traffic sources are greedy — that is, they always send as fast as they can. Thus, once they have started, no random decisions are possible. Start time was varied as a parameter in the previous section, and had a modest effect.

5.3.2 Adding Random Background Traffic

A technique used by some researchers to avoid misleading deterministic effects is to add small amounts of background traffic. Floyd & Jacobson report that traffic phase effects are largely eliminated when 15% of the packets through a gateway are 40-byte packets sent at random, exponentially distributed intervals [23]. Traffic from interactive telnet sessions is claimed to be similar to this — thus this source of randomization also reflects a realistic real-world phenomenon.

In this section we extend Floyd & Jacobson’s work by quantifying the amount of background traffic needed. The crucial question is how much background traffic is enough to eliminate systematic errors due to spurious determinism?

Background traffic was added in the form of small (40-byte) packets sent as part of TCP sessions, with exponentially distributed interarrival times (i.e. Poisson traffic sources.) The amount of background traffic is specified as a fraction of the link that it uses; thus in the simple fairness configuration with 1.5 Mb/s links, 1% background traffic corresponds to an average rate of 15 Kb/s.

Background traffic is injected at nodes A, B, and X in the simulation configuration, all terminating at Y (see Figure 10.) A characteristic of the simulation package used is that this kind of traffic must use TCP connections. To prevent TCP from throttling the background traffic, 10 TCP sources were placed at each node. Thus, there are 30 background TCP connections competing with the 2 connections of primary interest. Analysis of the traces showed that TCP’s congestion control mechanisms did not significantly change the nature of the background traffic from the source’s Poisson behavior.

Figure 17 shows the effect of increasing background traffic, while keeping configuration parameters constant. With no traffic, there is a large systematic error that gives a median log fairness value of -1.0. (With slightly different configuration parameters, the log fairness value could have been quite different.) With increasing amounts of background traffic, the median log fairness shifts right as far as +0.5, and then back towards the center as the background traffic reaches 20%. Further analysis presented below suggests that as much as 20% background traffic may be necessary to avoid systematic errors.

“Enough” background traffic can be defined as the level of background traffic at which the distribution of fairness results is relatively independent of small perturbations in the network configuration.

For a given amount of background traffic, the following algorithm was used to determine whether it is sufficient to isolate the results from the exact choice of network configuration parameters.

```
repeat N times {
     Choose a randomly perturbed configuration
     repeat M times {
          simulate (yields a single log fairness result)
     }
     report distribution of above log fairness results
}
```

Measuring Configuration Sensitivity

The algorithm reports *N* results, each the distribution of simulating *M* times with a given perturbed configuration. We consider that if the *N* distributions reported by the algorithm are similar, then the system is insensitive to small changes in the configuration. In the results presented here, *N* and *M* were both set to 100. Because it is hard to plot the distribution of a set of distributions, the median of each set of M simulations is taken, and the distribution of the medians is graphed.

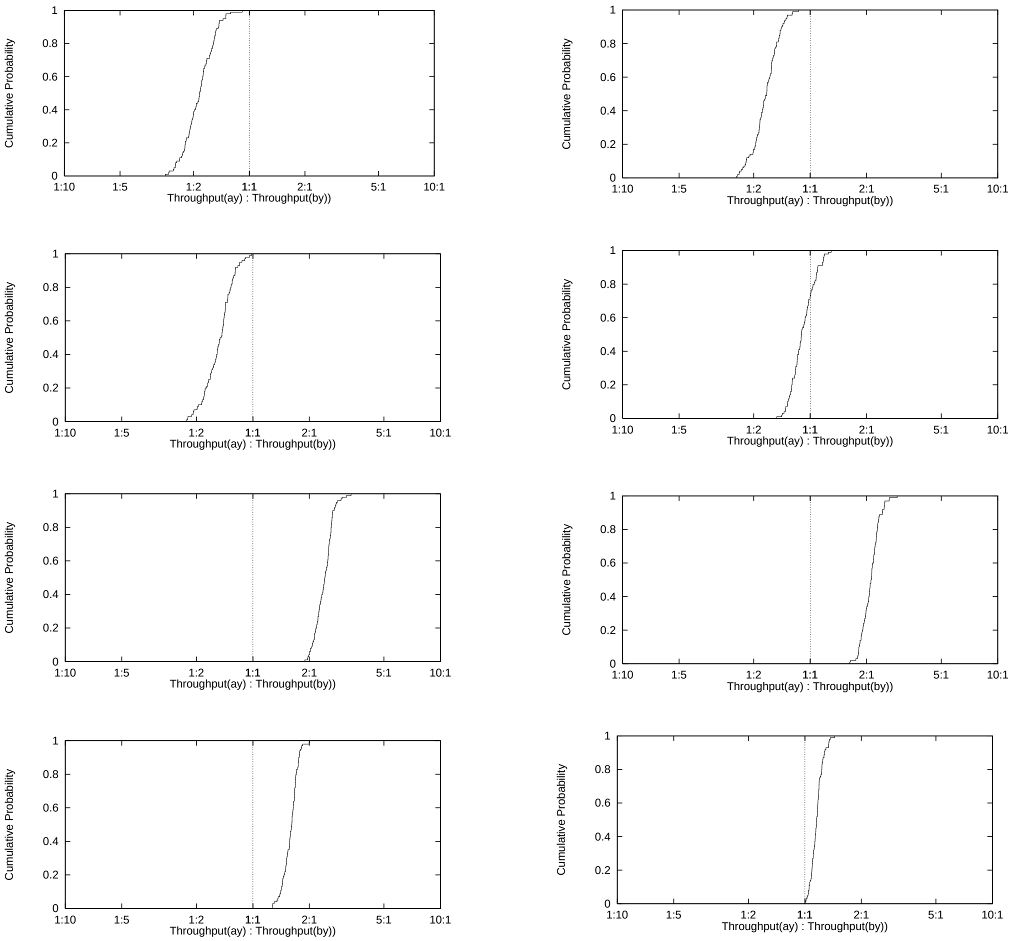


FIGURE 17. Adding increasing amounts of background traffic (40-byte packets). Graphs, left to right and top to bottom, have 0.1%, 0.2%, 0.5%, 1%, 2%, 5%, 10%, and 20% background traffic (as a fraction of bytes, not packets) added. The distribution is narrow at all amounts of randomness. The center of the distribution shifts to the right with up to 2% random traffic, and then shifts back toward the center.

The distributions for several amounts of background traffic are shown in Figure 18, and the amount of variation is quantified in Table 5. It appears that 5.0% background traffic is sufficient to reduce the variance due to configuration perturbations to nearly as small a value as 20% background traffic. However, with 2.0% background traffic the results are more than twice as sensitive to configuration perturbations. Thus, we conclude that 5.0% background traffic is a good practical choice.

It is noteworthy that even 0.05% background traffic dramatically reduces the range of unfairness. With no background traffic, log fairness values of +/- 2.0 are reasonably likely; with 0.05% background traffic the range of log fairness drops to +/- 1.0. 0.05% background traffic corresponds to 140 packets over the 60 seconds of simulated time.

Floyd & Jacobson's [23] study recommended values for the background traffic that correspond to about 0.5% of the capacity of the bottleneck link. This value is an order of magnitude lower than the one suggested by the results presented here. As shown in the third graph in Figure 18, a background traffic level of 0.5% still allows 5% changes in configuration parameters to generate a variation in log fairness with standard deviation 0.474, or a factor of 3.0 in fairness.

Background Traffic Level	Sensitivity (Std. dev. of log fairness with 5% configuration parameter variation)
none	0.453
0.05%	0.427

TABLE 5. Decrease in sensitivity to configuration perturbations with increasing background traffic level. Summarizes information in Figure 18.

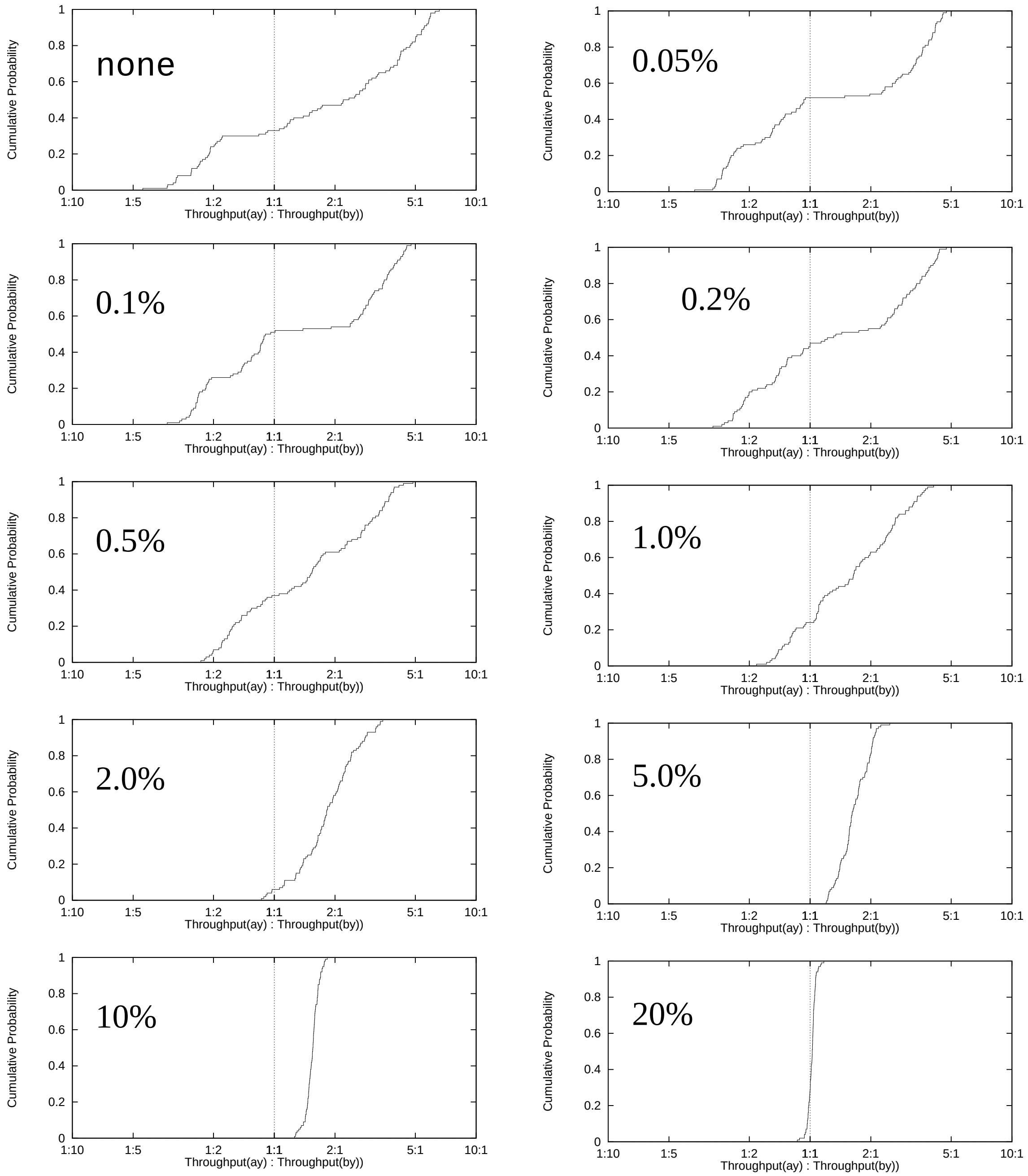


FIGURE 18. Decrease in sensitivity to configuration perturbations with increasing background traffic (40-byte packets). Each graph shows the cumulative distribution of median values for log fairness over the range of perturbed configurations. Graphs show different values for the amount of background traffic (large label), increasing left to right and top to bottom.

Background Traffic Level	Sensitivity (Std. dev. of log fairness with 5% configuration parameter variation)
0.1%	0.403
0.2%	0.374
0.5%	0.314
1.0%	0.238
2.0%	0.143
5.0%	0.073
10%	0.032
20%	0.020

TABLE 5. Decrease in sensitivity to configuration perturbations with increasing background traffic level. Summarizes information in Figure 18.

5.3.3 Randomizing Processing Time

Floyd and Jacobson [23] also suggested the addition of random delays to the packet processing time as a way of avoiding phase effects. We quantify the effect of this approach here, similarly to the previous section, by determining how much random delay must be added to make the system relatively insensitive to small configuration perturbations. This is modeled by delaying each call to `tcp_output` by a random time, uniformly distributed in some range. Here, the range is expressed as a fraction of the time required to send a a full-size data packet on the output link. A value larger than 1.0 would limit the maximum rate of the TCP to less than the link rate, thus we only show results for random delays with a maximum value of one packet transmission time.

Figure 19 shows the decreasing sensitivity to configuration perturbations with increasing amounts of random delay. When a large fraction of a packet transmission time is added, sensitivity to the configuration drops sharply. Table 6 shows the sampled standard deviation of log fairness as a function of amount of random processing delays.

Random Processing Delay (% of packet transmission time)	Sensitivity (Std. dev. of log fairness with 5% configuration parameter variation)
none	0.453
20%	0.362
40%	0.250
60%	0.148
80%	0.079
100%	0.057

TABLE 6. Decrease in sensitivity to configuration perturbations with increasing random delays added to host processing time. Summarizes information in Figure 19.

5.3.4 Summary

Adding random background traffic and adding random processing delays have both been shown to reduce the sensitivity of a simple test network to small configuration changes. The amounts of randomness needed are small: either 5% background traffic, and one packet transmission time are sufficient to reduce the variation in log fairness due to configuration perturbations from multiple orders of magnitude to several percent.

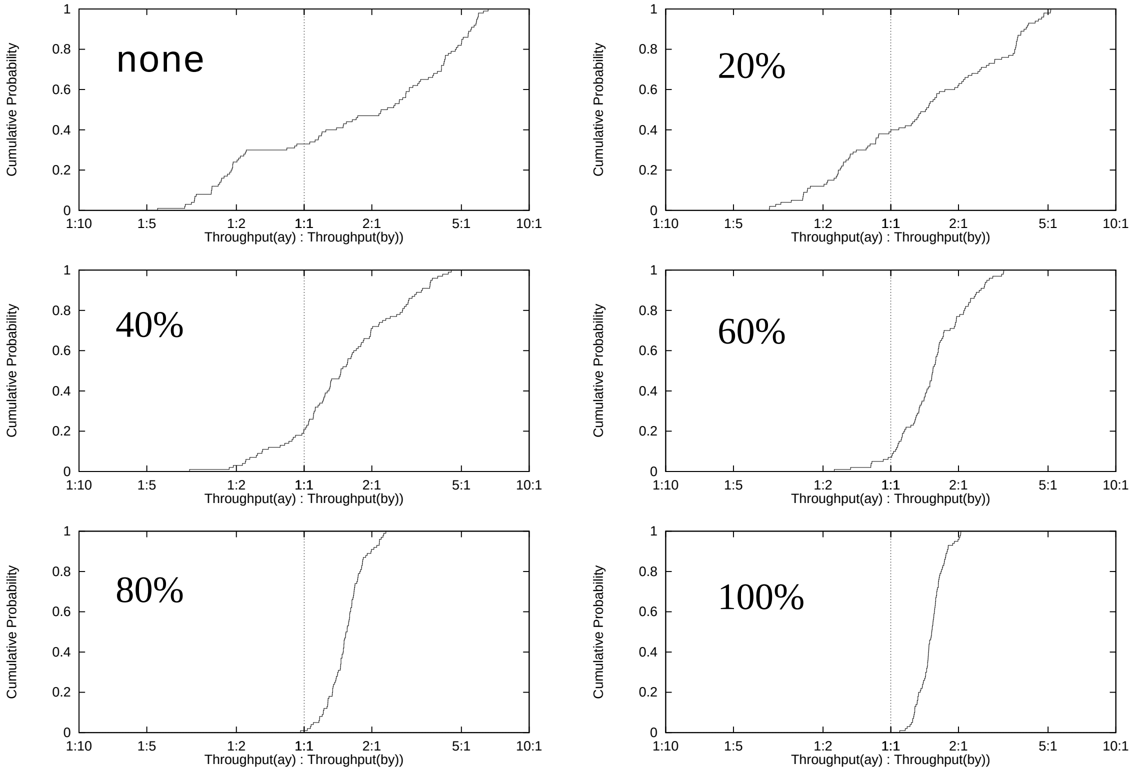


FIGURE 19. Decrease in sensitivity to configuration perturbations with increasing random processing delays. Each graph shows the cumulative distribution of median values for log fairness over the range of perturbed configurations. Graphs show different values for the amount of added random processing delay (large label), increasing left to right and top to bottom.

The techniques described here are analogous to effects found in the real world, although they are not very exact models. Low level background traffic probably exists in most real-world networks. Processing delay in hosts can vary, depending on interrupt conflicts, cache effects, collisions in multiple access protocols such as Ethernet, and many other phenomena. Neither model is very realistic: background traffic is not Poisson and processing time is not uniformly distributed. However, there is no particular reason to believe that more realistic models of background traffic or processing delay would yield more meaningful results. However, 5% background traffic seems like a more realistic assumption than 100% random processing delay.

We return to the question that the network configuration is designed to test: does TCP share bandwidth fairly when the competing sessions have different-length access links? Table 7 shows the average and standard deviation (across configuration perturbations) of the median (across many tests) of log fairness. Both techniques give similar values, and we conclude that in this configuration, a reasonable value for log fairness is +0.23. This corresponds to the short connection getting 1.7 times as much bandwidth as the long connection.

Methodology	Log Fairness
100% random packet delay	+0.221 +/- 0.059
5% background traffic	+0.244 +/- 0.082

TABLE 7. Log fairness values with standard deviations over perturbed configuration space for two recommended methodologies.

5.4 Randomizing Network Protocols for Measurability

The first subsection of this chapter showed that randomly perturbing network configurations could expose the range of spurious determinism that could lead to systematic errors. The second section showed how random effects, often present in the real world, could be included in simulated network models. This section will discuss the design of network control algorithms with substantial built-in randomness, and show that such protocols are inherently more amenable to making robust, reproducible, and broadly applicable measurements of their performance in any scenario, not just scenarios with carefully constructed background traffic or other forms of injecting randomness.

This section discusses TCP/IP systems. There are three parts to TCP/IP’s network control protocol: the TCP sender algorithm, the TCP ack algorithm, and the router scheduling & packet dropping algorithm. Each has potential for randomization. Conventionally, TCP/IP systems implement the following things deterministically:

- **Packet dropping**. An arriving packet is dropped at a router if and only if the current queue length is greater than some fixed threshold.
- **Timer Expiry**. Berkeley-derived TCP implementations periodically check for timer expiration every few hundred milliseconds. The accuracy of these timers is high enough that two workstations can remain synchronized for long periods of time.
- **Round trip time estimation**. Round trip times are calculated by subtracting two timestamps, both of which are quantized to an integral multiple of the TCP timer granularity, typically 500 mS.
- **Window limits**. TCP will only send a packet if it fits inside the current congestion window, which is measured in bytes. This results in deterministic quantization error equal to half the size of a packet.
- **Window increase/decrease.** TCP increases its window a deterministic amount when data is acknowledged, and decreases it a deterministic amount when packet loss is detected.

Each of these deterministically made decisions can have a large impact on system performance. In this work, we discuss only the first. Randomizing other aspects of TCP/IP appears to be a promising direction for future research.

Random drop and random early detection gateways have been proposed in the literature [22] as a way of improving fairness. Both involve randomizing the decision as to which packets to drop under congestion, but they do it in different ways.

Random drop gateways, rather than dropping newly arrived packets when the queue is full, replace a randomly chosen packet already in the queue in order to make room for a newly arrived packet.

Random early detection gateways drop newly arriving packets with some probability based on the queue length or a smoothed version of the queue length. Typically, no packets are dropped when the queue is less than half full. When the queue is at least half full, incoming packets are randomly dropped with a probability that increases from zero when the queue is half full, to a few percent when the queue is nearly full.

Figure 20 shows how random drop gateways are much less sensitive to configuration perturbation than are conventional drop-tail gateways. The average log fairness is +0.146 with standard deviation 0.0239, which is slightly more fair, and has substantially less configuration sensitivity than the results shown in Table 7 for a drop-tail gateway with 5% background traffic or 100% random packet delay.

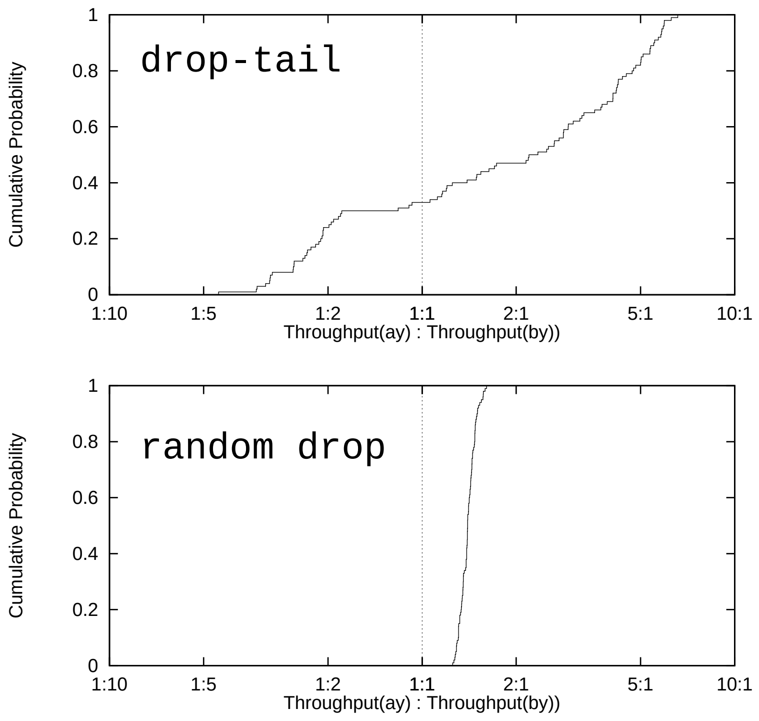


FIGURE 20. Random-drop gateways make the system less sensitive to configuration perturbations. Drop-tail gateway is never fair within about an order of magnitude, but random drop gateway is consistently close to fair, with the shorter link having a slight advantage. Each graph shows the cumulative distribution of median values for log fairness over the range of perturbed configurations. Top graph is (deterministic) drop-tail; bottom graph is random drop.

5.5 Conclusions

Extreme sensitivity to small perturbations in network configurations is an indicator of irreproducible, non-robust results. The first section of this chapter developed several methods for perturbing networks, and derived estimates of how much perturbation was necessary to elicit the full range of behaviours from almost identical systems. Adding 5% randomness to link lengths, start times, queue sizes, and window sizes is recommended as a starting point. It may be that some systems require more randomness than this; experimentation may be required to find the best value. Not too much precision will be lost in measurements of systems which require less randomization than this.

Adding random traffic to the network was shown to be a fruitful technique for making networks less sensitive to configuration changes. Adding 5% background traffic, or 100% random packet delay, caused the system to become much more fair, and the results much more reproducible, robust, and broadly applicable.

Although deterministic networks can be measured fairly accurately by carefully adding background traffic and/or random packet delays, it was shown that networks that are inherently random can be measured at least as accurately without any extra effort. With a random drop gateway, not only did fairness increase slightly, but sensitivity to configuration perturbations decreased.

The advantage of building randomized network protocols is clear. Randomized networks have been shown to be fair, and measurements of them to be broadly applicable, with or without carefully constructed background traffic.

In real-world WANs, amounts of background traffic adequate to avoid the repeatable phase effects evident without any background traffic may exist. On LANs, however, no background traffic may exist for long periods of time, and thus randomized networks may confer a major practical advantage for LAN applications.

Random drop gateways are a convenient way to add randomness to a network, and they also have performance advantages as described in the next chapter. However, the benefits of randomized networks can also be achieved by modifying the congestion control algorithms in the end-system TCP implementations.

Chapter 6
Randomizing TCP/IP for Improved Performance

The preceding chapter demonstrated that randomization could make TCP/IP less sensitive to small configuration changes, and therefore easier to make robust performance measurements of. In this chapter, it is shown that randomized versions of TCP/IP can perform significantly better, on average, than conventional implementations.

It has been shown previously by Floyd and Jacobson [22] that adding randomization to the packet dropping algorithm at routers can make starvation (where one connection gets much less than its fair share) much less likely. Here we corroborate these results by showing that the average transfer time is significantly reduced for random-drop gateways.

In addition, this chapter describes some new techniques for randomizing host implementations, and shows that benefits similar to randomizing router implementations can be achieved. This may be an attractive alternative to changing router implementations, both because it is more in keeping with the “end-to-end” philosophy behind TCP/IP, and because in practice host implementations are easier to change. One proposed modification, that requires only small modifications to a conventional TCP implementation, is shown to improve performance almost as much as random-drop gateways.

6.1 Introduction

The simulation configuration used throughout this chapter is shown below in Figure 21. A number of senders all compete to transmit through a bottleneck link with rate proportional to the number of senders.

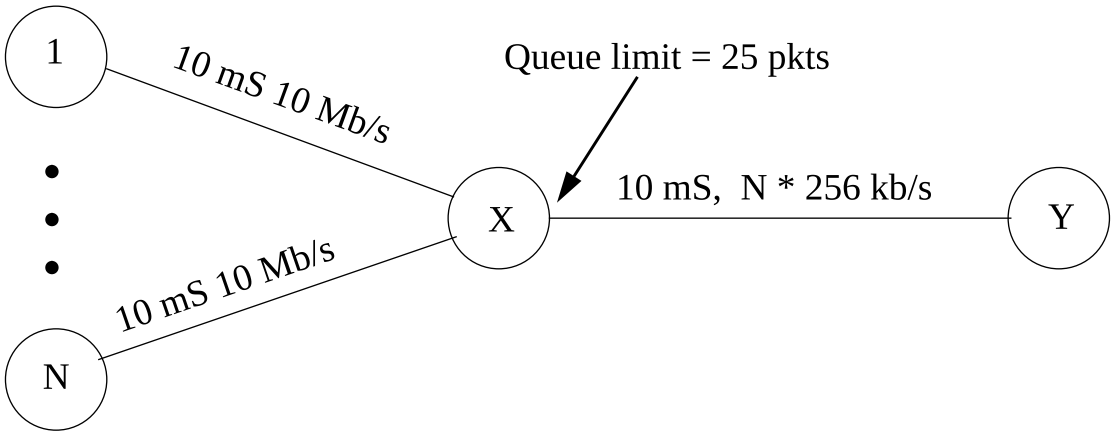


FIGURE 21. Simple Performance Testing Configuration. N TCP sessions are set up: A_1 to Y through A_N to Y. All senders are greedy. Baseline configuration parameters are shown.

The delay time metric is a measure of the average time spent waiting for network transfers to complete. In order to have fixed-time simulations, it is computed according to EQ 4. For each sender, the simulation time is divided by the amount of data it was able to transfer (measured by the largest acknowledgment sequence number received by the sender) and the average of these numbers is taken. The metric reflects the expected waiting time of a sender in a large population to send a document, in seconds per packet worth of document size. Thus, it corresponds directly to how long users must wait for results.

$$\frac{1}{N}\sum_{i=1}^{N}\frac{\text{sim time}}{D_{iY}} \qquad \textbf{(EQ 4)}$$

Lower values of this metric are better. The metric tends to give better scores to algorithms that share the link fairly, because this minimizes the sum of the inverse bandwidth terms. This corresponds to the fact that an unfair system reduces some users waiting time a little, but increases others by a lot.

6.2 Randomizing Window Computations at Sender

In the TCP implementation, there are several computations to compute the window size based on acknowledgment arrivals and detected losses. In conventional implementations, all these computations are simple deterministic formulas. This section explores the effect on performance of adding randomization at various points in the computation.

All the changes described here require no architectural changes to the code. They simply involve replacing non-randomized formulas with randomized ones.

6.2.1 Poisson Window Increase

In TCP Tahoe and most others, the congestion window is increased according to the following algorithm during congestion-avoidance mode. `Cwnd` is the congestion window, measured in bytes, and MSS (maximum segment size) is the usual size of a data packet.

```
When normal in-sequence acknowledgment received:
     cwnd = cwnd + $MSS^2$/cwnd;
```

Conventional (Tahoe) Window Increase Algorithm

Since the throughput of the system is approximately one congestion window of data per round trip time, the net effect is to increase the window by one packet per RTT.

Although cwnd is represented in bytes, under normal conditions it is only the corresponding number of packets that affects the sending algorithm. Thus, rounding cwnd down to the nearest multiple of the segment size (typically 512 bytes) would not affect packet transmission.

In the standard algorithm, the congestion window increases at regular, deterministic intervals. In scenarios with a small number of TCPs, increasing the congestion window is usually the direct cause of packet loss. Thus, deterministic phasing between the window increase algorithms of two competing connections is a potential cause of persistent unfairness and poor performance.

The window increase algorithm can be randomized in various ways. First, we experimented with increasing it in increments of the MSS, at exponentially-distributed intervals, but at the same average rate. The algorithm is as follows:

```
When normal in-sequence acknowledgment received:
     if (random(0..cwnd) <= MSS)
          cwnd = cwnd + MSS;
     }
```

Poisson Window Increase Algorithm

That is, with probability MSS/cwnd, the congestion window is increased by one packet. We refer to this scheme as Poisson window increase. This can be implemented very efficiently.

Results are shown in Figure 22 as cumulative distributions of total delay time, using the simulation topology from the beginning of this chapter except that both access links are 10 mS long.

6.2.2 Randomized Slow Start

A potential place to add randomness to TCP’s congestion control algorithm is by picking a random rate of increase each time the connection enters the slow-start phase. Experiments were made with these modifications, but no statistically significant effects on performance could be discerned.

6.2.3 Randomized Window Limit

The improved performance of the Poisson Window Increase scheme suggests that smooth increases in the number of outstanding packets may not be optimal. An intuitive reason for this is that by increasing the window smoothly, a TCP can maintain a full queue at a router for a long period of time, which tends to ensure large numbers of lost packets if another TCP recovers from timeout and begins slow-start.

Sending packets in somewhat irregular bursts may cause individual packet losses before he router queue becomes completely full, allowing the sender to detect congestion and recover using TCP’s fast recovery mechanism. In effect, randomness added to the window size can simulate the effect of Random Early Detection [22], by causing occasional packet drops before the queue is full.

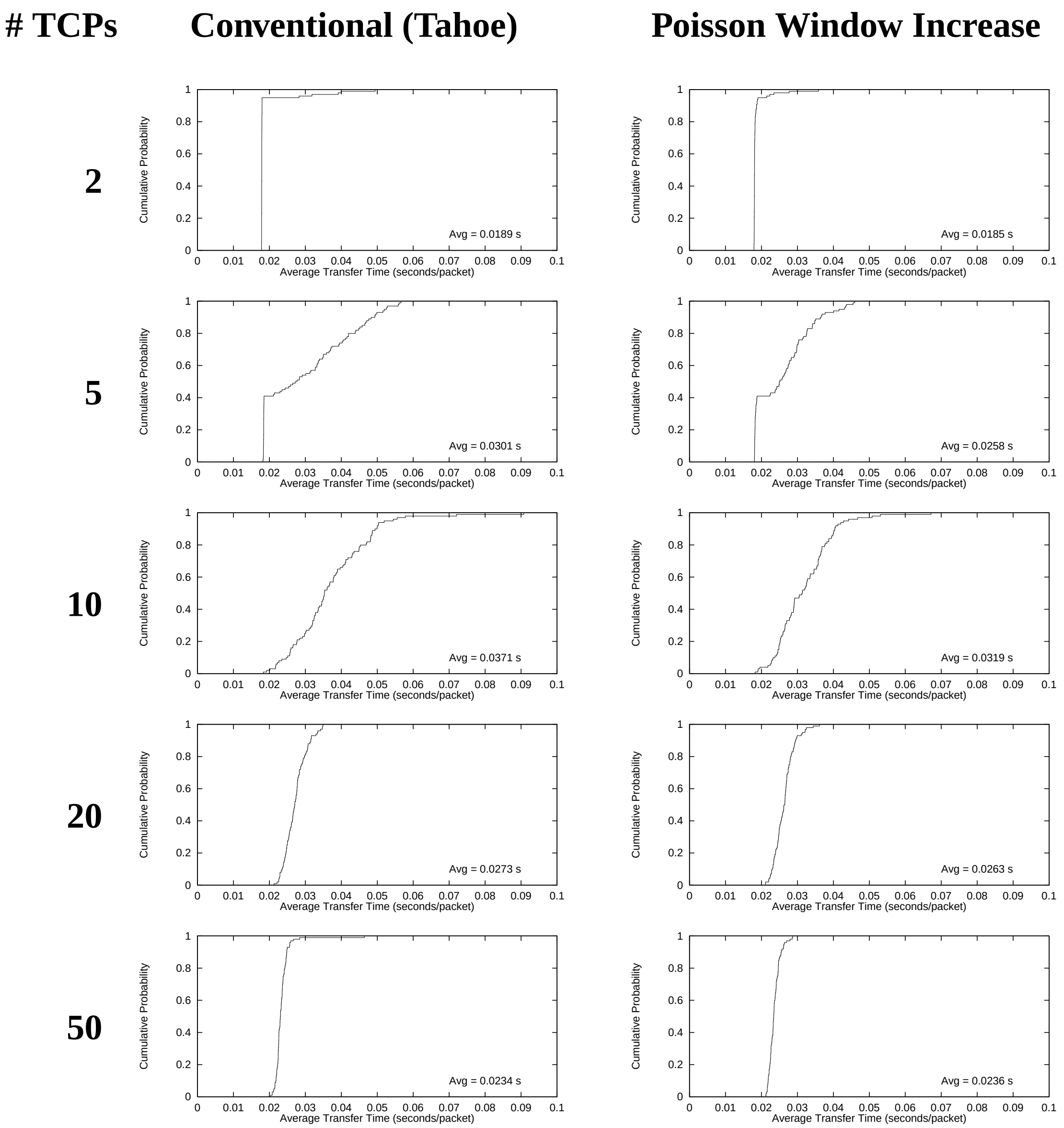


FIGURE 22. Performance Results: Poisson Window Increase Algorithm for 2, 5, 10, 20, and 50 TCP connections. Graphs show cumulative distributions of total transfer time across 100 simulations, with 5% random parameter variation and different seeds for the random number generators. In each row, the graph with the line farthest left has the best performance (lowest transfer time.) Average transfer time for (reported in lower right corner of each graph) drops by as much as 25% at lower numbers of TCPs. With 50 TCPs, performance improvement is only 2%.

To experiment with this system, the sending algorithm was modified. The Tahoe algorithm is essentially as shown below.

```
When in-sequence ACK received:
     w=cwnd
     w=min(w,wnd)
     while (data available && unacked_data + MSS <= w) {
          send next data packet
     }
```

Conventional (Tahoe) Sending Algorithm

The expression `unacked_data` is calculated as the difference between the starting sequence number of the next data packet to be sent, and the largest received ACK sequence number. It is increased by MSS every time a data packet is sent. `Wnd` is the window limit provided by the receiver, which is large enough that it does have any effect.

To add some random noise to the window, the following modified algorithm was used. It adds a uniform random variable to the current window every time the system is about to send one or more packets.

```
When in-sequence ACK received:
     w=cwnd + MSS*random(-wnd_noise .. +wnd_noise)
     w=min(w,wnd)
     while (data available && unacked_data + MSS <= w) {
          send next data packet
     }
```

Noisy Window Sending Algorithm

A range of values for wnd_noise were simulated, and the results are presented in Figure 23. Although significant reductions in total delay are achieved with multiple packets of added randomness, there are likely to be undesirable effects with small windows.

To add some burstiness to the window, the following modified algorithm was used. The modified algorithm uses a window that is larger by one packet, with a probability of bump_prob. The intent is to provoke a loss before the router buffer becomes persistently full. This scheme can be used alone, or with the Poisson Window Increase scheme.

```
When in-sequence ACK received:
     w=cwnd + ((random(0..1) < bump_prob) ? MSS:0)
     w=min(w,wnd)
     while (data available && unacked_data + MSS <= w) {
          send next data packet
     }
```

Bursty Window Sending Algorithm

A range of values for bump_prob were simulated, and the results are presented in Figure 24.

6.3 Randomizing Gateway Drop Policies

Ample evidence has already been presented [22] that random drop gateways improve fairness, and therefore should exhibit lower average transfer times. In this section we present the results of simulations that corroborate these results, mainly for the purpose of allowing a side-by-side comparison with the performance achievable by adding randomization to the sender implementation.

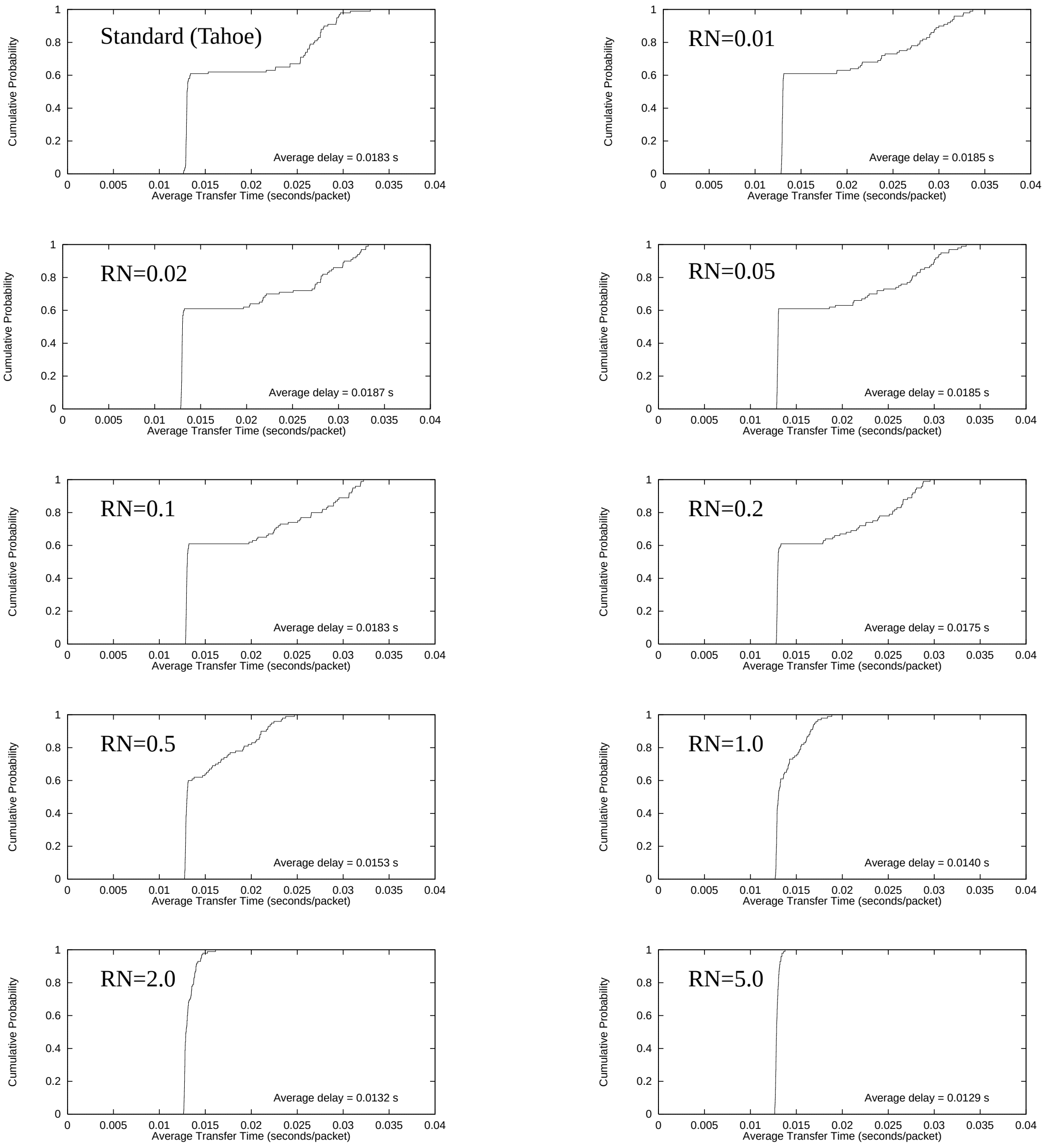


FIGURE 23. Performance Results: Noisy Window Sending Algorithm with 2 connections and varying amounts of noise. Benefit is small below `wnd_noise` of 0.5, but substantial performance increases are seen at larger values. However, large values of wnd_noise probably have undesirable effects that are not evident here.

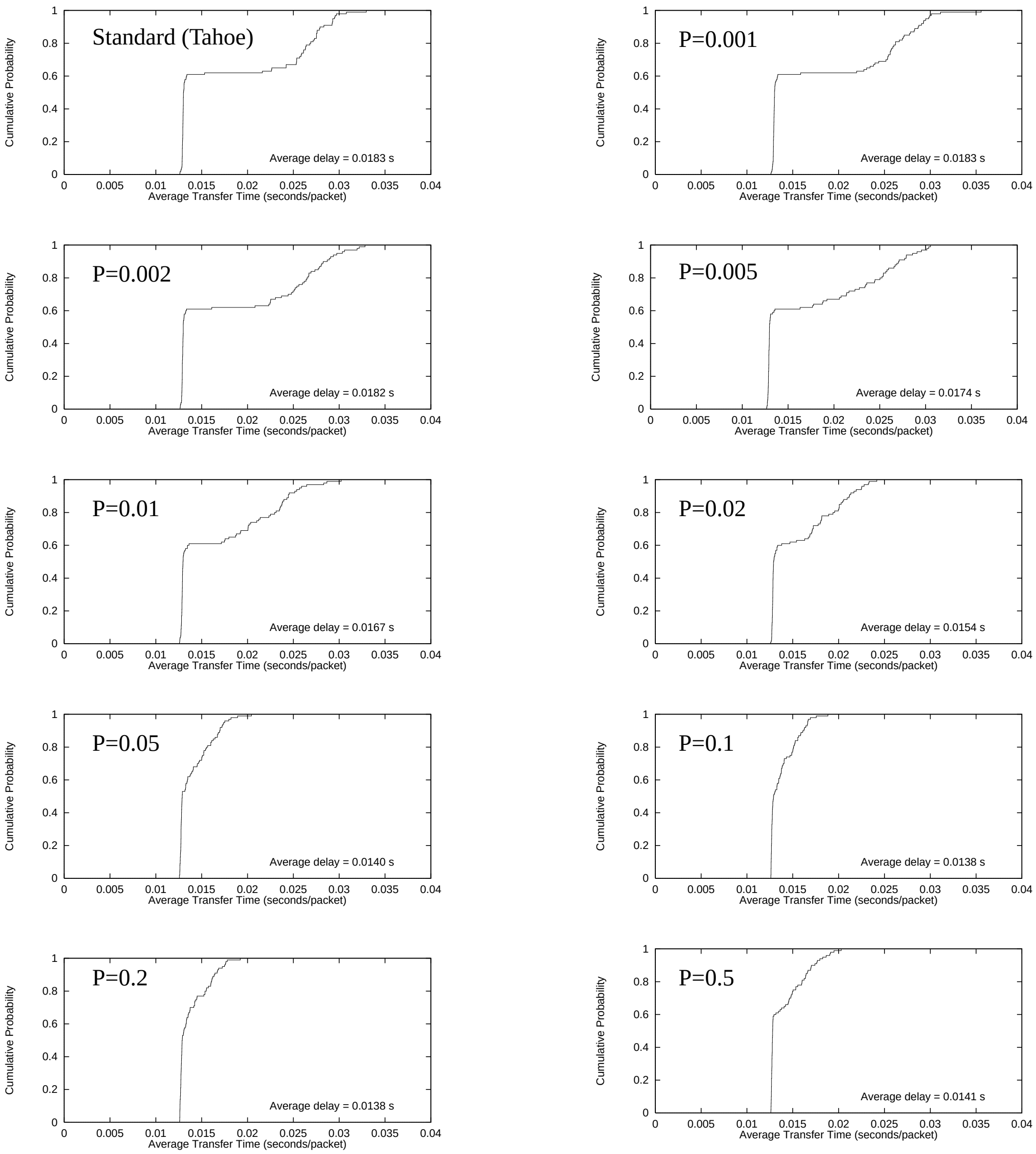


FIGURE 24. Performance Results: Bursty Window Sending Algorithm. Probability of using larger-by-one window increases left-to-right, top-to-bottom. Best results are obtained with P=0.1 or P=0.2. Probabilities less than 0.01 have little effect. Best performance increase is 25%, somewhat better than the Poisson Window Increase Algorithm.

Random-drop gateways differ from standard FIFO gateways in the way they choose the packet to be dropped. A conventional gateway will only buffer a finite number of packets; when a packet arrives and the queue is full, the arriving packet is dropped. A random-drop gateway instead always queues received packets, but drops a randomly chosen packet from the queue if the queue is full.

Using random-drop gateways improves average transfer time significantly. Figure 25 shows how the distribution of average transfer time differs for random-drop configurations.

6.4 Comparison of All Randomized Schemes

This section presents overall performance comparisons for each of the schemes previously described in this chapter. For the Bursty Window scheme, the value of P used is 0.1, which the results in Figure 24 indicate to be the best choice. For the Noisy Window scheme, the amount of noise is 2.0 packets.

Figure 26 shows performance results for the various schemes across a range of numbers of competing connections. Random-drop gateways are the clear winner. They perform near the theoretical maximum performance for all numbers of connections.

6.5 Conclusions

The results presented in this chapter show that both randomizing the window increase algorithm in the sender and using random drop gateways produce substantial performance benefits of up to 37% reduction in average transfer time. The improvements are more significant with smaller numbers of connections.

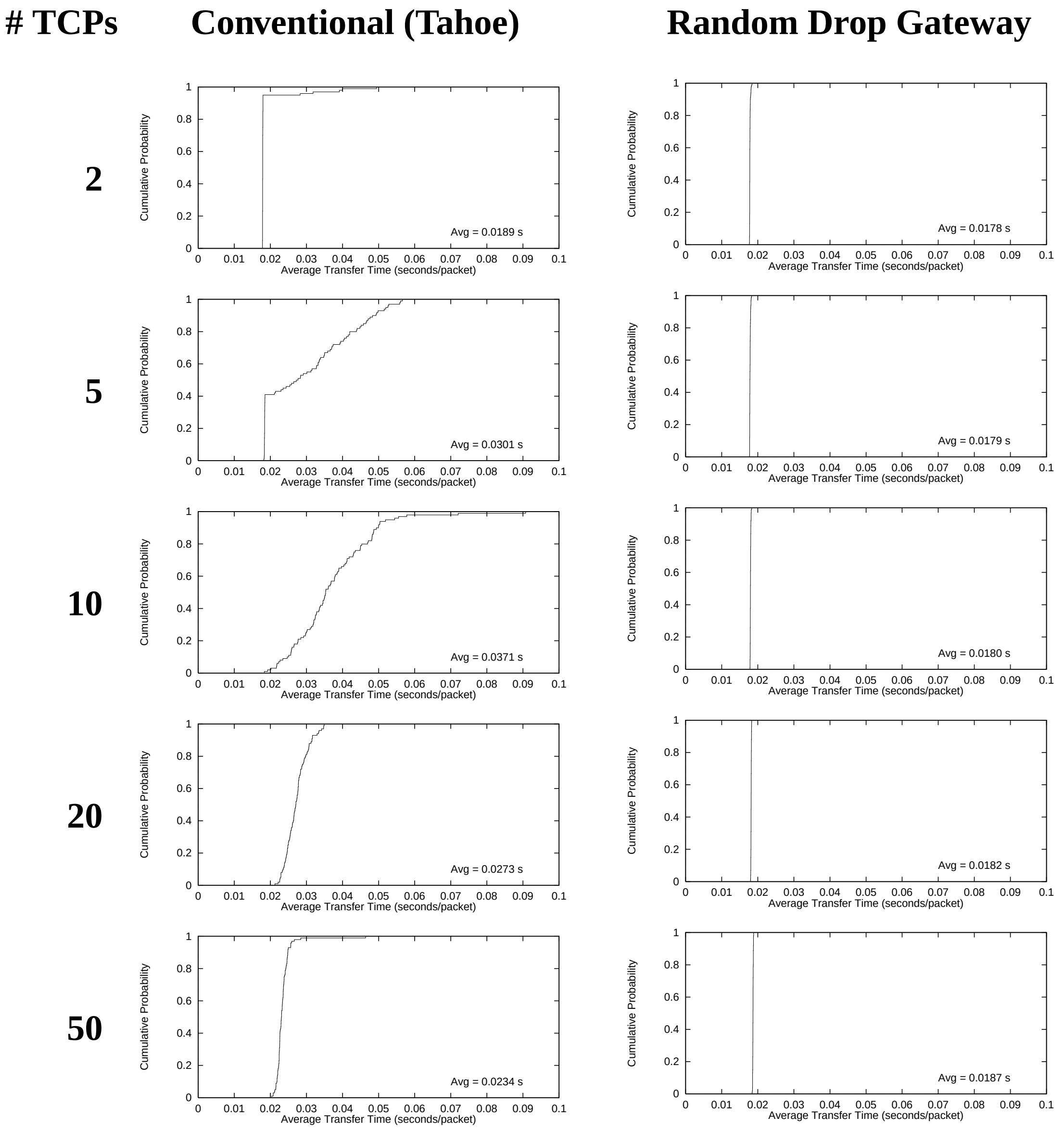


FIGURE 25. Performance Results: Random Drop Gateways for 2, 5, 10, 20, and 50 TCP connections. Graphs show cumulative distributions of total transfer time across 100 simulations, with 5% random parameter variation and different seeds for the random number generators. In each row, the graph with the line farthest left has the best performance (lowest transfer time.) Average transfer time for (reported in lower right corner of each graph) drops by as much as 37% on average. Improvement is larger with smaller numbers of connections.

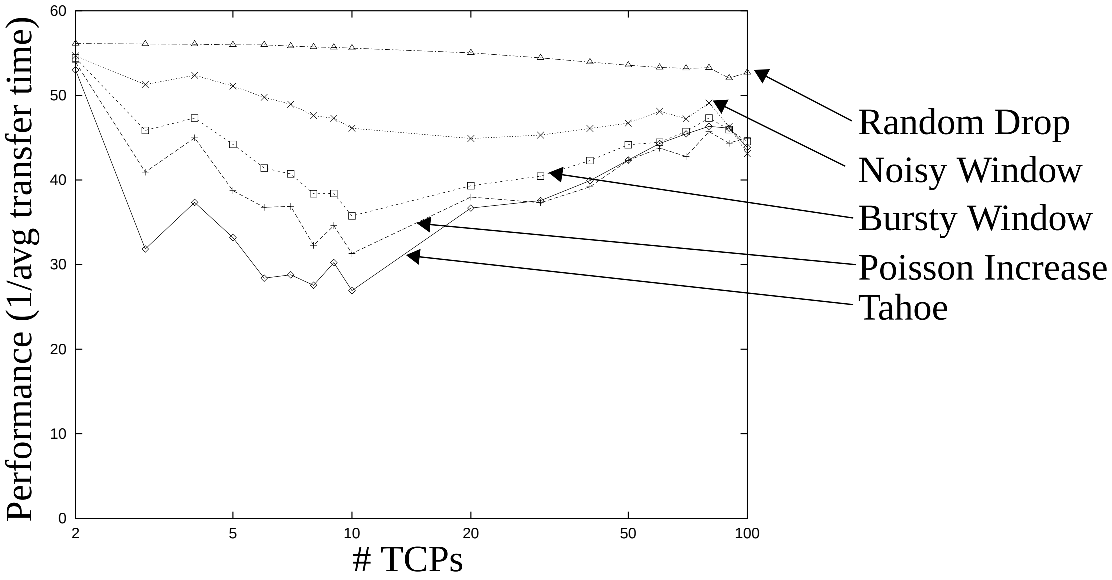


FIGURE 26. Overall Performance Comparison of All Randomized Schemes. Performance values are 1/average transfer time. Number of TCP sessions varies from 2 through 50. Best performance is from random drop gateways, but randomized congestion window schemes come close.

The most aesthetically appealing of the algorithms presented is the Poisson Window Increase algorithm. This extremely simple algorithm can be implemented in a few lines of code, and does not change the average-case behavior of the window increase algorithm.

The Noisy Window algorithm gives the most performance improvement of all the host-based algorithms, and also exhibits a very smooth variation with number of connections. The Bursty Window algorithm performs slightly better than the Poisson Window Increase algorithm, but probably not enough to justify its complexity and difficult analysis.

None of the host schemes, however, perform as well as the gateway-based scheme. However, the host schemes are still interesting for the following reasons.

First, there is some historical evidence that it is easier to change host implementations than router implementations. Although there are many more hosts than routers, host software is upgraded frequently for other reasons. In a research environment, it is often very easy to experiment with new TCP implementations in various Unix derivatives with freely available source.

Second, the host-based schemes follows the end-to-end principle [39], which says that anything that can be done in the end-systems should be done there, rather than in the network. Rather than citing philosophical justifications for this principle, we simply point to its success as a core principle of the most successful data network ever.

As a project for future work, a real implementation of the various schemes in the BSD kernel would be a valuable contribution. Each of the various degrees of randomness should be tunable to allow the most convenient experimentation. “TCP Monte Carlo” is proposed as a name for this implementation.

Chapter 7
Conclusions

By injecting randomization into otherwise deterministic systems according to some simple principles, this thesis has proposed improved ways of measuring both congestion control and memory system performance, and has also demonstrated ways in which the performance of the TCP/IP congestion control system can be improved using randomization.

First, several techniques to add randomness to congestion control simulation configurations and to program memory layout optimization systems were developed and evaluated.

Robust, reproducible, and predictive performance measurements are important, but it is difficult to make such good measurements of complex systems. One reason is that very small changes in any aspect of the system or its experimental configuration can cause large variations in performance. Worse, measurements of the change in performance when a new technique is applied are subject to the sum of two variations in performance. Although each measurement may be repeatable in the sense that running the experiment or simulation again produces the same result, the results are hard for other researchers to reproduce, not broadly applicable, and may change due to any other small change in the system.

For simulation-based TCP/IP performance research, specific guidelines specifying appropriate amounts of each kind of randomization were proposed that should help produce robust, reproducible, and predictive results.

For both systems, randomized methods were shown to reduce dramatically reduce sensitivity and therefore the error to be expected in measurements. In TCP/IP systems, random perturbation can be used to expose the configuration sensitivity of congestion control simulations. Comparing the performance distributions over a hundred slightly perturbed configuration was shown to be more meaningful and reproducible than comparing individual performance results. In measuring memory system performance, the median performance over a large number of runs with random procedure ordering was shown to be a much more reliable performance metric.

The randomization techniques also provide an estimate of the error that should be expected if the techniques are not used: that is, if measurements are reported for single, deterministic simulations. Error estimates for a common simulation configurations suggest that deterministic simulations under these conditions may include fairly large errors. Fairness in congestion control systems was shown to vary by more than an order of magnitude due to insignificant changes in link propagation delays. Runtime of most SPEC95 benchmarks exhibited more than 10% variation (all variations are reported as the 90% confidence interval.) When all eight integer benchmarks are averaged, the variation is still 3.5%, leading to a 8.0% error in comparing two versions of a program or compiler at 90% confidence.

A second contribution of this thesis has been to describe ways of building randomization into the core of TCP/IP congestion control algorithms so that future performance measurements of these systems would be inherently free of large systematic errors. Although it probably does not make sense to redesign TCP in this light, it is recommended as a design principle for new network protocols.

A third contribution of this thesis has been to demonstrate that the randomized versions of TCP perform better, because reducing the variation in bandwidth allocation between connections reduces the average transfer time across all connections. Based on simulation results, levels of randomness were selected that gave the best performance in terms of average transfer time.

By modifying the congestion window computation algorithm in the sender, performance improvements of up to 32% were achieved over a conventional implementation (TCP Tahoe). The above-mentioned figure is the average across a wide range of numbers of competing greedy connections. The largest benefits were seen with 20 connections or less. The most promising algorithm adds a uniformly distributed number to the value of the congestion window before using it to limit the sending algorithm. Various other algorithms were proposed and analyzed, showing lower but still encouraging performance improvements.

Random drop gateways, originally proposed by Floyd & Jacobson [22], improve performance even more, by an average of 37% across the simulations performed in Chapter 7. However, because the practical networking community has been much slower to adopt changes to routers than changes to host implementations, the randomized window computations (which can be implemented purely in the sender) may prove to be an easier path to improved performance.

7.1 Lessons Learned

Based on the present author’s experience in implementing several randomized systems for making the performance measurements reported in this thesis, it is not hard to add randomization to existing systems. Thus, it is highly recommended as a tool to improve measurability as well as overall performance.

When randomization is added to a system, individual experiments are not repeatable. On the whole, I argue that it is worthwhile to sacrifice repeatability for reproducibility and robustness. Adding randomness may make it much more difficult to create repeatable test sequences in order to track down problems. Possible solutions to this problem include means for specifying initial random seeds and flags for turning off randomness. These considerations are important in compiler development, but probably not in networks where there is usually unavoidable randomness.

When applied to compiler optimizations, randomness may have an interesting advantage which seems worthy of further research. In most programming languages, it is possible to write programs which are ambiguous: i.e. can be interpreted in multiple different ways by the compiler. “Correct” programs would perform correctly under any valid interpretation. Examples of ambiguous constructs in C include writing past the end of an array (where the memory could be empty, or nonexistent, or contain another important data structure) and hidden memory aliases (the ANSI C specification allows memory references through incompatible pointer types to be reordered.) If the compiler used in development happens to choose a benign interpretation of an incorrect ambiguous construct, the bug may not be noticed until the compiler is upgraded, or the program is ported to a new machine. Because bugs get more difficult to fix as time elapses since the code was written, a randomized compiler which found bugs earlier might improve the development process. Best results would come from testing programs with the randomness level set very high, and comparing outputs against normal operation.

A further benefit in system development is that a given set of test cases can exercise a larger range of system states when randomness is added. This may reduce the amount of work needed to assemble an acceptably comprehensive test suite.

For compiler development, repeatability can be achieved by careful management of pseudorandom seeds. A suggestion is to accept a seed on the command line, and generate a seed for each function based on a hash function of the command line seed and the function name. The seed information should be stored in the object file produced so that results can be repeated if necessary.

7.2 Future Work

It is likely that randomness can be profitably applied to many more systems than congestion control and program memory layout optimization. In earlier unpublished work by the present author, encouraging results were obtained for adding randomness to a RISC instruction scheduler. Tuning the instruction selection heuristic function, which weighs the effects of each instruction on register pressure, cycle count, and progress on the critical path was significantly aided by randomness. In general, algorithms driven by heuristic functions may benefit from adding random noise to the result of the function.

7.3 Summary

This thesis presented a simple principle, that arbitrary decisions should be made randomly, and applied it to network congestion control and compiler memory layout systems in which many more or less arbitrary decisions were previously made deterministically. In both cases, system performance became more reproducible and robust.

References

1. R.M. Karp. An Introduction to randomized algorithms. *Discrete Applied Math. 34*, 165-201.

2. D. Knuth. *The Art of Computer Programming, Vol I: Fundamental Algorithms*. Addison Wesley, 3rd ed. 1997.

3. D. Knuth. *The Art of Computer Programming, Vol III: Sorting & Searching*. Addison Wesley, 3rd ed. 1997.

4. R. Motwani, P. Raghavan. Randomized Algorithms. *ACM Computing Surveys*, Vol 28, No. 1. March 1996.

5. R. Motwani, P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

6. J.B. Chen. *The Impact of Software Structure and Policy on CPU and Memory System Performance*. Ph.D. Thesis, published as technical report CMU-CS-94-145 from Carnegie Mellon University.

7. C. Small, N. Ghosh, H. Saleeb, M. Seltzer, K. Smith. Unpublished work.

8. T. Debiec. Point of View. *Cabling Installation and Maintenance*, October 1996. PennWell Publishing.

9. J.M. Hammersly, D.C. Handscomb. *Monte Carlo Methods*. Methuen, 1964.

10. W. Richard Stevens. *TCP/IP Illustrated, Volume 3*. Addison Wesley, 1996.

11. M. Mathis, J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. *Proceedings of SIGCOMM ‘96.*

12. Z. Liu, E. Yan, P. Danzig, J. Ahn. An Evaluation of TCP Vegas. *Proceedings of ACM SIGCOMM ‘95 Conference.*

13. L. Brakmo, L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communication*, Vol 13, No. 8 (October 1995) pages 1465-1480.

14. L. Brakmo, S. O’Malley, L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. *Proceedings of SIGCOMM ‘94* (Aug. 1994) pages 24-35.

15. Nikolas Gloy, Trevor Blackwell, Michael D. Smith, Brad Calder. Procedure Placement using Temporal Ordering Information. *Proceedings of the 30th Annual IEEE/ACM International. Symposium. on Microarchitecture*, December 1997.

16. Cliff Young, David S. Johnson, David R. Karger, Michael D. Smith. Near-optimal Intraprocedural Branch Alignment. *Proceedings of ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pp. 183-193, June 1997.

17. L. Brakmo, L.L. Peterson. Experiences with Network SImulations. *Proceedings of SIGMETRICS ‘96* (May 1996).

18. Cliff Young. Ph.D. Thesis. Harvard University, 1997.

19. R.W. Quong. Expected I-Cache Miss Rates via the Gap Model. *Proceedings of the 21st International Symposium on Computer Architecture*, pp 372-383, April 1995.

20. K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, V. 26 N. 3, July 1996, pp. 5-21.

21. S. Floyd. *Ns Simulator Tests for Random Early Detection (RED) Gateways*, LBNL Technical Report, October 1996.

22. S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, V.1 N.4, August 1993, p. 397-413.

23. S. Floyd and V. Jacobson. On Traffic Phase Effects in Packet-Switched Gateways. *Internetworking: Research and Experience*, V.3 N.3, September 1992, p.115-156.

24. N. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, vol IT-31 pp 119-123, March 1985.

25. R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, vol 19, pp 295-404, July 1976.

26. H.W. Holbrook, S.K. Singhal, D.R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. *Proceedings of ACM SIGCOMM '95*. pp 328-341.

27. W.H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1988.

28. Y.C. Ho, X. Cao. Perturbation analysis and optimization of queueing networks. *Journal of Optimization Theory and Applications*, 40(4), pp 559-582.

29. N. Weste, K. Eshraghian. *Principles of CMOS VLSI Design*, 2 ed. Addison-Wesley, 1993.

30. V. Jacobson. Congestion Avoidance and Control. *Computer Communication Review*, 18(4), pp 314-329.

31. L.S. Brakmo, S.W. O'Malley, L.L. Peterson. TCP Vegas: New Techniques for COngestion Detection and Avoidance. *Computer Communication Review* 23(4), pp 24-34.

32. W.R. Stevens, G.R. Wright. *TCP/IP Illustrated*, Volume 2. Addison-Wesley, 1995.

33. A.S. Tannenbaum. Computer Networks 2ed. Prentice Hall, 1989.

34. H.T. Kung and T. Blackwell, A. Chapman. Credit Update Protocol for Flow-Controlled ATM Networks: Statistical Multiplexing and Adaptive Credit Allocation. In *Proceedings of ACM SIGCOMM '94*.

35. H.T. Kung and R. Morris. Credit-Based Flow Control for ATM Networks. *IEEE Network Magazine*, March 1995.

36. S.J. Gould. *The Mismeasure of Man*. W.W. Norton & Company, 1996.

37. D. Mosberger, L.L. Peterson, P.G. Bridges, S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proceedings of ACM SIGCOMM* '96.

38. Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*. Digital Equipment Corporation, 1995.

39. J.J. Saltzer, D.P. Reed, D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, vol.2, no.4, p. 277-88. 1984.

40. D. Lin, and H.T. Kung. TCP Fast Recovery Strategies: Analysis and Improvements. In *Proceedings of INFOCOM '98*. San Francisco, March 1998.

41. D. Lin. *Internet Congestion Control: Cooperative End-System and Gateway Algorithms*. Ph.D. Thesis, Harvard University, 1998.

42. R. Morris. S*calable TCP Congestion Control*. Ph.D. Thesis, Harvard University, 1998.

43. J.L. Hennessy, D.A. Patterson. *Computer Architecture, a Quantitative Approach*. 2ed. Morgan Kaufmann.

44. A. Eustace, A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing SYstems*. pp 303-314. January, 1995.

45. K. Pettis and R. Hansen. Profile GUided Code Positioning. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. pp 242-251. May, 1989.