On Reducing the Processing Cost of On-Demand QoS Path Computation

George Apostolopoulos
Department of Computer Science
University of Maryland
College Park, MD 20742

Satish K. Tripathi
Bourns College of Engineering
University of California
Riverside, CA 92521-0425

Abstract

Quality of Service (QoS) routing algorithms have become the focus of recent research due to their potential for increasing the utilization of an Integrated Services Packet Network (ISPN) serving requests with QoS requirements. While heuristics for determining paths for such requests have been formulated for a variety of QoS models, little attention has been given to the overall processing complexity of the QoS routing protocol. Although on-demand path computation is very attractive due to its simplicity, many believe that its processing cost will be prohibitive in environments with high request rates. In this work, we study alternatives to on-demand path computation that can reduce this processing overhead. In addition to the well known solution of path pre-computation we introduce and study path caching, an incremental modification of on-demand path computation. Our simulation results show that caching is an effective alternative to path pre-computation and that both path caching and pre-computation can achieve significant processing cost savings without severely compromising routing performance.

1 Introduction

Although a considerable amount of work has been done on the problem of supporting QoS in the network, only recently the role that routing will play in an integrated services network came into attention. The main body of the QoS routing work so far focuses on the path finding problem: given the QoS metrics, the network state and the QoS requirements of a request, find a path that maximizes the chances of the request for successful resource reservation and has the minimum negative impact on the network's ability to accept future requests. A variety of heuristics for the unicast version of the above problem under different QoS models have been discussed in the literature [1, 3, 4, 5, 6, 7, 8]. Some ([1, 3, 7, 8]), operate under a bandwidth based QoS model where requests express their QoS requirements in terms of desired bandwidth. This model has the advantage of simplicity, and is the basis of the Controlled Load service model proposed by the Intserv working group. Most of the path finding heuristics assume that paths can be calculated on a per request basis, resulting in an *on-demand* mode of operation. While calculating paths on-demand leads to a simple implementation and reduces storage requirements, many believe that it may not be practical in environments with high rates of QoS requests because of its high processing overhead. Although this claim is hard to verify due to the limited experience with realistic QoS aware network environments, alternatives to on-demand routing have already been proposed [1, 7, 9, 10]. These alternatives are based on the path pre-computation principle: paths to destinations are computed asynchronously to the request arrivals and are used to route multiple requests, reducing in this way the per request processing overhead.

In all the aforementioned path pre-computation proposals, when paths are pre-computed, all possible paths to all destinations have to be computed and stored in a QoS routing table. This may prove inefficient both in terms of processing and storage if most of the pre-computed paths are not used. In addition, in all the path pre-computation proposals so far, the set of pre-computed paths is not updated between pre-computations. It is possible that allowing the addition of new "good" paths to the set of pre-computed paths will improve routing performance without incurring the cost of pre-computing a completely new set of paths.

In this work, we propose a caching architecture that is more flexible than the so far proposed path pre-computation approaches. In this caching architecture, paths are computed only on-demand and then are stored in a path cache to be reused for future requests. If a request can not be routed using a cached path then an on-demand computation is used to determine a QoS path for it and the new path is added in the path cache, updating in this way its contents. To the best of our knowledge, the only previous work on path caching appeared in [17]. Although the approach is similar, our work is considerably different as will become apparent when we present our caching scheme in more detail. In ad-

dition, a major goal of this work is to compare both the routing performance and the processing cost of path caching to that of a path pre-computation architecture and investigate the overall cost effectiveness of path caching as a method for reducing the processing cost of determining QoS paths.

This paper is organized as follows: In Section 2, the algorithms used are discussed and path caching is introduced. Section 3 introduces the simulation environment and in Section 4 the different approaches are compared based on the simulation results. In the last Section we summarize our findings.

2 Algorithms for Path Computation

We consider only link state routing protocols where paths are calculated at the source and requests are routed using source routing. There is some evidence that this will be the architecture of choice for the standard QoS routing algorithms. Private Network to Network Interface (PNNI) [14], an already standardized QoS routing protocol, is based on this architecture. Moreover, the QoS routing proposals in IETF's OSPF working group are based on the link state architecture and can operate in both source routing and hop-by-hop mode.

In a link state routing protocol, each node maintains a link state database that contains a description of the network state as known to the node. The link state database is updated from link state reports generated by other nodes. Link state updates are distributed to all nodes using flooding. In contrast to previous studies ([1]), we use a link update generation model proposed in [7] that initiates a new link state update when the available bandwidth in a link changes significantly since the last time advertised. A link state *update threshold* determines the percentage of change necessary for re-advertising the value of the available link bandwidth. If b_{last} is the most recently advertised available bandwidth value and b_{cur} is the current value, an update is originated only if $|b_{cur} - b_{last}|/b_{last} > th$, where th is the link state update trigger threshold.

Recent simulation studies ([1]) showed that from among the heuristics proposed for routing requests with bandwidth requirements, the shortest path heuristics [1, 7, 8] perform better than the widest path heuristic [3] that prefers widest over shortest paths. The width of the path, also called bottleneck capacity, is defined as the minimum available bandwidth over all the links in the path. We will use the widest-shortest path heuristic [7] as the basis of the path computing algorithms.

On-demand widest-shortest paths are computed as follows: Links that have insufficient available bandwidth for the request that is being routed are pruned from the network topology before the path is calculated. Then the minimum hop count paths between the source and the destination are discovered and the widest one is used to route the request. If there are more than one widest-shortest paths one of them must be chosen.

From among the multiple path pre-computation proposals, we chose to use in this work the approach described in [7] mainly because it is the only path pre-computation solution that has been proposed to IETF's OSPF working group. [7] uses a modified Bellman-Ford algorithm to pre-compute the widest bottleneck capacity minimum hop path to each destination. In addition, paths that are longer than the minimum hop path (alternate paths) but have larger bottleneck capacity than the shorter paths recorded for this destination are also stored. If there are multiple paths with equal bottleneck capacity (both minimum hop and alternate) all paths are stored. When a request arrives, a path is selected among the pre-computed paths as follows: paths are checked for feasibility in order of increasing length and the shortest feasible one is selected. As in the on-demand case, a path is feasible if its bottleneck capacity (as calculated during the time of path pre-computation) is larger or equal to the request requirements. If there are multiple feasible paths with the same hop count, a path is selected using the technique discussed above for on-demand path computation. If there are no feasible paths the request is routed over the longest available path for the destination.

For comparison purposes we also use a static path computation algorithm. This operates in exactly the same way as the on-demand algorithm except that the link capacity is used as the value for the available bandwidth, making path selection insensitive to variations of resource availability in the network. If there are multiple minimum hop paths with the same (static) bottleneck capacity, one of the them is selected at random using the same load balancing technique as in the previous two cases, only that now the link capacity to the first hop of the path is used instead of the available bandwidth on the interface to the first hop of the path.

2.1 Path Caching

Path caching attempts to reduce the processing complexity of on-demand path computation without compromising its on-demand nature and its ability to compute paths for individual requests when necessary. Path caching reduces the number of path computations by reusing already calculated paths. Already discovered paths to a particular destination are stored in a path cache associated with this destination. Future requests are routed on-demand only if they can not be routed using the contents of the path cache. A different path cache is associated with each destination node. Clearly, the path cache needs to be flushed and re-populated in case of topology changes (i.e. change in link status).

The path cache can be implemented in a variety of ways. We chose a simple implementation where a path is repre-

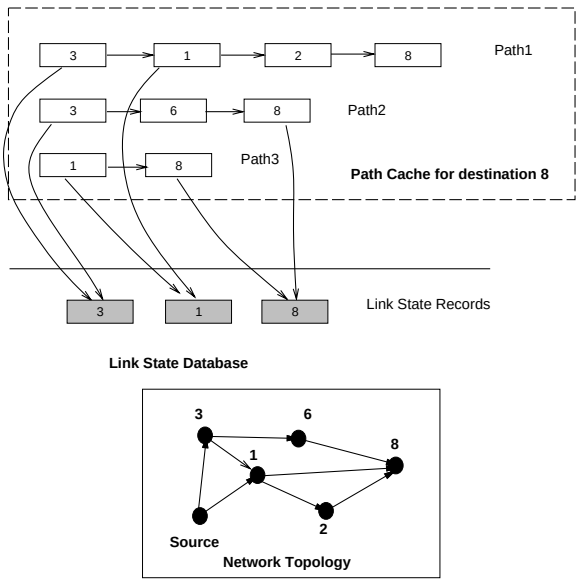


Figure 1. Example path cache organization

sented as a list of node structures. Each node structure is essentially a pointer to the corresponding entry in the link state database (Figure 1). This organization allows for simple traversal of the link state database and easy reconstruction of the path when it is needed for routing a request but is not optimize for storage.

For each cached path the hop length and the bottleneck bandwidth capacity are maintained in the cache along with the sequence of nodes in the path. The value of the bottleneck capacity does not necessarily reflect the most recent information in the link state database, since it is not re-computed each time a link state update is received. Only feasible cached paths are considered for routing requests. A cached path is feasible if its bottleneck capacity is larger than the bandwidth requirements of the request. If the cache does not contain any feasible path, the request can not be routed using the cache contents and will have to be routed on-demand.

2.2 Operation

When a new request arrives, the cache containing the paths to the destination node is searched for a feasible path. Such a path is selected using a *cache selection policy*. If such a path exists, then it is used to route the request. If there is no feasible path the request is routed on-demand. The on-demand computation may fail to discover an appropriate path, in which case the request is rejected. If a path is found though, it is added in the cache for the destination. If the cache is full, a *cache replacement* policy determines which cached path will be replaced by the new path. In order to make sure that the cached paths reflect to a reasonable degree the current network state, a *cache update* policy determines how the cached paths are updated. In the rest of this section we will discuss these cache managements policies in detail.

2.2.1 Cache Selection Policy

In accordance with previous studies, in order to minimize the consumption of network resources, the shortest of the feasible cached paths should be used for routing a request. Choosing a longer path results in using resources on more links and potentially penalizes the ability of the network to accept future requests. Still, depending on the topology of the network, there may exist multiple paths of the same (minimum) hop length. In this case a tie-breaking mechanism is needed. In this work we explore three such mechanisms:

- *Round-Robin*: For each path a counter of the times the path is used is maintained. When there are multiple feasible paths with the same hop count, the least used of the paths is chosen. This attempts to distribute the load among the multiple paths on a request basis. The usage counter is reset each time the path is added or removed in the cache and when the information about the bottleneck capacity of the path is updated (as we will describe in the next section).

- *Widest path*: The path with the largest bottleneck capacity among the feasible equal hop length paths is chosen. This implements a "worse-fit" policy, in an attempt to reduce bandwidth fragmentation (see [16] for a discussion of bandwidth fragmentation).

- *Tightest path*: The feasible path with the smallest bottleneck bandwidth is chosen. This policy, in contrast to the previous one, attempts to pack requests and leave large chunks of bandwidth available for future larger requests.

In all the above cases, in the occasion that there is also a tie in the amount of available bandwidth, a path is selected among the ones with the same hop length and bottleneck capacity at random with uniform distribution.

2.2.2 Cache Replacement Policy

As a result of successful on-demand path computations performed for requests that could not be routed using a cached path, new information needs to be added to the cache. If the path discovered is already in the cache, its bottleneck capacity will be updated. Otherwise, the path will be added in the cache. If the cache is full another path will have to

be replaced. The choice of the path to be replaced is such so that a path with low chances of being used later will be replaced. Note that since the new path is the result of a on-demand computation that had to be performed because there was no feasible path in the cache; it follows that the newly discovered path will have bottleneck capacity larger than all the currently cached paths.

The search for a path suitable for replacement first considers paths that are longer than the path that is to be added in the cache. If longer paths exist they are ideal candidates for replacement since they will never be selected after the new path is added in the cache. Indeed, the cache selection policy will always prefer the new path to the longer and less wide path. In the next step, paths that have length equal to the new one are considered. If such paths exist, one of them is selected for replacement in agreement with the cache selection policy. i.e., if the widest cached path selection policy is used, then the narrowest of the paths with hop length equal to the new path is selected, if the tightest policy is used then the widest path is chosen and in the round-robin policy, the most used path is replaced. The last case is for the cache to contain only paths that are shorter than the new path. In this case it is not obvious which path is the one that has less chances of being used for routing future request. As a heuristic, we choose a path among *all* the cached paths again in accordance to the cached path selection policy.

2.2.3 Cache Update Policy

Clearly, the network conditions (in our case the available bandwidth in each interface) change continuously, resulting in new feasible paths to destinations that need to be discovered and added to the path cache, and in new values for the bottleneck bandwidth capacity of already cached paths. The contents of the path cache can be updated either by invalidating cached paths or by accessing the link state database and re-computing updated values of the bottleneck bandwidth capacity of the cached paths. Note that the link state updates generated by nodes are used only to update the local link state database. The information about cached paths in not modified on receipt of a link state update. When cached paths are invalidated, future requests are forced to be routed on-demand, discovering in this way the new network state. As a result, path invalidation is expected to result in a larger overall number of on-demand path computations. On the other hand, re-computing the bottleneck capacity of cached paths will lead to fewer on-demand computations since the paths are still in the cache and can be used for routing future requests. Still, by only updating cached paths the discovery of new paths may be delayed resulting in sub-optimal routes. We investigate the performance of the two alternatives using three different cache update policies:

- *Periodic Invalidation*: All the cached paths for all destinations are periodically invalidated, forcing future requests to be routed on-demand and re-populate the path caches.
- *Periodic Update*: All cached paths for all destinations are updated periodically by accessing the link state topology database and re-computing bottleneck bandwidth values.
- *Individual Invalidation*: Each cached path has a lifetime associated with it. After the path's lifetime has expired it is invalidated.

All the above cache update policies require performing operations periodically or in the case of individual invalidation setting the lifetime of a cache entry. We will collectively refer to the update period of all the above policies as the *cache update period*. This term will also include the lifetime of a cache entry when the individual invalidation policy is used and the path pre-computation period when paths are pre-computed.

After presenting the details of our path caching architecture we can contrast it with the previous work on caching presented in [17]. The main difference is that [17] allowed only a single path to be cached per destination. We allow caching multiple paths per destination achieving more flexibility for routing requests and also in an attempt to exploit topology characteristics like availability of multiple equal hop paths between sources/destinations. paths and As a result of caching multiple paths per destination, cache management in the form of cache selection and cache replacement is now an important component of the caching architecture. In addition, we investigate the effects of the cache size to the performance of the caching scheme. Finally, cached paths in [17] were invalidated when a sufficiently large number of link state updates was received for any of the links belonging to the cached path. We believe that the implementation of this policy is non-trivial in terms of processing that has to be performed on receipt of a new link state update. We chose to investigate different, lower cost mechanisms for keeping cached paths up to date.

3 Simulation Environment and Performance Metrics

We have developed a simulator based on the Maryland Routing Simulator (MaRS) [13]. MaRS was extended to perform source routing and the link state update generation mechanism has been modified to support triggered updates. A simple resource reservation protocol has been implemented for unicast connections. The traffic load is expressed in terms of connection requests. If a request is established there is no actual packet traffic over its path.

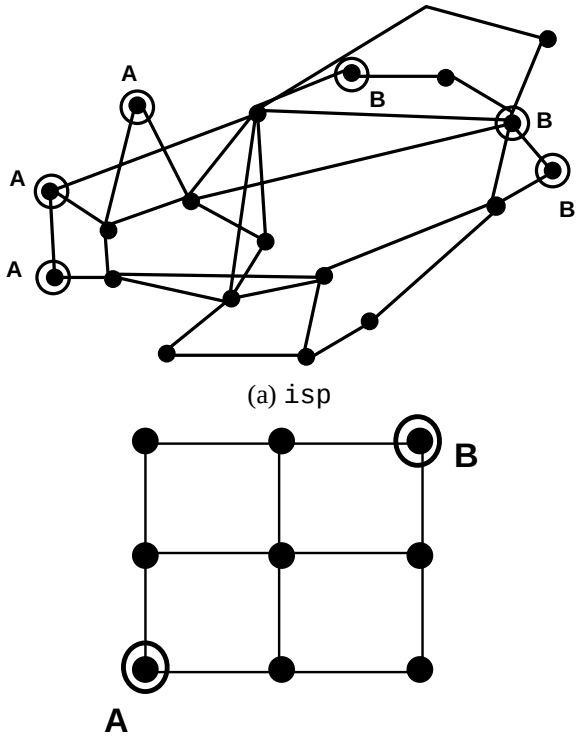


Figure 2. Topologies used in the experiments

The current load for the network links is determined by the list of reservations that is maintained for each interface by the reservation protocol. Effectively, all connections are assumed to be constant bit rate and to conform to their stated bandwidth requirements. This is clearly a simplification but issues like traffic policing, shaping, and modeling are orthogonal to the issues we study here. The topologies used in the experiments are shown in Figure 2. The `isp` is a typical topology of a fairly large ISP provider [1]. The `mesh` topology is more compact and we expect the differences in running time between computing a path to all destinations and a single destination to be small. In addition, in contrast to `isp`, the `mesh` topology has a larger number of equal hop multi-paths. The propagation delay in all links is set to 1 millisecond. It is assumed that links never fail.

We believe that QoS routing will be more beneficial in cases where temporary mismatches between traffic pattern and network topology will create increased loads on parts of the network. These conditions can occur either as a result of a network failure (link failure) or simply because of changing traffic patterns. In order to recreate these conditions in our simulations we determine the link capacities so that the topology is dimensioned for uniform traffic. This dimensioning results in link capacities between 30 and 80 MBits/sec for the `isp` topology and 100 and 140 MBits/sec for `mesh`. We create conditions of non-uniform traffic by establishing two sets of non-uniform traffic nodes. Nodes that belong to one of the sets can request connections only with nodes in the other set and have a mean request arrival rate different than the nodes that do not belong to any of the sets. If the destination set contains more than one node, the destination is chosen randomly among the nodes in the destination set with uniformly distributed probability. The nodes belonging to each set for both topologies are shown in Figure 2 marked with A and B correspondingly. In the case of non-uniform traffic there are two request arrival rates. The *background* rate for traffic between nodes that do not belong to any of the sets of non-uniform traffic nodes and the *foreground* rate for the nodes in the sets. The background rate is chosen so that background traffic levels are large enough to limit the routing options available to primary traffic, i.e., reducing the levels of background traffic will reduce the overall blocking ratio.

A basic dimension of the comparison between the different routing architectures is their behavior under different workloads. To capture the effects of call duration and requested bandwidth we use two different workloads which combine different duration and request size. The parameters of the workloads are summarized in Table 1. Bandwidth requirements are uniformly distributed between the minimum value of 64 kbit/sec and maximum value shown while call duration is exponentially distributed with mean of 3 minutes. MIT stands for mean inter-arrival time. The traffic loads are chosen in such a way so that blocking ratios are kept in the 2%-30% range. The arrival rates are Poisson distributed with the mean shown in Figure 1.

All experiments were performed with a link state update triggering threshold of 10%. This number results in frequent link state updates, ensuring that link state information is accurate. We also performed experiments with larger threshold values that resulted in link state information inaccuracy. Although we do not report the detailed results here due to space limitations, the overall behavior of the different alternatives is similar to what is described in Section 4 even when link state information is inaccurate.

Each node in the network has an individual request arrival process, independent from the ones of the other nodes. The network was simulated until 100,000 connections were requested. The first 30,000 connection requests were used to warm up the network and are ignored when calculating the routing performance and the processing cost.

3.1 Higher Level Admission Control

It is well known [11, 12] that excessive alternate routing can actually reduce routing performance in conditions

mesh	1 Mbit/sec	6 Mbit/sec
Foreground MIT	2.5 sec	20 sec
Background MIT	5 sec	40 sec
isp	1 Mbit/sec	6 Mbit/sec
Foreground MIT	.65 sec	5 sec
Background MIT	1.3 sec	7.5 sec

Table 1. Workload parameters

of high load, since traffic following alternate routes can interfere with minimum hop traffic competing for the same links. In order to address this problem, we investigate high level admission control policies similar to trunk reservation. Assuming that explicit routing is used, we propose a trunk reservation approach that may result in rejecting requests routed over alternate paths during the resource reservation phase, even when there are sufficient resources to satisfy the request. A local per node check determines if the request is allowed to continue reserving resources over the path depending on both the resources that remain available on the link after the reservation and the relative length of the path, i.e., how much longer it is compared to the minimum hop path. This information is easy to compute if after each topology change the minimum hop to each destination is computed. When a reservation is attempted through a node, the quantity $(b_i^{avail} - b_{req})/b_i^{capacity}$ is calculated for the outgoing link i, where b_i^{avail} is the amount of available bandwidth on link i, and $b_i^{capacity}$ is the capacity of link i. The resource reservation for the request is allowed to continue only if this fraction is larger than a trunk reservation level which depends on the length of the path. If the request fails this test, it is rejected. Computing the trunk reservation level based on the request's requirements and the residual capacity of the link allows us to reject requests only when they really would have resulted in overloading a link. Having different trunk reservation levels for increasingly longer paths allows us to penalize longer paths more and control alternate routing better. For the experiments in this paper, the trunk reservation levels were set to 2% for one hop longer paths, 5% for two, three and four hops longer, and 10% for all longer paths.

3.2 Performance Measures

In most circuit switched routing performance studies the connection blocking ratio is used as a measure of routing performance. The connection blocking ratio is defined as the percentage of connection requests rejected out of the total number of requests. As mentioned in [1], this is not necessarily an accurate measure since connections have different bandwidth requirements. Thus we mainly report the bandwidth acceptance ratio, the sum of bandwidths of connection requests accepted over the sum of bandwidths of all the connection requests.

To compare the processing cost of the alternative routing architectures we use the processing cost model described in [2]. In this model the routing algorithm is broken down into a number a elementary operations and the cost of each of these operations is measured in carefully designed benchmark experiments. These operations are: a) initialization, b) accessing the link state database, c) data structure operations. Then, in the actual simulations, the number of times that each elementary operation was executed is determined and the aggregate cost of the routing algorithm is computed by summing the products of counts and cost for each elementary operation. The number computed this way corresponds to the total time spent in routing protocol processing for a given simulation run. For path caching, we need to accommodate new elementary operations that are unique to path caching. These operations are: a) updating the cache, b) adding a path in the cache, c) cache lookup, d) invalidating a cached path.

We derive the cost of the individual operations by profiling the execution of benchmark experiments using the `pixie` profiler available in the Digital Unix platform used for the simulations. The profiler gives the total real time spent in each function of the simulation. By associating the total time spent for operations of a particular category with the number of such operations we can derive an approximate cost per operation. The details of the benchmark experiments are described in [2]. The cost of cache operations in measured in conjuction with how many iterations the main loop of these operations required. The number of iterations depends on the number of cached paths. The results are summarized in Table 2. The numbers are derived for the architecture we used for the simulations: Alphaserver 2100 4/274 servers with 4 Alpha 21064A CPUs at 275 MHz, 256 Mbytes of real memory, and Digital Unix operating system. Cache related costs do not depend on the cache size because of the way they are computed. The cost of initialization is large since part of initialization is spent in releasing the memory of the data structures for the paths used since the last path pre-computation. This cost clearly depends on the size of the network. The cost of all other operations is relatively independent of the size of the network topology and size.

In all the numbers reported the 95% confidence interval is under .3% in the `isp` and 1% in the `mesh` topology for the bandwidth acceptance ratio and under .5% for the processing cost values in both topologies. Confidence intervals were calculated using the Student's t distribution.

Operation	isp	mesh
Node (Avg/Node)	5	3
Iteration (Avg/Round)	2	1.5
Data Structure (Avg/Operation)	4	4
Initialization (Avg/Path Computation)	120	47
Cache Update (Max/Path)	5	
Caching a Path (Max/Path Cached)	15	
Cache Lookup (Max/Cached Path)	3	
Path Selection (Max/Path)	3	

Table 2. Cost of the operations performed by the routing algorithms (in μsec)

4 Performance Comparison

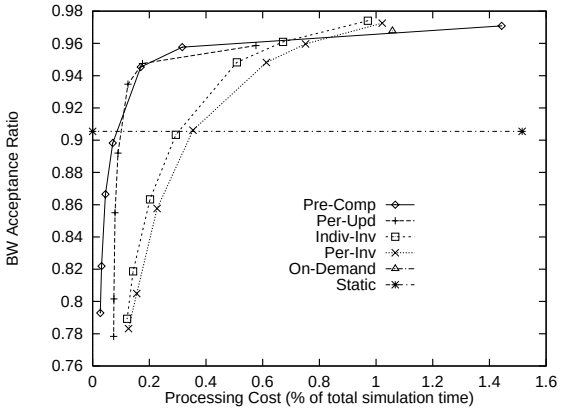
First we determine appropriate values for the path pre-computation period in the particular topologies and workloads used. These values are chosen so that the routing performance of the various alternatives measured using the bandwidth acceptance ratio is better or similar to the performance of static path computation. The values for the update period calculated this way and used in the experiments are 5, 10, 30, 60, 120, 180, 220, and 280 seconds. For the larger values of the update period routing performance is actually worse than static path computation in some combinations of topology and workload, nevertheless we use them for comparison purposes. A disadvantage of having fixed update periods is that it has varying effects under different workloads. For example, when the same network topology is used, requests belonging to a workload with requests for large amount of bandwidths, must have a slower arrival rate (since we want to keep the rejection rate low) than requests of a workload with smaller request sizes. For slow arrival rates, path pre-computation is performed more often in terms of number of requests between successive path pre-computations. Still, having a fixed period will most probably be the simplest alternative in a real network. In addition, even for the simplified environment of our experiments, it is not obvious how to choose equivalent pre-computation periods for different workloads since due to non-uniform traffic different nodes have different request rates.

The cache size for the path caching experiments was set to four paths. Choosing a small cache size is important since it is desirable to minimize the storage overhead of the path caches. Nevertheless, we verified that in the traffic workloads and topologies used in the experiments, increasing the cache size did not have any effect in the routing performance or the processing cost of all path caching alternatives. This is an indication that small caches of very few paths are sufficient for the effective operation of the path caching schemes.

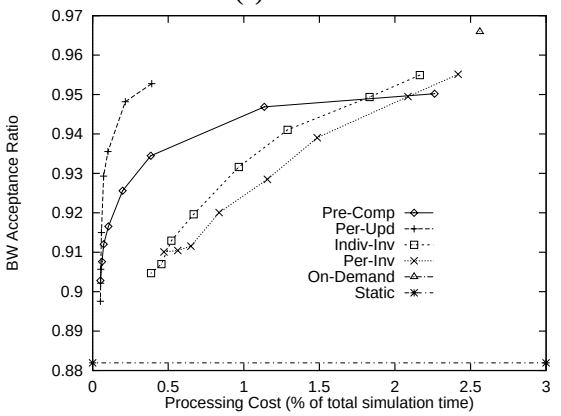
We compare the different approaches by evaluating their performance/cost tradeoffs. We accomplish this by plotting the bandwidth acceptance ratio against the processing cost for the different values of the cache update or path pre-computation period. This results in a curve for each different alternative. The points of the curve that have larger costs (x-axis coordinate) correspond to small periods. For on-demand path computation there is only a single point since its routing performance and cost does not dependent of the update period. The processing cost of static computation is negligible but in the graphs we plot a line instead of a single point on the y-axis to make comparisons with the routing performance of the other alternatives easier.

In Figure 3 we show the performance of path pre-computation and caching for both topologies for requests up to 1 Mbit/sec, for the widest path cache selection policy. In this figure we see that for larger update periods all alternatives can achieve significant processing cost savings when compared to on-demand routing. For small update periods all alternatives have routing performance similar to on-demand routing but with cost that can be similar or even exceed the cost of on-demand routing. Indeed, path pre-computation and periodic cache update incur a processing cost each time paths are re-computed or the cache is updated. If the update period is sufficiently small, their cost can exceed that of on-demand path computation. On the other hand, the cache invalidation policies do not involve any processing cost for cache management. As a result, their processing cost can only be smaller than that of on-demand computation.

Path pre-computation and periodic cache update achieve better processing cost savings for similar values of the update period. This is more visible in part (b) of Figure 3, in the `isp` topology. For large values of the update period, the cost of pre-computation and periodic update levels off. This reflects that the processing due to path computation is so small that path selection cost becomes the dominant cost component. On the contrary, the cost of the other two cache policies continues to decrease as the update period increases. This cost decrease is due to the fact that with larger update periods, cached paths are updated less often and tend to be used more, reducing in this way the number of on-demand path computations. Path pre-computation has processing cost similar to that of periodic cache update in the `mesh` topology. A larger difference in favor of periodic cache update exists in the `isp` topology. This is due to the fact that in the larger `isp` topology the cost of computing a path to a single destination is much lower than the cost of computing paths to all destinations. This cost difference is not very pronounced in the `mesh` topology. Note also that in the `mesh` topology the routing performance of all alternatives appears to deteriorate quickly and become worse

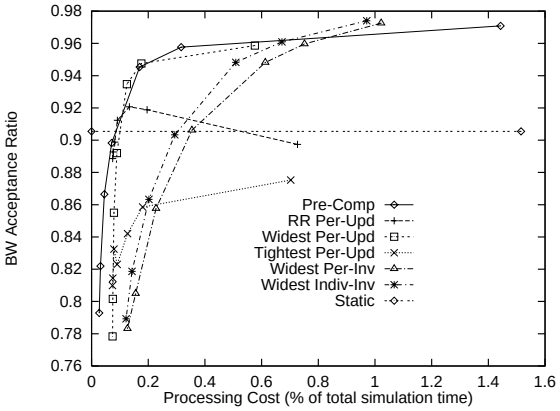


(a) `mesh`

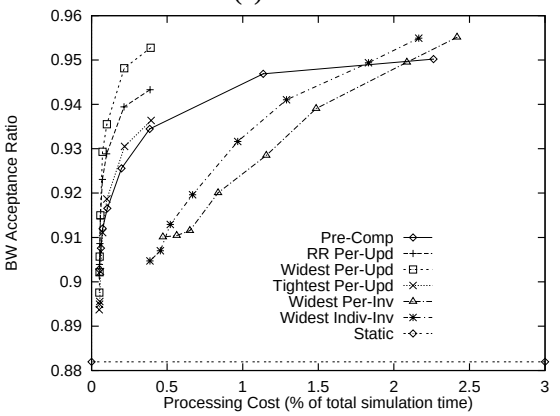


(b) `isp`

Figure 3. Performance comparison, 1 Mbit/sec



(a) `mesh`



(b) `isp`

Figure 4. Effects of cache selection policy, 1 Mbit/sec

than static routing even for moderately small values of the update period, while this is not the case in the `isp` topology. This is mainly a result of the fact that in the `mesh` topology most calls (especially between the source and destination of hot-spot traffic) are routed over the equal hop (and in this case also minimum hop) multi-paths. This is in favor of the static routing algorithm that can also use all the available minimum hop multi-paths. In the `isp` topology, the existence of alternate paths results in significantly improved routing performance of all alternatives over static routing. The results for the 6 Mbit/sec requests are similar to the 1 Mbit/sec case and are not shown here.

4.1 Effects of Cache Selection Policy

In Figure 4 we show the effects of the cache selection policy for both topologies for requests up to 1 Mbit/sec. It turns out that the type of cache selection policy affects the observed behavior of the cache based alternative only when the periodic update is used. Using the periodic update policy results in a average number of cached paths that is larger than when the invalidation based policies are used. This is intuitive since the periodic update policy never invalidates paths in contrast to the other two policies that periodically invalidate individual or all the cached paths. With more paths in the cache, the effects of the cache selection policy become more pronounced. For the periodic cache update policy, the widest path cache selection policy results in better routing performance for smaller update period values for both topologies. The narrowest path policy always results in the worse routing performance. For the other cache policies the type of cache selection policy does not have any significant effect since the cache occupancy and the hit ratios are lower. For these policies we show the curve for only the widest cache selection policy.

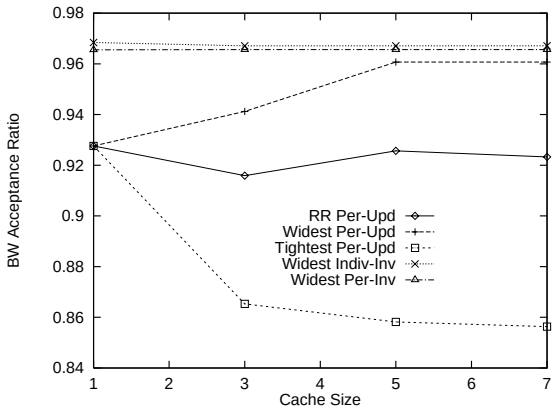
Overall, we observe that only the widest cached path selection policy achieves good performance when the periodic update cache update policy is used. This is intuitive since,

with increasing cache update periods, the bottleneck bandwidths of the cached paths become increasingly less accurate. Choosing the tightest path will soon result in overflowing the chosen path since the reduction of the path's available bandwidth will be discovered later, when the cache is updated again. The round-robin cache selection policy ignores completely the available bandwidth capacity and fails to achieve good performance when caches are updated often. Only choosing the widest cached path combines protection against the infrequent updating of the cached paths with good routing performance for smaller update periods.

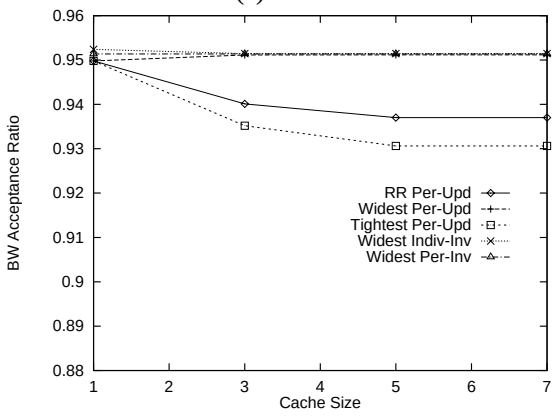
5 Other Considerations

5.1 Cache Size

In Figure 5 we show how the cache size affects the routing performance of the different cache based alternatives. In part (a) of the figure we show data for the `mesh` topology and in (b) for `isp`. Again, requests are up to 1 Mbits/sec and the link state update threshold was set to 10%. The update period was set to 5 seconds. The invalidation based policies do not show any important dependence on the cache size, mainly since the cache occupancy is low most of the times. Nevertheless, there are some interesting variations in the behavior of the periodic cache update policy. In the `mesh` topology, when the widest cached path selection is used, performance increases with increasing cache size. This is reasonable since this topology has multiple equal hop paths that should be in cache to achieve good performance. The more paths exist in the cache, the more information is available about the network each time cached paths are updated and the load balancing performed by choosing the widest of them performs better. This is an indication that in some types of topologies, caching multiple paths per destination is advantageous to a single path cache architecture similar to [17]. The performance of choosing round robin between the cached paths does not really depend on the cache size. Due to the high degree of link sharing between the minimum hop multi-paths, even rotating between few multi-paths can achieve the same link load as rotating between multiple paths. Selecting the tightest cached path performs bad when the cache size is not 1, showing that this policy is not very appropriate for this topology. In the `isp` case there will be less equal hop multi-paths in the caches due to the topology. As a result, the variations in the routing performance due to varying cache size are much less pronounced. For the periodic cache update policy, routing performance of the widest cached path selection policy is independent of the cache size and the tightest cached path selection policy performs bad again. In the `isp` topology, the routing performance of the round-robin cached path selection policy seems also to be slightly



(a) `mesh`



(b) `isp`

Figure 5. Effects of cache size

decreasing with a cache that is larger than 1 path, indicating again that this selection policy does not perform well. Again, for the invalidation based cache update policies the type of cached path selection policy does not affect their performance. As a result, in Figure 5, for the invalidation based policies we show only the curve for the widest cached path selection policy.

6 Conclusions

From our experiments it turns out that:

- Path caching is a viable method for reducing the processing cost of on-demand QoS path computation, at least when the widest-shortest path selection criterion is used.

- The periodic cache update policy coupled with the widest cached path selection policy achieves the best processing cost/routing performance trade-off,

in many cases significantly better than path pre-computation.

- The tightest cached path selection policy performs bad in all cases considered here.
- Invalidation based cache update policies achieve significant processing cost savings but in general less than path pre-computation and periodic cache update.
- Topology can play an important role in both the processing cost and the routing performance of the different caching policies. In particular, the amount of equal hop multi-paths and the density and diameter of the topology can determine the relative costs of single-destination and all-destination path computations and the routing performance of the different cached path selection policies.
- Small cache sizes are sufficient for achieving good routing performance and processing savings. There is always though a dependence on the network topology. In some cases, the cache should be large enough to hold a specific number of minimum length equal hop paths as was the case in the `mesh` topology.

References

[1] Q. Ma and P. Steenkiste, On Path Selection for Traffic with Bandwidth Guarantees, Proceedings of IEEE International Conference on Network Protocols, Atlanta, Georgia, October 1997.

[2] G. Apostolopoulos, R. Guérin, S. Kamat, and S. K. Tripathi, "QoS Routing: A Performance Perspective." To appear in Proceedings SIGCOMM'98.

[3] Z. Wang, and J. Crowcroft, Quality of Service Routing for Supporting Multimedia Applications, IEEE Journal Selected Areas in Communications, 14(7):1228-1234, 1996

[4] W. C. Lee, M. G. Hluchyj, and P. A. Humblet, Routing Subject to Quality of Service Constraints in Integrated Communication Networks, IEEE Networks, pages 46-55, July/August 1995

[5] R. Widyonon, The Design and Evaluation of Routing Algorithms for real-time Channels, Technical Report TR-94-024, University of California at Berkeley, June 1994

[6] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, Two Distributed Algorithms for the Constrained Steiner Tree Problem, In Proceedings of 2nd International Conference on Computer Communication and Networking, pages 343-349, 1993

[7] R. Guerin, D. Williams, and A. Orda, QoS Routing Mechanisms and QSPF Extensions, in proceedings of GLOBECOM 1997

[8] Q. Ma, P. Steenkiste, and H. Zhang, Routing High-Bandwidth Traffic in Max-Min Fair Share Networks, in ACM SIGCOMM'96, 206-217, 1996

[9] J.-Y. Le Boudec and T. Przygienda, A Route Precomputation Algorithm for Integrated Services Networks, Journal of Network and Systems Management, vol. 3, no. 4, pages 427-449, 1995

[10] A. Shaikh, J. Rexford and K. Shin, Efficient Precomputation of Quality-of-Service Routes, to appear in Proc. Workshop on Network and Operating Systems Support for Digital Audio and Video, July 1998.

[11] R. S. Krupp, Stabilization of Alternate Routing Networks, in IEEE International Communication Conference, Philadelphia, PA, 1982

[12] R.J. Gibbens, F. P. Kelly, and P. B. Key, Dynamic Alternate Routing - Modeling and Behaviour, in Teletrafic Science for New Cost-Effective Systems, Networks and Services, ITC-12, Elsevier Science Publishers, 1989

[13] C. Alaettinoglu, A. U. Shankar K. Dussa-Zieger, and I. Matta. Design and Implementation of MaRS: A Routing Testbed, Journal of Internetworking Research and Experience, 5(1):17-41, 1994 RFC 2178

[14] Private Network to Network Interface Specification, Version 1.0, af-pnni-0055.000, ATM Forum, March 1996

[15] L. Breslau, Adaptive Source Routing of Real Time Traffic in Integrated Service Networks, Ph.D. Thesis, University of Southern California, December 1995

[16] S. Gupta, Performance Modeling and Management of High-Speed Networks, Ph.D. Thesis, University of Pennsylvania, 1993

[17] M. Peyravian and A. D. Kshemkalyani, Network Path Caching: Issues, Algorithms and a Simulation Study, Computer Communications, vol. 20, pages 605-614, 1997