Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-99-27

1999-01-01

Optimal Flow Aggregation

Subhash Suri, Tuomas Sandholm, and Priyank Warkhede

Current IP routers are stateless: they forward individual packets based on the destination address contained in the packet header, but maintain no information about the application or flow to which a packet belongs. This stateless service model works well for best effort datagram delivery, but is grossly inadequate for applications that require quality of service guarantees, such as audio, video, or IP telephony. Maintaining state for each flow is expensive because the number of concurrent flows at a router can be in the hundreds of thousands. Thus, stateful solutions such as Intserv (integrated services) have not been adopted for... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Suri, Subhash; Sandholm, Tuomas; and Warkhede, Priyank, "Optimal Flow Aggregation" Report Number: WUCS-99-27 (1999). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/497

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/497

Optimal Flow Aggregation

Subhash Suri, Tuomas Sandholm, and Priyank Warkhede

Complete Abstract:

Current IP routers are stateless: they forward individual packets based on the destination address contained in the packet header, but maintain no information about the application or flow to which a packet belongs. This stateless service model works well for best effort datagram delivery, but is grossly inadequate for applications that require quality of service guarantees, such as audio, video, or IP telephony. Maintaining state for each flow is expensive because the number of concurrent flows at a router can be in the hundreds of thousands. Thus, stateful solutions such as Intserv (integrated services) have not been adopted for their lack of scalability. Motivated by this dilemma, we formulate and solve the flow aggregation problem, where we give an efficient algorithm for computing the smallest set of aggregated flows that encode the forwarding state of individual flows. Our hope is that such aggregation of state information might increase the viability of Intserv-type protocols.

Optimal Flow Aggregation

Subhash Suri, Tuomas Sandholm and
Priyank Warkhede

WUCS-99-27

October 1999

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Optimal Flow Aggregation

Subhash Suri Tuomas Sandholm Priyank Warkhede

Department of Computer Science
Washington University
St. Louis, MO 63130

{suri,sandholm,priyank}@cs.wustl.edu

October 7, 1999

Abstract

Current IP routers are stateless: they forward individual packets based on the destination address contained in the packet header, but maintain no information about the application or flow to which a packet belongs. This stateless service model works well for best effort datagram delivery, but is grossly inadequate for applications that require quality of service guarantees, such as audio, video, or IP telephony. Maintaining state for each flow is expensive because the number of concurrent flows at a router can be in the hundreds of thousands. Thus, stateful solutions such as Intserv (integrated services) have not been adopted for their lack of scalability. Motivated by this dilemma, we formulate and solve the *flow aggregation* problem, where we give an efficient algorithm for computing the smallest set of aggregated flows that encode the forwarding state of individual flows. Our hope is that such aggregation of state information might increase the viability of Intserv-type protocols.

1 Introduction

Current IP networks provide one simple service: the best effort packet delivery, in which no guarantee is made about when or if a packet will be delivered. This simple model allows IP routers to be *stateless:* a router does not need to know anything about the potentially large number of individual connections passing through it; it simply forwards each IP packet based on the destination address contained in the packet header. The routing table entries are highly aggregated—a single entry like 128.01.* provides the next hop information for all destinations that start with prefix 128.01. When multiple entries match a packet's destination, the router uses the longest matching prefix rule to forward the packet [2, 5, 11, 14, 16, 19].

The best-effort service model works well when there is no congestion in the network and the end applications are relatively insensitive to delay (such as file transfer). In reality, the network is frequently and heavily congested, and a large number of emerging applications are real-time, meaning they are extremely sensitive to delay, such as audio, video, or IP telephony. During network congestion, a router needs to give priority to real-time traffic over non-real-time traffic, and thus adopt a "differentiated services" model. Such a differentiated services model is also attractive to ISPs (Internet Service Providers), who need better traffic management so they can offer different quality services to different customers at different prices.

A differentiated service model can be implemented by maintaining per-flow state in the routers, as proposed by protocols like RSVP and Intserv [1, 15]. Stateful routers can provide more powerful and flexible services such as bandwidth allocation, end-to-end latency bounds, protecting well-behaving flows from misbehaving ones, and end-to-end congestion control [8]. Unfortunately, maintaining per-flow state in routers can be prohibitively expensive because the number of flows can be in the hundreds of thousands. Therefore stateful routers do not scale to large sizes as well as the stateless routers. In this paper, we formulate and solve a problem, called *flow aggregation*, which we hope can make stateful routers more scalable. Before we describe flow aggregation, let us briefly explain how current stateless routers forward packets.

In the IP address scheme, each network is assigned a network address, and each host in that network uses the network address as its prefix. Each router maintains a routing table, containing a set of network address prefixes; associated with each prefix is a "next hop" label. Thus, an entry $\langle 128.01.*, A\rangle$ says that a packet whose destination address starts with 128.01 should be forwarded to router A; the router A will forward the packet closer to the packet's ultimate destination. (The symbol '$*$' is the wildcard character.)

The routing table entries are highly aggregated—while there are millions of IP hosts, the largest backbone routers have about 50 thousand prefixes [12]. This aggregation has several advantages—smaller table size reduces table memory, improves search time, and it also reduces the routing update traffic (routers communicate using the Border Gateway Protocol to inform each other about link or router failures, which triggers modifications to the next hop field). The aggregation does have a cost—to look up a packet's next hop, we need to find the longest prefix matching the header, which is a more complicated operation than a simple index into a table. For instance, suppose that a router has three prefixes $0*$, $010*$, and $01010*$, with corresponding next hops A, B and C. Then, a packet with destination address 010101 matches all three but is sent to C. On the other hand, a packet with address 011001 is sent to A.

1.1 Flow-Based Routing

The simple stateless routing, which works well when the network has sufficient capacity and no congestion, is grossly inadequate for real-time applications, such as audio or video, that have stringest delay requirements. *Stateful* routers can implement more sophisticated routing and packet scheduling by using additional packet header fields and by maintaining information about flows and applications. For

instance, an ISP can provide guaranteed quality of service to a company by routing all traffic between two company sites along a high bandwidth channel, which requires the routers in the ISP network to maintain state information over network address pairs $(src, dest)$. In this paper, we will use $(src, dest)$ pairs to illustrate ideas, though all the results carry over for any header field pair, such as destination address and host application.

A *flow* is defined as a pair $(src, dest)$, where *src* and *dest* are network address prefixes, each at most w bits long; in IP version 4, these addresses are at most 32 bits. We define a *flow routing entry* to be a tuple $\langle (src, dest),\ action \rangle$, where *action* is the routing action associated with the flow $(src, dest)$. The routing action typically is the address of the next hop router to which the packet should be sent, but its exact semantics is irrelevant to our abstract framework; in some applications, the action could also take the form of "do not forward the packet" which is useful for access control: an ISP may not permit certain flows to pass through its network [3, 4].

We say that a flow routing entry $(src, dest)$ *matches* a packet P if *src* is a prefix of the packet's source address, and *dest* is a prefix of the packet's destination address. Thus, a packet with header $(0011, 1100)$ matches the flow $(00*, 1*)$, but not the flow $(00*, 10*)$. Let $\mathcal{D}$ denote a table of N flow routing entries. Given packet header P, it is possible that more than one flow entries of $\mathcal{D}$ match P, in which case we define the best matching flow, as follows. Suppose two flow entries, F_1 and F_2, match P. We say that F_1 is a better match than F_2 if each field of P has a longer match with F_1 than F_2. The *best matching flow* of P is the flow that is a better match than any other flow in $\mathcal{D}$. For instance, if we consider a packet header $(0011, 1100)$, and two flow entries $F_1 = (001*,\ 110*)$, and $F_2 = (00,\ 1*)$. Then, F_1 is the best matching flow for the packet.

In order for the best matching flow to be well-defined, the flow entries must be *consistent*, that is, there cannot be two flow entries that partially overlap in the flow address space. We say that a flow routing table $\mathcal{D}$ is consistent if for any two flow entries F_i and F_j either F_i and F_j are disjoint, or one is a subset of the other. Because the primary motivation for flow-based routing is to uniquely classify flows, we will be interested only in consistent flow routing tables.

1.2 Flow Aggregation and Our Contribution

In a consistent flow routing table, each packet header has a unique best matching flow. We say that two flow tables are *equivalent* if each possible packet header receives the same routing action in both tables (using the best matching flow rule). We can define the *flow aggregation* problem as follows: Given a flow routing table $\mathcal{D}$, compute another table $\mathcal{D}'$ that is equivalent to $\mathcal{D}$ and has the smallest possible number of flow entries. As an example, consider a flow table with the following four entries: $\langle (00*, 10*),\ A \rangle$, $\langle (00*, 11*),\ A \rangle$, $\langle (01*, 10*),\ A \rangle$, $\langle (01*, 11*),\ B \rangle$. The smallest equivalent table for this example has two entries: $\langle (0*, 1*),\ A \rangle$, $\langle (01*, 11*),\ B \rangle$.

Our main result is a fast algorithm for determining the optimal aggregation. If the input table has N flow entries, and K distinct routing actions, and each field (source or destination) has at most w bits, then our algorithms runs in worst-case time $O(NKw^2)$; using quadtree style path-compression [13], the worst-case time can be improved to $O(NK)$, assuming w word size.

A pragmatic question one can ask is this: how are flow entries generated, and should one expect any significant aggregation to be achieved? Indeed, if the flow routing entries were manually generated by a network manager, then one would not expect any significant aggregation by running our algorithm. We expect the flow entries to be generated automatically by various algorithms that are being proposed for dynamic routing and traffic engineering. These protocols can generate a large number of flow entries, and since the number of distinct next hops at each router is much smaller (tens, or at most a few hundred in very large backbone routers) than the number of flows, significant aggregation may be achievable. There are also proposals for using packet traces at ISP boundaries to build virtual circuit paths, such as in multi-protocol label switching (MPLS), which are basically flows routing entries. Like the IP prefix

aggregation in stateless routers, flow aggregation has the benefits of improved lookup time and reduced memory. (Reducing memory also leads to improvement in the lookup time, because a smaller data structure may fit entirely in the fast cache.)

1.3 Previous Work

A lot of work has been done in the networking community on congestion control and end-to-end delay bounds *assuming* that routers maintain flow information [1, 7, 8, 15]. However, we have not seen any algorithmic work on aggregating flow state. Flow aggregation bears some resemblanse to the following image compression problem, which is NP-complete [9]: given an $n \times n$ array of 0's and 1's and an integer K, is there a set of K rectangles that precisely cover all the 1's? In flow aggregation, however, the rectangles of different colors can nest. The flow aggregation problem with *inconsistent flows*, where one uses priority to define best matching flow, however appears to be NP-complete. In this paper, we will exploit the geometric structure of consistent flows to develop an efficient dynamic programming algorithm.

The one-dimensional version of our algorithm solves the flow aggregation problem when flows are defined simply by destination-address prefixes. This turns out to be the *prefix table aggregation* problem, which as we recently learnt, was solved independently by Daves et al. [6], preceding our work by a few months. The main focus and result in [6] is prefix compaction, while our main motivation and contribution is flow aggregation, which is a two-dimensional problem. We do not believe that the algorithm in [6] generalizes to flow aggregation, and we think our geometric interpretation and resulting dynamic programming are central to solving the flow problem. We describe the one-dimensional version of our dynamic program in Section 3 primarily to lay the groundwork for the two-dimensional flow aggregation problem.

1.4 Organization

Our paper is organized as follows. In Section 2, we formulate the flow aggregation problem as a geometric compression problem. In Section 3, we develop our main ideas by describing our algorithm in one dimension. In Section 4, we present our main result: the flow aggregation algorithm. In Section 5, we present some extensions and experimental results. Finally, we conclude in Section 6.

2 Flow Entries as Rectangles

We interpret each flow entry as a geometric rectangle in the two-dimensional IP address space—the two axes are the source and the destination addresses. Since each address uses w bits, the domain is the integer line $[0, 2^w - 1]$ along each axis. The source and destination fields are network address prefixes, and each such prefix encodes a *contiguous range* of addresses. For instance, the prefix $101*$ corresponds to the closed interval $[1010\cdots0,\ 1011\cdots1]$. The prefix ranges have the property that either two ranges are disjoint, or one contains the other. The range of s_1 contains the range of s_2 precisely when s_1 is a *prefix* of s_2. For instance, the range of $10*$ is a superset of the range of $10110*$, but the ranges of $1010*$ and $110*$ are disjoint. A packet header has fully specified source and destination addresses, and thus corresponds to a *point* in the two-dimensional space.

A flow (s, d) corresponds to the rectangle whose projections are the ranges of s and d in their respective dimensions. We denote this rectangle by $R(s, d)$—the points of $R(s, d)$ are precisely the packet headers that match the flow (s, d). To emphasize that we are dealing with special rectangles, we will use the term *prefix rectangle*. We say that two prefix rectangles are *consistent* if they are either disjoint, or one contains the other. The flow table $\mathcal{D}$ is consistent if all its flow entries are pairwise consistent. Figure 1 shows examples of consistent and inconsistent rectangles.

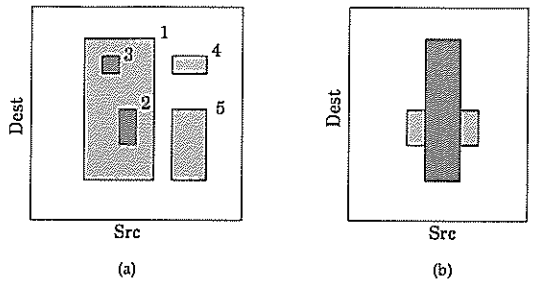

Figure 1: (a) An example showing 5 consistent rectangles. The number at the top-right corner of each rectangle gives its color (action). A point that lies in both rectangles 1 and 3, receives the color 3. A point lying in rectangle 4 gets the color 4. (b) An example of two inconsistent rectangles.

Consider a flow routing table $\mathcal{D}$ with N flow entries. These flows map to N prefix rectangles in the two-dimensional space $[0,\ 2^w - 1] \times [0,\ 2^w - 1]$. We let each distinct *action*, associated with our flows, to be represented by a color, where colors are integers numbered from one to K. Thus, we can think of a flow tuple $\langle (s, d),\ action_i \rangle$ as a prefix rectangle with color i. Since each packet must be classified into some flow, we assume, without loss of generality, that the prefix rectangles of $\mathcal{D}$ completely cover the two-dimensional space $[0,\ 2^w - 1] \times [0,\ 2^w - 1]$. The *flow classification* induced by $\mathcal{D}$ is the mapping from packet headers (points) to the set of colors. Using the best matching flow rule, each packet header receives a unique color: the color assigned to a point is the color of the *smallest* rectangle containing the point. (Refer to Figure 1.)

We can now formulate the flow aggregation problem. Given N prefix rectangles with colors in $\{1, 2, \ldots, K\}$, determine the smallest set of consistent prefix rectangles and their colors that induce the same coloring as the input set. Figure 2 shows an example. We begin by considering the problem in one dimension to help us develop the main idea for our algorithm.

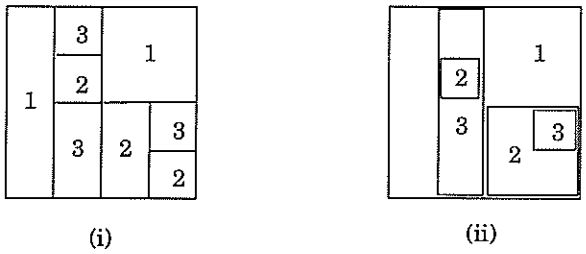

Figure 2: An example of flow aggregation. Fig. (i) shows an input with 8 rectangles, and Fig. (ii) shows an optimal solution using 5 rectangles.

3 Aggregation in One Dimension

Consider a set of N prefixes $\mathcal{D} = \{s_1, s_2, \ldots, s_n\}$, where each s_i is a binary bit string of length at most w, and the ith string is assigned color c_i, with $c_i \in \{1, 2, \ldots, K\}$. Each string s_i corresponds to a contiguous interval on the line $[0,\ 2^w - 1]$, which we call the *prefix range* of s_i, and denote by $R(s_i)$. The set of N prefix ranges partitions the line $[0,\ 2^w - 1]$ into at most $2N - 1$ "elementary intervals," where each elementary interval is the interval between two consecutive range endpoints. Assign to each elementary interval the color of the *smallest range* containing that interval. Under this coloring rule, the prefix set $\mathcal{D}$ is a mapping from the points of the line $[0,\ 2^w - 1]$ to the color set $\{1, 2, \ldots, K\}$. Given a point P, we let $\mathcal{D}(P)$ denote the color assigned to P by the set $\mathcal{D}$. Figure 3 shows an example, where a set of

prefixes partition the line into six elementary intervals. The colors assigned to these intervals, in left to right order, are 2, 1, 2, 3, 2, 3.

We say that two prefix sets $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent* if they induce the same coloring on the line $[0,\ 2^w - 1]$. That is, $\mathcal{D}(P) = \mathcal{D}'(P)$, for all $P \in [0,\ 2^w - 1]$. The one-dimensional prefix aggregation problem can be formulated as follows: Given a set of prefixes $\mathcal{D}$, find the smallest prefix set $\mathcal{D}'$ that is equivalent to $\mathcal{D}$. Figure 3 (ii) shows the optimal solution for the example in (i); the number next to each prefix range is its color.

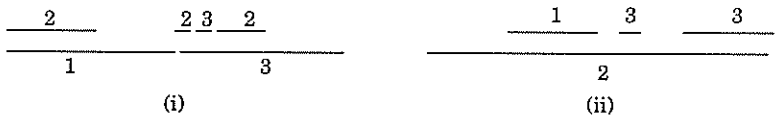

Figure 3: An example of aggregation in one dimension. Fig. (i) shows an input with 6 prefix ranges, and Fig. (ii) shows an optimal solution using 4 prefix ranges.

Our algorithm uses dynamic programming to compute the optimal set $\mathcal{D}'$. We divide a prefix range into two halves, and then try to combine their optimal solutions. One difficulty with this obvious approach is that the combined cost may depend on the actual subproblem solutions. Consider, for instance, the case where we have four equal-length elementary intervals colored 1, 2, 3, 1. The left half subproblem has an optimal solution $\{1, 2\}$; the right half subproblem has an optimal solution $\{3, 1\}$. But adding them together does not give the optimal solution, which has only three prefixes. With this motivation, let us introduce the concept of a *background prefix*.

Consider a prefix s, and its range $R(s) \subseteq [0,\ 2^w - 1]$. Suppose we just want to solve the coloring subproblem for the range $R(s)$. We say that a solution G for $R(s)$ contains a *background prefix* if $s \in G$; that is, one of the prefix ranges in G is the whole interval $R(s)$. The *background color* of G is the color of the background prefix. Fig. 3 (ii) shows an example that has a background prefix with color 2, while the set of prefixes in Fig. 3 (i) does not contain a background prefix. Our dynamic programming algorithm will use the key observation that it is sufficient to consider solutions in which background colors are well-defined.

Lemma 3.1 *Every solution of the coloring problem for a prefix range $R(s)$ can be modified into a solution of equal cost with a background prefix.*

PROOF. Consider a solution without a background prefix. Pick a prefix p in this solution such that the range $R(p)$ is not contained in any other prefix's range. Replace p by s, and give it p's color. □

3.1 The Dynamic Programming Algorithm

We are given a set $\mathcal{D} = \{s_1, s_2, \ldots, s_n\}$ of N prefixes, where each s_i is a binary bit string of maximum length w, and the ith string is assigned color c_i, with $c_i \in \{1, 2, \ldots, K\}$. Consider the coloring induced by $\mathcal{D}$ on the line $[0,\ 2^w - 1]$: a point has the color of the smallest prefix range in which it lies. (Note that the length of a prefix range $R(s_i)$ is inversely proportional to the number of bits in the prefix s_i.) We start by building a partition of $[0,\ 2^w - 1]$ in which each piece is monochromatic and each interval has length a binary power. That is, we recursively divide the line $[0,\ 2^w - 1]$ into two equal halves until each piece is monochromatic. Because $\mathcal{D}$ has N prefixes, and each prefix has at most w bits, our final subdivision has size at most wN.

Let $p_1, p_2, \ldots, p_M$, where $M \leq wN$, denote the prefixes that correspond to the monochromatic intervals in the final subdivision. We call the $R(p_i)$'s *monochromatic binary* intervals. These intervals are the basic subproblems for our dynamic program's initialization. Given an arbitrary prefix range $R(s) \subseteq [0,\ 2^w - 1]$, and a color $c \in \{1, 2, \ldots, K\}$, let us define

$cost(s, c)$ = value of an optimal solution for the range $R(s)$ with background color c.

We initialize this cost function for the monochromatic binary intervals $R(p_i)$, as follows. Let $c_0(p_i)$ be the color of the interval $R(p_i)$—that is, $c_0(p_i)$ is the color induced on $R(p_i)$ by the input prefix set $\mathcal{D}$. Then, for $i = 1, 2, \ldots, M$, we let $cost(p_i, c) = 1$ if $c = c_0(p_i)$, and $cost(p_i, c) = \infty$ otherwise. The following lemma gives the general formula for this cost function. Given a prefix s, we use $s0$ and $s1$ to denote strings obtained by appending to s a 0 and a 1, respectively.

Lemma 3.2 *Let s be an arbitrary prefix. Then, for $c_i = 1, 2, \ldots, K$,*

$$cost(s, c_i) = \min \begin{cases} cost(s0, c_i) + cost(s1, c_i) - 1 & \\ cost(s0, c_i) + cost(s1, c_j) & c_i \neq c_j \\ cost(s0, c_j) + cost(s1, c_i) & c_i \neq c_j \\ cost(s0, c_j) + cost(s1, c_l) + 1 & c_i \neq c_j, c_l \end{cases}$$

PROOF. Omitted from this extended abstract for lack of space. □

If the input $\mathcal{D}$ has N prefixes, the number of colors is K, and the prefixes are w bits long, then the dynamic program based on Lemma 3.2 takes $O(NKw)$ time and space. When we implemented our algorithm, we found that the worst-case memory requirement for this algorithm was infeasibly large to be of practical value. For instance, for the practical values of interest $N = 50,000$, $w = 32$, and $K = 256$, the dynamic program needs to construct a table of size 4×10^8. Even assuming that each entry takes just 1 word of memory, the worst-case memory requirement for this algorithm is 3200 MB of memory! This motivated us to look for an improved algorithm, which we describe in the next section. Not only does the new algorithm require significantly less memory in practice, but it is also simpler and faster.

3.2 An Improved Dynamic Program

Intuitively, maintaining K distinct solutions, one for each background color, for every subproblem seems like an overkill. (If we were only interested in the *value* of the solution then, we could of course choose not to store the intermediate solutions. However, they are needed for constructing the optimal prefix set.) But as we saw earlier, keeping just one optimal solution does not work. However, we show below that storing just the background colors that give the smallest cost for each subproblem suffices. Let s be an arbitrary prefix, and let $\mathcal{L}(s)$ be the list of background colors that give the minimum cost solutions for $R(s)$. That is,

$$\mathcal{L}(s) = \{c_i \mid cost(s, c_i) \leq cost(s, c_j), \quad 1 \leq i, j \leq k\}$$

Again, we initialize these lists for the monochromatic binary intervals by setting $\mathcal{L}(p) = \{c_0(p)\}$. The following lemma shows how to compute these lists in a bottom-up merge. (Recall that $s0$ and $s1$ are prefixes obtained by appending 0 and 1 to the prefix s.)

Lemma 3.3 *Suppose s is an arbitrary prefix. Then,*

$$\mathcal{L}(s) = \begin{cases} \mathcal{L}(s0) \bigcap \mathcal{L}(s1) & \text{if } \mathcal{L}(s0) \bigcap \mathcal{L}(s1) \neq \emptyset \\ \mathcal{L}(s0) \bigcup \mathcal{L}(s1) & \text{otherwise.} \end{cases}$$

PROOF. We first consider the case $\mathcal{L}(s0) \cap \mathcal{L}(s1) \neq \emptyset$. Since all colors in the intersection set $\mathcal{L}(s0) \cap \mathcal{L}(s1)$ are equivalent, it would suffice to show that $cost(s, c_i) < cost(s, \bar{c})$ whenever $c_i \in \mathcal{L}(s0) \cap \mathcal{L}(s1)$ and $\bar{c} \notin \mathcal{L}(s0) \cap \mathcal{L}(s1)$. It is easy to see that

$$cost(s, c_i) = cost(s0, c_i) + cost(s1, c_i) - 1.$$

(That is, the minimum is achieved by the first term in the expression of Lemma 3.2.) Assume, without loss of generality, that $\bar{c} \notin \mathcal{L}(s0)$. Then we must have $cost(s0, \bar{c}) \geq 1 + cost(s0, c_i)$, and $cost(s1, \bar{c}) \geq cost(s1, c_i)$; if $\bar{c} \notin \mathcal{L}(s1)$, the second inequality is strict. Now, it is easy to check that

$$cost(s, \bar{c}) = \min \begin{cases} cost(s0, \bar{c}) + cost(s1, \bar{c}) - 1 \\ cost(s0, c_i) + cost(s1, \bar{c}) \end{cases}$$

Since $cost(s0, \bar{c}) \geq 1 + cost(s0, c_i)$, and $cost(s1, \bar{c}) \geq cost(s1, c_i)$, it follows that $cost(s, \bar{c}) \geq cost(s0, c_i) + cost(s1, c_i) > cost(s, c_i)$, which proves the claim.

Next, consider the case $\mathcal{L}(s0) \cap \mathcal{L}(s1) = \emptyset$. In this case we show that $cost(s, c_i) < cost(s, \bar{c})$ whenever $c_i \in \mathcal{L}(s0) \cup \mathcal{L}(s1)$ and $\bar{c} \notin \mathcal{L}(s0) \cup \mathcal{L}(s1)$. Let us assume that $c_i \in \mathcal{L}(s0)$, and thus $c_i \notin \mathcal{L}(s1)$. Then,

$$cost(s, c_i) = cost(s0, c_i) + cost(s1, c_j),$$

for any $c_j \in \mathcal{L}(s1)$. Now, since $\bar{c} \notin \mathcal{L}(s0) \cup \mathcal{L}(s1)$, we have $cost(s0, \bar{c}) \geq 1 + cost(s0, c_i)$, and $cost(s1, \bar{c}) \geq 1 + cost(s1, c_j)$. Since

$$cost(s, \bar{c}) = \min \begin{cases} cost(s0, \bar{c}) + cost(s1, \bar{c}) - 1 \\ cost(s0, \bar{c}) + cost(s1, c_j) \\ cost(s0, c_i) + cost(s1, \bar{c}), \end{cases}$$

it follows that $cost(s, \bar{c}) \geq 1 + cost(s0, c_i) + cost(s1, c_j) > cost(s, c_i)$, which completes the proof. □

Lemma 3.3 gives a straightforward dynamic programming algorithm. Starting from the initial color lists of the monochromatic binary intervals, the algorithm computes the lists for increasing longer prefix ranges. When computing the list for prefix s, we set $\mathcal{L}(s) = \mathcal{L}(s0) \cap \mathcal{L}(s1)$ if $\mathcal{L}(s0) \cap \mathcal{L}(s1) \neq \emptyset$; otherwise $\mathcal{L}(s) = \mathcal{L}(s0) \cup \mathcal{L}(s1)$. Once all the lists have been computed, we can determine an optimal color assignment by a top-down traversal. (Details are presented in the full paper.)

The worst-case complexity of the preceding algorithm is $O(NKw)$, since there are $O(Nw)$ subproblems, and the size of a color list is at most K. Thus, from a *worst-case* point of view, the dynamic program based on Lemma 3.3 is not much better than that of Lemma 3.2. However, in practice we found that the list sizes were much smaller than the total number of colors, and thus the memory requirement was substantially improved. We next describe our main result: the dynamic programming algorithm for the flow aggregation.

4 Optimal Flow Aggregation

Consider a set $\mathcal{D}$ of N consistent flows. Each flow (s, d) corresponds to a rectangle $R(s, d)$ in the two-dimensional space $[0,\ 2^w - 1] \times [0,\ 2^w - 1]$. The color of $R(s, d)$ is the color (action) associated with flow (s, d). Using the best matching flow rule, the set $\mathcal{D}$ gives a mapping from the set of points $[0,\ 2^w - 1] \times [0,\ 2^w - 1]$ to the set of colors. Let $\mathcal{D}(P)$ denote the color assigned to point P by $\mathcal{D}$. Geometrically, $\mathcal{D}(P)$ is the color of the smallest rectangle containing P. Our goal is to find the smallest set of consistent flows $\mathcal{D}'$ that realizes the same coloring map as $\mathcal{D}$; that is, $\mathcal{D}(P) = \mathcal{D}'(P)$ for all points P. Our algorithm generalizes the dynamic program of the preceding section.

We start with the observation that any solution can be modified to contain a background flow. The *background flow* for a prefix rectangle $R(s,d)$ is the flow (s,d). We say that a solution G for the rectangle $R(s,d)$ contains the background flow if $(s,d) \in G$. The background color of G is the color assigned to the flow (s,d). Fig. 2 (ii) shows an example that has a background flow of color 1; the set of flows in Fig. 2 (i) does not contain a background flow. The following generalizes the background prefix lemma; we omit its easy proof in this abstract.

Lemma 4.1 *Every solution of the coloring problem for a prefix rectangle $R(s,d)$ can be modified into a solution of equal cost with a background flow.*

Given a prefix rectangle $R(s,d)$, and a color $c \in \{1,2,\ldots,K\}$, define

$$cost(s,d,c) \;=\; \text{value of an optimal solution for rectangle } R(s,d) \text{ with background color } c.$$

The following lemma gives the general formula for this cost function. (Recall that we use the notation $x0$ (resp. $x1$) to denote the bit string x with 0 (resp. 1) appended.)

Lemma 4.2 *Given a prefix rectangle $R(s,d)$, and a color $c_i \in \{1,2,\ldots,K\}$, we have*

$$cost(s,d,c_i) \;=\; \min \begin{cases} cost(s0,d,c_i) \;+\; cost(s1,d,c_i) - 1 & \\ cost(s0,d,c_i) \;+\; cost(s1,d,c_j) & c_j \neq c_i \\ cost(s0,d,c_j) \;+\; cost(s1,d,c_i) & c_j \neq c_i \\ cost(s,d0,c_i) \;+\; cost(s,d1,c_i) - 1 & \\ cost(s,d0,c_i) \;+\; cost(s,d1,c_j) & c_j \neq c_i \\ cost(s,d0,c_j) \;+\; cost(s,d1,c_i) & c_j \neq c_i \end{cases}$$

PROOF. Omitted in this extended abstract. □

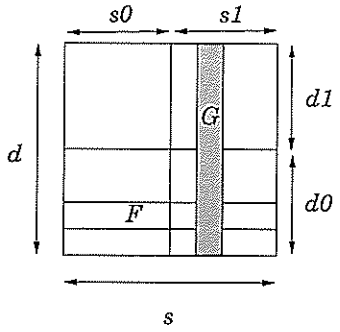

Figure 4: F spans $R(s,d)$ along the s-axis; G spans it along the d-axis.

The key insight in the preceding dynamic program is the following geometric fact: since we are computing consistent rectangles, an optimal solution cannot have two prefix rectangles that cross each other. Thus, an optimal solution for $R(s,d)$ with background color c must be composed of either the optimal solutions of the left and right half subproblems, or the top and bottom half subproblems. More specifically, let us introduce the following definition.

We say that a prefix rectangle $R' = (s',d')$ spans $R(s,d)$ along the s-axis (resp. d-axis) if $s = s'$ and d is a prefix of d' (resp. $d = d'$ and s is a prefix of s'). Figure 4 illustrates this definition. Consistency implies that an optimal solution of $R(s,d)$, with any background color, cannot have rectangles spanning $R(s,d)$ along both axes. The absence of a rectangle spanning along s-axis (resp. d-axis) allows one to combine the left and right (resp. top and bottom) subproblem solutions.

4.1 An Improved Algorithm

As in the one-dimensional case, the dynamic program can be improved in practice (though not in the worst case) by maintaining the list of only those background colors that give optimal solutions. Let $\mathcal{L}(s,d)$ denote the list of colors that achieve minimum cost for the coloring subproblem $R(s,d)$. That is,

$$\mathcal{L}(s,d) = \{c_i \mid cost(s,d,c_i) \leq cost(s,d,c_j), 1 \leq c_i, c_j \leq K\}$$

We use the notation $cost(s,d)$ to denote the minimum cost of $R(s,d)$ over all colors; that is, $cost(s,d) = \min_i cost(s,d,c_i)$. In the following, we use the term "input rectangle" to mean the prefix rectangle corresponding to a flow in the input set $\mathcal{D}$.

Flow-Aggregate (s, d)

1. If no input rectangle of $\mathcal{D}$ lies entirely inside $R(s,d)$, then all points mapped to the region $R(s,d)$ receive the same color c. In this case, we set $\mathcal{L}(s,d) = \{c\}$, $cost(s,d) = 1$, and **return**.

2. If no input rectangle of $\mathcal{D}$ spans $R(s,d)$ along the s-axis, then do the following:

   if $\mathcal{L}(s0,d) \cap \mathcal{L}(s1,d) \neq \emptyset$ **then**
   $cost_v(s,d) = cost(s0,d) + cost(s1,d) - 1$; $\mathcal{L}_v(s,d) = \mathcal{L}(s0,d) \cap \mathcal{L}(s1,d)$
   else $cost_v(s,d) = cost(s0,d) + cost(s1,d)$; $\mathcal{L}_v(s,d) = \mathcal{L}(s0,d) \cup \mathcal{L}(s1,d)$

   Otherwise, set $cost_v(s,d) = \infty$.

3. If no input rectangle of $\mathcal{D}$ spans $R(s,d)$ along the d-axis, then do the following:

   if $\mathcal{L}(s,d0) \cap \mathcal{L}(s,d1) \neq \emptyset$ **then**
   $cost_h(s,d) = cost(s,d0) + cost(s,d1) - 1$; $\mathcal{L}_h(s,d) = \mathcal{L}(s,d0) \cap \mathcal{L}(s,d1)$
   else $cost_h(s,d) = cost(s,d0) + cost(s,d1)$; $\mathcal{L}_h(s,d) = \mathcal{L}(s,d0) \cup \mathcal{L}(s,d1)$

   Otherwise, set $cost_h(s,d) = \infty$.

4. **if** $cost_v(s,d) > cost_h(s,d)$ **then**
   $cost(s,d) = cost_h(s,d)$; $\mathcal{L}(s,d) = \mathcal{L}_h(s,d)$
   else if $cost_v(s,d) < cost_h(s,d)$ **then**
   $cost(s,d) = cost_v(s,d)$; $\mathcal{L}(s,d) = \mathcal{L}_v(s,d)$
   else $cost(s,d) = cost_v(s,d)$; $\mathcal{L}(s,d) = \mathcal{L}_h(s,d) \cup \mathcal{L}_v(s,d)$

The code above describes a generic call on an arbitrary prefix rectangle $R(s,d)$. The initial call is made on the subspace $R(*,*)$, corresponding to the rectangle $[0,\ 2^w - 1] \times [0,\ 2^w - 1]$. In the code, $cost_h, cost_v, \mathcal{L}_h$ and $\mathcal{L}_v$ are temporary variables used for comparing the solutions obtained by either combining the left and right halves of $R(s,d)$, or the top and bottom halves. Due to lack of space, we omit the proof of correctness of this algorithm.

In order to analyze the running time of this dynamic program, we observe that a subproblem $R(s,d)$ makes a recursive call only if $R(s,d)$ contains at least one input rectangle of $\mathcal{D}$ inside it. We can show that the total number of subproblems is $O(Nw)$, the cost of deciding if a rectangular region is spanned by some filter is $O(w)$, and the cost of maintaining color lists per subproblem is $O(K)$. Thus, the total time and space complexity of the algorithm is $O(NKw^2)$ in the worst case.

Theorem 4.3 *Given a set of N consistent flows, with K distinct colors and at most w-bit prefixes, we can compute an optimal flow aggregation in $O(NKw^2)$ worst case time.*

5 Extensions and Experimental Results

5.1 Improving Time Complexity by Path Compression

The dynamic programs of Sections 3 and 4 can be improved to eliminate the w factors, thus resulting in the worst-case running time and space $O(NK)$. The w factors arise due to long non-branching paths in the recursion tree. A standard quadtree style path compression can eliminate such paths, by shrinking the rectangle $R(s,d)$ in each step to ensure that each recursive call separates two input rectangles.

5.1.1 Minimizing the Bit Complexity

We have used the number of flows as our complexity measure. Instead one could ask to minimize the total *bit complexity* of the flow routing table. Algorithms that use tries or bit vectors for flow classification [2, 7, 10, 17] are sensitive to the total number of bits in the routing database. Given a flow $f = (s,d)$, let $b(f)$ denote the bit length of s plus the bit length of d. Then, the *bit complexity* of a flow routing table $\mathcal{D} = \{f_1, f_2, \ldots, f_n\}$ is $\sum_{i=1}^{n} b(f_i)$. We could ask for a routing table of minimum bit complexity that is equivalent to $\mathcal{D}$. It turns out that our dynamic programs also minimizes the bit complexity of the output table.

5.2 Experimental Results

We implemented our dynamic programming algorithms, for both one- and two-dimensional aggregation. We do not have any publically available flow databases to test our two-dimensional algorithm, since the stateful routers are still in their infancy. On the other hand, prefix tables are widely available for large backbone routers, so we were able to test our one-dimensional aggregation algorithm. We ran our algorithm on three publically available routing tables, obtained from the Mae-East Exchange Point [12]. The number of prefixes in these databases varied from about 8000 (Paix) to about 41000 (Mae-East). The total number of colors (distinct next hops) varied from 17 to 58. Table 1 below shows our results. While one-dimensional results are no indication of the two dimensional problem, it should be encouraging that our prefix aggregation algorithm achieves compression of 30-40% even in these highly aggregated prefix tables. It therefore appears likely that significant aggregation might be possible in the flow routing tables, which are going to be automatically generated.

Database	Input	Output	Reduction	Memory	Time
Mae-East	41455	23680	42.88%	3.8 MB	2.73 s
PacBell	24728	14168	42.70%	2.1 MB	1.85 s
Paix	7982	5888	26.23%	0.8 MB	0.72 s

Table 1: One dimensional prefix aggregation. When multiple next hops were available for a prefix, we initialized the corresponding color list with all those next hops. The input and output are the number of prefixes.

6 Concluding Remarks

We gave an efficient algorithm for computing an optimal flow aggregation for reducing state information in IP routers. The algorithm is relatively simple, and exploits some basic geometric properties of consistent prefix rectangles in two dimensions. The basic dynamic progrmming algorithm runs in $O(NKw^2)$ worst-case time, for N flows with K colors and w bit prefixes. While the improved dynamic program does not reduce the worst-case complexity, it should be substantially better in practice. With path compression in the recursion tree, the worst-case time can be reduced to $O(NK)$.

The IP routers certainly need to move beyond the current best-effort service model, if they are to be used for advanced services like audio, video, or IP telephony. The past history has shown that highly stateful solutions like ATM (asynchronous transfer mode) have failed to be widely adopted despite their many ability to provide quality of service. Achieving similar capabilities in IP routers with minimal per-flow state appears to be the most

promising alternative. Our hope is that algorithms like ours for flow aggregation will make stateful routers more scalable, and thus more acceptable.

References

[1] R. Braden, L. Zhang, S. Berson, S. Herzog and S Jamin. Resource Reservation protocol (RSVP)–Version 1, Functional Specification. RFC 2205, Sept. 1997.

[2] A. Brodnik, S. Carlsson, M. Degermark and S. Pink. Small Forwarding Table for Fast Routing Lookups. *Computer Communication Review*, October 1997.

[3] D. B. Chapman and E.D. Zwicky. *Building Internet Firewalls.* O-Reilly & Associates, Inc., 1995.

[4] W. Cheswick and S. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, 1995.

[5] G. Cheung and S. McCanne. Optimal Routing Table Design for IP Address Lookups Under Memory Constraints. *Proc. of INFOCOM '99.*

[6] R. Daves, C. King, S. Venkatachary and B. Zill. Constructing Optimal IP Routing Tables. *Proc. of INFOCOM '99.*

[7] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *Proc. of ACM SIGCOMM*, 1998.

[8] A. Demers, S, Keshav and S. Shenker. Analysis and simulation of a fair queueing algorithm. Journal of Internetworking Research and Experience, pp. 3–26, 1990.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, 1979.

[10] T. V. Lakshman and D. Stidialis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *Proc. of ACM SIGCOMM*, 1998.

[11] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multi-way and Multicolumn Search. *Proc. of IEEE INFOCOM '98.*

[12] Merit, Inc. ftp://ftp.merit.edu/statistics/ipma. Routing Table Snapshot, 14 Jan '99, Mae-East NAP.

[13] J. Mitchell, D. Mount and S. Suri. Query-Sensitive Ray Shooting. *Int. Journal of Computational Geometry & Applications*, pp. 317–347, 1997.

[14] S. Nilsson and G. Karlsson. Fast Address Look-Up for Internet Routers. *Proceedings of IEEE Broadband Communications 98*, April 1998.

[15] S. Shenker, R. Braden and D. Clark. Integrated services in the Internet Architechture: an overview. Internet RFC 1633, June 1994.

[16] V. Srinivasan and G. Varghese. Faster IP Lookups using Controlled Prefix Expansion. *Sigmetrics'98.*

[17] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast Scalable Level Four Switching. *Proc. of ACM SIGCOMM*, 1998.

[18] V. Srinivasan, S. Suri and G. Varghese. Packet Classification using Tuple Space Search. *Proc. of ACM SIGCOMM*, 1999.

[19] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. *Computer Communication Review*, October 1997.