Understanding mergeability

July 22, 2017

This document is a supplement to the 2017 SIGCOMM submission entitled "Language-Directed Hardware Design for Network Performance Monitoring". It provides details of the merge procedure introduced in Section 3.2 and proofs for the theorems outlined in Section 3.4.

1 Details of the merge procedure in Section 3.2

For queries that are linear in state, the state update takes the form $\mathbf{S} = A(\mathbf{p}) \cdot \mathbf{S} + B(\mathbf{p})$, where S is a vector of state, and A and B are functions of the previous k packets (denoted by $\mathbf{p}$). Here k is an arbitrary integer.

1.1 Single packet history

Consider the case where $k = 1$. If the state is $\mathbf{S}$ at any point in time, then after N packets, $\mathbf{S_N} = A^N \cdot \mathbf{S}+$ terms independent of S. Here, A^N is shorthand for $A(p_N) \cdot A(p_{N-1}) \cdot \ldots \cdot A(p_1)$ Therefore to merge an evicted value $\mathbf{S_N}$ with the existing value $\mathbf{S}_{backing}$ in the backing store, we need to replace $\mathbf{S}$ with $\mathbf{S}_{backing}$ in the expression for $\mathbf{S_N}$ by computing:

$$\mathbf{S_N} - A^N \cdot \mathbf{S} + A^N \cdot \mathbf{S}_{backing} = \mathbf{S_N} + A^N(\mathbf{S}_{backing} - \mathbf{S})$$

The merge procedure is straightforward: the switch keeps A^N as auxiliary state and passes it with $\mathbf{S_N}$ to the backing store to complete the merge. The backing store already knows S, since it is the default starting value for the state, which does not change.

1.2 Bounded packet history

The required auxiliary state is more complex for larger values of k. If $k = 1$, $A(p)$ and $B(p)$ are known to the switch for every packet. However, for larger values of k, the switch needs access to older packet fields to compute $A(\mathbf{p})$ and $B(\mathbf{p})$. This means the values of A and B are themselves incorrect for the first $k - 1$ packets after an eviction, an issue that must be addressed by the merge procedure.

Consider Figure 1 for $k = 3$. Once the switch performs an eviction at T_1, the switch cannot properly update the its state for the next two packets (first packets of the subsequent epoch). It thus starts updating its state from the third packet onwards and entrusts the backing store to fill the "hole" caused by missing the first two state updates.

To perform the merge successfully, the backing store needs two additions pieces of information, in addition to the query-specific state A^N discussed in the previous section:

- The last $k - 1$ packets from the *previous epoch*, called the *exit log* R of the previous epoch.
- The first $k - 1$ packets from the *current epoch*, called the *entry log* X of the current epoch.

Upon merging, the backing store receives S, R, and X, and must merge them with $S_{backing}$. First, the backing store runs the actual aggregation function over R starting from $S_{backing}$: $S' = g(S_{backing}, R)$. Doing this requires the backing store to use the exit log of the *previous* epoch. The backing store then performs a standard merge $S'' = m(S', S)$ and stores X for use in the *next* epoch's merge.

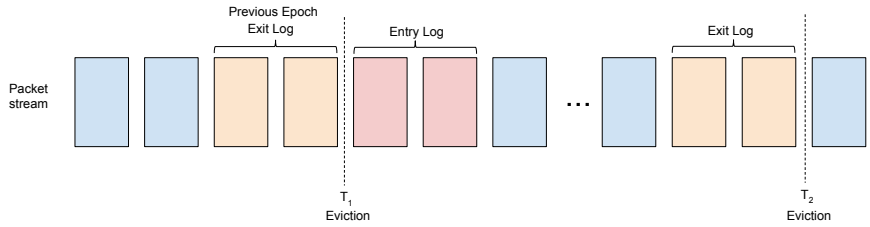

Figure 1: Entry and exit log for an eviction, with $k = 3$.

2 Proofs of theorems in Section 3.4

2.1 Notation

A programmer supplies an aggregation function f. f takes as inputs an n-bit *state* vector and a p-bit *packet header* vector and returns an updated n-bit *state* vector. f captures incremental computations over packet streams, such as a count of packets or an exponentially weighted moving average of packet latencies. f can only represent incremental computations where the size of state is fixed and doesn't grow with the length of the packet stream. For example, f can't represent a median over the packet stream.

Given f, we want two implementation functions: an incremental function implementation g and merge function implementation m that can merge results from running f over separate packet streams. Specifically, can we merge two results from running f on two packet streams into one result, equivalent to running f on the concatenated stream? Formally, we suppose that the state used by the implementation is $n' \geq n$ bits. The paper refers to the merge function using a auxiliary bits. Here, $a = n' - n$.

$$
\begin{aligned}
f : \{0,1\}^n \times \{0,1\}^p &\to \{0,1\}^n \text{(programmer's incremental computation)} \\
g : \{0,1\}^{n'} \times \{0,1\}^p &\to \{0,1\}^{n'} \text{(implementation's incremental computation)} \\
m : \{0,1\}^{n'} \times \{0,1\}^{n'} &\to \{0,1\}^{n'} \text{(implementation's merge operation)} \\
h : \{0,1\}^{n'} &\to \{0,1\}^n \text{(transform implementation state to programmer state)} \\
s_0 &\in \{0,1\}^n \text{(programmer's start state)} \\
s'_0 &\in \{0,1\}^{n'} \text{(implementation's start state)} \\
m(s'_0, s') &= s' \quad \forall s \in \{0,1\}^n (s'_0 \text{ is an identity for } m) \\
(g, m, h, s'_0) &\text{ merges } (f, s_0) \text{ if} \\
s_0 &= h(s'_0) \text{(implementation's start matches with programmer's start)} \qquad (1) \\
f(f(f(s_0, P_1), P_2), \ldots, P_n) &= h(g(g(g(s'_0, P_1), P_2), \ldots, P_n)) \ \forall n \in \mathbb{N} \quad P_1, P_2, \ldots, P_n \in \{0,1\}^p \qquad (2) \\
h(g(g(g(s'_0, P_1), P_2), \ldots, P_{i+j})) &= h(m(g(g(g(s'_0, P_1), P_2), \ldots, P_i), \\
&\qquad g(g(g(s'_0, P_{i+1}), P_{i+2}), \ldots, P_{i+j}))) \\
\forall i, j \in \mathbb{N} \quad P_1, P_2, \ldots, P_{i+j} &\in \{0,1\}^p \qquad (3)
\end{aligned}
$$

Here f, g, m, and h are assumed to be efficiently computable functions. We use the notation $f(s, \{p_1, \ldots p_k\})$ to denote the composition of f over a sequence of packets, similar to a fold function. Additionally, we say that a merge function successfully merges an aggregation function if Equation 3 holds.

2.2 The closure graph

Before proceeding with the proofs, we introduce the directed *closure graph* of f, denoted $G(f)$. Suppose that at some point during the aggregation, the programmer specified state is s. After running through some

additional packets, that state is updated to s'. If s is known, then computing s' is straightforward: simply execute the aggregation f on all the packets seen in the interim. However, if s is *not* known, we can instead use the packets seen to compute a function that tells us how to get from *any* state s to the final desired state s'. That is, given a sequence of packets $\{p_k\}$, we can compute an *iterated* function $f_i \in \{0,1\}^n \to \{0,1\}^n$, such that for any state s, $f_i(s) = f(s, \{p_k\})$. For an empty sequence, the corresponding iterated function is the identity. Our goal is to store some compact representation of the iterated function in the on-chip cache. This iterated function is updated with each new packet that is seen, so that when a key is evicted from the on-chip cache, an identifier for the updated iterated function is sent to the backing store. The backing store then applies this iterated function to the state $s_{backing}$ stored from a previous eviction to get the new value.

How does one update the iterated function stored on the switch? The vertices of the closure graph of f are iterated functions f_i, and we let $|G(f)|$ indicate the number of vertices. Since there are a finite number of functions $\{0,1\}^n \to \{0,1\}^n$, the size of the graph is bounded. Each vertex has an outgoing edge corresponding to every possible packet. For a given packet p, the iterated function f_i has an edge to the function f_j satisfying $f(f_i(s), p) = f_j(s)$ for all s. The closure graph thus has the following property: given a sequence of packets $\{p_1, \ldots p_k\}$, start at the identity function and, on the ith step, follow the edge corresponding to each packet p_i. The ending vertex of this process is the iterated function that captures the effect of this packet sequence on any starting state. We say that this ending vertex is the result of *updating the iterated function* by the given sequence of packets. To update the iterated function stored on the switch given a new packet, simply follow the edge labeled with that packet.

2.3 Proofs of theorems

Theorem 1 (Publication Theorem 3.1)**.** *A merge function exists to successfully merge an aggregation function f, provided it can use up to $n2^n$ auxiliary bits.*

Proof. We can represent each iterated function f_i in $n2^n$ bits, using a list of 2^n $n-$bit numbers, where the kth item is $f_i(k)$. The switch stores this representation. For each new packet p, the switch computes the update to each number in the list: $f_i(k) \to f(f_i(k), p)$. Upon a merge, the switch sends this representation to the backing store, which can then evaluate $f_i(s_{backing})$, where $s_{backing}$ is the value stored in the backing store. □

Note that the switch needs to tell the backing store what update to perform on the value it currently has. Each iterated function in the closure graph is a possible update, so the switch needs to convey at least $\log |G(f)|$ auxiliary bits for the merge to be possible.

As an example, consider a simple counter, where $f(s,p) = s + 1$ and $s_0 = 0$. If the counter has n bits, the closure graph is a ring of size 2^n, where the ith vertex encodes the iterated function $f_i(s) = s + i$. Note that all edges from the ith vertex point to the $i + 1$st vertex. For every observed packet, the switch walks one step further around the ring. This is implemented efficiently via adding 1 to an n-bit counter. Upon merging after K packets, the iterated function on the switch is f_k, where $k = K \mod 2^n$. Upon merging, this is sent to the backing store, which then knows to add k to the existing backing store value.

In general, requiring $n2^n$ bits is typically infeasible for even moderate values of n. This is where the linear-in-state and associative conditions come in. We show that either of these conditions is enough to require only $O(n)$ auxiliary bits of space.

Theorem 2 (Publication Theorem 3.2)**.** *Aggregation functions that are either linear-in-state or associative have a merge function that uses only $O(n)$ auxiliary bits.*

Proof. The associative condition is trivial: by definition, no auxiliary bits are required. Suppose f is an aggregation function that is linear in state, i.e. $f(\mathbf{S}, p) = \mathbf{A}(p) \cdot \mathbf{S} + \mathbf{B}(p)$, where $\mathbf{S}$ is a state vector, and $\mathbf{A}, \mathbf{B}$ are matrices depending only on the incoming packet. Then:

$$f(\mathbf{S}_0, \{p_1, \ldots, p_k\}) = \mathbf{A}(p_k) \ldots \mathbf{A}(p_1) \cdot \mathbf{S}_0 + \mathbf{C}(p_1, p_2, \ldots, p_k) \equiv \mathbf{A}' \cdot \mathbf{S}_0 + \mathbf{C}(p_1, \ldots, p_k)$$

where $\mathbf{C}$ is some function of only the packets and $\mathbf{A}'$ is a composition of the $\mathbf{A}$ matrices for the observed packets. The switch keeps track of $\mathbf{A}'$ and $\mathbf{C}$, which requires $O(nd^2)$ space where d is the number of pieces of state, each of which we assume has size n. These matrices are sent to the backing store upon eviction, at which point the merge operation computes the new backing store value:

$$\mathbf{S}_d \leftarrow \mathbf{C}(p_1, \ldots, p_k) + \mathbf{A}' \cdot \mathbf{S}_d$$

It's easy to verify that this is the proper merge procedure. □

However, there are some functions that are not mergeable with so few auxiliary bits. Take the following TCP-non-monotonic function:

```
def nonmt(maxseq, count, tcpseq):
  if maxseq > tcpseq:
    count = count + 1
  else:
    maxseq = tcpseq
  nm_q = groupby(pktstream, 5tuple, nonmt);
```

Theorem 3 (Publication Theorem 3.3)**.** *The TCP-non-monotonic function requires at least $n2^n$ auxiliary bits to merge.*

Proof. Two pieces of state must be stored: the max sequence number and the count. For simplicity, we assume each piece of state requires n bits. We present a family of packet sequences, such that there is an injection between the sequences and vertices in the closure graph of f. In other words, starting from the identity, updating the identity iterated function by each sequence in this family will result in a distinct iterated function.

Consider a family of packet sequences parameterized by a tuple $(a_0, a_1, \ldots a_{2^n-1})$: the sequence consists of a_i packets with sequence number i, in increasing order of i. For each such tuple, performing an update U of the identity iterated function by the sequence for that tuple results in the following final iterated function:

$$U(a_0, \ldots, a_{2^n-1}) = f \qquad \text{such that} \qquad f(x, y) = \left(2^n - 1, (y + \sum_{i=0}^{x-1} a_i) \mod 2^n\right)$$

where x and y are the max sequence number and count, respectively. To see why this is the case, recall that the iterated function is taking some *previous* max seq. number and count and trying to update it by the packet sequence. If the max sequence number was $x = k$ before, then the count will increase by the number of packets with sequence number $< k$, which is $\sum_0^{k-1} a_i$.

Lemma 3.1. *U is an injection.*

Proof. If $(a_0, \ldots, a_{2^n-1}) \neq (a'_0, \ldots, a'_{2^n-1})$, then there is some k such that $a_k \neq a'_k$ but $a_j = a'_j$ for all $j < k$. Note: $k = 0$ may satisfy this condition. Let $U(a_0, \ldots, a_{2^n-1}) = f$ and $U(a'_0, \ldots, a'_{2^n-1}) = f'$. Then, for any y:

$$f(k+1, y) - f'(k+1, y) = \left(0, \sum_{i=0}^{k} a_i - \sum_{i=0}^{k} a'_i\right) = (0, a_k - a'_k) \neq (0, 0)$$

so $f \neq f'$, making U an injection. □

There are 2^{n2^n} such packet sequences, meaning that the closure graph has at least 2^{n2^n} distinct vertices, and thus number of auxiliary bits needed is at least $\log 2^{n2^n} = n2^n$. □

Now we turn to the question: given an aggregation function, how easy is it to compute a merge function that uses the minimal number of auxiliary bits? Since we know that the minimal number of bits is $\log |G(f)|$, we can construct $G(f)$. This approach is extremely inefficient, but it works.

Theorem 4. *Given an aggregation function f, an algorithm exists to compute a merge function using the minimal number of auxiliary bits.*

Proof. To construct the closure graph, we start with a single node representing the identity function. Starting at this node f_0, we enumerate all p-bit values for a single packet P, and compute the updated iterated function f_P satisfying $f_P(s) = f(f_0(s), P)$ for all s. If this iterated function has not been seen yet, we create a new node for it. We join two nodes by an edge labeled with the packet value that causes the transition between those two nodes and repeat this process from each newly created node until all edges out of a node lead back to existing nodes[1].

There are 2^{n2^n} possible iterated functions. We assume that given an iterated function, it is possible to find that function in the existing partial closure graph in time $O(n2^n)$, the number of bits needed to represent the number of iterated functions in the worst case. Then, since we must enumerate every possible p-bit packet the total runtime is:

$$O(\#\text{ iters} \cdot \#\text{ packets} \cdot \#\text{ time to test if function has been seen}) = O\left(2^{n2^n} \cdot 2^p \cdot n2^n\right)$$

□

This algorithm is intractable in practice. However, a polynomial time algorithm is also likely intractable. We demonstrate this hardness result by considering a decision version of this problem: given an aggregation function f and merge function m, does m successfully merge f for all possible packet inputs? This problem turns out to be co-NP-hard.

Before we launch into the hardness proof, we must limit the scope of the problem. "All possible packet inputs" includes packet sequences of arbitrary length. However, we can restrict ourselves to sequences up to 2^n packets.

Theorem 5. *To determine whether m merges f on every packet sequence, it is sufficient to consider packet sequence up to length 2^n.*

Proof. We show it is sufficient to check equation 2 on packet streams of length $L <= M = 2^{max(n,n')}$. This check fails only if there is a stream P_{false} of length $L > M$ that falsifies equation 2. However, there must then also be a smaller stream P_{small} of length $L' <= M$ that falsifies equation 2.

To see why, let's say packet A within P_{false} first falsifies equation 2. If $A's$ position within P_{false} is less than or equal to M, the substream up to and including A is P_{small}. If not, running f repeatedly on the substream of packets until A will result in some state value s being repeated twice, after seeing (say) packets P_1 and P_2. We can remove all packets after P_1 and before P_2, creating a shorter stream that still falsifies 2. This process can be repeated until the stream length is less than or equal to M. This will eventually happen because there will always be a repeated state value by the pigeon-hole principle because the number of packets in a stream is greater than the number of distinct states. □

Now we know that checking whether m successfully merges f is actually possible. However, it is likely intractable. We'll show this by showing that it is hard to perform the simple action of *verifying* that a *given* merge function successfully merges a given aggregation function.

Theorem 6 (Publication Theorem 3.4)**.** *Given a merge function m and aggregation function f, verifying that m successfully merges f is co-NP hard.*[2]

Proof. We'll show that $MERGE$ is co-NP-hard by reducing $TAUTOLOGY$ (a known co-NP-complete problem) to $MERGE$.

An instance of $TAUTOLOGY$ is a function t. The output is 1 when t is a tautology, 0 otherwise. $TAUTOLOGY(t)$ can be reduced to[3] $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$,[4]where

[1]We need a way to check node equality, i.e., function equality. We do this by a brute force check on all inputs to both nodes/functions.

[2]co-NP is made up of problems that are complements of problems in NP.

[3]This is a polynomial many-to-one reduction or a Karp reduction.

[4]t'(P) = (NOT t(P))

1. $s \in \{0,1\}$
2. $t : \{0,1\}^p \to \{0,1\}$
3. OR is the boolean OR function on two bools.
4. I is the identity function.

First, we observe that $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$ decides whether statement 4 is true:

$$t'(P_1) \oplus t'(P_2) \oplus \ldots t'(P_{i+j}) = (t'(P_1) \oplus t'(P_2) \ldots t'(P_i)) OR (t'(P_{i+1}) \oplus t'(P_{i+2}) \ldots t'(P_{i+j})) \quad (4)$$
$$\forall i, j \in \mathbb{N} \quad P_1, P_2, \ldots, P_{i+j} \in \{0,1\}^p$$

Next, we prove a lemma that shows the reduction from $TAUTOLOGY$ to $MERGE$.

Lemma 6.1. *Statement 4 is true iff* $t'(P) = 0 \ \forall P$.

Proof. Let's suppose $t'(P) = 0 \ \forall P$. It is easy to see that statement 4 reduces to 0 on both the LHS and RHS. If on the other hand, $t'(P^*) = 1$ for some P^*. Then, we set:

1. $i = 1$
2. $j = 2k - 1$
3. $P_1 = P_2 = \cdots = P_{i+j} = P^*$

The LHS is an XOR over an even number of 1s, which is 0. The RHS is an OR of 1 and something else, which is 1. So, we have found one setting of the quantified variables that falsifies statement 4. □

It is straightforward to see how $TAUTOLOGY(t(P))$ reduces to $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$, showing that $MERGE$ is co-NP-hard. □