

# Addressing OSPF Load in IP/MPLS Networks

## Abstract

There is an increasing trend towards deploying IP/MPLS networks that provide value added services such as virtual private networks, guaranteed bandwidth pipes, etc. to network users. Such networks are required to be highly reliable and robust even under peak load and/or unanticipated situations of link/router failures. TE mechanisms typically involve path computation based on information obtained through OSPF or other link state routing mechanisms and then using a signaling mechanism such as RSVP-TE to set up the computed path in the network. As routers in such a network have to endure a higher computation and protocol processing load compared to routers in a best effort IP network, these mechanisms will have to be significantly scalable in robust large-scale TE networks. Path computation mechanisms have been studied extensively and it is generally accepted that such mechanisms can be implemented within the means of current router architectures. The link state routing overhead however, can still be significant even when using intelligent path computation mechanisms. Proposed mechanisms to decrease this overhead require changes to the protocols and are therefore not easily deployable given the large installed base of protocol implementations.

In this paper, we show that OSPF protocol overhead is indeed a significant component of the load offered on routers and that this load can become unacceptable as the degree of the node increases. We then outline an implementation mechanism that takes advantage of the evolving router architectures which helps scale OSPF implementations significantly.

## Keywords

OSPF, link state routing, traffic engineering, code profiling

## I. INTRODUCTION

The Internet is a collection of network domains or Autonomous Systems (ASes) where each AS uses an *interior routing protocol* within the domain and an *exterior routing protocol* to maintain inter-AS connectivity [?]. OSPF [?] is one of the most widely used interior routing protocols while BGP [?], [?] is in predominant use as an exterior routing protocol. Traditionally, routing has been performed along the shortest path from source to destination with best effort service. OSPF computes routing tables using *shortest path trees* (SPT) while BGP mostly uses shortest AS-path length as the metric. OSPF is a link state routing protocol and relies on reliable flooding procedures to maintain a consistent snapshot of network topology at every router in the network. Each router then, typically, computes the shortest path to every other router in the network and uses this information in data forwarding. *Dijkstra's* shortest path computation algorithm is widely employed in OSPF implementations [?], [?].

It is well known that as the size of the network increases OSPF implementations can become unstable due to processing overload caused by excessive flooding and/or by frequent *Dijkstra* executions during periods of network instability. The solution adopted by OSPF in scaling to large networks is a 2-level area hierarchy wherein a network is split into multiple areas all interconnected through a special area known as the backbone

area. The flooding scope of topology information is limited to a given area, and information related to a given area is made available to other areas in a summarized manner. It is possible, however, that in a sufficiently large network, one or more areas may themselves be very large. OSPF implementations also apply some intelligent mechanisms to help scalability. For example, to avoid overwhelming the router CPU(s), routers typically recompute the SPT periodically instead of recomputing as soon as a change is detected. In addition, jittered timers are employed to alleviate flooding spikes during certain link state information flooding procedures. A combination of such techniques has helped OSPF implementations perform very well in real commercial networks.

Three emerging developments, however, motivate the need for OSPF implementations that can scale further. They are (1) *deployment of Internet traffic engineering (TE) mechanisms*, (2) *emerging high capacity router architectures*, and (3) *renewed interest in millisecond convergence*. The reasons as to why each of these increases the OSPF processing load on the routers are discussed below.

First, in the context of emerging IP/MPLS networks, there is an acute desire on the part of network service providers to deploy traffic engineering (TE) mechanisms in their networks. The goal of this deployment is to optimize network capacity utilization while also providing service guarantees to their network users. Interest in TE mechanisms has prompted router vendors to define various extensions to existing protocols to enable TE deployment. Indeed, within the IETF, OSPF-TE [?] and RSVP-TE [?] have emerged as the routing and signaling components, respectively. OSPF-TE refers to the standard OSPF protocol plus all its TE related extensions. RSVP-TE refers to the RSVP mechanisms defined for setting up MPLS Label switched paths (LSPs). TE related extensions to OSPF involve advertisement of additional link state information such as available bandwidth for every link in the network. As new LSPs are set up or old ones removed, such link information changes. Whenever such a change is deemed significant, routers re-advertise this information using flooding procedures. In general terms, TE issues have been studied under the umbrella of QoS routing. Traffic demand patterns clearly play a role in determining the frequency of TE related link state advertisements. If the frequency of such advertisements is very low, the information available in every routers link state database can become very stale. It has been shown that stale information may limit the benefits of richer network connectivity. It has also been suggested that in order to capitalize on dense network topologies, link state updates should be more frequent and as a result there is a need for techniques for dealing excessive link state traffic [?].

Secondly, since OSPF is a link state routing protocol, the flooding load offered on a router is related to its degree and the "connectedness" of the network. Information about a router in the network is received from every neighbor through which that router can be reached. Therefore, the larger the degree and the "richer" the connectivity, the larger the amount of redundant information is received at a router. Existing high end routers

can support up to a hundred interfaces and could possibly have up to have up to tens of OSPF neighbors. An important recent development is the trend towards very high capacity switch fabrics enabling next generation routers with multiple terabits of backplane capacity and potentially many hundreds of interfaces. Such next generation routers could possibly have hundred or more OSPF neighbors, thus causing significantly higher OSPF protocol load. It has been shown in [?] that this high degree combined with link or router flapping leads to very high protocol overhead for that time interval.

Thirdly, independent of whether TE mechanisms are deployed or not, there has been an increased interest in addressing convergence issues in routing protocols. For link state routing protocols, it has been shown in [?], that better convergence times can be achieved with sub-second hello intervals and incremental & frequent SPT computations. Clearly, the more frequently the SPT computation needs to be performed, the higher the load on the router.

The object of this paper is two-fold and is motivated by the emerging developments discussed above. First, we propose a code profiling technique to measure various components of the protocol processing load offered on a router. We then apply this technique to a real OSPF implementation written by John Moy [?], [?]. We describe our experiments and explain our results which validate that flooding overhead and neighbor maintenance form a significant component of protocol processing load on routers with large degrees, and indeed that this load increases non-linearly with increasing degree. Second, we outline a novel OSPF implementation mechanism that relies on distributing certain OSPF functions across multiple processors typically available in current high end routers and in emerging next generation routers. Our mechanism does not need any changes to the non OSPF modules that may or may not interact with the OSPF module on the main processor on a router.

The rest of the paper is organized as follows. In the next section, we cover the preliminaries including an overview of the OSPF routing protocol and a discussion on the evolution of router architectures. Section 3 describes the objectives of our experiments and describes our code profiling methodology that is used in our experiments. Section 4 describes and provides a discussion of the results of the experiments. Section 5 proposes a new OSPF implementation mechanism and discusses its merits over traditional implementations. Section 6 concludes this paper.

## II. PRELIMINARIES

### A. *Evolution of Router Architectures*

Two major components of a typical router are its data plane functionality and its control plane functionality. The data plane functionality includes IP packet classification, enforcing QoS, IP lookup and forwarding, while the control plane functionality includes the routing protocols and the associated logic that

is used to populate the routing table to be used in the data plane. This is illustrated in Figure. 1(a).

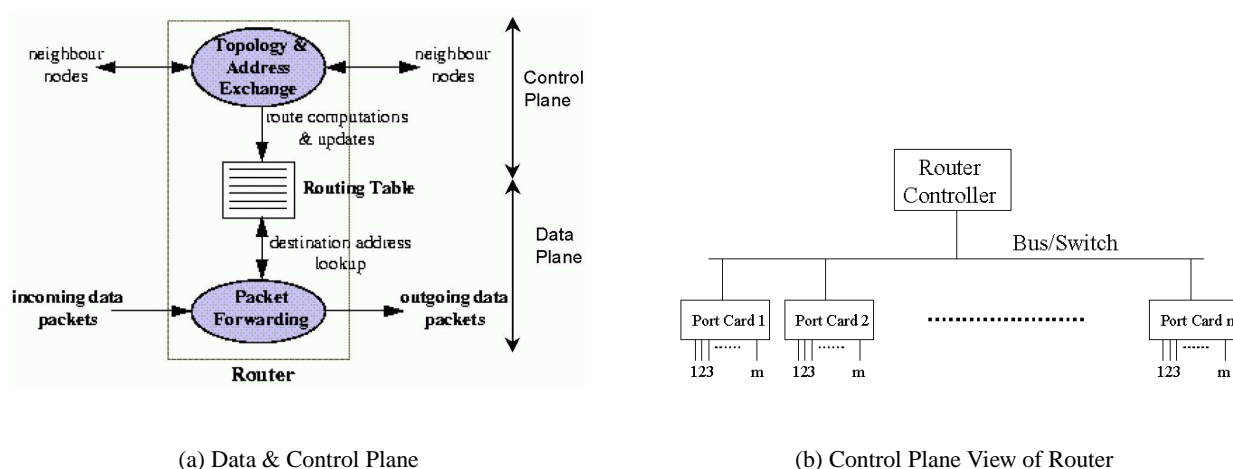


Fig. 1. Router Architecture

Router architectures have evolved significantly over the past few years and continue to evolve. Initial architectures had IP data forwarding and IP routing protocol processing all implemented in software supported on a single CPU. A lot of research has since been done in the data plane aspect of the routers. This coupled with advances in hardware technology has made hardware based wire speed IP forwarding engines implementing the data plane functionality feasible. Indeed, current high end router architectures typically have one or more such forwarding engines interconnected through a high capacity switch fabric and controlled by a central router controller. The control plane functionality is implemented in software executing on the central router controller. The routing table and other configuration information needed at each of the forwarding engines is communicated to them by the controller using an in-band mechanism using the switch fabric or using another mechanism connecting the router controller and the forwarding engines such as an ethernet bus or switch internal to the router chassis. Within the chassis, the forwarding engine and some router interfaces are usually co-located on port cards. This is illustrated in Figure. 1(b).

An important recent development, is the trend towards very high capacity switch fabrics enabling next generation routers with multiple terabits of backplane capacity and hundreds of interfaces. Already, it is not uncommon to find routers with upto tens of interfaces participating in link state routing within an OSPF area. Next generation routers could possibly have hundred or more interfaces within an OSPF area. It is conceivable that such routers can have multiple CPUs on their router controllers and also have dedicated CPUs on the port cards for performing localized functions such as programming and/or polling the forwarding engines. Notice that while the data plane functionality has evolved significantly from the original all software architecture, the control plane functionality remains concentrated on the router controller.

When IP control packets destined to the router controller arrive at one of the router's interfaces on one

of its port cards, they are either sent to the router controller through the switch fabric or otherwise over the internal bus/switch mechanism.

## B. OSPF Overview

OSPF RFC [?] is the authoritative source for information on OSPFv2. The following is a brief overview of the protocol. Each router in an OSPF network has a *link state database* (LSDB) which comprises of *link state advertisements* (LSA). At a given router, these LSAs are either self originated, or are obtained from its neighbors using the OSPF protocol. The following types of LSAs are defined: *Router LSA*, *Network LSA*, *External LSA*, *Summary LSA*, *ASBR-summary LSA*, and *Opaque LSA*. The router LSAs and the network LSAs together provide the topology of an OSPF area. Summary LSAs are generated by routers that have interfaces on the backbone area and also other areas. External LSAs are used to advertise reachability to IP prefixes that are not part of the OSPF domain. TE related information is carried in opaque LSAs. Each LSA has a standard header that contains advertising router id, LStype, LS id, age, sequence number and a checksum. The LS type, LS id and the Advertising router id together identify an LSA uniquely. For each LSA in the LSDB, the checksum is verified every CheckAge seconds (typically 300) - if this check fails, something has gone seriously wrong on the router. For two instances of an LSA, the age, sequence number and the checksum fields are used in determining the more recent instance.

In addition to the LSDB, each router in an OSPF network keeps the following information

- for each interface over which this router has an OSPF peering to one or more neighbors
  - an OSPF interface finite state machine which keeps track of the underlying interface state and the capacity in which OSPF is interacting with its neighbors on this interface - it could be DR (Designated Router), BDR (Backup DR), DROther or P2P. The notion of DR, BDR and DROther is only used in the case of broadcast capable interfaces.
  - a neighbor finite state machine for each neighbor that was discovered/configured on this interface - this state machine tracks the state of the communication between this router and the neighbor over this interface.

The objective of the OSPF protocol is to keep the LSDBs of all the routers in the network synchronized with each other so that all routers have a consistent snapshot of the network. Each LSA in the LSDB is aged with time. For self originated LSAs, each LSA is refreshed periodically. When the age of a self originated LSA reaches MaxAge, the LSA is first flushed out of the OSPF domain by flooding the maxage LSA and then re-originated with the initial age. For the LSAs originated by other routers, if the age reaches MaxAge, they are removed from the LSDB as soon as it is not involved in the process of initial database synchronization with any of its neighbors. If for any reason, a router wants to flush one of its self-originated LSAs from the

OSPF domain, it sets its age to MaxAge and floods it.

There are essentially five types of packets exchanged between OSPF neighbors: Hello packets, Database description packets, Link state request packets, Link state update packets, Link state acknowledgment packets. The exact use of these messages is included in the following subsections. Figures 2 and 3 give a high level view of some of the important OSPF control flows discussed below.

### B.1 Establishing and Maintaining Neighbor Relationships

When the OSPF protocol is enabled on an interface, it periodically multicasts hello packets on that interface. Hello packets are used to first discover one or more neighbors and where necessary carry all the information to help the DR election process. Among other things, hello packets also carry the identities of all other routers from which the sending router has received hello packets from. When a router receives a hello packet that contains its own identity, it concludes that bidirectional communication has been established between itself and the sender of the hello packet. When bidirectional connectivity is established, the router decides the capacity in which it must be an OSPF speaker on this interface.

Once the capacity is established, the two neighboring routers must decide if they indeed should exchange their LSDBs and keep them synchronized. If database exchange needs to be performed, a neighbor relationship (adjacency) is established. For example, if a router is speaking OSPF in the capacity of DROther over an interface, it would decide not to establish an adjacency with another router that is participating as DROther on that interface - DROther speakers only establish adjacencies with DR and BDR speakers.

Once the neighbor relationships (adjacencies) are established and the DR election is done, hello packets are used as keep-alives for the adjacency and are also used in monitoring any changes that can potentially result in changes in the DR status.

### B.2 Initial LSDB Synchronization

If the decision is to establish an adjacency, the router is agreeing to keep its LSDB synchronized with its neighbor's LSDB over this interface at all times. The router enters a master/slave relationship with its neighbor before any data exchange can start. If the neighbor has a higher id, then this router becomes the slave. Otherwise, it becomes the master.

LSDB synchronization between two routers is achieved in two parts: Firstly, each router records the summaries of all LSAs in its LSDB, except the ones with maxage, at the time at which the master/slave relationship was negotiated. Each router then sends this summarized information, contained in data description packets, to the other router. When a router receives this summary information from its neighbor, it compares the summary with the contents of its own LSDB and identifies those LSAs for which the neighbor has a more recent instance. It then explicitly requests these more recent LSAs by sending link state requests packets to

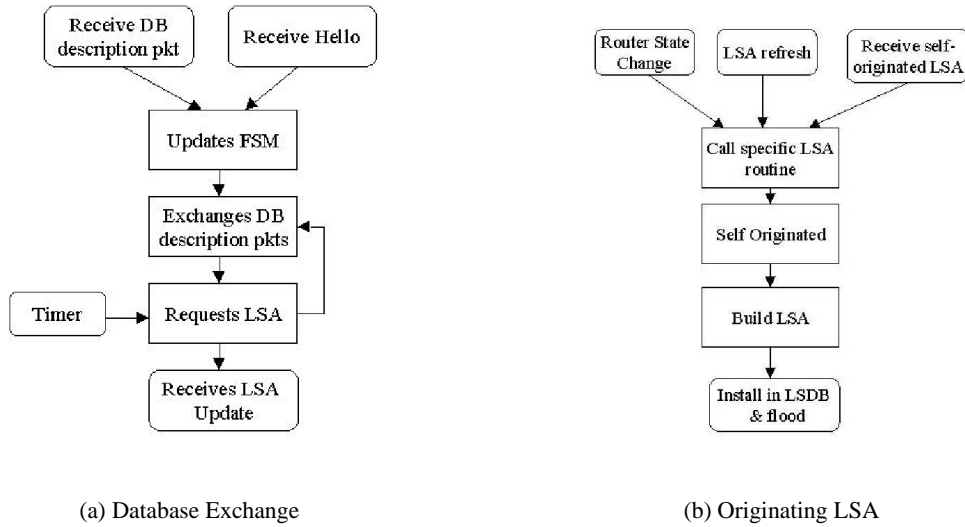


Fig. 2.

its neighbor. When each of these routers receives a link state request packet from its neighbor, it responds by sending the requested LSAs in link state update packets to the neighbor. Secondly, once the routers exchange their summary information, each router includes the other in its flooding procedure.

Note that, in the above, once the summarization of the LSAs is done, sending data description packets to the neighbor, responding to link state requests from the neighbor and also including the neighbor in the flooding procedure can all happen concurrently. The adjacency establishment is complete when all LSAs requested by each of the neighbors are received. A router includes a link to its new neighbor in its router LSA after it receives all the LSAs that it requested from that neighbor.

### B.3 Reliable Flooding Procedure

The flooding procedure at a router  $R$  is invoked in two scenarios - first is when it intends to originate/refresh an LSA and second when it receives a new LSA or an update to an existing LSA from its neighbor. The flooding procedure is described below.

When an updated non self-originated LSA  $L$  is received from a neighbor  $N$ , one of following scenarios can happen

- No previous instance of  $L$  exists in the LSDB;  $L$  is a new LSA. If the age of  $L$  is maxage and if there are no neighbors in the dbexchange process, then send an ack to  $N$  and discard  $L$ . Otherwise, timestamp  $L$ , ack it and install in the LSDB.
- An older version of  $L$  exists in the LSDB. If the older version was received less than  $\text{minLSarrival}$  time ago, discard  $L$ . Otherwise, timestamp  $L$ , ack it and install in the LSDB. If there are any neighbors from whom this router is expecting acks for the older version of  $L$ , stop expecting such acks.

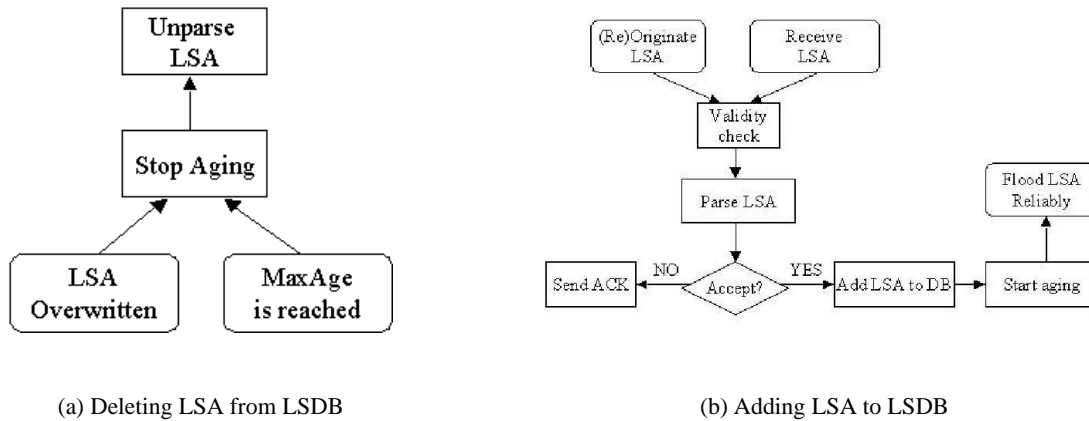


Fig. 3.

- A newer version of  $L$  exists in the LSDB. Three cases are of interest:
  - If  $N$  and  $R$  are still in the database exchange process, and if  $N$  had previously sent a database description packet suggesting that it had a newer version than the one in the LSDB, then this is an error. The database exchange process with  $N$  has to start all over again.
  - If the age of the newer version is maxage and its sequence number is maxseqno, then discard  $L$ . The intent here is to let the seqno wrap around.
  - If the newer version was received more than minLSinterval time ago, then send the newer version of  $L$  to  $N$ . Do not expect an ack from  $N$  and do not send an ack for  $L$ .
- The version in the LSDB is the same as  $L$ . In this case, check if this is an implicit ack. If it is an implicit ack, then no need to ack it unless  $N$  is the DR. If not treated as an implicit ack, send an ack to  $N$ .

If  $L$  was installed in the LSDB above, then it needs to be sent to all neighbors except  $N$  and other DROther/BDR speakers for which  $N$  is the DR (note that if  $R$  was part of more than one area, then the scope of flooding for  $L$  would depend on the LSA type of  $L$ ). In order to ensure reliable delivery of  $L$  to its neighbors,  $L$  is retransmitted periodically to each neighbor  $M$  until an ack is received from  $M$ . An ack could be implicit. In the last scenario above,  $L$  could be treated as an implicit ack from  $N$  if this router was waiting for an ack from  $N$  for the version in the LSDB.  $R$  uses this same approach when originating or refreshing LSAs that it itself generates.

When sending  $L$  to a neighbor  $M$  with which  $R$  is still in the database synchronization phase,  $L$  must be compared with the instance of  $L$  that was described in the database description packets sent by  $M$ . Two cases are of interest

- $L$  is an older version. In this case, no need to send  $L$  to  $M$ .
- $L$  is same or more recent. In this case, it is no longer necessary to request the version of  $L$  from  $M$ . So, stop asking for  $L$  when sending link state request packets to  $M$ . If  $M$  and  $N$  are on the same broadcast

capable interface and if  $N$  is the DR, then it is not necessary to send  $L$  to  $M$ . In all other cases send  $L$  to  $N$ .

Note that the above procedure sometimes causes acks to be received even when they are not expected. Such acks are simply discarded.

A maxage LSA  $L$  is removed from the LSDB when no acks are expected for  $L$  from any neighbor and  $R$  is not in database synchronization phase with any of its neighbors.

In some cases,  $R$  can receive a self-originated LSA  $L$  from one of its neighbors  $N$ . If  $L$  is more recent than the one in the LSDB ( $L$  must have been originated by a previous incarnation of  $R$ ), then  $R$  must either flush  $L$  by setting its age to maxage and flooding it, or it must originate a newer instance of  $L$  with its sequence number being one more than that of  $L$ .

### III. MEASUREMENT OBJECTIVES AND METHODOLOGY

In this section, we define the measurement related terminology we use in the rest of this paper and describe our objectives and our methodology. We will refer to the number of OSPF neighbors (in an area) of a router as the degree of the router. We view the OSPF load offered on a router as consisting of two parts: *core* load and *neighbor overhead* load. For a given network, core load is defined as the OSPF load offered on a router with degree one. This load includes the SPT computation load and also LSA processing load assuming exactly one OSPF neighbor. The neighbor overhead load is defined as the load that is offered in addition to the core load on routers with degree more than one. Note that for routers with degree one, neighbor overhead load is zero.

Our primary objective is to understand the dynamics of neighbor overhead load for routers with large degrees. Based on previous work relating to SPT computation loads, we take for granted that the problem of optimizing core load is addressed by works such as smart (pre-)computation mechanisms and also by incremental techniques. In their paper, Apostolopoulos et. al. [?], indicated that the operation of LSA receiving/sending takes significant time relative to the time taken by other components of their protocol implementation. We measure the ratio of the total OSPF load and the corresponding core load for routers with representative degrees. This ratio is referred to as the *reducible overhead factor* signifying that it is possible to reduce the load by this factor while still being able to maintain consistent link state routing functionality. We consider such measurements in both stable and unstable networks with possibly multiple link flaps.

We performed the above mentioned measurements on a Linux machine using a real OSPF implementation that is available under the gnu license and is written by John Moy [?]. An OSPF network simulator that allows one to create arbitrary network topologies is part of this package. Each router in the simulated network executes as a separate linux process and a separate main process coordinates the communication and

synchronization requirements needs of the routers in the networks. We instrumented the protocol code in [?] to measure the time spent on various critical OSPF control flows described briefly in section II. We did this in such a way that we could identify the CPU time used by each router corresponding to the core load and also the neighbor overhead load. The division of work between core load and neighbor overhead load was identified at the functional flow level and those functions were instrumented with profiling information. To do such a fine degree profiling of functions, an extremely small granularity timer is essential. This is lacking in the current non-RT systems. The systems can only give timing in the order of 1 jiffy, which is in the order of 10 milliseconds. Therefore, we used Pentium inline assembly code [?] for accessing the register values and getting the exact timing of nanosecond granularity. The CPU after every cycle updates the *Time Stamp Counter*(TSC) by 1. So, the number of cycles elapsed can be measured between any two events. The number of cycles when multiplied by the cycle length (inverse of clock frequency) will give the time taken. As the experiments were done in a single processor 1 Ghz machine, the cycle length was 1 nanosecond. Hence the granularity by which the timing could be measured is nanosecond. This timer when put at the beginning and end of the function-to-be-profiled to give the exact timing information, as was necessary. The assembly code that we used is given below.

```

pushl %edx
pushl %eax
RDTSC ~V Reads Time Stamp counter
movl %%edx,%0 : ~S=g~T (high)
movl %%eax,%0 : ~S=g~T (low)
popl %edx
popl %eax

```

The measurements involved each router dumping out the CPU time used for core load and neighbor overhead load every ten simulated seconds. At the end of each simulated run of a sample OSPF network, we computed the reducible overload factor using this series of ten second data bins. We found that these ten second bins were just right for us to sample the load on the routers. We wanted the bin size to be small enough for us to capture peak CPU activity and at the same time large enough to allow completion reasonable processing of OSPF packets.

To verify the fact that we did not leave out any major function of a OSPF control flow, for each router, we compared our measured time with actual CPU time the whole router process was taking. There is an overhead on each process in the simulator for communicating with the main process for synchronizing and exchanging simulator control data. This communication overhead was separately profiled. The total CPU time taken by the process can be seen from the `/proc/[PID]/stat` file for that process. Our profiled code (both the communication overhead as well as the actual OSPF control flow) was very close to the total CPU time. Small variations were due to the fact that the `stat` file was updated not continually but after every interval

of time. As can be seen from Table III, the required OSPF code was wholly profiled, except for the small difference of 1.6% attributed to profiling error margin.

TABLE I  
AVERAGE PERCENTAGE OF PROFILED CODE

OSPF control flow	OSPF simulator overhead	Sum
82.1	16.3	98.4

We discuss our experiments based on the above methodology and the results we obtained in the next section.

#### IV. EXPERIMENTAL RESULTS

In this section, we simulate the following two network topologies using the instrumented simulator described in the previous section.

##### A. Topologies

These network topologies have both large & small degree as well as different network sizes.

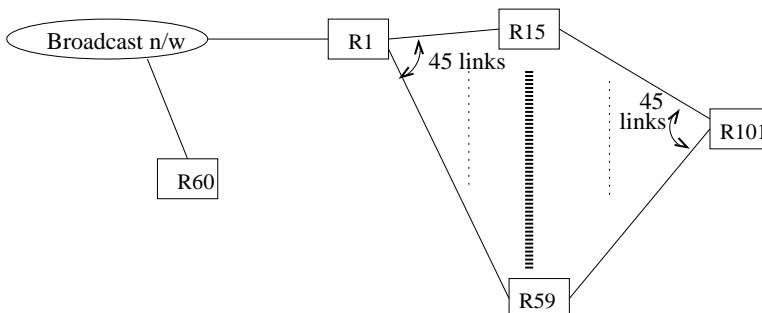


Fig. 4. Network 1

1. This network was designed to show the increase in total load with increase of degree of the nodes and size of the network.

Figure 4 is a 48-node network. There are 45 routers in a row from  $R_{15}$  to  $R_{59}$  - they simulate any network cloud, and the interconnection among themselves does not matter for the routers outside this cloud. The cloud should have the minimal property of being reachable through every link of routers  $R_1$  and  $R_{101}$ . Both  $R_1$  and  $R_{101}$  are connected with all the 45 nodes from  $R_{15}$  to  $R_{59}$ .

It is split into 2 areas.  $R_1$  and  $R_{60}$  form a broadcast area network. Other area is made up of 90 point-to-point links between routers. Division of the network between two areas is done so that summary LSAs

of one area will be flooded into the other area. The 2 areas add summarization overhead on the  $R_1$  which is a part of both areas, they increasing it's load.

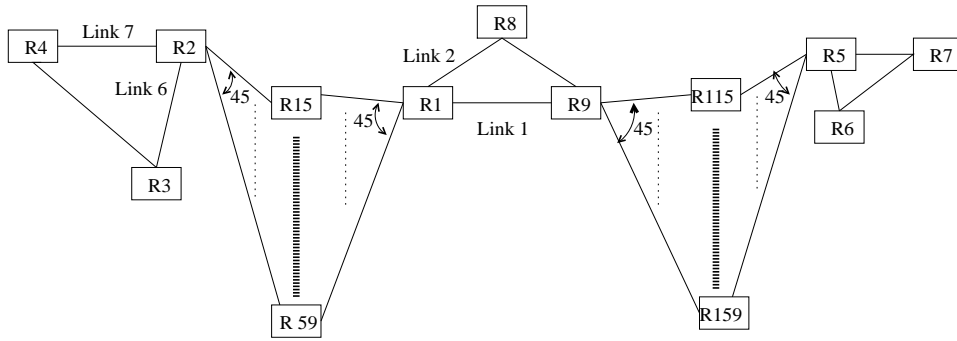


Fig. 5. Network 2

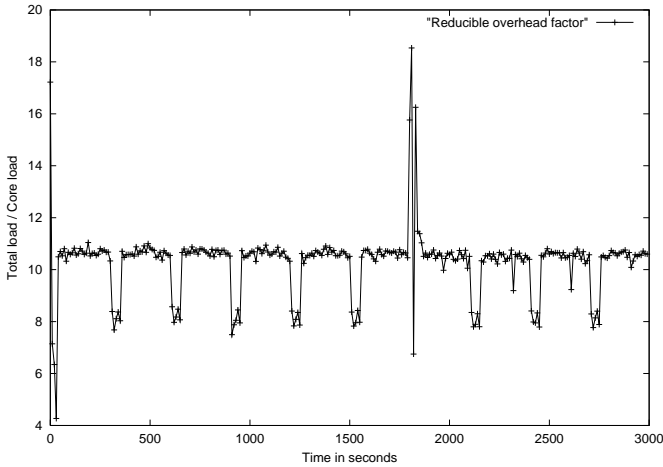
2. This larger network was designed to show the effect the effect of increased size as well as flapping of links/routers.

Figure 5 is a 99-node network. There are 45 routers in each of the two rows shown by dotted lines - between  $R_{15}$  and  $R_5$  & between  $R_{115}$  and  $R_{159}$ . All the routers are in one area. 4 routers -  $R_1$ ,  $R_2$ ,  $R_5$  and  $R_9$  have a high degree of 47. Both  $R_1$  and  $R_2$  are connected with all the 45 nodes from  $R_{15}$  to  $R_{59}$ , while  $R_5$  and  $R_9$  are connected with all the 45 nodes from  $R_{115}$  to  $R_{159}$ . Degrees of other routers are 2. In this network the topology is that of two equal parts on both sides. The three routers in the middle is a substitute for a network cloud, connecting both sides. The route costs are specified in the figure so as to route traffic through specific routes. In the 3 routers in the middle, the top 2 links have high cost compared to the horizontal link1 between  $R_1$  &  $R_9$ . Hence all the traffic between the two dumbbells are routed through the low cost link. Breaking of that low cost link forces all traffic between the two parts to flow along the alternate high cost route. So any change in those critical links might change the routes for a whole lot of the nodes, creating a lot of OSPF control overhead. Whereas, breaking of other links will have less effect. This will translate to different amounts of LSA updates.

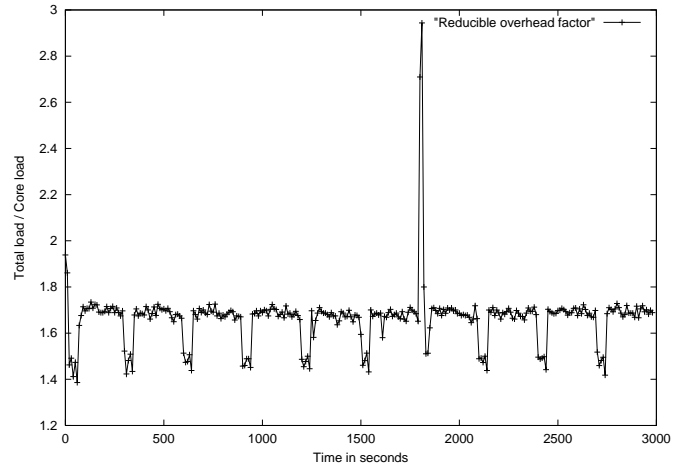
## B. Results and analysis

In the experiments, the routers came up in a sequence. The network condition did not vary for the length of the experiment. The load on the router increases with increase of both the degree of each node as well as size of the network.

In experiment 1, we take network 1 and simulate is for 3000 secs and 34000 secs. In Figure 6(a), we see the *reducible overhead factor* to be 11 for router  $R_1$  of degree 46. This shows a big difference difference in the total load compared to the core load. The core load does not have the big overhead in terms of flooding



(a) Node  $R_1$ : Degree 46



(b) Node  $R_{59}$ : Degree 2

Fig. 6. Network 1

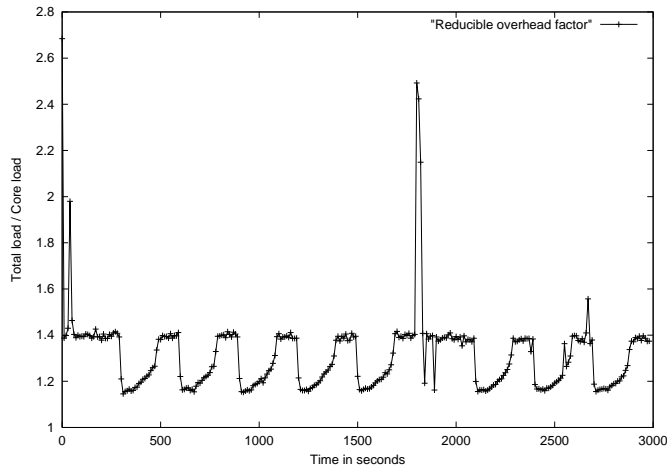
& receiving of control packets from multiple neighbors.

At regular intervals of 5 minutes or 300 seconds, we see a decrease in the *reducible overhead factor*. According to OSPF specification, the LSAs in the database are checksum verified every 5 min interval. This verification adds up to the core load, significantly reducing the scale of improvement. As the verification is done within a range, it appears distributed over time - hence the glitch below stays for tens of seconds. Router  $R_{60}$  in Figure 7(a) is much more affected by this aging, as its LSA database is much larger because of the summary LSAs of the other area in it's LSDB. Compared to it, the router  $R_1$  recovers much faster because of the smaller size of its LSDB. This checksum calculations for detecting corruption in memory can be seen to have a significant overhead on the controller. With the reliability of memory systems today, it is worth looking into whether there is a need for this very frequent checksums.

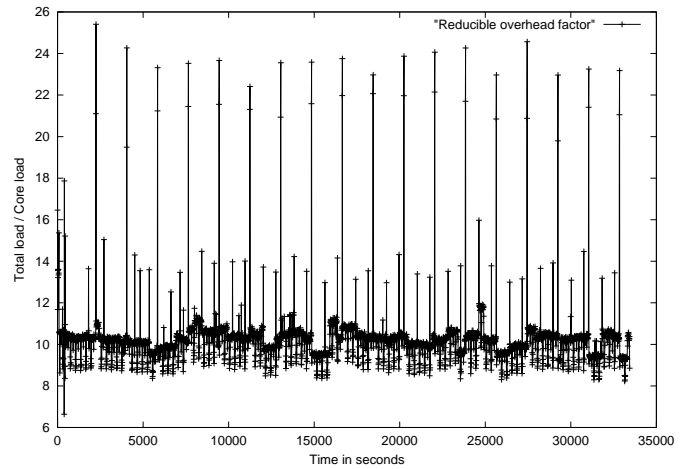
At regular intervals of 30 minutes or 1800 seconds, we see a sharp increase in the *reducible overhead factor*. According to OSPF specification, the LSAs are reoriginated and the old LSAs flushed. This produces the sharp spike of improvement, as lots of flooding of LSAs occur. This gives a big increase in terms of the reduction of load on the controller during this time, every 30 minutes. This scenario models any time when there is a large number of LSAs being flooded. This models the case of a large number of TE LSAs in the network. Hence we show that large number of TE LSAs can be accommodated in this way scalably.

For a much longer run of almost 10 hours or 34000 seconds, scenario is pretty much the same. Figure 7(b) correspond to the network of scenario 1 for router  $R_1$ , for a run of 34000 seconds. The reducible factor is shown to be consistently 11, matching that of Figure. 6(a).

Router  $R_{60}$  in Figure 7(a) is much more affected by this aging, as its LSA database is much larger

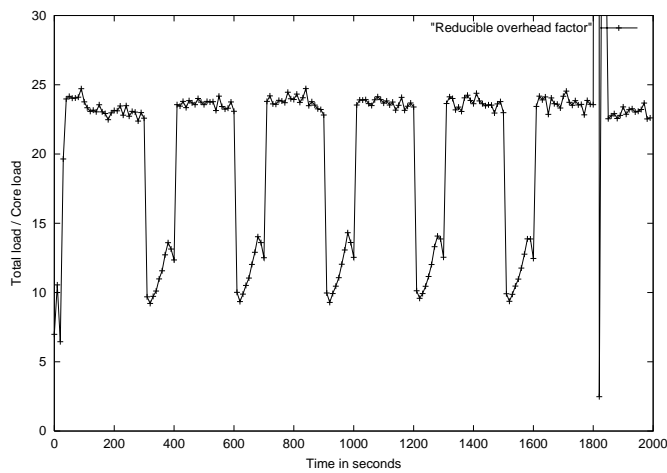


(a) Node  $R_{60}$ : Degree 1

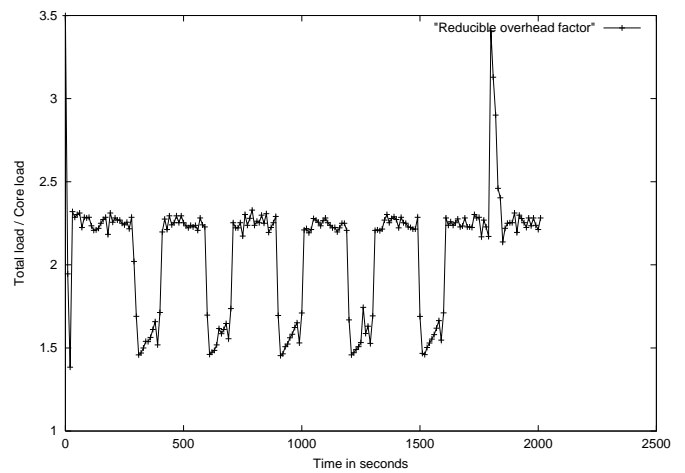


(b) Node  $R_1$ : Degree 46

Fig. 7. Network 1



(a) Node  $R_1$ : Degree 47



(b) Node  $R_{59}$ : Degree 2

Fig. 8. Network 2

because of the summary LSAs of the other area in its LSDB. Compared to it, the router  $R_1$  recovers much faster because of the smaller size of its LSDB. With the reliability of memory systems today, it is worth looking into whether there is a need for this very frequent checksums.

We now repeat the same experiment with a larger network. Figure 8(a) and Figure 8(b) correspond to the Network 2 (Figure 5) for a run of 2000 seconds.

The figures for Network 2 show similar characteristics. As the second network is double the size of the first one with 99 nodes, the amount of improvement is also larger. More the number of LSAs in the network,

more is the factor of improvement - this logic is proven by the above graphs. Hence, in term of scalability over large area networks, this scheme suggests an even better performance.

The amount of improvement is 23 for Figure 8(a) and is 2.3 for Figure 8(b).

Figure 9 corresponds to the amount of improvement with increasing degree of nodes.

With increasing degree of a node in the network, the amount of improvement increases more quickly than linearly. The two lines in Figure 9 correspond to the 48 node network and the 99 node network. This non-linear increase is due to the removal of receiving and flooding overhead from the core load.

Also, larger networks have more improvement than smaller networks. Larger networks generate more redundant data than smaller networks because of the inherent property of OSPF, in which all received control packets are broadcasted out though all the ports. The basic objective of scalability is thus achieved.

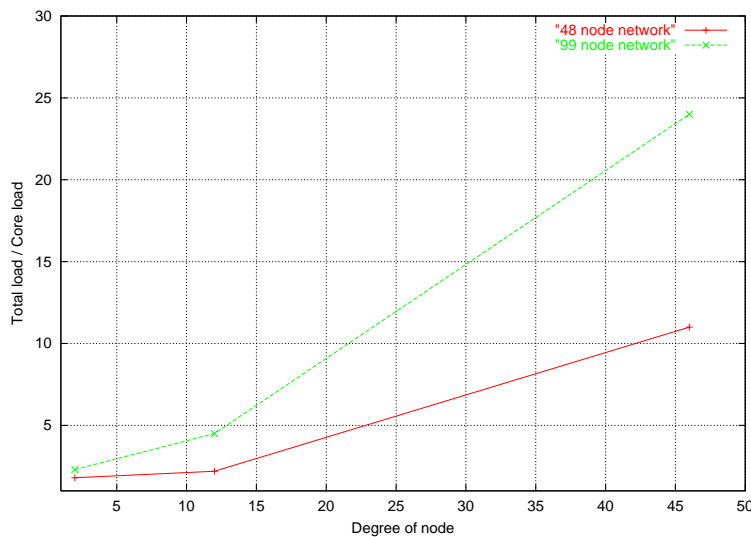


Fig. 9. Increasing gain with increasing size of network

We did another set of experiments to show the effect of unstable network on the *reducible overhead factor*. The results presented here are done on Network 2 (Figure 5) for 700 seconds. Unstable networks are those where a link or a router may go down and then come back up at any time. Also a link may flap, which implies that it may continuously go up and down leading to repeated SPF calculation.

Figure. 10 depicts the experimental case. It shows the time when the specified link or router goes down or comes up. Link  $L_1$  is more important compared to all other links, as all traffic from left half to right half passes through it because of its lower cost. Change in the status of that link will trigger a change in all the routes which use that link. Enough time is given for the system to stabilize before a new link or router flap is done. Whenever the improvement came back to the stable scenario, after the effect of any link or router

failure or booting, some other link or router is again flapped. So, the effect of each failure is carefully isolated.

1. Controller load reduction with the change in the link & router status.

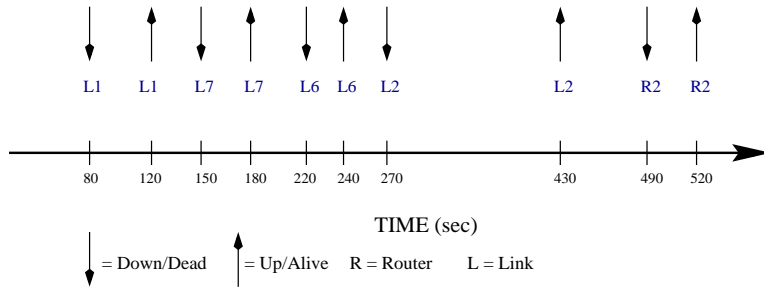
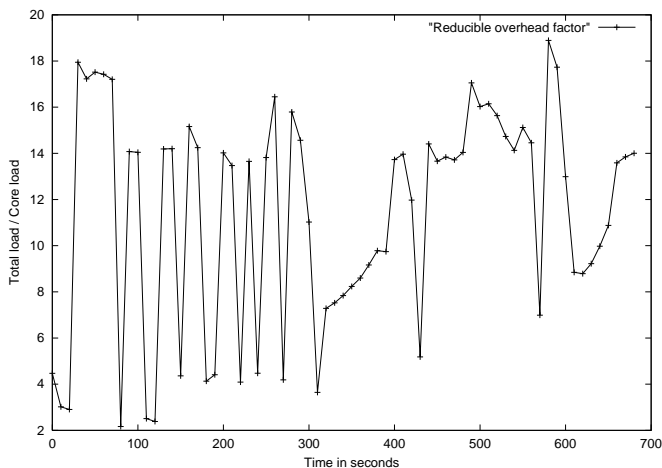


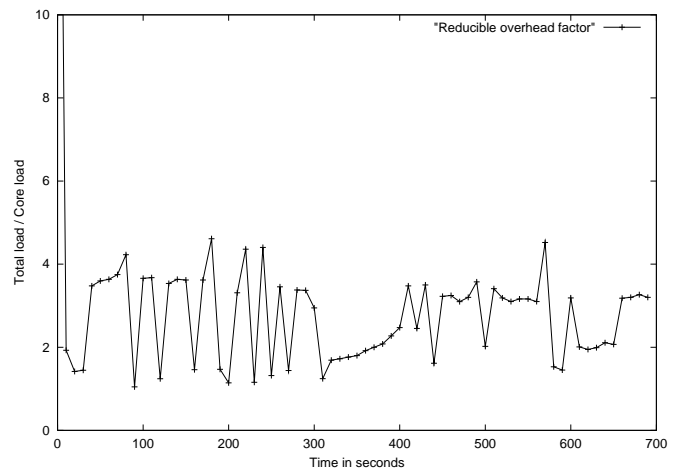
Fig. 10. Link/Router flaps in the simulation on Network 2

The amount of improvement is shown for two representative routers Router 9 & Router 115.

In Figure 11(a) for router 9, the average improvement for static case is 14 times. In Figure 11(b) for router 115, the amount of improvement is 3.5 times for static case. The improvement in the face of route flapping goes down to about 4 times for router  $R_9$  and to about 1.5 times for router  $R_{115}$ . The change is due to the large number of LSA update packets that are generated at each link failure or link up. Many of the LSA updates are actual updates which need to be made on the LSDB and hence need to be handled by the controller. So, the controller has to process all of these messages for addition and deletion from the LSDB. This significantly pushes up the controller load and thus we see the reduction in gain. But, when Link  $L_1$  flaps at 80 and 120 seconds, the controller load is further reduced to about 2 for router  $R_9$ . This shows the importance of link  $L_1$  compared to other links, as more routes change because of its status change. This can be more clearly seen in the next section, which actually shows the amount of CPU time

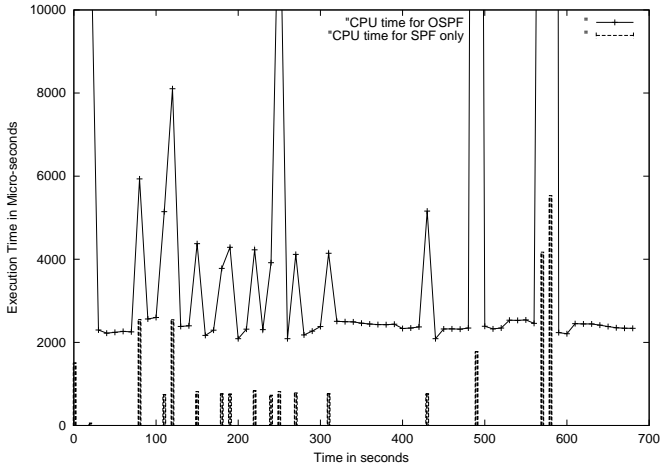


(a) Node  $R_9$ : Degree 47

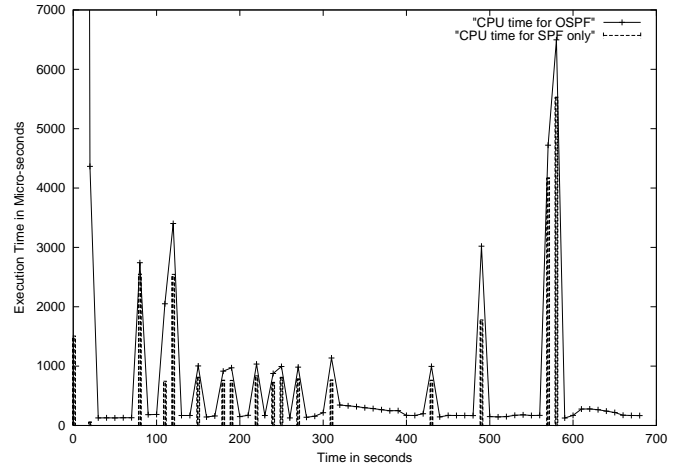


(b) Node  $R_{115}$ : Degree 2

Fig. 11. Network 2: Unstable



(a) Centralized OSPF and SPF time



(b) Distributed OSPF and SPF time

Fig. 12. Network 2: Unstable

necessary when these route flaps happen.

2. The actual CPU time taken by the OSPF control overhead and also the CPU time for the SPF computation only in both the implementations.

Figure.12(a) shows the SPF computation Vs. Time for total OSPF load. We can see that it comes to about 1/3 of the total OSPF load on the average. So, the other load on controller is significant, consisting of flooding and receiving LSAs mostly importantly. At the points where the route flaps occurred, the increase in the time of core load can be clearly seen. Link  $L_1$  flapping at 80 and 120 seconds creates a lot of LSA generation thereby using almost 3 times the normal OSPF load, compared to other link flaps which produce 2 times the normal OSPF load. But around 500 seconds, the flapping of router  $R_2$  produces a huge OSPF load. It goes way beyond the scale of the graph as depicted here. The number of times SPF computation is done as well as the time for each SPF computation also increases significantly.

Figure.12(b) shows SPF computation Vs. Time for core OSPF load. In this case, most of the core load comprises of SPF calculation. The SPF calculation is very difficult to distribute, and hence is part of the core load.

With the advocating of sub-second SPF calculations for milli-second convergence, this fact is worth noting. The core load thus wholly comprises of SPF computation, with other OSPF operations seen to be negligible.

## V. A NOVEL OSPF IMPLEMENTATION TECHNIQUE

In this section, we outline an OSPF implementation strategy for high end routers which have two or more CPUs. As described earlier, evolving routers have a router controller that executes the control plane

software including OSPF and possibly other routing protocols such as BGP. Multiple protocols typically interact with each other for enforcing route redistribution policies and also typically involve a common routing table management mechanism. For this reason, any changes to the OSPF protocol need to be made without impacting the interaction of OSPF with the rest of the control plane software. Fortunately, any interaction with other parts of the system only involves the part of OSPF that contributes towards its core load.

We therefore propose to keep the core load of OSPF along with other control plane software and distribute the functionality of OSPF that contributes to the neighbor overhead load across one or more CPUs typically available in a high end router. We will refer to the CPU that has the OSPF core load along with other software as the *core CPU*. All other CPUs available on the router will be referred to as *secondary CPUs*. We further assume that each of these CPUs has an independent operating system instance executing on it and that all CPUs are connected as a loosely coupled distributed system with a messaging based communication available between them. We will refer to the secondary CPUs that are involved in OSPF related processing as the *OSPF supplemental CPUs*. Also, our reference to a CPU is really meant to imply software executing on that CPU.

Our distributed mechanism works as follows: First, for each neighbor, an OSPF supplemental CPU is assigned such that that CPU takes care of all the neighbor management processing such as initial hello exchanges, keep-alive hello exchanges and also DR election where appropriate. The supplemental CPU is required to keep the core CPU informed of neighbor state.

Second, for each LSA in the LSDB, the core CPU picks one of the supplemental CPUs as a *delegate CPU*. The delegation of LSA processing to the delegate CPUs is done based on a load balancing heuristic. The assignment of LSAs to port cards is done when either Maximum Age for that LSA is reached or when self-originated LSAs are refreshed, or whenever a new LSA comes into existence.

An LSA that is received by the router (from any of its interfaces) is always directed to that LSA's delegate CPU. The delegate CPU performs all the necessary validation and forwards this LSA to the core CPU only if it is required to be installed in the core CPUs LSDB. Specifically, when a delegate CPU receives an LSA, if Maximum Age is reached for that LSA, it ceases to be the delegate and forwards it to core CPU. It then decides if the LSA needs to be sent to the core CPU. If not, it simply performs the OSPF flooding acknowledgement procedures. Also, for self-originated LSAs, if the LSA needs to be refreshed, a delegate CPU ceases to be the delegate and informs the core CPU about the need for such a refresh. The core CPU can then perform a new delegate assignment. In the context of receiving LSAs, it must be noted that there is a need to keep the age of an LSA consistent at the core CPU and its delegate. We perform aging of an LSA both at the core and its delegate CPUs. This is important because the acceptance of an incoming LSA can be based on the comparison of its age with that of the instance in the LSDB. If this age difference is such that the

maximum drift in the LSDB age will affect the acceptance or rejection of the LSA, then that LSA is sent to the core CPU for a decision. The core CPU which keeps the exact age can decide whether to accept or reject that LSA. Else, if this drift does not affect the age difference, the delegate can itself take a decision whether to accept or reject the LSA. This will significantly reduce the processing overhead in the core CPU.

When the core CPU receives an LSA and installs it in its LSDB, it may need to perform reliable flooding procedures to send that LSA to its neighbors. The core CPU then sends a copy of the LSA to each of the supplemental CPUs that is performing neighbor management for at least one of the neighbors to which the LSA needs to be sent. Note that if the CPUs are connected on a shared bus, only one copy of the LSA needs to be broadcast from the core CPU. In any case, the number of copies that the core CPU has to make will be sufficiently less than the number of neighbors to which the LSA needs to be sent. Each supplemental CPU then sends a copy of that LSA to each appropriate neighbor under its control and also takes care of timer based retransmissions where necessary. When this is done successfully, each supplemental CPU sends a message to the core CPU indicating the status.

It must be noted that the core CPU is the only place where the complete LSDB is available. Since this complete LSDB needs to be accessible during initial database exchange procedures, such database exchange procedures are always performed at the core CPU. A supplemental CPU which manages a neighbor has the responsibility of indicating the need for database exchange to the core CPU.

Clearly, the above description of the distribution mechanism is an outline of the implementation. A detailed description is provided in a full paper [?]. The above implementation brings the OSPF load on the core CPU close to the core load offered by OSPF. In other words, our procedure can reduce the load on the core CPU by as much as the reducible overhead factor discussed previously. Any further improvements can only be done by LSDB partitioning and distributed SPT computation. Even if it is considered that such a partitioning is feasible, such a distribution is not easily achieved without impacting the rest of the control plane software on the core CPU.

## VI. CONCLUSION

In this paper, we discussed the emerging trends in the areas of Internet traffic engineering and evolution of router architectures. We motivated the need for highly scalable OSPF protocol implementations in making large IP/MPLS networks with traffic engineering feasible. We identified a chunk of the OSPF load offered on routers with multiple OSPF neighbors that does not contribute to consistency of the LSDB itself but is a necessary component in achieving reliable flooding. We showed through simulation that such load is indeed significant relative to the overall OSPF load and outlined an implementation technique that takes advantage of multiple processors on emerging high end routers in processing such load in a distributed manner. Although

the procedures described in this paper are focussed on OSPF, they can be easily adapted to other link state routing protocols such as IS-IS.