Chapter 10: Maintaining State with Slower Memories

May 2008, Santa Rosa, CA

Contents

List of Dependencies

- **Background:** The memory access time problem for routers is described in Chapter 1. Section 1.5.2 describes the use of load balancing techniques to alleviate memory access time problems for routers.

Additional Readings

- **Related Chapters:** The general load balancing technique called constraint sets, used for analysis in this chapter, was first described in Section 2.3. Constraint sets are also used to analyze the memory requirements of other router architectures in Chapters 2, 3, 4 and 6.

Table: *List of Symbols.*

C	Number of Updates per Time Slot
E	Number of State Entries
h	Number of Banks
M	Total Memory Bandwidth
R	Line Rate
S	Speedup of Memory
T	Time slot
T_{RC}	Random Cycle Time of Memory

Table: *List of Abbreviations.*

ISP	Internet Service Provider
I/O	Input-Output (Interconnect)
GPP	Generalized Ping-Pong
SMA	State Management Algorithm
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory

"To ping, pong, or ping-pong?"

— The Art of Protocol Nomenclature[†]

10

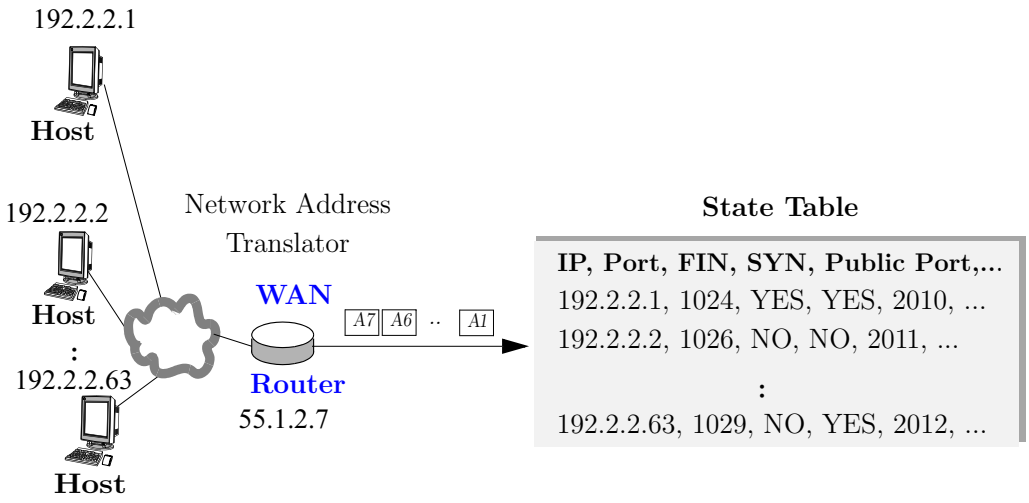
Maintaining State with Slower Memories

10.1 Introduction

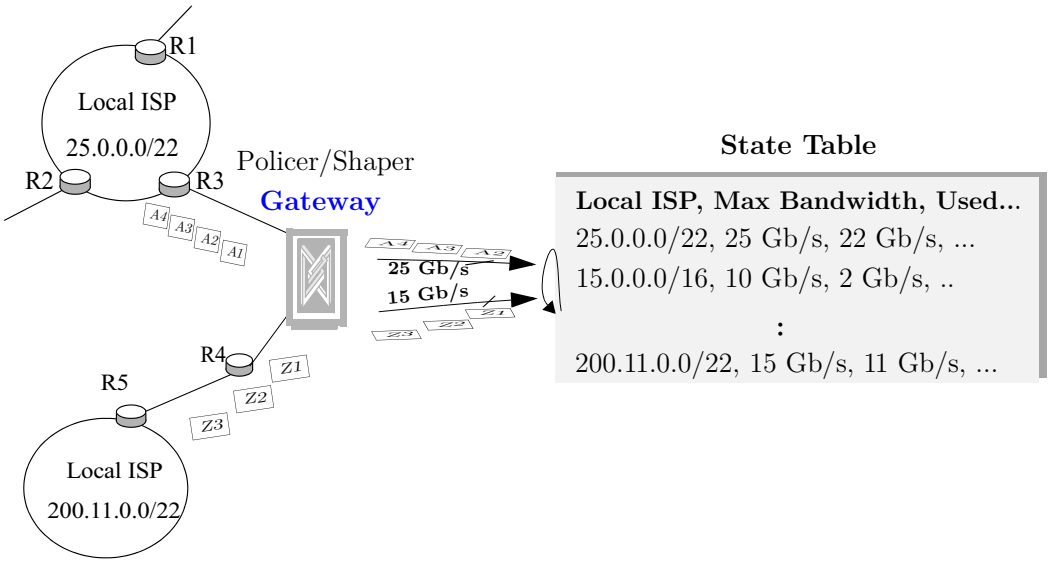
Many routers maintain information about the state of the connections that they encounter in the network. For example, applications such as Network Policing, Network Address Translation [208], Stateful Firewalling [194], TCP Intercept, Network Based Application Recognition [209], Server Load balancing, URL Switching, *etc.*, maintain state. Indeed, the very nature of the application may mandate that the router keep state.

The general problem of state maintenance can be characterized as follows: When a packet arrives, it is first classified to identify the connection or *flow* (we will describe this term shortly) to which it belongs. If the router encounters a new flow, it creates an entry for that flow in a flow table and keeps a state entry in memory corresponding to that flow. If packets match an existing flow, the current state of the flow is retrieved from memory. Depending on the current state, various actions may be performed on the packet — for example, the packet may be accepted or dropped, it may receive expedited service, its header may be re-written, the packet payload may be encrypted, or it may be forwarded to a special port or a special-purpose processor for further

[†] *Ping* is an application to troubleshoot networks. *Pong* is an internal Cisco proposal for synchronizing router clocks (currently subsumed by IEEE 1588). *Ping-pong* is a technique that is generalized in this chapter to maintain state entries on slower memories.



(a) State Maintainence for Network Address Translation



(b) State Maintainence for Policing and Bandwidth Shaping

Figure 10.1: *Maintaining state in Internet routers.*

processing, *etc.* Once the action is performed, the state entry is modified and written back to memory.

✎**Example 10.1.** Figure 10.1 shows two examples of routers maintaining state — (a) A network address translator [208] keeps state for connections, so that it can convert private IP addresses to its own public IP address when sending packets to the Internet, (b) A gateway router belonging to a backbone Internet service provider (ISP) maintains state for packets arriving and departing from local ISPs, so that it can police data and provide the appropriate bandwidth based on customer agreements and policy.

We are not concerned with how the various flows are classified and identified, or the actions that are performed on the packet once the state entry is retrieved. The task of flow classification is algorithmic in nature, and a number of techniques have been proposed to solve this problem [134, 184]. Similarly, the actions that are performed on the packet (after the state entry is retrieved) are compute-intensive and are not a focus of this chapter. We are only concerned with how the actual state entry is updated because the operation is memory-intensive.

☞**Observation 10.1.** A *flow* is defined by the specific application that maintains state. The granularity of a flow can be a single connection or a set of aggregated connections. For example, a network address translator, a stateful firewall, and an application proxy maintain flow state entries for each individual TCP/UDP connection they encounter. A policer, on the other hand, may keep state entries at a coarser level — for example, it may only maintain state entries for all packets that match a certain set of pre-defined policies or an access control list.

In summary, a state entry needs to be read, modified, and written back to memory in order to update its state. This operation is usually referred to as a "read-modify-write" operation. The set of tasks required to perform state maintenance is shown

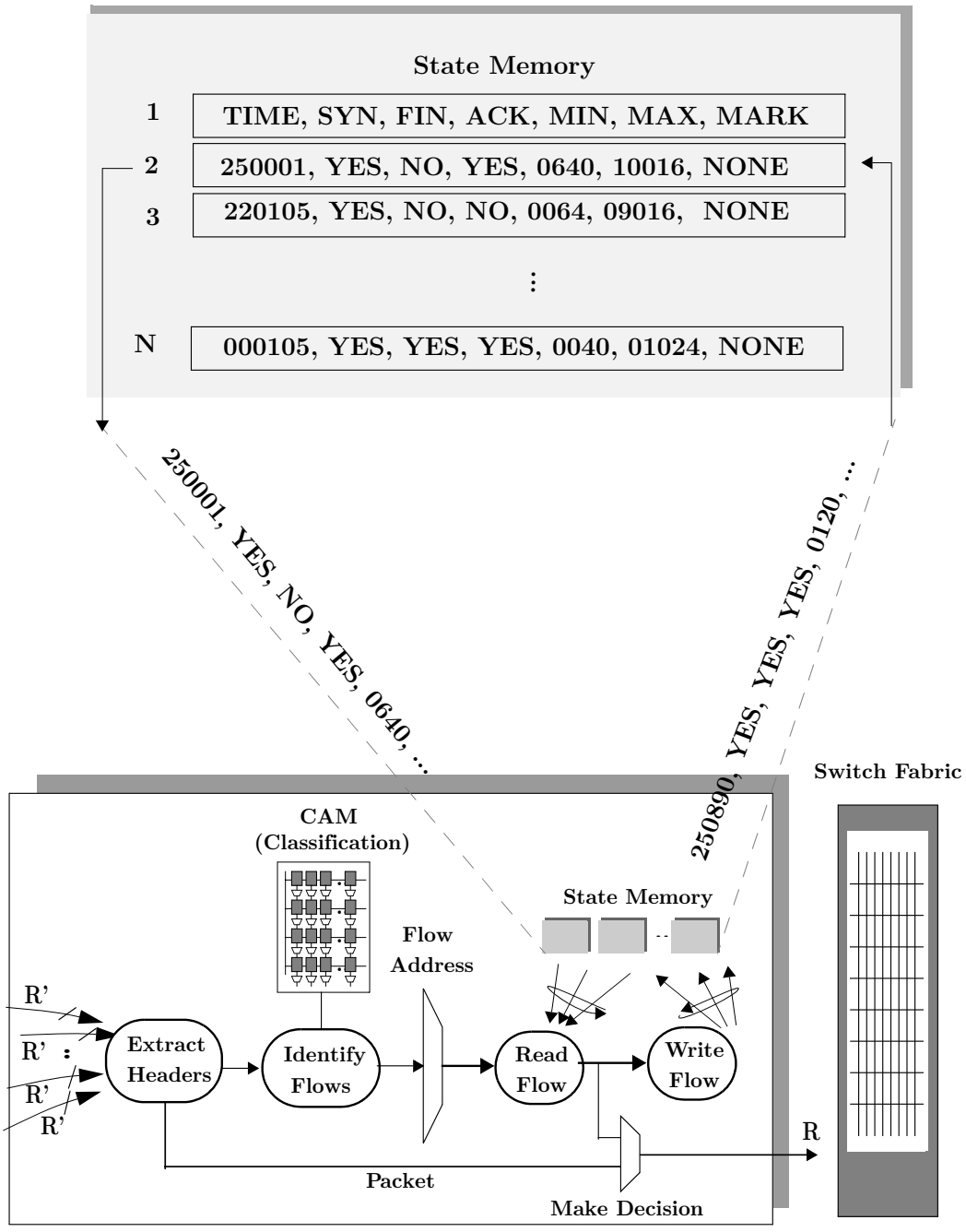


Figure 10.2: *Maintaining state on a line card.*

in Figure 10.2. As we will see, the rate at which state entries can be updated is bottlenecked by memory access time. It is our goal to study and quantitatively analyze the problem of maintaining state entries [210]. We were not aware of any previous work that describes the problem of maintaining state entries at high speeds. And so we are motivated by the following question — *How can we build high-speed memory infrastructure for high-speed routers, particularly when the updates need to occur faster than the memory access rate?*

10.1.1 Characteristics of Applications that Maintain State

We are interested in a general solution (which does depend on the characteristics of any one particular application) that can maintain state entries at high speeds. The applications that our solution must cater to share the following characteristics:

1. Our applications maintain a large number of state entries, one for each flow. For example, a firewall keeps track of millions of flows, while an application proxy usually tracks the state of several hundreds of thousands of connections. These applications require a large memory capacity, making it unfeasible (or at least very costly) to store them on-chip in SRAM. Instead, it becomes necessary to store the state entries in off-chip memory.
2. Our applications update their state entries frequently, usually once for each packet arrival. In the worst case, they must keep up with the rate at which the smallest-sized packets can arrive.
3. Our applications are sensitive to latency. This is because the packets in our applications must wait until the state entry is retrieved before any action can be performed. So our solution should not have a large memory latency. More important, the latency to retrieve a state entry must be predictable and bounded. This is because packet processing ASICs often use pipelines that are several hundred packets long – if some pipeline stages are non-deterministic, the whole pipeline can stall, complicating the design. Second, the system can lose throughput in unpredictable ways.

4. Our applications mandate that state entries be updated at line rate. There must be no loss of performance, and no state entry must be left unaccounted for.

10.1.2 Goal

Our goal is to build a memory subsystem that can update state entries at high speeds that make no assumptions about the characteristics of the applications or the traffic. Indeed, similar to the assumptions made in Chapter 7, we will mandate that our memory subsystem give deterministic performance guarantees and ensure that we can update entries at line rates, even in the presence of an adversary. Of course, it is also our goal that the solution that caters to the above applications can update state entries at high speeds with minimum requirements on the memory.

☞**Observation 10.2.** Routers that keep measurement counters (as described in Chapter 9) also maintain state. Thus, the storage gateway, the central campus router, the local bridge, and the gateway router in Figure 9.1 may all maintain state. Conceptually there are two key differences between applications that keep state and applications that maintain statistics counters — (1) Applications do not need the value of the measurement counter (and hence do not need to retrieve the counter) for each packet, and (2) Counting is a special transformation of the data (*i.e.*, depending on the way counters are stored and their value, only a few bits are modified at a time), whereas a state-based application can transform the read value in a more general manner.

10.1.3 Problem Statement

The task of state maintenance must be done at the rate at which a smallest-size packet can arrive and depart the router line card. For example, with a line rate of 10 Gb/s, a "read-modify-write" operation needs to be done (assuming 40-byte minimum-sized packets) once every 32 ns. Since there are two memory operations per packet, the memory needs to be accessed every 16 ns. This is already faster than the speed of the

most widely used commodity SDRAM memories, which currently have an access time of ∼50 ns. It is extremely challenging to meet this requirement because, unlike the applications that we described in the previous chapters (packet buffering, scheduling, measurement counters, *etc.*), there is no structure that can be exploited when updating a state entry. Indeed, any state entry can be updated (depending on the packet that arrived). The memory subsystem must cater to completely random accesses. Further, as line rates increase, the problem only becomes worse.

✎**Example 10.2.** At 100 Gb/s, a 40-byte packet arrives in 3.2 ns, which means that the memory needs to be accessed (for a read or a write) every 1.6ns. This is faster than the speed of the highest-speed commercial QDR-SRAM [2] available today.

10.2 Architecture

In what follows, we will first describe the main constraint in speeding up the memory update rate in a standard memory, and then describe techniques to alleviate this constraint. We note that read-modify-write operations involve two memory accesses — (1) the state entry is read from a memory location, and (2) then later modified and written back. Since the state entry for a particular flow resides in a fixed memory location, both the read and write operations are constrained, *i.e.*, they must access the same location in memory.

In order to understand how memory locations are organized, we will consider a typical commodity SDRAM memory. An SDRAM's internal memory is arranged internally as a set of banks. The access time of the SDRAM (*i.e.*, the time taken between consecutive accesses to any location in the memory) is dependent on the sequence of memory accesses. While there are a number of constraints on how the SDRAM can be accessed (see Appendix A for a discussion), the main constraint is that two consecutive accesses to memory that address the same bank must be spaced apart by time T_{RC} (which is called the random cycle time of the SDRAM). However, if the two consecutive memory accesses belong to different banks, then they need to

be spaced apart by only around T_{RR} time (called the cycle time of DRAM when there is no bank conflict).

Unfortunately, the bank that should be accessed on an SDRAM depends on where the state entry for the corresponding flow to which an arriving packet belongs is stored. Since we do not know the pattern of packet arrivals, the bank that is accessed cannot be predicted a priori. In fact, for any particular state entry, the read and the write operations must access the same bank (because they access the same entry), preventing consecutive memory operations from being accessed any faster than T_{RC} time. Since designers have to account for the worst-case memory access pattern, the state entries can be updated only once in every two random cycle times in any commodity DRAM memory.

☞**Observation 10.3.** In general for any SDRAM, $T_{RR} < T_{RC}$, and so if one could ensure that consecutive accesses to DRAM are always made to different banks, then one can speed up the number of accesses that an SDRAM can support by accessing it once every T_{RR} time. On the other hand, if these memory banks were completely independent (for example, by using multiple physical DRAMs, or if the memory banks were on-chip), and we could ensure that consecutive memory accesses are always to different banks, then depending on the number of banks available, we could potentially perform multiple updates per T_{RC} time.

10.2.1 The Ping-Pong Algorithm

How can we ensure that consecutive memory accesses do not access the same bank? Consider the following idea:

Idea. *We can read the state from a bank that contains an entry, and write it back to an alternate bank that is currently not being accessed.*

Indeed, the above idea is well known and is referred to in colloquia as the "ping-pong" or "double buffering" algorithm [211, 212]. The technique is simple — a read access to an entry, and a write access to some other entry, are processed simultaneously. The read access always gets priority and is retrieved from the bank that contains the updated state entry. If the write accesses the same bank as the read, then it is written to an alternate bank. Thus every entry can potentially be maintained in one of two banks. A pointer is maintained for every entry, to specify the bank that contains the latest updated version of the state entry.

☞**Observation 10.4.** Note that instead of a pointer, a bitmap could be maintained to specify which of the two banks contains the latest updated version of the entry. Only one bit per entry is needed to disambiguate the bank. In contrast, a pointer-based implementation that allocates a state entry to an arbitrary new location (when it updates the state entry) requires $\log 2E$ bits per entry, where E is the total number of state entries. Of course, in order to maintain bitmaps, when a state entry is written to a bank, it must be written to a memory address that is the same as its previous address, except that it is different in the most significant bits that represent the bank to which it is written. Note that with the above technique, the memory update rate can be speeded up by a factor of two, at the consequence of losing half the memory capacity.

10.3 State Management Algorithm

The ping-pong technique can speed up state updates by a factor of two. However, it is not clear how we can extend the above algorithm to achieve higher speedup. The problem is that the read accesses to memory are still constrained. We are looking for a general technique that can be used to speed up the performance of a memory for any value of the speedup S. This motivates the following idea:

Idea. *We can remove the constraints on the read by maintaining multiple copies of the state entry on separate banks, and alleviate the write constraints by load balancing the writes over a number of available banks.*

The above idea is realized in two parts.

1. **Maintaining multiple copies to remove bank constraints on read accesses:** In order to remove the read constraints on a bank, multiple copies of the state entry are maintained on different banks. Depending on the banks that are free at the time of access, any one of the copies of the state entry is read from a free bank.
2. **Changing the address and maintaining pointers to remove bank constrains on write accesses:** When writing back the modified state entry, again depending on the banks that are free, multiple copies of the modified state entry are written to different banks, one to each distinct bank. Since the memory location to the modified state entry changes, a set of pointers (or a bitmap as described in Observation 10.4) to the multiple copies of the state entry for that flow is maintained. Note that we need to maintain multiple copies of the updated state entry, so that when the corresponding state entry is read (in future), the read access has a choice of the banks to read the updated state entry from.

We will now formally describe a State Management Algorithm called "Generalized Ping-Pong" (GPP-SMA) that realizes the above idea. Assume that GPP-SMA, performs C read-modify-write operations every random cycle time, T_{RC}. In what follows, C is a variable, and we can design our memory subsystem for any value of C. For GPP-SMA, C is also the number of copies of a state entry that must be maintained for each flow. Clearly this is the case, since the C read accesses in a random cycle time must all be retrieved from C distinct memory banks. GPP-SMA maintains the following two sets.

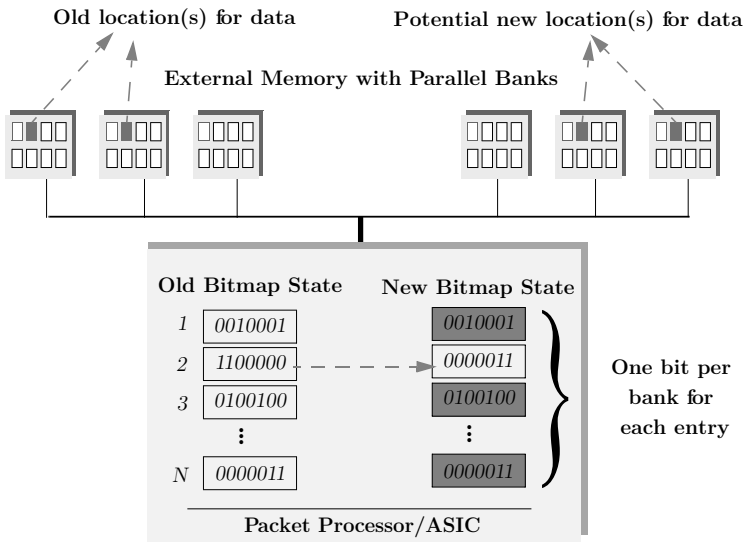


Figure 10.3: *The read-modify-write architecture.*

ℵ**Definition 10.1. Bank Read Constraint Set (BRCS):** This is the set of banks that cannot be used in the present random cycle time because a state entry needs to be read from these banks. Note that at any given time $|BRCS| \leqslant C$, because no more than C read accesses (one for each of the C state entries that are updated) need to be retrieved in every time period.

ℵ**Definition 10.2. Bank Write Constraint Set (BWCS):** This is the set of banks that cannot be used in the present random cycle time because data needs to be written to these banks. During any time period, the state entry for up to "C" entries needs to be modified and written back to memory. Since C copies are maintained for each state entry, this requires $C \times C = C^2$ write accesses to memory. All these C^2 write accesses should occur to C^2 distinct banks (as each bank can be accessed only once in a time period). Hence $|BWCS| \leqslant C^2$.

Figure 10.3 describes the general architecture, and how state entries are moved from one bank to another after they are updated. We are now ready to prove the

Algorithm 10.1: The Generalized Ping-Pong SMA.

```
1  input  : Requests for Memory Updates.
2  output: A bound on the number of memories and total memory bandwidth
           required to accelerate memory access time.

3  C ← Number of updates per random cycle time
4  U ← (1, 2, ..., h) /* Univeral set of all banks */
5  for each time period T do
6      BRCS ← ∅
7      BWCS ← ∅
8      for each read update request to entry e do
9          /* Retrieve set of all banks for entry e */
10         AvailableBanks ← RetrieveBanks(e) \ BRCS
11         /* Choose bank and read entry */
12         b ← AnyBank(AvailableBanks)
13         Read entry e from bank b
14         /* Update Bank Read Constraint Set */
15         BRCS ← BRCS ∪ b

16     for each write update request to entry e do
17         /* Choose C banks for every entry */
18         B_1, B_2, ... B_C ←PickBanksForWrite(U \BRCS \BWCS)
19         Write entry e to chosen C banks B_1, B_2, ... B_C
20         /* Update Bank Write Constraint Set */
21         BWCS ← BWCS ∪ B_1, B_2, ... B_C
22         /* Update bitmap for entry e */
23         UpdateBankBitmap(e, B_1, B_2, ... B_C)
```

main theorem of this chapter:

Theorem 10.1. *(Sufficiency) Using GPP-SMA, a memory subsystem with h independent banks running at the line rate can emulate a memory that can be updated at C times the line rate (C reads and C writes), if $C \leqslant \frac{\lfloor\sqrt{4h+1}-1\rfloor}{2}$.*

Proof. In order that all the C reads and C^2 writes are able to access a free bank, we need to ensure that GPP-SMA has access to at least $C + C^2 = C(C + 1)$ banks. In

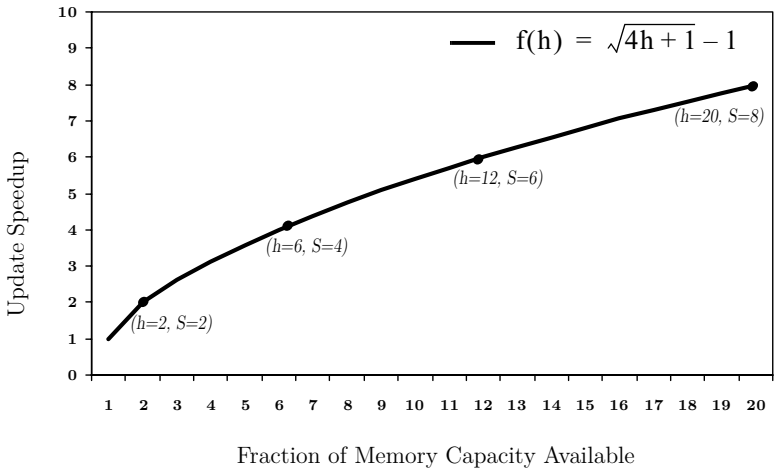


Figure 10.4: *Tradeoff between update speedup and capacity using GPP-SMA.*

any given time period, GPP-SMA first satisfies all the read accesses to memory. If we have at least C banks, then clearly it is possible to satisfy this and read from C unique banks, because each state entry is kept on C unique banks. However, we need more banks. The remaining C^2 writes must be written to C^2 distinct banks (which are distinct from the banks that were used to read C entries in the same random cycle time). Hence, there need to be at least $h \geqslant C + C^2$ independent banks. Solving for C, we get,

$$C \leqslant \frac{\lfloor \sqrt{4h+1} - 1 \rfloor}{2}. \tag{10.1}$$

Note that our analysis assumes that up to C banks are independently addressable in the random cycle time of the memory.[1] This completes the proof. ❒

[1]Commercial DRAM memories limit the maximum number of banks that can be addressed by any single chip in a given random cycle time, due to memory bandwidth limitations. This limit can be increased by using multiple DRAM devices. Of course, if the memory banks are on-chip this is not an issue.

✂Box 10.1: Adversary Obfuscation✂

A large portion of the cost (in terms of memory bandwidth, cache size, *etc.*) of deploying the memory management algorithms described in this thesis comes from having to defend against worst-case attacks by an adversary. In certain instances, the cost of implementing these algorithms may be impractical, *e.g.*, they may require too many logic gates, or use a cache size too large to fit on packet processing ASICs. We could reduce this cost (*e.g.*, our algorithms could be simplified, or the caches can be sized sub-optimally), if we could somehow prevent worst-case adversarial attacks or bound the probability of worst-case adversarial attacks.

In order to achieve this, we have deployed and re-used in multiple instances [33] (and this has also been proposed by other research [204] and development groups [213]) an idea called *adversary obfuscation*. The basic technique is simple —

> **Idea.** *"If the first access or value (*e.g.*, addresses, block offsets, initialization values) in a sequence of operations that manipulate the data structures of the memory management algorithms can be obfuscated, then the relation between an adversarial pattern and the state of the memory management algorithm cannot be deterministically ascertained".*

For example, the above idea can be implemented by randomizing the first[a] access (or value) used by certain data structures, by the use of a programmable seed that is not available to the adversary at run-time. Adversary obfuscation can be overlaid on many of our deterministic memory management techniques to reduce the probability of an adversarial attack to a pre-determined low probability that is acceptable in practice (for example, reducing the chance that an adversary is successful to less than 1 in 10^{30} tries).

☞**Observation 10.5.** Note that *adversary obfuscation* is a natural consequence of the GPP-SMA algorithm. Every time a new entry is added, or an existing entry updated, there is a natural choice of (a subset of h) banks that the new entry must be written to, C times. Since this choice is made at run-time, an adversary cannot predict the banks on which an updated entry is located. Over time, GPP-SMA exercises choice in every time slot, and the probability of an adversarial pattern becomes statistically insignificant.[b]

[a] This can also be done periodically rather than just at the first access.

[b] Of course, the function that exercises this choice can itself be seeded by a programmable value created at run-time, and hence not available to the adversary.

Figure 10.4 shows how GPP-SMA trades off update speedup with capacity. Note that by definition the update speedup $S = 2C$, because with a standard memory only one state entry can be updated every 2 random cycle times, whereas GPP-SMA performs C updates per random cycle time.

From the above analysis, only the even Diophantine[2] solutions for h and Diophantine solutions for C that satisfy Equation 10.1 are true solutions.[3] For example, only the following values of $(h, C) = (2, 1), (6, 2), (12, 3) \ldots$ are valid. This is because the above analysis assumes that C updates are performed in a time period T_{RC}, where C is assumed to be an integer. But if we are interested in non-integer values of C (for example, $C = 2.5$ updates per random cycle time), we could do the above analysis over a different time period which is a multiple of T_{RC}, say yT_{RC}, such that yC is an integer. Then it is possible to find non-integer values of C. This will result in a tighter bound for the number of banks required to support non-integer values of C. In practice, non-integer values of C are almost always required.

✎**Example 10.3.** Consider a 10 Gb/s line card with a state table of size 1 million entries, which are 256 bits in width. Suppose that a minimum-sized packet is 40 bytes, and so it arrives once every 32 ns. The state entries need to be updated once every 32 ns. Suppose we used a 1 Gb DRAM that has a random access time $T_{RC} = 50$ ns (*i.e.*, and so can be updated once every 100 ns, *i.e.*, over three times slower than the rate that is required). Then GPP-SMA, with a DRAM having 4 independent banks, can speed up the memory subsystem to update an entry every 25 ns (such that the DRAM has a capacity of 256 Mb), which is good enough for our purposes.

10.3.1 Consequences

We consider a number of consequences of the GPP-SMA algorithm.

[2]The term *Diophantine* refers to integer solutions.

[3]This is because for any value of C, we need $h = C(C + 1)$ banks, which is always an even number.

1. **Special case for Read Accesses:** If state entries are only read (but not modified), then only the first part of GPP-SMA (*i.e.*, maintaining multiple copies of the state entries) needs to be performed when a state entry is created. In such a case, GPP-SMA degenerates to an obvious and trivial algorithm where state entries are replicated across multiple banks to increase the read access rate supported by the memory subsystem.
2. **Special case for the Ping-Pong Algorithm:** Note that a special case for GPP-SMA (Algorithm 10.1), with $h = 2$ and $C = 1$, leads to the ping-pong algorithm described in Section 10.2.1. Since $C = 1$, no copies need to be maintained. In such a case, only the second part of the GPP-SMA algorithm (*i.e.*, load balancing the write accesses) needs to be performed.
3. **Applicability to Other Applications:** GPP-SMA makes no assumptions about the memory access pattern. It is orthogonal to the other load balancing and caching algorithms described throughout this thesis. And so, it can be used in combination with these techniques.

✎**Example 10.4.** For example, GPP-SMA with h banks can be applied in combination with the buffer and scheduler caching algorithms described in Chapters 7 and 9 respectively to reduce their cache size by a factor of $\sqrt{h}$.

10.4 Implementation Considerations

We consider a number of implementation-specific considerations and optimizations for GPP-SMA.

1. **Optimizing the Data Structure:** It is easy to implement GPP-SMA in hardware. For every memory access, the bank that contains the latest updated version of the state entry needs to be accessed. This can be done by maintaining C pointers to the C location(s) where the most updated version of the state entry is maintained. But if the memory addresses used for the "C" copies of a flow's state entry, which are on "C" different banks, are kept identical (except

for the bank number, which is different), then C separate pointers need not be maintained. We only require a bitmap of size $C(C+1)$ bits per entry to describe which C of the possible $C(C+1)$ banks have the latest updated version of the state entry for that particular entry. This reduces the size required to implement GPP-SMA.

2. **Using DRAM Bandwidth Efficiently:** GPP-SMA can be analyzed for a time period of yT_{RC}. In such a case, each bank can be accessed up to y times in a time period of yT_{RC}. When data is written back to a bank, multiple writes to that bank can be aggregated together and written to contiguous memory locations in an efficient manner.[4] This allows GPP-SMA to utilize the DRAM I/O bandwidth efficiently.

3. **Saving I/O Write Bandwidth:** GPP-SMA performs C updates per random cycle time and creates C copies for each update, for a total of C^2 writes per random cycle time. If the copy function was done in memory, or if the memory was on-chip, there would be no need to carry each of the C^2 writes on the memory interconnect. With a copy function in memory, the number of distinct write entries that are carried over the interconnect would reduce to C per random cycle time. This would save memory I/O bandwidth, reduce the number of ASIC-to-memory interface pins, and reduce worst-case I/O power consumption. Indeed, with the advent of eDRAM [26]-based memory technology (which is based on a standard ASIC process), it is relatively easy to create memory logic circuitry that is capable of copying data over multiple banks.

4. **Scalability and Applicability:** GPP-SMA requires h banks (and hence only gives $\frac{1}{h}$ of the memory capacity) in order to be able to speed up the updates by a factor of $\Theta(\sqrt{h})$. Clearly, this is not very scalable for large values of speedup unless the memory capacity is very large. Based on current technology constraints and system requirements, we have noticed that GPP-SMA is practical for small values of speedup ($S \leqslant 4$). As the capacity of memory improves (roughly at the

[4]DRAM memories are particularly efficient in transferring large blocks of data to consecutive locations. This is referred to as burst mode in DRAM. Refer to Appendix A for a discussion.

Table 10.1: *Tradeoffs for memory access rate and memory capacity.*

Banks per Entry	#Updates per TimeSlot	Memory Access Rate Speedup	Total Memory Bandwidth	Comment
h	C	S	$M \equiv C + C^2$	-
1	0.5	1	1	1 update in 2 T_{RC}'s
2	1	2	2	1 update per T_{RC}
4	1.33	2.67	4	4 updates in 3 T_{RC}'s
5	1.5	3	4.5	3 updates in 2 T_{RC}'s
6	2	4	6	2 updates per T_{RC}
12	3	6	12	3 updates per T_{RC}
h	$C \leqslant \frac{\lfloor\sqrt{4h+1}-1\rfloor}{2}$	$S \leqslant \lfloor\sqrt{4h+1}-1\rfloor$	$M \leqslant h$	C updates per T_{RC}

speed of Moore's law), higher values of update speedup may become practical, as more memory banks per area can be fitted on-chip.

☞**Observation 10.6.** The advent of on-chip memory technology such as eDRAM [26] has allowed ASIC designers the flexibility to structure memory banks based on their application requirements. Today, it is not uncommon to have $h \geqslant 32$ banks with eDRAM-based memory.[5]

Table 10.1 summarizes the performance of GPP-SMA, and describes the tradeoff between the number of banks used, and the update speedup.

10.5 Conclusions

A number of data-path applications on routers maintain state. We used the "constraint set" technique (first described in Chapter 2) to build a memory subsystem that can be used to accelerate the perceived random access time of memory. An algorithm called GPP-SMA was described which can speed up memory by a factor $\Theta(\sqrt{h})$, where h is the number of banks that a state entry can be load balanced over. Our

[5]As an example, Cisco Systems builds its own eDRAM-based memories [32] that have densities, capacities, and number of banks that cater to networking-specific requirements.

algorithm makes no assumptions on the memory access pattern. The technique is easy to implement in hardware, and only requires a bitmap of size h bits for each entry, to be maintained in high-speed on-chip memory. The entries themselves can be stored in main memory (for example, commodity DRAM) running $\Theta(\sqrt{h})$ slower than the line rate. For example, a state table of size $\sim$ 160 Mb, and capable of updating 40 M updates/sec, can be built from DRAM that allows up to 6 banks to be independently addressed in a random cycle time, with capacity 1 Gb and a T_{RC} of 50 ns (which could have supported no more than 10 M updates/s on its own).

Our technique has two caveats — (1) The DRAM capacity lost in order to speed up the random cycle time grows quadratically as a function of the speedup, limiting its scalability, and (2) The technique requires larger memory bandwidth.

While there are systems for which the above technique cannot be applied (*e.g.*, systems that require extremely large speedup), we have seen that these techniques are practical for small values of speedup ($S \leqslant 4$). While we have not used these techniques for current-generation 40 Gb/s line cards (mainly due to the use of eDRAMs [26] which were able to meet the random cycle time requirements at 40 Gb/s), we are currently negotiating deployment of the above technique for the next-generation 100 Gb/s Ethernet switch and Enterprise router line cards, where even eDRAM can be too slow to meet the random cycle time requirements.

Summary

1. A number of data-path applications on routers maintain state. These include: stateful firewalling, policing, network address translation, *etc.*
2. The state maintained is usually to track the status of a "flow". The granularity of a flow can be coarse (*e.g.*, all packets destined to a particular server) or fine (*e.g.*, a TCP connection), depending on the application.
3. When a packet arrives, it is first classified, and the flow to which it belongs is identified. An entry in the memory (to which the flow belongs) is identified. Based on the packet, some action is performed, and later the entry is modified and written back. This is referred to as a "read-modify-write" operation.

4. However, the large random access times of DRAMs make it difficult to support high-speed read-modify-write operations to memory.
5. We consider a load balancing algorithm that maintains copies of the flow state in multiple locations (banks) in memory. The problem is to ensure that irrespective of the memory access pattern, flow state entries can be read and updated faster than the memory access rate of any single memory bank.
6. We describe and analyze a state management algorithm called "generalized ping-pong" (GPP-SMA) that achieves this goal.
7. The main result of this chapter is that the generalized ping-pong algorithm, with h independent memory banks running at the line rate, can emulate a memory that can be updated at C times the line rate (C reads and C writes), if $C \leqslant \frac{\lfloor\sqrt{4h+1}-1\rfloor}{2}$ (Theorem 10.1).
8. GPP-SMA is resistant to adversarial memory accesses that can be created by hackers or viruses; and its performance can never be compromised, either now, or provably, ever in future.
9. GPP-SMA requires a small bitmap to be maintained on-chip to track the entries maintained in DRAM, and is extremely practical to implement.
10. GPP-SMA can speed up the memory access time of any memory subsystem. Hence the technique is orthogonal to the various load balancing and caching techniques that are described in the rest of this thesis. Thus it can be applied in combination with any of the other memory management techniques. For example, it can reduce the cache sizes of the packet buffering and scheduler caches described in Chapters 7 and 8.
11. The main caveat of the technique is that it trades off memory capacity for random cycle time speedup, and requires additional memory bandwidth, and so its applicability is usually limited to low values of speedup ($S \leqslant 4$).
12. At the time of writing, we are considering deployment of these techniques for the next generation of 100 Gb/s Ethernet switch and Enterprise router line cards.