CAMP: Fast and Efficient IP Lookup Architecture

Sailesh Kumar
Washington University
sailesh@arl.wustl.edu

Michela Becchi
Washington University
mbecchi@arl.wustl.edu

Patrick Crowley
Washington University
pcrowley@arl.wustl.edu

Jonathan Turner
Washington University
jon.turner@wustl.edu

ABSTRACT

A large body of research literature has focused on improving the performance of longest prefix match IP-lookup. More recently, embedded memory based architectures have been proposed, which delivers very high lookup and update throughput. These architectures often use a pipeline of embedded memories, where each stage stores a single or set of levels of the lookup trie. A stream of lookup requests are issued into the pipeline, one every cycle, in order to achieve high throughput. Most recently, Baboescu et al. [21] have proposed a novel architecture, which uses circular memory pipeline and dynamically maps parts of the lookup trie to different stages.

In this paper we extend this approach with an architecture called *Circular, Adaptive and Monotonic Pipeline* (*CAMP*), which is based upon the key observation that circular pipeline allows decoupling the number of pipeline stages from the number of levels in the trie. This provides much more flexibility in mapping nodes of the lookup trie to the stages. The flexibility, in turn, improves the memory utilization and also reduces the total memory and power consumption. The flexibility comes at a cost however; since the requests are issued at an arbitrary stage, they may get blocked if their entry stage is busy. In an extreme case, a request may block for a time equal to the pipeline depth, which may severely affect the pipeline utilization. We show that fairly straightforward techniques can ensure nearly full utilization of the pipeline. These techniques, coupled with an adaptive mapping of trie nodes to the circular pipeline, create a pipelined architecture which can operate at high rates irrespective of the trie size.

Categories and Subject Descriptors

C.2.6 [**Computer Communication Networks**]: Internetworking – Standards (e.g., TCP/IP)

General Terms: Algorithms, Design.

Keywords

Internet router, IP lookup, longest prefix match.

1. INTRODUCTION

Recent advances in optical and signaling technology have pushed network link rates beyond 10 Gbps, with 40 Gbps links now appearing. A line card terminating a 40 Gbps IP link needs to forward a minimum-sized packet within 8 ns. To do so, the outgoing line card must be identified based on the packet's destination address and the current set of IP routes. Thus, the routing table must be searched every 8 ns. This is challenging because: *i*) IP address lookup requires a longest prefix match, which in turn requires several sequential memory accesses per match, and *ii*) global routing tables contain over one hundred thousand prefixes and are growing. The dual challenges of serialized access and large datasets have inspired a number of novel algorithms and data structures. Many implementations rely on tree-based data structures, such as tries, to encode IP route tables. In these schemes, the longest prefix is found by traversing the trie from the root node to the matching leaf node, using a stride of one or more bits from the search string.

In order to forward packets at increasing link rates, modern routers employ specialized hardware based on these ideas to perform IP lookup. Memory bandwidth is an important concern in any implementation, whether it is based on off-chip memory or an ASIC. For example, at 40 Gbps rates, a multi-bit trie of stride 4 requires 8 memory accesses every 8 ns. Achieving this bandwidth using a single memory is challenging. A number of researchers have proposed a pipelined trie. Such tries enable high throughput because when there are enough memories in the pipeline, no memory stage is accessed more than once for a search and each stage can service a memory request for a different lookup each cycle.

Most recently, Baboescu et al. [21] have proposed a circular pipelined trie, which is different from the previous ones in that the memory stages are configured in a circular, multi-point access pipeline so that lookups can be initiated at any stage. At a high-level, this multi-access and circular structure enables much more flexibility in mapping trie nodes to pipeline stages, which in turn maintains uniform memory occupancy. In this paper, we extend this approach with an architecture called Circular, Adaptive and Monotonic Pipeline (CAMP). Our work, while also exploiting a circular pipeline, differs from the previous circular pipeline proposals in several ways.

First, CAMP differs in the way the trie is split into sub-tries. While [21] aims at having a large (~4000) number of equally sized sub-tries, our design strives for simplicity. Thus, CAMP splits a trie into one root sub-trie and multiple leaf sub-tries. Root sub-trie handles first few bits (say *r*) of the IP address, and it is implemented as a table, indexed by the first *r* bits of the IP address. With this, there may be up to 2^r leaf sub-tries; each of which can be independently mapped to the pipeline. By judiciously mappings these, the systems maintain near-optimal memory utilization, not only in memory space but also in number of accesses per pipeline stage.

Second, having a reduced number of sub-trie of different sizes, we propose a different heuristic to map them to the pipeline stages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'06, December 3-5, 2006, San Jose, California, USA.
Copyright 2006 ACM 1-59593-580-0/06/0012...$5.00.

As a matter of fact, our scheme proves to be much simpler, which also gracefully handles incremental updates.

Finally, our design uses a different mechanism to maximize pipeline utilization and handle out of order lookup conditions. In particular, we aim at having not more than one access per pipeline stage for any lookup. CAMP goes further and decouples the dependence of number of pipeline stages from the number of trie levels. Thus it can employ a large number of compact and fast pipeline stages to enable high throughput while consuming low power. With large number of stages, pipeline utilization may degrade significantly. To this end, CAMP employs effective schemes to achieve high utilization.

We also present an extensive analysis of the design tradeoffs and their impact on lookup rate and power consumption. For real routing tables storing 150 thousand prefixes, CAMP achieves 40Gbps throughput with a power consumption of 0.3 Watts. Projections on 250 thousand prefixes show a power consumption of 0.4 Watts at the same throughput.

The rest of the paper is organized as follows. In Section II we discuss the related work. In Section III we describe the operation of pipelined trie. In Section IV we present a heuristic to map trie nodes to pipeline stages. In Section 5 we present experimental results and in Section VI we discuss worst-case scenarios. Finally, in section VII, we summarize our findings.

2. BACKGROUND

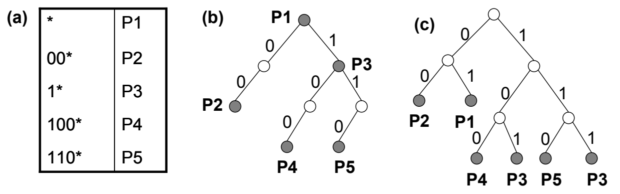
IP lookup consists of determining the longest prefix matching the destination address field within the routing database of variable length prefixes. Longest prefix match for IP lookup has been widely studied. Some well known IP-lookup mechanisms encompasses from TCAM [9][10] to Bloom filters [6] to hash tables [1] based schemes. Since these are not directly related to the current work, we will focus on the trie based lookup schemes.

2.1. Trie based IP Lookup

A large fraction of memory-based solutions use lookup tries. A trie is an ordered tree data structure associating a string s_x to each node n_x; s_x is not explicitly stored at any point of the tree, but can be derived by following the path from the root of the trie to the leaf node n_x. A basic property of tries is that all descendants of a node n_x share a common prefix, represented by the string associated to n_x. In context of IP lookup, a binary trie representing a routing table can be built by traversing each prefix from the leftmost to the rightmost bit, and inserting in the trie, a left child for each 0 and a right child for each 1. For an example, see Figure 1(a) and (b). Nodes corresponding to the valid prefixes must be marked with a prefix pointer. Lookup is performed by traversing the trie according to the bits in the IP address. When a leaf is reached, the last marked node traversed corresponds to the longest matching prefix.

As illustrated in Figure 1(b), each node contains two pointers: one into the array of child nodes and one prefix pointer. To reduce memory usage, *leaf pushing* (Figure 1(c)) has been proposed [1], wherein prefixes at non-leaf nodes (e.g.: P1, P3) are pushed down to the leaves. Thus, each node stores either a prefix pointer or a pointer to the array of children. However, leaf-pushed nodes may need to be replicated at several leaves (e.g.: P3). Therefore on average, leaf pushing does not halve the memory. Moreover, it also complicates the updates.

If several bits are scanned for each node traversal, then the resulting data structure is a *multibit* trie. The number of bits scanned



*	P1
00*	P2
1*	P3
100*	P4
110*	P5

Figure 1: (a) Routing table; (b) corresponding unibit trie; (c) corresponding leaf-pushed unibit trie.

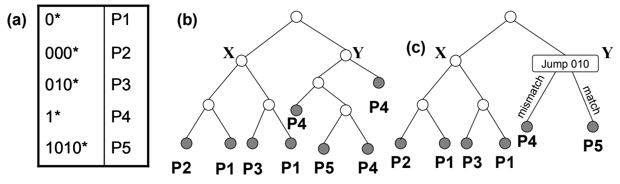
once is *stride*. A node with stride d will have a maximum of 2^d child nodes. In multi-bit trie, some prefixes may be expanded to align to the stride boundaries, which may increase the size of the routing table. However, during a node traversal, multiple bits are scanned, which reduces the number of scans. Since the time to complete a lookup is determined by the trie depth, the choice of stride depends upon the *lookup time-memory tradeoff*: lower strides allow a more compact data structure but require more memory accesses, whereas higher strides reduce the lookup time at the cost of more memory.

Controlled prefix expansion has been introduced [2] in order to address the above issue. Given the maximum number of memory accesses allowed for a lookup (i.e., trie depth), this technique uses dynamic programming to determine the stride leading to the minimum total memory. However, this involves two important limitations: first, it is suitable for building a trie from scratch but does not support incremental updates; second, while reducing the total memory, this technique does not control the *per level* memory occupancy in a pipelined trie. The reason for this will be explained shortly.

2.2. Pipelined IP Lookup Tries

An effective way to tackle the time-memory tradeoff is to recognize that tries are well suited for *data structure pipelining* [7][8]. A common way to pipeline a trie is to assign each trie level to a different stage so that a lookup request can be issued every cycle, thus increasing the throughput. Besides increasing the throughput, such pipelined implementations are also suitable for handling updates. In fact, as proposed in [7], software preprocessing of prefix insertions and deletions can be exploited in order to determine the necessary per-level modifications to be performed in the trie. In a second phase, those write operations can be inserted in the pipeline in the form of “write bubbles”. Because of the sequential character of the pipeline operation, straightforward techniques can prevent write operations from interfering with the existing lookups.

In a pipelined implementation, it is desirable for nodes to be distributed uniformly across pipeline stages. [7] applies an extended version of controlled prefix expansion to achieve this objective. Rather than minimizing the total memory, the modified algorithm aims at minimizing the size of the largest trie level, while still keeping the total memory low. Through the use of variable-stride tries (having a fixed per level stride but allowing different strides at



0*	P1
000*	P2
010*	P3
1*	P4
1010*	P5

Figure 2: (a) Routing table; (b) corresponding unibit leaf pushed trie; (c) unibit trie with jump nodes.

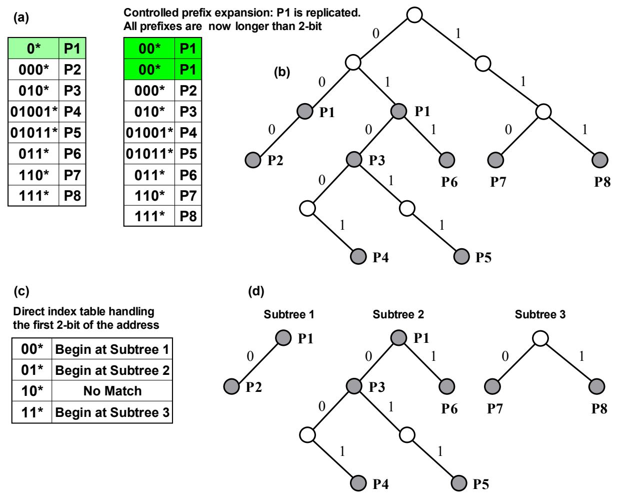
different levels), it achieves a discretely balanced prefix distribution across pipeline stages.

An alternative approach is presented in [8], where *height*-based (rather than level-based) pipelining is proposed. The work does not aim at balancing memory utilization; rather, it focuses on guaranteeing worst case performance bounds. In particular, it focuses on leaf-pushed unibit tries using a technique called *jump nodes*, which limits the number of copies of a leaf-pushed node. The usage of jump nodes is exemplified in Figure 2, where all descendants of node *Y* represent either the prefix *P4* (leaf-pushed) or *P5*. Clearly, all internal nodes in the subtree rooted at *Y* can be condensed into a jump node carrying the information about the remaining portion of *P5*. In [8], authors argue that jump nodes ensure that the number of leaves in a leaf pushed unibit trie is equal to the number of prefixes, which enables *O(1)* updates. Unfortunately, since not all the copies of leaf-pushed nodes can be removed by using jump nodes (see *P1* in Figure 2), such claims are incorrect. Moreover, height-based pipelining leads to unbalanced stages; as a workaround, hardware-based pipelining has been proposed, which, adds to complexity and power consumption.

As we have already mentioned, the most recent and the most efficient pipelined trie has been proposed in [21], which uses a circular pipeline with dynamic pipeline entry points.

2.3. Efficient Encoding of Multibit-Trie Nodes

The last relevant aspect studied in the literature is the use of compression to reduce memory requirements. In particular, the Lulea scheme [4] is suited for tries using leaf pushing, whereas the Tree Bitmap algorithm [5] focuses on non-leaf-pushed multibit tries. Specifically, Tree Bitmap allows *O(1)* updates as compared to Lulea, while requiring comparable memory. As mentioned previously, due to the limited effectiveness of leaf pushing in reducing the total memory requirements, we focus on non-leaf-pushed tries. Our infrastructure in a way is orthogonal to these compression techniques. Therefore, we first present our mapping algorithm on a simple unibit trie and then show its adaptation to multibit tries and Tree Bitmap scheme.


(a)
Controlled prefix expansion: P1 is replicated.
All prefixes are now longer than 2-bit
(b)
0
1
P1
P2
P3
P4
P5
P6
P7
P8
(c)
Direct index table handling the first 2-bit of the address
(d)
Subtree 1
Subtree 2
Subtree 3

0*	P1
000*	P2
010*	P3
01001*	P4
01011*	P5
011*	P6
110*	P7
111*	P8

00*	P1
00*	P1
000*	P2
010*	P3
01001*	P4
01011*	P5
011*	P6
110*	P7
111*	P8

00*	Begin at Subtree 1
01*	Begin at Subtree 2
10*	No Match
11*	Begin at Subtree 3

Figure 3: (a) Routing table (prefixes shorter than 2-bits are expanded using controlled prefix expansion) (b) unibit trie of six levels; (c) Direct index table for first 2-bits, (d) resulting 4 sub-tries of four levels each.

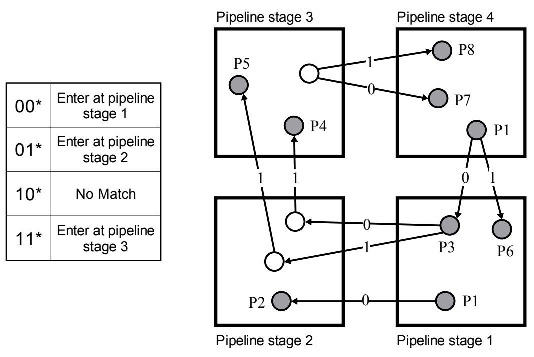
3. CAMP

Pipelining is an effective way to achieve high lookup rates. Previous pipelined schemes are based on the assumption that the pipeline is linear, and has a unique entry and exit point; moreover, it is assumed that a global mapping is performed on the entire trie. We remove both assumptions based on the observation that practical prefix-sets present us considerable opportunity to split a trie into multiple sub-tries; thus, different pipeline entry points can be assigned to them. This leads to many mapping opportunities, from which assignments may be chosen to achieve balanced pipeline stages. Moreover, it also eliminates two important limitations faced by any global mapping based scheme, namely, 1) the number of pipeline stages is bound to the maximum prefix length, and 2) adding a memory bank requires a complete remapping (in scenarios of an overflow generated by a sequence of prefix insertions).

We introduce Circular Adaptive and Monotonic Pipeline (CAMP) using a set of 8 small prefixes shown in Figure 3 along with the corresponding binary trie. Pipelining this trie will require 6 stages. A level-based mapping will result in 1, 2, 3, 5, 2 and 2 nodes in stages 1 to 6, respectively, while a height-based mapping will result in 6, 4, 2, 1, 1 and 1 nodes. Thus, both of these mapping creates unbalanced pipeline and the degree of imbalance is dependent upon the prefix set.

We now consider splitting this trie into four sub-tries. Since prefix P1 is only 1-bit long, we first expand it to 2 bits using controlled prefix expansion (see Figure 3). Now, all prefixes in the database are longer than 2-bits; therefore, the upper two levels of the trie can be stored in a direct index table, which leaves us with three sub-tries of four levels each. More generally, when a routing database contains prefixes all of which are longer than *x*-bits (shorter prefixes are expanded to *x*-bits), then the first *x* levels of the trie can be replaced by a direct index table containing 2^x entries, each of which points to one of the up to 2^x sub-tries with height at most $32 - x$.

With multiple subtries, we now seek to obtain a balanced mapping of nodes to pipeline stages. We exploit the fact that requests can enter and exit at any stage, thus roots of sub-tries can be mapped to any stage. If we also allow a request to wrap-around through the pipeline (i.e., by taking advantage of the circular pipeline), we can get a high degree of flexibility in mapping. Nodes descended from the root of a sub-trie can be stored at subsequent pipeline stages, wrapping around once the final stage is reached. In the example above, the 3 sub-tries constructed from the 8 prefix table can be mapped to a four stage circular memory pipeline with dynamic entry points as shown in Figure 4. Note that the first two



00*	Enter at pipeline stage 1
01*	Enter at pipeline stage 2
10*	No Match
11*	Enter at pipeline stage 3

Figure 4: A four stage circular pipeline and the way the three subtries in Figure 3 are mapped onto them.

bits are used to determine the entry stage into the pipeline and subsequent bits are processed within different pipeline stages.

3.1. General Dynamic Circular Pipeline

A general circular pipeline may not require a node to be stored in a stage *adjacent* to the parent node's stage. For example, the two nodes of the first sub-trie in the previous example can be stored at any two distinct stages, because, irrespective of the way they are stored, a lookup request for this sub-trie will access each stage only once. However, this will require the pipeline to insert no-ops when request traverses a stage where the required node is not present. Supporting no-ops increases the flexibility in storing the nodes of various sub-tries which can lead to more balanced pipeline stages. On the other hand, as will be shown later, it may complicate the update scenario.

A general circular pipeline has three important properties, *i*) it allows dynamic entry and exit points, *ii*) it is circular, thus all neighboring stages are connected in one direction, and *iii*) it supports no-ops for which requests are simply passed over whenever the designated node is not found. The corresponding mapping algorithm maps the root of each sub-trie to some pipeline stage and subsequent nodes are mapped such that, *a*) a node is stored at a stage which is at least one ahead (including wraparound) of the stage where its parent is stored, and *b*) all lookup paths terminate before making a circle through the pipeline. Thus, nodes along any path are mapped in a monotonically increasing pipeline stage and every lookup is guaranteed to make at most one access to a memory stage.

It can be argued that the lookup throughput of a general circular pipeline matches that of any other pipeline because a lookup request accesses a memory at most once. However, allowing dynamic entry points introduces new problems due to request conflicts. A request contending to enter the i^{th} stage may have to wait until a bubble (idle cycle) arises there. In an extreme case, a request may have to wait for such a bubble indefinitely, if other requests are entering the pipeline every cycle and keeping its entry stage busy. This may lead to non-deterministic performance, low pipeline utilization and out-of-order request processing. However, as we will see next, relatively straightforward techniques coupled with a small speedup in pipeline operating-rate ensure deterministic performance.

3.2. Detailed Architecture of the CAMP

The schematic block diagram of a CAMP system is shown in Figure 5, which consists of a circular pipeline of memories. The first block performs a direct table lookup on the first *x*-bits of the address (*x* being the initial stride in the lookup trie), which provides the stage where the root node of the corresponding sub-trie is located. Subsequently, a lookup request to traverse through the sub-trie is dispatched into that stage. All requests are stored in the ingress FIFO in front of each memory stage. As soon as the corresponding stage receives a bubble (idle cycle), the request at the head of the FIFO is issued into the pipeline. Once a request traversing through the pipeline reaches the stage containing the last node, it comes out with either the valid next hop information or a no match.

The ingress FIFO in front of each stage plays a critical role in improving the efficiency. Consider a system without such queues. It is possible that a stream of *n* lookup requests enters a given stage resulting in a train of *n* requests in the pipeline. All subsequent requests contending to enter the pipeline may have to wait for *n*

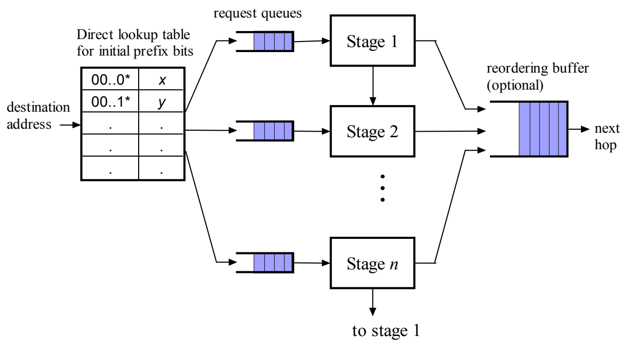


Figure 5: Schematic block diagram of a CAMP system

cycles. Thus, the efficiency can be as low as 50%, because the pipeline services *n* requests and then waits for *n* cycles before servicing subsequent requests. The worst-case efficiency can be even lower. Consider a situation when a request enters the pipeline and the next waits for 1 cycle. After it enters, the third has to wait for 2 cycles. Thus, the i^{th} request waits for i–1 cycles, which will lead to a very low efficiency.

The ingress FIFO serves as a reorder buffer, which obviates the above head of line blockings. If a request must wait for few cycles before it can be serviced, it stays in its queue and therefore does not block the subsequent requests. Quite intuitively, larger request queues will improve the efficiency of the pipeline, as they will provide extended immunity against requests which must block before being serviced.

While improving the dispatch rate into the pipeline, these FIFOs also leads to requests being serviced out of order. Therefore, an optional reorder buffer is present at the output, which restores the order of requests. Reordering is optional because the problem of out-of-order arises only among the packets destined to different destinations. A single TCP flow will never experience any reordering, as any two packets having the same prefix (thus designated to the same "next hop") always traverse thru the same path in the lookup trie. Hence, these requests will contend to enter the pipeline at the same stage, where they are always serviced in a first-in first-out order. We now introduce the metric of pipeline efficiency and characterize it for different pipeline configurations and input traffic patterns.

3.3. Characterizing the Pipeline Efficiency

One metric characterizing the efficiency of CAMP is *pipeline utilization*. Pipeline utilization is the fraction of time the pipeline remains busy provided that there is a continuous backlog of lookup requests. Another metric, which more directly reflects the performance, is *Lookup per Cycle* or *LPC*, i.e. the rate at which lookup requests are dispatched into the pipeline.

A linear pipeline guarantees an LPC of 1 however pipeline utilization can remain low if a majority of prefixes are not 32-bits long (hence they do not use all stages). In a CAMP pipeline, on the other hand, pipeline utilization can approach one therefore requests may be dispatched at rates higher than one per cycle. It can happen, *i*) when most requests do not make a complete circle through the pipeline, or *ii*) when there are more pipeline stages than there are levels in the trie. Thus, whenever some pipeline stages are not traversed by a request, new requests contending to enter there can be issued. Note that, practical IP lookups, where a majority of prefixes are only 24-bits long, leave a large fraction of stages unused.

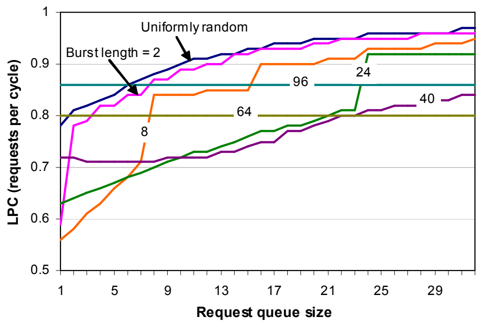


Figure 6: LPC of CAMP versus request queue size.

In order to evaluate the efficiency of CAMP, we have performed a first order analysis of the pipeline utilization and the resulting LPC. To simplify the analysis presented here, we assume that all requests make one complete circle through the pipeline and there are as many pipeline stages as there are levels in the trie (in this case pipeline utilization will be equal to the LPC). Later we consider scenarios, when requests do not make a complete circle. The only variant now is the entry point in the pipeline. We consider following four distributions of the entry points of the arriving requests: *i*) uniformly random, *ii*) uniformly random short burst of requests at each pipeline stage, and *iii*) uniformly long burst of request at each stage, and *iv*) weighted random, so some pipeline stages receives more requests than the others.

Not surprisingly, long burst of requests result in high utilization because when many requests arrive at a stage, they are serviced without conflict. On the other hand, when the burst lengths are comparable to the pipeline depth, trains of requests are created and subsequent bursts may have to wait before they can be dispatched. Uniformly and weighted random request arrivals can be modeled using a discrete time Markov chain, however, we skip the details due to space constraints.

We continue with the results from our software simulations, where we generate the above four request arrival patterns and measure the resulting LPC. Our representative setup has 24 pipeline stages and requests circle through all stages before exiting. In Figure 6, we report the LPC for different request queue sizes. It is clear that, a LPC of 0.8 can be achieved for all traffic patterns, once the request queue size is 32. This suggests that CAMP remains 80% efficient for practically all traffic patterns. In another experiment, we fixed the request arrival rate at 0.8 per cycle and request queue size at 32 and measured the discard rate and the average delay experienced by a request. After running the experiment for more than 100 million iterations, no requests were discarded and the average delay experienced by a request was only a few tens of cycles.

3.4. When is LPC greater than one?

While the LPC of a linear pipeline is always one, the LPC of CAMP can be engineered to be greater than one, which can improve the throughput. This is possible because CAMP enables a trie data-structure to be pipelined further, up to the number of stages much higher than the number of levels in the trie. For example, the mapping of the three sub-tries shown in Figure 3 to a six stage pipeline is shown in Figure 7. As we will see soon, with many sub-tries, it is not difficult to determine the offsets for each of them so that every stage of the pipeline remains nearly uniformly populated. When there are many stages in the pipeline, each sub-trie (and its lookup requests) will span only a fraction of all stages. This

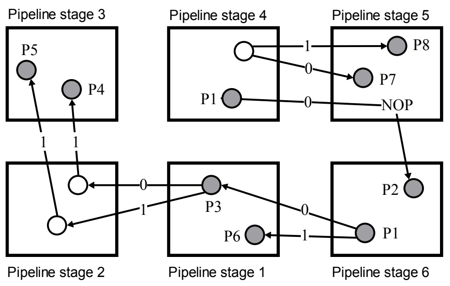


Figure 7: A six stage circular pipeline and the way the three sub-tries in Figure 3 are mapped onto them.

can lead to a dispatch rate higher than one per cycle, assuming that all arriving requests do not traverse the same sub-trie. In fact, when sub-tries and therefore the associated prefixes are nearly uniformly dispersed all around the stages (because stages are balanced), it is less likely that all lookup requests will contend to enter one stage. An orthogonal factor leading to higher LPC is the fact that most prefixes are in close vicinity of 24-bits.

It is neither difficult nor expensive to implement more pipeline stages than the levels in a trie. From a practical perspective, a multi-bit trie with the appropriate node encoding (tree-bit map or shape shifting trie), can not only reduce the total memory requirement by also effectively increase the number of stages in the pipeline. For example, a stride of *k* will reduce the number of levels (and the memory accesses) in a trie by a factor of *k*, which can directly lead to a *k*-times higher LPC.

We now present the LPC of a setup where there are 32-stages in the pipeline and the sub-tries contain the leftmost 24- prefix bits. Each sub-trie uses tree-bit map of stride 3, thus a single lookup path spans across at most 8 pipeline stages. In this experiment, we also assume that the average prefix length is 24-bits, thus a request on average traverses through only 6 stages. As plotted the LPC in Figure 8, the LPC ranges from 3 to 5, even for large bursts of traffic. For smaller bursts, which are more realistic, LPC is even higher.

4. MAPPING IP-LOOKUP TRIES TO CAMP

In order for the proposed infrastructure to operate, we need a mapping algorithm which assigns the trie nodes to the pipeline stages. The primary purpose of the algorithm is to achieve a uniform distribution of nodes to stages. In particular, the mapping should minimize the size of the biggest (and bottleneck) stage. This will not only enable high throughout but also reduce the chances of unbalanced pipeline during updates.

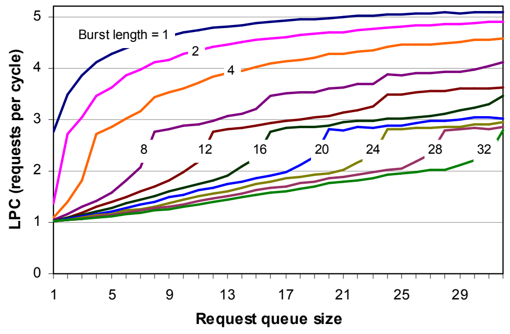


Figure 8: LPC of CAMP versus request queue size.

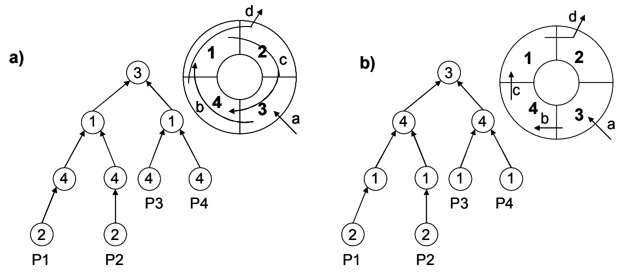


Figure 9: a) invalid assignment: matching P1 causes one extra loop of the circular pipeline; b) valid assignment: the circular pipeline is traversed only once.

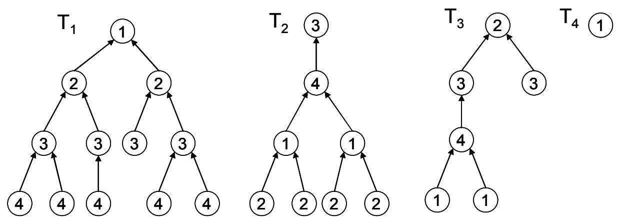


Figure 10: Example coloring with min-max heuristic.

4.1. Problem Formulation

We can formulate the above problem as a *constrained graph coloring problem*, where colors represent the pipeline stages, and graph represents the set of sub-tries. The following two constraints guide the coloring: *i*) every color should be nearly equally used, and *ii*) a relation of order, when traversing a sub-trie from the root to the leaves, must be associated with the color assignment. The first constraint captures the intent of achieving a uniform distribution of nodes across the pipeline stages. The second constraint arises due to the fact that nodes must be mapped to the circular pipeline in such a way that any lookup request makes at most one complete circle through the pipeline. Thus, all paths from root to leaf must be assigned distinct colors in a monotonic order (including wrap-around).

If we represent each colors by an integer, the relation of order is the "saturated <" relation. In other words, if we have *N* colors (1, 2, ..., *N*) then the following relation will hold: 1<2<...<N<1. A mapping which doesn't preserve such an order relation is exemplified in Figure 9(a). Such a mapping can lead a lookup to circle through the pipeline multiple times, thus reducing the overall throughput. A mapping which preserves the order relation is illustrated in Figure 9(b), where all paths from root to leaf (i.e. any lookup operation) traverse through a color at most once. Naturally there can be several mapping choices which will preserve the order relation and we are interested in those which lead to a nearly uniform usage of different colors. We believe that such a constrained graph coloring problem is NP-hard and can be reduced to the well known bin-packing problem therefore we present a heuristic algorithm to obtain a near optimal solution.

4.2. The Min-Max coloring algorithm

Several simple heuristics can be obtained to perform the coloring which preserves the order relation. For instance, each sub-trie can be colored by first randomly selecting a color for the root node and then incrementing the color when proceeding towards the leaves. While such a randomized scheme may lead to fairly balanced color distributions in case of a large number of sub-tries, it may be not satisfactory when there are not that many sub-tries or when some sub-tries are significantly larger than others. We therefore introduce a more effective coloring heuristic. In particular, if we color sub-tries sequentially, at each coloring step we want to exploit the information about the current status of the color distribution.

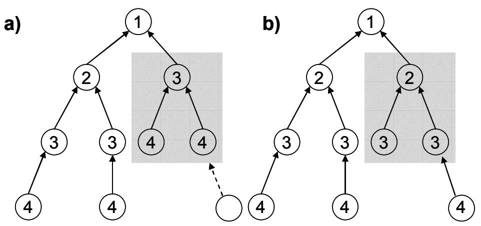


Figure 11: An insertion operation causes a subtree rotation in case of skip-level assignment.

A *min-max coloring heuristic* seeks to obtain uniform color usage by coloring the sub-tries in a sequence such that the larger sub-tries are colored before the smaller ones. Such a sequence is motivated by the well-known bin-packing heuristic and can be attributed to the fact that if tries are colored in a decreasing size sequence then the coloring of smaller tries can effectively correct the unbalances caused by the already colored bigger tries. Thus, min-max heuristic first sorts all sub-tries according to their size and then in a decreasing order, assigns colors to the nodes of the sub-tries. For the currently selected sub-trie, the coloring needs to restore any discrepancy in the color usage until now. Since, the choice of a color for the root determines the colors for all subsequent nodes therefore, min-max algorithm tries all possible colors to color the root node (subsequent nodes are colored with increasing color values) and tracks the color usage for each choice. Eventually it picks the one which results in the most uniform color usage and moves on to the next sub-trie. Figure 10 illustrates the application of the min-max heuristic on a set of four sub-tries.

4.3. Additional considerations

The above coloring heuristic only considers unitary increment between colors of a node and those of its children. It would be possible to add further flexibility in color assignment by removing this constraint, without affecting the correctness of the system. The shaded area in Figure 11(a) illustrates such possibility. The added flexibility may lead to more uniform usage of colors however it complicates the coloring. The complexity may be acceptable if the mapping were static, however in practical systems, updates often adds and removes nodes from the tries, in which case, remapping a large part of the trie may be needed if the unitary increment constraint was not applied. As an example, let us add a bottom right node to the tree shown in Figure 11(a). Since color 4 is already used at the leaf, the colors of the nodes in the shaded area must be reassigned, as illustrated in Figure 11(b). Due to these costly updates, we do not consider possibility of skipping colors between adjacent levels.

5. EXPERIMENTAL EVALUATION

In this section we evaluate the memory requirements and performance of CAMP and compare it with those of linear pipelined schemes proposed earlier. We first consider unibit tries and show how the selection of initial stride affects the node distribution across various stages. Subsequently, we analyze the impact of route updates on a balanced CAMP pipeline. Thereafter, we extend these analyses to a multi-bit trie implementation and show how having

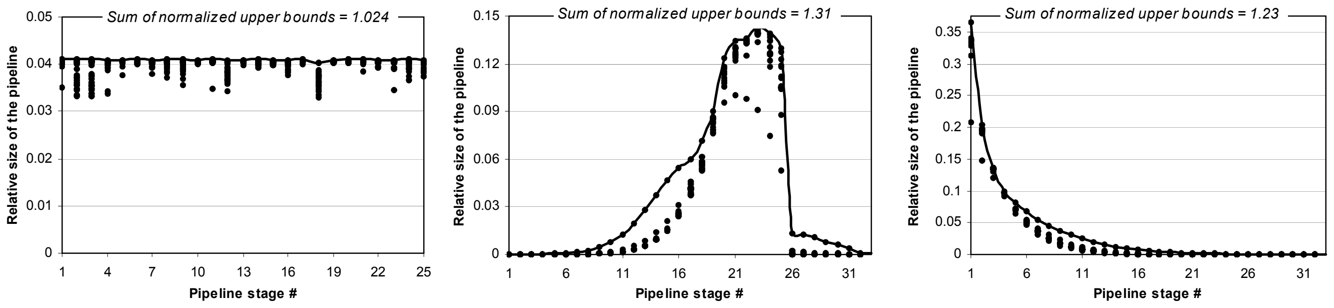


Figure 12: Normalized memory requirements of each pipeline stage in a binary trie a) CAMP using min-max heuristic, b) level to pipeline stage mapping, c) height to stage mapping. Leaf pushing was not done in these experiments.

more pipeline stages than trie levels in a CAMP system affects the node distribution. We conclude with a brief analysis of power dissipation and die area. Our study focuses mainly on practical databases: we therefore begin with a brief discussion of the IPv4 address allocation process and trends in BGP routing table growth.

5.1. BGP Routing Tables and Trends

BGP tables have grown steadily over past two decades from less than 5000 entries in the early 1990s to nearly 75,000 entries in 2000 to up to 135,000 entries today. The trends in the growth are well studied in [12][13], which highlight that 16 to 24-bit long prefixes makes up the bulk of the BGP table. It has been shown that a small fraction (<1%) of prefixes are longer than 24-bits and are likely to remain so in the near future due to the address aggregation and route aggregation techniques. The use of prefix length filtering also limits the propagation of longer prefixes throughout the global BGP routing domain.

Another important trend concerns updates in BGP tables. A majority of updates are linked to network link failure and recovery which removes a set of neighboring prefixes from the trie and quickly adds them back either due to the link recovery or due to the discovery of an alternative path.

To summarize the BGP trends: *i*) the number of prefixes in BGP tables has grown nearly exponentially and is likely continue to grow; *ii*) prefixes smaller than 26-bits make the bulk of the BGP table and is likely to remain so in the near future; *iii*) route updates can concentrate in short periods of time; however, updates rarely change the shape of the trie even after extended period of time.

We now discuss the memory requirement of pipelined tries. Unless otherwise specified, the experiments reported in this section are based on a dataset consisting of more than fifty BGP tables obtained from [11] and [15], containing anywhere from 50,000 to 135,000 prefixes.

5.2. Practical Considerations

Two important issues must be addressed when designing a CAMP pipeline: *i*) choice of the number of stages and *ii*) selection of the initial stride, which divides a trie into multiple sub-tries. We postpone their discussion to subsequent sections, and concentrate on another important design aspect. For a given number of stages and initial stride, how to dimension each stage and how does it compare with a linear pipeline?

To answer these questions, we determine the memory requirement of every pipeline stage for an array of routing tables in our dataset. Thereafter, from among all these data points, we compute the maximum memory requirement of every stage. Since some tables contain fewer prefixes than others, it is likely that they will require relatively less memory at each stage and hence may not contribute to the maximum computation. Therefore, we normalize the memory requirement of a stage for a given prefix set before considering it for the maximum computation. Thus the impact of prefix set's size are eliminated but that of the prefix trends and length distribution are preserved. This gives us a first order estimate of the memory required at each pipeline stage for the today's prefix sets.

In Figure 12(a) we plot the normalized size of each stage of a CAMP pipeline for all routing tables. The initial stride is set to 8, thus all subsequent uni-bit sub-tries require 25 pipeline stages. A "*dot*" represents the size of the corresponding pipeline stage for a prefix-set. The maximum size of each pipeline stage from among all dots is shown as an envelope in solid line. In Figure 12(b) and (c), we draw similar plots for a linear pipeline using a level to stage and height to stage mapping, respectively. We then add up the maximum size of each stage, represented by the envelope. This provides us the total memory overhead of each scheme (printed in the same plots). It can be noted that CAMP has a total memory overhead of 2.4% as compared to 23% in height to stage mapping and 31% in level to stage mapping. Thus, not only does CAMP allow a more balanced distribution of nodes to stages (highlighted by Figure 12), but it also reduces the total memory.

5.3. Initial stride and number of sub-tries

The selection of the initial stride determines the number of sub-tries a trie will be split into. Specifically, an initial stride of *k* will lead to up to 2^k sub-tries. A large number of sub-tries generally lead to more balanced pipeline stages. On the same dataset used in the previous analysis, we verified that the 2.4% memory overhead reported for an initial stride of 8 reduces to 0.02% and 0.01% for initial strides of 12 and 16, respectively. Larger initial strides, however, come at a cost. The direct indexed array which processes the initial *k*-bits and selects a sub-trie has 2^k entries. Therefore, an initial stride of 12, which requires a 4k entries table, is preferable over 16, which requires 64k entries table.

5.4. Incremental Updates

From the previous discussion it is clear that CAMP mapping algorithm leads to uniform pipeline utilization once an appropriate initial stride is chosen. We now study the effect of updates, which may disturb a balanced system. The goal of the discussion is two-fold: first, we seek to evaluate the degree of imbalance that can be introduced by incremental updates in extreme scenarios; second, we seek to determine a bound on the extra memory needed to compensate for the imbalance.

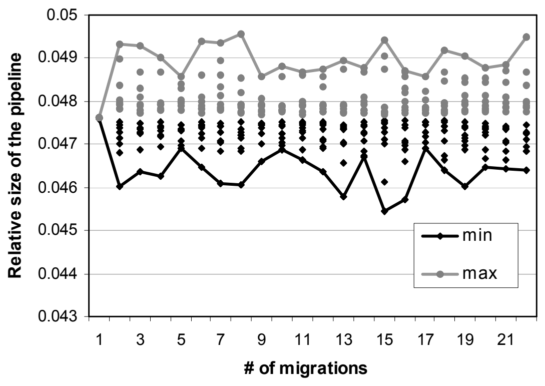


Figure 13: Successive migrations between a set of 22 distinct BGP tables. The upper and lower bound of the relative pipeline size are highlighted.

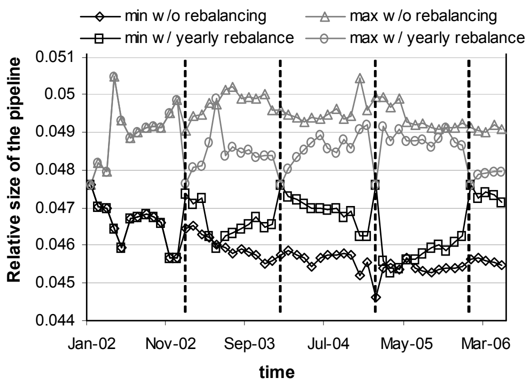


Figure 14: Effect of incremental updates over time; two scenarios are represented: once without and one with yearly rebalancing.

An extreme (and unlikely) scenario is created by considering a subset of BGP tables from [11], each containing nearly 105,000 prefixes, and simulating a sequence of migrations from one table to the other. System begins in a balanced state (an initial stride of 12 is assumed) and each successive migration incrementally removes all prefixes belonging to the previous table and adds the ones present in the new table. During migration, the node to stage assignment of already existing sub-tries is preserved (and extended to the newly added nodes of the same sub-tries), while the roots of the newly added sub-tries are assigned a random stage. The results of these experiments are reported in Figure 13, where several distinct simulations have been run starting from a different routing table. The sizes of the smallest and the largest pipeline stage, normalized with respect to the total table size, are shown by a sequence of min-max data points in black-gray shade. The upper and lower envelope of all max-min data points is drawn in the same plot. It is clear that, even in this extreme case, the imbalance leads to only 4% increase in the occupancy of the largest stage.

A more realistic scenario has been created by considering monthly snapshots of the rrc00 routing table over time, from 2002 till 2006 [15], during which, the table grew from 90126 prefixes to 135520 prefixes. Two cases are considered: in the first one, a balanced node to stage assignment is performed at the beginning (2002) and incremental updates are carried out until 2006 without any intermediate rebalancing. In the second case, the system is rebalanced, once every year, with a new (and balanced) node to stage assignment. Figure 14, reporting the result of this experiment, can be read as Figure 13 with the difference that the x-axis now reports the timestamp of each table snapshot. Without rebalancing, the maximum variation in the occupancy of the largest memory stage is 6%, while with rebalancing it is 4%. Note that such variation decreases every year; in particular, it is limited to less than 1% after 2006. In fact, as the routing table grows and the trie becomes relatively denser, it becomes more difficult to disturb a balanced system.

We conclude that, even in extreme update scenarios, the occupancy of a CAMP pipeline stage can increase only marginally. Hence, small memory over-provisioning should be adequate. Although there are effective methods to rebalance a CAMP system in face of real-time incremental updates, the limited amount of imbalance and the infrequent need of rebalancing renders them not worthwhile.

5.5. Multi-bit tries

Until now, we have only considered a uni-bit trie lookup. We now extend our evaluation to multi-bit tries where tree-bit maps are used to represent multi-bit nodes. The first design issue is to determine a stride which minimizes the total memory. We accomplish this experimentally by applying different strides on our datasets and measuring the total memory. The results are reported in Figure 15. It is obvious that strides of 3, 4 and 5 are the most appropriate choices.

When selecting the stride from among the three choices above, CAMP has relatively higher flexibility than a linear pipeline. In case of a linear pipeline, a higher stride will reduce the number of memory stages, which may increase the size of each stage. A linear pipelined trie will therefore generally prefer conservative strides (e.g.: 3) so as to keep the bottleneck stage smaller, even though this may lead to non optimal total memory. CAMP, on the other hand, may choose relatively higher stride due to the fact that pipeline stages are uniformly sized and no single stage is the bottleneck. Additionally, as we will show in the next subsection, CAMP exhibits more flexibility in selecting the number of pipeline stages which can reduce their size independent of the adopted stride.

5.6. Number of pipeline stages

A key property of CAMP is that the number of pipeline stages can be different from the number of trie levels. It enables a trie data-structure to be pipelined to many more stages. Besides reducing the size of each pipeline stage (thus enabling them to run faster), more stages also improves the overall LPC, leading to a higher throughput. Despite these obvious benefits, a large number of stages may lead to a relatively less balanced distribution of nodes across different stages.

We experimentally quantify the impact of number of stages on the node distribution. We keep an initial stride at 9 and the stride of each sub-trie is 5. In Figure 16, we vary the number of pipeline stages from 6 thru 30 and plot the excessive nodes allocated to the largest pipeline stage (percentage of the average number of nodes in a stage). Clearly, more stages result in higher imbalance as the largest stage is relatively more occupied. However, note that, even for 30 pipeline stages, the largest stage is less than 1% bigger than the average stage. Therefore, we can conclude that the overall impact of higher number of pipeline stages on the node distribution is very nominal.

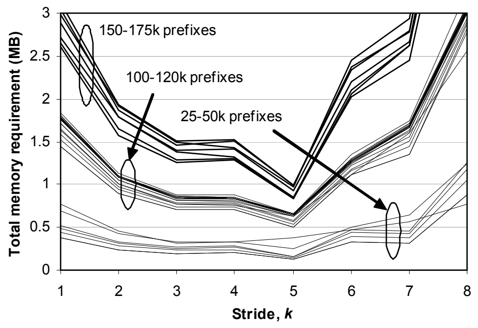


Figure 15: Total memory requirements of a tree-bit mapped multi-bit trie with different stride values.

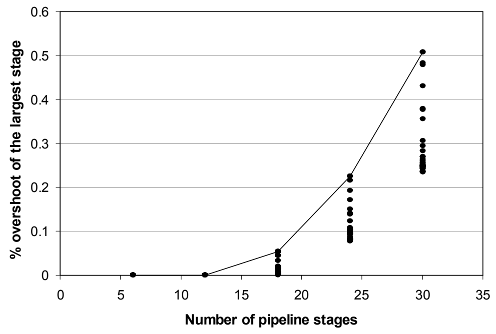


Figure 16: Percentage overshoot of size of the largest pipeline stage from the average pipeline stage size.

5.7. Power Dissipations and Area Estimates

We now characterize the power dissipation and die area of a CAMP system. The analysis is carried out assuming a 0.09μm CMOS process and using CACTI3.2 [16]. The evaluation considers large synthetic prefix sets, besides our original dataset. We allocate an additional 25% memory to account for pathological conditions which may arise in the future. Wherever there is choice, we pick optimum memory configuration (number of banks and the clock frequency), which meets a given throughput objective. Finally, throughout the experiments, we use a tree-bit mapped multi-bit trie of stride 5.

In Figure 17(a), we plot the power dissipation of the system for different link rates. As shown, the power dissipation for 1 million prefixes is 7 Watts when a 5-stage pipeline is used, and drops down to 3.4 Watts when a 10-stage pipeline is used. This can be explained as follow. The size of each stage is halved (from 1.6 MB to 0.8 MB) when doubling the number of stages. A single bank memory of these sizes has an access time of 4.2 ns and 2.2 ns, respectively. Therefore, achieving a 160 Gbps throughput requires a 4-bank and 2-bank memory, respectively, the former consuming 33% more energy per clock cycle. A smaller number of stages also lead to a lower LPC, thus requiring clocking the memory at higher rates.

Another interesting observation is that 1 million prefixes on a 10 stage pipeline dissipates less power than 600k prefixes on a 5 stage pipeline. In order to obtain an optimum number of pipeline stages which minimizes the power dissipation, we measure the power dissipation while varying the number of stages. In Figure 17(c) we plot the power dissipation of a 1 million and 600k prefix CAMP system providing 160 Gbps throughput. Power dissipation clearly drops as we increase the number of stages, however, beyond 15 stages, the reductions are nominal. It happens because every stage is 0.5 MB in a 15-stage pipeline, and a single banked memory of this size has access time of less than 2 ns, sufficient to provide 160 Gbps. Beyond 25 stages, the increase in power due to the more individual components exceeds the reductions due to higher LPC. Hence, the overall power dissipation begins to increase.

The cost and yield of an ASIC vastly depends on the die size, therefore, we also quantify the die size of a CAMP system. In Figure 17(b), we plot the area in cm^2, required by a 5- and 10-stage CAMP for different link rates. As expected, larger number of prefixes results in proportionally larger area. We report the area requirements as a function of the number of pipeline stages in Figure 17(c), which suggests that as the number of stages increases, area first decreases and then increases after certain point. However, area sensitivity is small, because area is mostly independent of the LPC and clock frequency and only loosely coupled to the number of banks.

6. WORST-CASE PREFIX SETS

We have till now considered only practical routing tables. For completeness, we briefly discuss a worst-case scenario for CAMP and describe a technique to handle it. Due to lack of space, we focus only on the main ideas.

What distinguishes CAMP from static pipelined tries is its ability to use uniformly occupied memories. Therefore, the worse-case for CAMP arises when it is not trivial to split a trie into multiple sub-tries and uniformly map them to different stages. Recall that we divided a single trie into up to 2^k sub-tries by separately considering the initial k-bits of the address. As shown in Figure 18(a), any trie which begins with a long skinny section is difficult to be split. If we attempt to split such a trie, it will require a large initial stride,

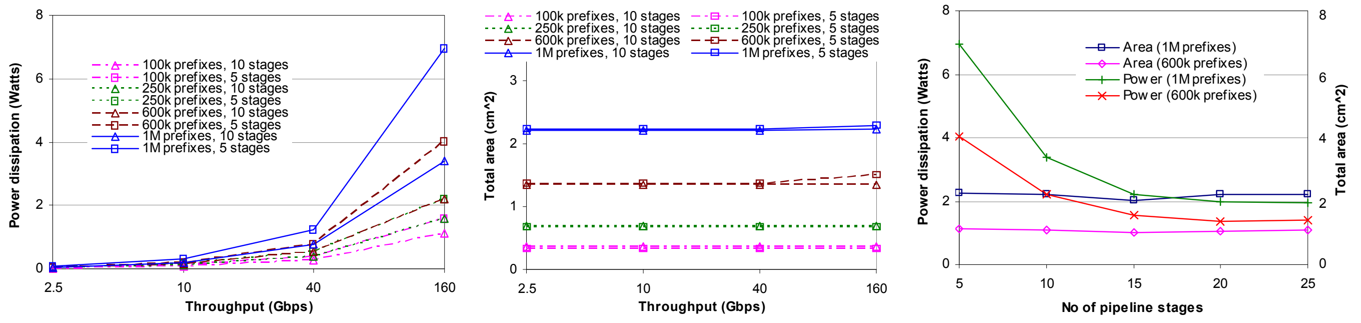


Figure 17: Power consumption and area estimates of different CAMP configurations.

which can make the direct index table ($2^{initial\ stride}$ entries) prohibitively large.

In order to handle these worst-case conditions, we propose an *adaptive CAMP*, which allows a trie to be split into a parent sub-trie and multiple child sub-tries. This way, not only can we directly control the number and size of sub-tries generated, but we can also ensure that the resulting sub-tries are equal in size. The process begins with assigning rank (total number of its descendents) to each node. We then distinguish all nodes with rank equal to the size of the sub-tries we want to generate. These nodes form a sub-trie of which they are the root. The procedure is illustrated on the above trie in Figure 18(b), where root node of each resulting sub-trie is shown.

This procedure can directly lead to a more balanced CAMP. However, a direct index table can no longer be employed for the parent sub-trie. To address this problem, we simply treat this sub-trie as other sub-tries and map it to the same pipeline. Thus, a request is first dispatched into the pipeline to parse the parent sub-trie, and then another request is dispatched to parse one of the child sub-trie. This may reduce the LPC, however allocating more pipeline stages can easily mitigate this issue.

7. CONCLUDING REMARKS

To summarize, we have introduced CAMP, which is an extension of a recently proposed novel IP lookup architecture based on a multi-point access circular pipeline of traditional memories. CAMP enables near optimal and uniform memory utilization in face a large number of updates. A key feature of the architecture is that the number of stages in the pipeline is decoupled from the number of levels in the trie. Hence, a large number of smaller memory stages can be employed, leading to a higher throughput at lower area and power dissipation. CAMP also ensures fast incremental updates, which has been validated on a collection of real and synthetic prefix sets.

There are many ways the CAMP architecture can be extended further. One possibility is to consider off-chip memories. Since the architecture provides a balanced usage of space and bandwidth, an array of high bandwidth off-chip memories can be employed and accessed in a pipelined order. Thus, CAMP is directly applicable to a modern network processor which contains several independent memory channels. Another possible research direction is further exploration of the proposed adaptive CAMP, so that it enables even simpler incremental update, which can provide deterministic performance irrespective of the dynamically changing prefix sets.

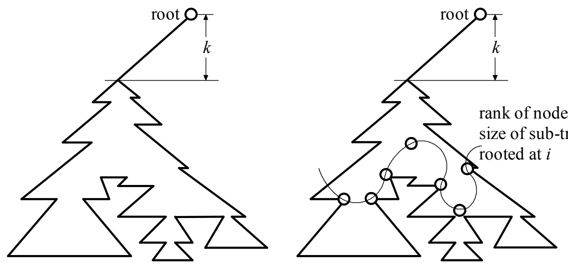


Figure 18: a) a worst-case prefix set, b) the way adaptive CAMP splits a trie into parent and child sub-tries.

References

[1] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proc. ACM SIGCOMM'97*, pp. 25-37.

[2] V. Srinivasan, and G. Varghese., "Fast Address Lookups using Controlled Prefix Expansion", in *ACM Transactions on Computer Systems*, vol. 17, no. 1, 1999, pp. 1-40.

[3] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," in *IEEE Journal on Selected Areas in Communications*, 2003.

[4] M. Degermark, A. Brodnik, S. Carlsson and S. Pink, "Small Forwarding Tables for Fast Routing Lookups", in *Proc. of ACM SIGCOMM 1997*.

[5] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software ip lookups with incremental updates", in *ACM SIGCOMM Computer Communications Review*, 34(2), 2004.

[6] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," in *Proc. ACM SIGCOMM 2003*.

[7] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines", in *Proceedings of INFOCOM 2003*, 2003

[8] J. Hasan and T.N. Vijaykumar, "Dynamic Pipelining: Making IP-Lookup Truly Scalable", in *Proc. ACM SIGCOMM 2005*, pp 205-216.

[9] A. J. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs", in *Proc. INFOCOM* 1993.

[10] Francis Zane, Girija Narlikar, and Anindya Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines", in *Proc. INFOCOM 2003.*

[11] BGP Table Data. http://bgp.potaroo.net, April 2006

[12] G. Huston, "Analyzing the Internet's BGP routing table", in *Internet Protocol Journal*, 4(1), 2001.

[13] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental Study of Internet Stability and Wide-Area Backbone Failures", Proc. 29th Annual International Symp. on Fault-Tolerant Computing, Madison, WI, June 1999.

[14] C. Labovitz, G. R. Malan, and F. Jahanian, "Origins of Internet Routing Instability," In Proc. of Infocom '99, New York, NY, March 1999.

[15] Routing Information Service. http://www.ris.ripe.net

[16] CACTI.http://www.research.compact.com/wrl/people/jouppi/CACTI.html

[17] Haoyu Song, Jonathan Turner and John Lockwood, "Shape Shifting Tries for Faster IP Route Lookup", in *ICNP 2005*, 11/2005.

[18] S. Suri, G. Varghese, and P. Warkhede, "Multiway range trees: Scalable IP lookup with fast updates", GLOBECOM 2001.

[19] S. Nilsson and G. Karlsson, "Fast Address Lookup for Internet Routers," in Proc. of IEEE Conf. on BroadBand Communications Tech., 1998.

[20] Timothy Sherwood, George Varghese and Brad Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," In Proceedings of the 30th Annual ISCA, pages 288-299, 2003.

[21] Florin Baboescu, Dean M. Tullsen, Grigore Rosu, Sumeet Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," in ISCA 2005.

[22] R. Graham, F. Graham, and G. Varghese, "Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines", in the Proceedings of SPAA'04, Barcelona, Spain, (2004), pp. 103—111.