In Proceedings of the 32nd International Symposium on Computer Architecture, June, 2005

A Tree Based Router Search Engine Architecture With Single Port Memories

Florin Baboescu[†], Dean M. Tullsen[†], Grigore Rosu[‡], Sumeet Singh[†]

[†]Department of Computer Science and Engineering
University of California, San Diego
{baboescu, tullsen, susingh}@cs.ucsd.edu

[‡]Department of Computer Science
University of Illinois, Urbana-Champaign
grosu@cs.uiuc.edu

Abstract

Pipelined forwarding engines are used in core routers to meet speed demands. Tree-based searches are pipelined across a number of stages to achieve high throughput, but this results in unevenly distributed memory. To address this imbalance, conventional approaches use either complex dynamic memory allocation schemes or over-provision each of the pipeline stages. This paper describes the microarchitecture of a novel network search processor which provides both high execution throughput and balanced memory distribution by dividing the tree into subtrees and allocating each subtree separately, allowing searches to begin at any pipeline stage.

The architecture is validated by implementing and simulating state of the art solutions for IPv4 lookup, VPN forwarding and packet classification. The new pipeline scheme and memory allocator can provide searches with a memory allocation efficiency that is within 1% of non-pipelined schemes.

1 Introduction

The rapid growth of the Internet has brought great challenges in deploying high-speed networks. One particular challenge is the need to provide high packet forwarding rates through the router. This paper presents a novel architecture for a network processor which features a complexity-effective organization of pipelined computational cores. This architecture allows the problem to be partitioned in a way that balances both computation and memory, allowing the entire architecture to compute at high rates.

Network search engines capable of providing IP lookup, VPN forwarding, or packet classification are a major component of every router. With the increase in link speeds, increase in advertised IP prefixes, and deployment of new network services, the demands placed on these network search engines are increasingly causing them to become a potential bottleneck for the router. This paper considers the architecture of programmable network search engines. Other, more expensive, custom solutions are discussed in Section 5.

Memory access times and costs become dominant factors in a high-speed network processor. While network processors have received considerable attention in the commercial [1] and in the research [23, 6] communities, most of the commercial implementation have used a collection of multithreaded CPU cores. This allows a single memory to hold the entire database (thus no memory balance or fragmentation issues), but do not scale to the bandwidths required for future processors.

Most algorithmic-based solutions for network searches can be regarded as some form of tree traversal, where the search starts at the root node, traverses various levels of the tree, and typically ends at a leaf node. This computation is easily pipelined onto multiple computational elements, allowing different levels of the tree to be partitioned onto private memories associated with the processing elements – no data sharing is required, except for the state that follows the thread of computation through the pipeline. Unfortunately, this arrangement results in highly unbalanced memories, to accommodate databases (trees) that are typically unbalanced in unpredictable ways. For example, binary tries on typical IP prefix tables are highly unbalanced. As a result, despite a wide variety of academic and commercial solutions, only a few solutions do well in terms of performance, efficiency, and cost, and none of them provide a general solution for all three types of searches.

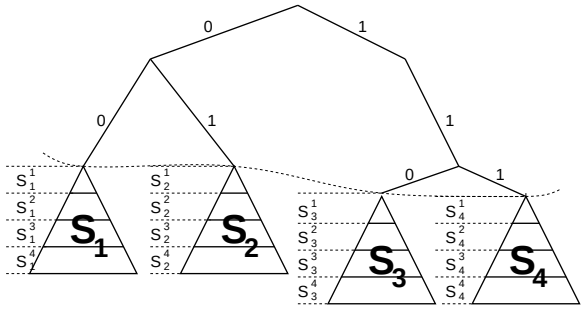

Figure 1. An example of a basic tree based search structure. The tree is split into four subtrees $S_1, \ldots S_4$. Each subtree has up to 4 levels. We call S_i^j the level j into the subtree S_i.

Basu et al. [8] identify memory balance as a critical issue in the design of IP lookup engines. Their technique to reduce memory imbalance is to design the tree structure to minimize the stage that has the largest memory. Even with their new algorithm, the memory allocated to one stage varies from nearly 0 to 150Kbytes for various IP tables (of sizes 100,000 to 130,000 prefixes). The worst case bound for a million prefixes is 11 Mbytes per stage (88 Mbytes across all eight stages). This more than doubles the total amount of memory that is used in a non-pipelined implementation.

To address this imbalance, conventional approaches use ei-

ther complex dynamic memory allocation schemes (dramatically increasing the hardware complexity) or over-provision each of the pipeline stages (resulting in memory waste). The use of large, poorly utilized memory modules results in high system cost and high memory latencies, which can have a dramatic effect on the speed of each stage of the pipelined computation, and thus on the throughput of the entire architecture.

By contrast, this paper describes a novel memory allocation algorithm that allows searches to logically start from *any* stage in the pipeline. This eliminates the memory imbalance, because any subtree of the search structure can be allocated across the pipeline starting at any stage. This degree of freedom greatly reduces memory imbalance compared to prior schemes and enables smaller, cheaper, faster processing elements. Thus, while previous schemes had virtually unbounded imbalance, we present one scheme that is within 1% of perfect balance, even for highly imbalanced trees.

The rest of this paper is organized as follows. Section 2 introduces our solution for solving the memory allocation problem for each pipeline stage without generating access conflicts. We introduce a linear algorithm for subtree allocation which we show can allocate the subtrees with at most 1% memory waste; however, as shown in Appendix B, the problem of *optimally* allocating subtrees on a pipeline ring is NP-complete. Section 3 provides an overview of network search applications: IP lookup, VPN forwarding, and packet classification. In Section 4 we evaluate our solution on all three application types introduced in Section 3, using both real life and synthetically generated routing tables and classifiers. Section 5 presents related work in the pipeline design of network processors as well as in network search applications. Section 6 concludes.

2 Towards a Balanced Memory Distribution in a Pipelined Search Architecture

Memory distribution per pipeline stage varies widely in the case of a conventional tree based search implementation of IP lookups and VPN searches (as shown by Basu et. al [8], and by our results in Figure 3). Further, the results show no correlation between the position of a particular pipeline stage and the amount of memory that needs to be allocated to that stage.

Prior pipelined network search algorithms require all searches to start from the first pipeline stage, going next to the second, and so on. Instead, we introduce our first contribution: an additional degree of freedom for the search operation. We allow the search to start at *any* stage in the pipeline. For every search, the starting position is picked using a hash function based on information in the packet header. For IP lookups the hash function is made up of a set of variable length IP prefixes. For decision-tree based packet classification, the hash function may use some of the most significant bits in two or three different fields of the packet header.

Figure 1 shows a tree based search structure. To keep the explanation simple, let us assume that the tree has four subtrees, called $S_1 \ldots S_4$. Furthermore, the depth of each subtree is four levels. We assume that this search structure is implemented on a four stage pipeline. The stages of the pipeline are called $P_1 \ldots P_4$. The first level of the subtree S_1, called S_1^1, is stored and processed by the pipeline stage P_1. The second level, S_1^2, is stored and processed by the pipeline stage P_2, and so on. The second subtree is processed starting with stage P_2, S_2^1 on P_2, S_2^2 on P_3, S_2^3 on P_4 and S_2^4 on P_1, respectively.

Similarly, the third subtree S_3 starts on pipeline stage P_3, while the fourth subtree S_4 starts on pipeline stage P_4. This allocation scheme tries to balance the load on each of the pipeline stages. By doing so, the pipeline allocates nearly equal amounts of memory to each stage, by virtually allocating a "subtree" in each of the stages. E.g., the first pipeline stage stores the first level in the first subtree (S_1^1), the second level in the fourth subtree (S_4^2), the third level in the third subtree (S_3^3), and the fourth level in the second subtree (S_2^4).

In practice, we relax these two simplifications in this illustration. We allow more subtrees than pipeline stages (processing elements), thus implying multiple subtrees may have the same start node. We also allow the maximum depth of each subtree to be less than or equal to the number of pipeline stages.

However, introducing this new degree of freedom that allows search tasks to start execution from any pipeline stage impacts the throughput of the system. This is because of potential conflicts between the new tasks and the ones that are in execution. In theory, the number of conflicts can be unbounded. However, next we will present an alternative to the conventional pipelined organization that eliminates all conflicts.

2.1 Our Solution to Guarantee Pipeline Throughput

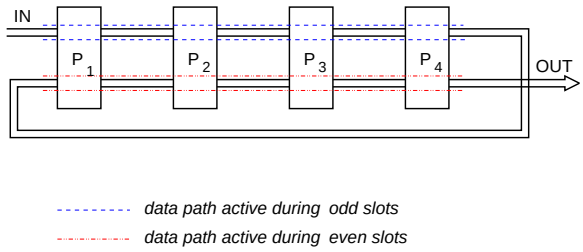

Figure 2. A random ring pipeline architecture with two data paths: first path is active during the odd clock cycles, used during the first traversal of the pipeline; second path is active during the even cycles to allow a second traversal of the pipeline.

We need to deal with two problems that create conflicts: (1) since levels are assigned to our pipelined processing elements in a circular fashion, most threads must wrap around to the beginning of the pipeline to complete execution; (2) computation for a new task can start at any processor.

We want to guarantee that for any stream of tasks, in each interval of time u the tasks that are already present in the pipeline progress to the next stage while ensuring that the next incoming task can also be accommodated.

Our solution, which represents the second contribution of this paper, is shown in Figure 2. It modifies the regular pipeline structure and behavior as follows.

Each pipeline stage works at a frequency $f = 2 * F$, where F is the maximum throughput of the input. All tasks traverse the pipeline twice and are inserted at the first pipeline stage, irrespective of their starting stage (for execution) in the pipeline.

Each pipeline stage accommodates two data paths (virtual data paths – they can share the same physical wires). The first data path (represented by the top lines) is active during the odd clock cycles and it is used for a first traversal of the pipeline. During this traversal a task T_i traverses the pipeline until its starting stage i and continues the execution until the last stage of the pipeline. The execution of a task always starts on the first traversal through its start processor. The second data path is traversed during even cycles and allows the task to continue its execution on the pipeline stages that are left. Once a task finishes executing, its results are propagated to the output through the final stage.

The number of stages in the pipeline must be at least equal to the maximum number of stages that are required for the execution of any task.

For example, consider the four stage pipeline in Figure 2. A task that must start executing in pipeline stage 3 is inserted in pipeline stage 1. It traverses the pipeline only in the odd cycles until it reaches stage 3 where it starts executing. Its results are forwarded to pipeline stage 4 also during an odd cycle. However, the results of the execution on stage 4 are moved forward to pipeline stage 1 for execution during the next even cycle. The task finishes its execution on pipeline stage 2. The final results are moved to the output via pipeline stages 3 and 4 during even cycles.

Our solution guarantees the following features: **1)** an output rate equal to the input rate, **2)** all the tasks exit in order, and **3)** all the tasks have a constant latency through the pipeline equal to $N * \frac{1}{F}$ where N is the total number of pipeline stages.

In summary, we provide a new pipeline architecture that allows the injection and removal of tasks each from a single processor, while communication between processors occurs only between neighbors in a linear ordering of the processors; this eliminates (1) the need for a scheduler for both input and output of the task and (2) the communication complexity. We also address the memory imbalance between the pipeline stages by allowing the execution of the tasks to start at any position in the pipeline. Section 4 evaluates how our new allocation scheme reduces the memory imbalance in the implementation of different network search applications.

This architecture requires that the time per processing step be half that of a more conventional pipelined configuration to maintain the same throughput. We show in Section 4 that the reduction in memory size easily allows those gains.

2.2 Selecting the Subtrees

To apply this new allocation scheme we need to first partition the tree into subtrees. Ideally, the subtrees to be allocated should have relatively equal size (approximately the same number of nodes).

We provide an iterative algorithm that takes as input the original trie [1] and at each step identifies one subtrie that contains a number of nodes which is the closest to a desired value (threshold). The subtrie is entirely eliminated from the original trie and saved into a list together with the prefix associated with its root node. The algorithm continues until the number of nodes left in the trie is less than the threshold.

The result of the algorithm is a list of tuples. Each tuple is made up of the root node of a subtrie together with the longest matching prefix of this node.

2.3 The Allocation of the Subtrees

The algorithm above splits the original tree into subtrees of relatively equal size. The next step is to allocate these subtrees to the circular pipeline such that the amount of memory used by each of the pipeline stages is relatively equal. As shown in Appendix B, the problem of finding an *optimal allocation* of each of the subtrees on the pipeline stages is intractable. Therefore, the best one can do is to develop heuristics for "good enough" subtree allocation on pipeline stages.

We propose a simple *linear time* solution for the allocation problem. In Section 4 we experimentally show that our solution leads to a very small memory waste, within 1% of the total memory size. Our heuristic considers one subtree at a time, randomly picked from the set of subtrees identified using the algorithm described in the previous section, and allocates it such that the level in the new subtree that requires the minimum amount of memory corresponds to the pipeline stage that already uses the largest amount of memory.

3 Network Search Applications

We evaluate our new pipeline architecture and task allocation algorithm using state of the art solutions for different types of network searches that are typically done in a router: IP lookups, VPN forwarding and packet classification. The features of these searches are summarized in Table 1.

In Appendix A we give details of each of the IP lookup algorithms that we implement and evaluate in Section 4. The VPN forwarding algorithms use the same data structures as in the case of IP lookup. In essence a router that provides VPN forwarding must execute two IP lookup operations for each search, as is given in RFC2547 [20]. It first executes an IP lookup based on the source IP field. The result of this determines the routing table that is used for the second IP lookup based on the destination IP field.

In packet classification each packet is matched against a prioritized set of rules made up using two or more fields (e.g. IP source and destination fields, port fields, etc.). A packet can be matched by several rules. The search determines the highest priority rule that matches each packet.

Decision-tree based packet classification algorithms [28, 15, 26] appear to be the most promising category of algorithmic solutions to the packet classification problem. We implement HyperCuts [26], a recent decision tree based packet classification algorithm introduced by Singh, et al. The scheme is based on a pre-computed decision tree which is traversed for each packet that needs to be classified. The computation at each stage in the tree uses several bits in the packet header as an index into an array of child pointers to identify the next child node to be traversed.

[1] A trie is a binary prefix tree.

Application	*Number of Entries*	*Number of fields*	*Type of Matches*
IP Lookup	$> 150K$	1	LPM
VPN Forwarding	$> 500K$	2	$EM + LPM$
Packet Classification	$\sim 10,000$	> 2	$EM + LPM + RM$

Table 1. Network Search Applications. LPM stands for "longest prefix match", EM for "exact match",a nd RM for "range match".

4 Evaluation

In this section we evaluate our Ring Pipeline architecture using the network search algorithms described in the previous section. Our architecture uses private single port memories for each of the pipeline stages. This contributes to an increase in the amount of memory needed due to increased fragmentation. We seek to balance memory for two reasons, to minimize cost (memory waste) and to maximize performance (minimize the access time of the largest memory). Thus, this section focuses on the following two critical questions: **1)** What is the overall waste in the memory space due to our new model? **2)** What is the maximum throughput and expected latency our scheme can provide? We start with the latter question.

4.1 Search Latency and Throughput

Each pipeline stage requires a computation phase and a memory access phase. Although the memory is uniport, our design allows two words located at a small distance one from another to be read in one memory access, as in [4]. The memory access time is similar to the access time of a regular uniport memory. We first investigate the relationship between per-stage memory allocation and the memory access time. Table 2 shows that the memory access time increases significantly with the size of memory. When our balanced allocation algorithm is applied, we find that all searches analyzed in this research, except one, can be implemented with memory latency less than $2ns$. The one exception corresponds to a VPN forwarding application that contains a large number of small destination IP routing tables. Even in this case the memory access time is less than $3ns$.

Memory Size(Kbits)	*Area(cm^2)*	*Access Time(ns)*
128	0.005	0.565
256	0.009	0.651
512	0.020	0.828
1024	0.037	0.919
2048	0.069	1.242
4096	0.134	1.520
8192	0.275	2.487

Table 2. The memory access time and area estimates for different sizes of on-chip SRAM using $0.09\mu m$ technology. The estimates are obtained using the *memory generator* application CACTI [24].

In order to determine both the search latency as well as the throughput of the searches using our architecture, we synthesized in Verilog the computational logic for each pipeline stage for both Eatherton's IP lookup algorithm and the HyperCuts algorithm using $0.13\mu m$ technology. The longest path delay in the computation of the next node address in both algorithms is smaller than $1\ ns$. This combines with a $2\ ns$ memory access time to a allow a $3\ ns$ execution delay per pipeline stage. The size of all the computation logic for all 8 stages is smaller than $0.125mm^2$.

Given the architecture of Section 2, a pipeline running at $330MHz$ (3 ns per stage) achieves a search throughput of $6\,ns$ per packet. This value is adequate for OC-768 (40Gbps) links that require a throughput of $8\ ns$ per packet for a minimum size (40 bytes) packet.

All the searches through the pipeline have a latency that is constant and is double the latency of a one way pipeline traversal. The overall latency of a search operation using the Eatherton algorithm [11] for the IPv4 lookup is $8 * 2 * 3\ ns = 48\ ns$ assuming an eight-stage pipeline with $3ns$ per stage. We measured the mean packet latency for different loads on a CISCO GSR router. In our evaluation the smallest mean packet latency was approximately $50\mu s$. Thus our search latency is less than 0.1% of the total mean packet latency. Consequently we conclude that the search latency of our solution has virtually no impact on the overall packet latency.

4.2 Memory Distribution per Pipeline Stage

We next evaluate the efficiency of our pipeline scheme to equally distribute memory across pipeline stages. We do this by simulating the behavior of our architecture for all three types of applications: IP lookups, VPN based lookups, and packet classification. We evaluate these models using both real life routing tables and classifiers, as well as synthetically generated ones that allow us to simulate large configurations. In the figures that follow all the memory values are expressed in $Kbits$.

4.2.1 Evaluation of IP Lookup

We first evaluate our pipeline architecture by a software simulation of the memory requirements for Eatherton's IP lookup algorithm [11]. The real life routing tables were extracted using instances of the BGP routing tables available at RIPE [21] and RIR [16] on Sept. 22, 2003 and parsed using the ($route\_btoa$) software available at [17]. We extracted the routing tables associated with ATT (AS7018), Sprint (AS1239), Level 3 Communications (AS3356) and France Telecom (AS5511). Because the results are very similar, we only display the results for ATT. To test the scalability of the algorithm we synthetically generate tables using two different models of routing table growth: one developed by the Network Processing Forum (NPF) [2], and one developed by Narayan, et al. [18].

The graph on the left in Figure 3 shows the results of using the Eatherton algorithm with a regular pipeline in which the search starts with the first pipeline stage, continues with the second, and so on. These results motivate our scheme by showing that one cannot identify a clear pattern of memory size allocation per pipeline stages.

Table 3 and Figure 3 show that our pipeline scheme has a double benefit. It eliminates the need for dynamic memory allocation per pipeline stage and it provides a better throughput. For example, an IP prefix table with about $500,000$ entries requires almost $11Mbits$ of memory for one stage (the sixth pipeline stage). As a result the memory access time increases to about $3.5ns$. In comparison, our new pipeline scheme has a maximum of $2.9Mbits$ of memory allocated per stage. As a result the memory access time is reduced to $1.4ns$.

In all these simulations we use a relatively naive split of the original search trie into 32 subtries (using the first 5 bits in the IP address field). These subtries are allocated to the pipeline stages starting from various positions. In this case the total memory across the pipeline stages is within 30% of the ideal memory allocation space (Table 3). Note that in the case of a conventional pipeline with statically allocated memory the total amount of memory to be used increases 206% over the non-pipelined implementation.

IP Table	*Total*	*BPW*	*CPW*
ATT	$7,727K$	30%	206%
A100K	$4,550K$	6.25%	206%
A200K	$10,674K$	15%	206%
A300K	$13,585K$	8.8%	206%
A400K	$18,440K$	4.9%	206%
A500K	$23,226K$	9%	206%
NPF	$23,583K$	10.4%	202%

Table 3. Eatherton Algorithm on a random access pipeline model - Total memory utilization and the percentage of wasted memory if each of the pipeline stages has allocated the maximum amount of memory that is required by the pipeline stages. The third column shows our balanced pipeline waste (BPW) while the fourth column shows the memory waste in a conventional pipeline (CPW).

Reducing the waste due to over-provisioning: Although our results above show that the total memory across the pipeline stages is within 30% of the ideal memory allocation space, we would like to provide even tighter bounds on the amount of memory that is wasted due to over-provisioning.

Our allocation algorithm assumes the trie is made up of a number of relatively equal subtries. Finding the perfect combination for allocating each of the subtries on the pipeline stages has an exponential complexity as we show in Appendix B. Instead, we propose a much simpler linear solution in which at each step one subtrie is considered for allocation. The subtrie is allocated such that the level in the subtrie that requires the minimum amount of memory corresponds to the pipeline stage that currently uses the largest amount of memory.

To reduce the degree of waste, we find it is sufficient to increase the number of subtrees, allowing finer-grain placement into memory. Thus, there are two question to be asked: **1)** how to split the trie into relatively equal sized subtries and **2)** what is a sufficient number of subtries such that the amount of waste due to over-provisioning is less than, for example, 1%.

We split the original trie into subtries of relatively equal size using the algorithm described in Section 2.2. We determine the minimum number of subtries that are required to achieve an overall waste due to over-provisioning that is less than 1% through a series of evaluations using both real life routing tables as well as synthetically generated ones. In the case of a balanced trie this number is small and it is equal to the depth of the trie. This number increases when the trie shape becomes more irregular. The multi-bit trie, with strides of size 4 that is used in the Eatherton algorithm has a more regular structure than a regular unibit trie. Therefore, for this experiment, we use the unibit trie search structure, which we expect to have the largest degree of irregularity. Our results shown in Table 4 can be directly extended to the equivalent multi-bit tries. We use a 24-stage pipeline to accommodate the larger depth of the unibit trie. The third column represents the number of subtries that we create. The subtries are distributed among the 24 stages of the pipeline. The maximum number of nodes allocated for a pipeline stage is given in the 4th column. The 5th and 6th columns represent the average and maximum percentage of memory wasted due to over-provisioning. This over-provisioning is a result of allocating for each pipeline stage the amount of memory needed by the largest stage.

The results show that, in the worst case, the original trie needs to be split into $4,000$ subtries to reduce the overall waste to below 1%.

Update Operations. This analysis assumes a static database, but balance will be impacted over time by update operations. We next consider the effect of these update operations on the overall memory balance per pipeline stage. In our evaluation we use the same worst-case uni-bit trie data structures.

We consider a routing table associated with AS9177 (NEXTRANET, Switzerland), collected by RIPE rrc00 [21]. The original routing table is collected at the beginning of Aug. 9th, 2003. Updates are recorded through the end of Aug 15th. at 00:00 UTC.

The routing table is represented using a trie which is split into $4,102$ subtries that were allocated to a 24 stage pipeline using our algorithm. During the update process 227 new subtries were created. Each subtrie was associated with a new branch in the trie. Each new subtrie is inserted into the pipeline in such a way as to try to avoid having any memory allocation in the pipeline stage with the largest number of entries already allocated. Our results show that at any moment in time the maximum "waste" per pipeline stage remains smaller than 0.5%.

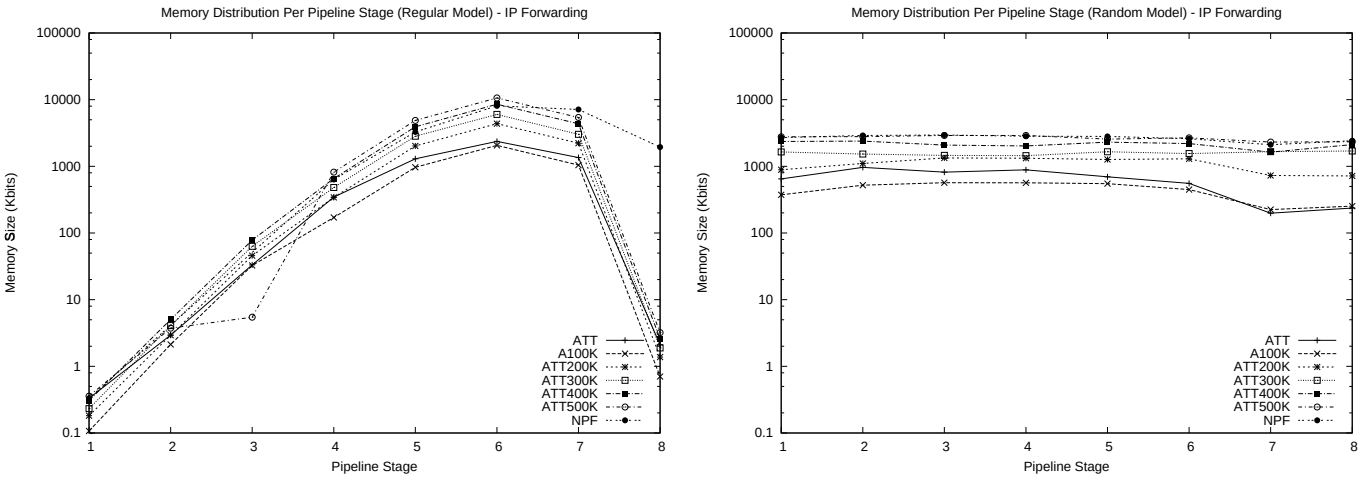

Figure 3. The memory utilization per pipeline stage using the Eatherton Algorithm [11] on a conventional pipeline architecture (left) and our balanced architecture (right). The values represent the amount of memory in bits that is used in each pipeline stage. A100K - A500K are synthetically generated routing tables using the model described by Narayan et al [18] while NPF is a 500K entries synthetic routing table generated using the model proposed by the NPF Forum.

IP Table	*No. of Prefixes*	*No. of Subtries*	*Max. No. Nodes*	*AMW*	*MMW*
ATT	$122,636$	$4,090$	$14,632$	0.23%	$0.37\ \%$
FT	$123,875$	$4,077$	$14,763$	0.19	0.27
$L3C$	$123,271$	$4,105$	$14,700$	0.25	0.39
$SPRINT$	$122,750$	$4,097$	$14,662$	0.20	0.33
$A50K$	$53,323$	$4,120$	$6,739$	0.09	0.18
$A100K$	$100,312$	$4,231$	$12,664$	0.18	0.35
$A200K$	$211,033$	$4,272$	$26,635$	0.46	0.62
$A300K$	$290,995$	$4,017$	$36,985$	0.59	0.86
$A400K$	$411,469$	$4,277$	$51,634$	0.28	0.39
$A500K$	$511,634$	$3,950$	$64,284$	0.33	0.54
NPF	$524,218$	$4,146$	$59,320$	0.35	0.61

Table 4. IP lookup using a single-bit trie search structure. The trie is split into a number of subtries, each subtrie with a number of nodes close to a given *threshold*. The number of subtries is shown in column 3. The maximum number of nodes allocated for a pipeline stage is given in column 4. Columns 5 and 6 show the average memory waste (AMW) and maximum memory waste (MMW) due to over-provisioning.

4.2.2 Evaluation of VPN Forwarding

We simulate a VPN forwarding engine using a similar search structure as in regular IP lookup. The only difference in this case is that the driver of the search engine must compute a hash function based on a tag value that is provided by the VPN application. The computed value determines the pipeline stage from which the search starts. The search structure and the search itself is implemented using the same algorithm designed by Eatherton [11].

No publicly available VPN forwarding tables exist. As a result, we do the evaluation using a set of synthetic tables that are generated using similar techniques to the ones used to generate the IP lookup tables. Our results shown in Table 5 correspond to three different cases: **1)** all the sets of tables contain about 1000 entries per set (AllSmall), **2)** all the sets of tables contain about $10,000$ entries per set (AllLarge) and **3)** the set contains a mix of small size tables and large size tables (Mixed). Each set contains about one million prefix entries.

VPN Set	*Total*	*BPW*	*CPW*
AllSmall	$55,599,512$	4.8%	82%
AllLarge	$41,440,848$	7.2%	89%
Mixed	$48,674,552$	3.2%	129%

Table 5. VPN forwarding using a random access pipeline model - Total memory utilization and the percentage of wasted memory for our balanced pipeline (BPW) and conventional pipeline (CPW).

Our results in Table 5 show that by using our new pipeline architecture for VPN applications, the total amount of wasted memory does not exceed 7.2%. It corresponds to a situation in which the set contains only a small number of relatively large VPN tables. In contrast a conventional pipeline architecture contributes to an increase in the memory of up to 129%.

4.2.3 Evaluation of the Packet Classification Algorithm (HyperCuts)

We next evaluate how a decision tree based classification scheme behaves on our new pipeline scheme using five-dimensional classifiers. We simulate the HyperCuts algorithm [26] on synthetically generated classifiers with up to 20,000 rules. The classifiers that we use are generated using the methodology described by Singh, et al. [26]. We consider classifiers with 5,000 (L5K), 10,000 (L10K) and 20,000 (L20K) rules.

Unlike in IP lookup, in tree based packet classification the largest amount of memory is allocated toward the earlier stages in the pipeline. Also the memory allocation per pipeline stage varies widely. For example, in the case of a 20,000 rules classifier the amount of memory allocated per stage varies from $480Kbits$ to $1Kbit$. Our new pipeline scheme brings down the maximum amount of memory that needs to be allocated per pipeline stage by a factor greater than two. For example in the case of a $20,000$ rules classifier, the maximum amount of memory that is allocated per pipeline stage drops to $215Kbits$ from $480Kbits$. In our simulation we used the subtrees originated in the second level nodes (the root node is at level 0).

The results in Table 6 show that a conventional pipeline architecture implementation may require more than three times the amount of memory used by the non-pipelined version. In contrast our pipelined scheme increases the memory usage by only 30%. This amount of wasted memory due to over-provisioning may be further reduced by using a larger number of subtrees.

DB	*Total*	*BPW*	*CPW*
$L5K$	$1,188K$	30%	55%
$L10K$	$2,496K$	32%	238%
$L20K$	$2,592K$	32%	238%

Table 6. HyperCuts algorithm using a random access pipeline model - Total memory utilization and the percentage of wasted memory for our balanced pipeline (BPW) and conventional pipeline (CPW).

5 Related Work

Extensive work has been done on processor ring communication [9, 10, 7, 3, 19]. Hierarchical ring buses as an alternative to the scalability and cost problems of the crossbar switches are addressed in [3, 19]. In both cases each element in their architecture is capable of controlling the insertion of data on the ring through a system of FIFOs. Coffman, et al. [10] further analyze the features of the processor-ring communication for large rings and prove boundary conditions for the task waiting times.

Packet forwarding in high speed routers has been a well studied area. There has been extensive research both in the IP lookup problem [11, 22] as well as packet classification [5, 14, 15, 28, 26, 27]. Most of this work deals with non-pipelined architectures, and the focus is to minimize the depth of the search structures.

Basu and Narilkar [8], in the context of a specific lookup algorithm that uses fixed stride multi-bit tries, show that the memory in some stages varies dramatically across databases, even in the face of their proposed algorithms to minimize the variation. For example, assuming an eight stage pipeline, their results show cases in which for two different databases the memory space to be allocated to a pipeline stage varies from almost 0 up to $150KB$ while in the case of another pipeline stage the memory space varies from about $150KB$ to up to $300KB$.

There is little work that addresses the memory limitation in the case of network search engines. The problem was introduced by Sikka, et.al [25] in the context of tries where it was left as an open problem. Basu and Narilkar [8] propose an approximate solution to the problem of trie memory allocation across stages, but they are less than successful at solving it. They propose a way to reduce the memory imbalance by minimizing the stage that has the largest memory. Baer, et al. [6] propose a cache based solution to reduce the memory capacity and the amount of memory multibanking. However, their solution can not provide deterministic throughput for any pattern of input packets and can not provide tight bounds for the worst case. Sherwood, et.al. [23] investigate the use of wide word pipelined memory that allows concurrent accesses. None of these architectures pipeline the computation across multiple processors.

Hardware based solutions based on Ternary CAMs provide an attractive solution to ASIC-based designs that implement tree based algorithmic solutions for searches. TCAMs are content addressable memories in which each bit is allowed to store a 0, 1 or a *"don't care"* value. A TCAM essentially compare each packet address with every address the search engine holds in its database, using parallel lookups on associative memory. However TCAMs have limitations: (1) large cell size (about 16 transistors per bit), (2) high power consumption $(10-15W$ at $133Msps)$, (3) very high cost per chip $(\$200 - \$300)$ and (4) can not provide a general, efficient, single chip solution for all of the algorithms our solution addresses [13].

6 Conclusion

In this paper we propose a general, pipelined, multiprocessor architecture for tree based algorithmic solutions. This architecture can be implemented using equal sized memories for each pipeline stage, limiting the need for over provisioning. This allows computation, even on highly unbalanced trees, to be partitioned into pieces that equalize both computation and memory allocation. This results in minimized memory cost and maximized packet throughput.

This solution achieves very low communication complexity because each pipeline stage communicates only with its immediate neighbors and all tasks enter and exit the pipeline through a single stage. It does not require any centralized scheduling mechanism. Our architecture also provides tight latency bounds for searches.

We evaluate our pipeline task allocation algorithm and our new multiprocessor pipeline architecture by implementing state-of-the-art tree-based network search algorithms for IP lookup, VPN forwarding, and packet classification. We demonstrate a memory allocation heuristic which can, in linear time, allocate subtrees with only 1% waste.

Our implementation can be used on high speed routers with OC-768 links that run at 40Gbps and require a throughput of $8ns$ per packet. We show that we can provide IP lookup, VPN forwarding, and packet classification at a rate of $6ns$ per packet while the overall latency is constant at $48ns$.

Acknowledgments

The authors would like to thank the reviewers for helpful feedback, Alexander Tudor for significant help with the simulation tools, and George Varghese for helpful discussions. This research was funded in part by NSF grant CCR-0311683 and by joint NSF/NASA grant CCR-0234524.

References

[1] Network processor forum. http://www.npforum.org.

[2] Network processor forum benchmark working group. http://www.npforum.org/benchmarking/index.shtml.

[3] C. Amerijckx and J. Legat. A low-power multiprocessor architecture for embedded reconfigurable systems.

[4] F. Baboescu, S. Rajgopal, N. Richardson, and L.-B. Huang. A scalable ip lookup low-power implementation for oc-768 links. Workshop for Application Specific Processors(WASP), 2004.

[5] F. Baboescu and G. Varghese. Scalable packet classification. In *Proc of ACM Sigcomm 2001*, september 2001.

[6] J.-L. Baer, D. Low, P. Crowley, and N. Sidhwaney. Memory hierarchy design for a multiprocessor look-up engine. In *IEEE PACT*, 2003.

[7] L. A. Barroso and M. Dubois. The Performance of Cache-Coherent Ring-based Multiprocessors. In *20th Annual International Symposium on Computer Architecture*, pages 268–277, May 1993.

[8] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *Proc. of Infocom*, march 2003.

[9] E. Coffman, L. Flatto, E. N. Gilbert, and A. G. Greenberg. An approximate model of processor communication rings under heavy load. *Information Processing Letters*, 64(2):61–67, 1997.

[10] E. Coffman, N. Kahale, and F. T. Leighton. Processor-ring communication: A tight asymptotic bound on packet waiting times. 1996.

[11] W. Eatherton. Hardware-based internet protocol prefix lookups. In *Eatherton, Will. Hardware-Based Internet Protocol Prefix Lookups. Washington University Electrical Engineering Department, MS thesis*, may 1999.

[12] M. R. Garey and D. S. Johnson. Computers and intractability - a guide to the theory of np-completeness. pages 213–224. W.H. Freeman and Company, New York, 1979.

[13] P. Gupta. Algorithmic search solutions: Features and benefits. In *NPC-West 2003*, october 2003.

[14] P. Gupta and N. McKeown. Packet classification on multiple fields. In *Proc of ACM Sigcomm 1999*, september 1999.

[15] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Proc of Hot Interconnects VII*, august 1999.

[16] D. Mayer. University of oregon route views project. 2003. ftp://ftp.routeviews.org/pub/routeviews.

[17] U. Michigan. Multi-threaded routing toolkit. 2003. http://www.mrtd.net/.

[18] H. Narayan, R. Govindan, and G. Varghese. The impact of address allocation and routing on the structure and implementation of routing tables. In *Proc. of ACM Sigcomm 2003*, august 2003.

[19] Ravindran and Stumm. A performance comparison of hierarchical ring- and mesh-connected multiprocessor networks. In *3rd International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 1997.

[20] E. Rosen and Y. Rekhter. BGP/MPLS VPNs. RFC 2547, 1999.

[21] RRC. Routing information service raw data. 2003. http://data.ris.ripe.net/.

[22] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. In *IEEE Network Magazine, vol. 15, no. 2*, 2001.

[23] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. 2003. 30th Annual International Symposium on Computer Architecture.

[24] P. Shivakumar and N. Jouppi. Cacti. http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[25] S. Sikka and G. Varghese. Memory-efficient state lookups with fast updates. In *Proc of ACM Sigcomm 2000*, september 2000.

[26] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. of ACM Sigcomm 2003*, august 2003.

[27] V. Srinivasan and al. Fast and scalable layer 4 switching. In *Proc of ACM Sigcomm 1998*, september 1998.

[28] T. Woo. A modular approach to packet classification: Algorithms and results. In *Proc. of Infocom*, 2000.

A IP Lookup

$Prefix$	$Value$
P_1	$0000001*$
P_2	$000000000*$
P_3	$01101100*$
P_4	$0110110100*$
P_5	$0110110101*$
P_6	$11001*$
P_7	$111101000*$
P_8	$11110101*$
P_9	$11110101110*$
P_{10}	$01100*$
P_{11}	$011011*$
P_{12}	$*$

Table 7. A simple example of a routing table with 12 prefixes.

The IP lookup operation requires a longest matching prefix computation at wire speeds. In IPv4 for example, at every hop (router), for each packet the 32 bit IP destination address is matched against a databases of IP prefixes. Each prefix entry consists of a prefix and a next hop value. For a better understanding of the problem, let's consider the following toy example based on an IP lookup database consisting of the following 12 prefixes shown in Table 7. If the router receives a packet with the destination address that starts with 11110101110 then the next hop value associated with the prefix P_9 is selected.

There are many solutions in the literature for the IP lookup problem ranging from binary search to trie lookup [22]. In the

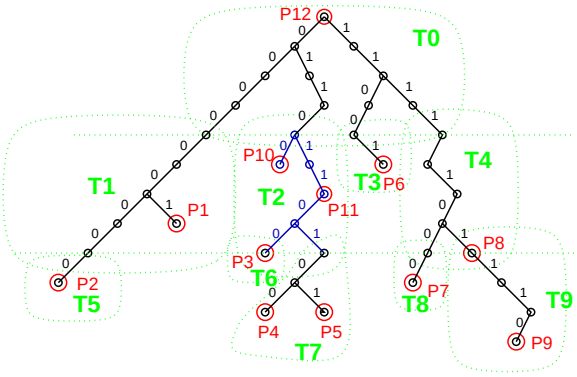

Figure 4. The trie lookup structure associated with the routing table given in Table 7.

evaluation of our new pipeline scheme we use the algorithm invented by Eatherton [11]. This algorithm offers both excellent throughput as well as fast update rates.

Eatherton's algorithm uses a trie as the basic search structure. The trie is organized into subtries with fixed depth(for example 4) marked with dotted lines in Figure 4. As a result, the initial trie is now represented as a tree in which each node is associated with a subtrie in the original representation.

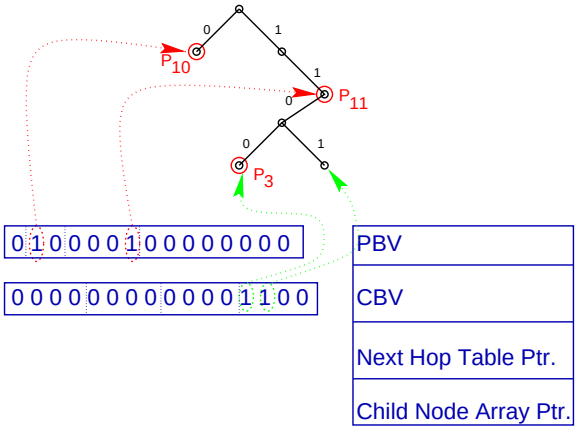

Figure 5. Each subtrie in the original trie may be represented as it is shown here. This picture shows the representation of the subtrie T_2 from Figure 4.

Each subtrie is represented with the help of two bit vectors. Figure 5 shows the representation of the subtrie T_2 from Figure 4 and two arrays that store the child nodes and the next hop information associated with the current node.

A first bit vector that we call PBV describes the distribution of the nodes associated with valid prefixes inside of the subtrie. This bit vector represents a linearized format of the original subtree: each row of the subtree is captured top-down from left to right. Each bit is associated in order with the prefixes: **, 0*, 1*, 00*, 01*, 10*, 11*, . . . , 111**. Two bits are set in PBV; they correspond to the valid prefixes P_{10} and P_{11} existent in the subtrie as it is shown in Figure 5 (The node associated with the prefix P_3 does not belong in this subtrie. Instead it is the root node of one of its child subtrie). The next hop information associated with each of the valid prefixes is kept in a table.

The second bit vector which we call CBV describes the child distribution. There are at most 2^4 children and a bit is set whenever a child exists at the end of that path through the trie. Thus, in Figure 5 we only have two bits set corresponding to two child subtries associated with the prefixes 1100 and 1101, respectively.

In summary, the search structure in the Eatherton algorithm is a tree which in every node stores: two bit vectors, a pointer to the block of child nodes, and a pointer to an array of next hop data. In order for the scheme to work efficiently all child nodes of a given parent must be stored contiguously in memory, to maximize locality, and minimize memory access time. Similarly, all the next hop information associated with valid prefix nodes in the associated subtrie is stored as a contiguous block in memory.

A search operation executes as follows. Assume that we need to identify the longest matching prefix associated with a destination address 01101101010. The algorithm considers strides of 4 bits of address at a time. It starts by reading the child bit vector associated with the root node and it determines if there is a child subtrie with the root at the position 0110. This corresponds to the seventh bit in the CBV being set. This bit is set which means that the search continues to the next node by using the next four bits of the address. In parallel it determines if there is any matching prefix in this node. If there is a match, the algorithm remembers it and continues the search recursively by going to the next child node. When the search fails, the last matching prefix represents the longest matching prefix for the search.

B Optimally Allocating Subtrees on a Pipeline Ring is an NP-Complete Problem

We here show the intractability of the problem of optimal placement of subtrees on a pipeline ring. In fact, we show that the simpler problem of deciding whether M given trees of height H can be allocated on a *ring* of H cells such that each cell contains an equal number of nodes is *NP-complete*. Here the complete tree information, i.e., the parents of each node, is not needed; only the number of nodes per level is neccessary to schedule a placement. Therefore, a tree of height H is encoded as H numbers $(l_1, l_2, ..., l_H)$ in the range $1..2^H$. Hence, one only needs $O(H^2)$ space to store a tree, despite the fact that the *weight* of the tree, that is, the total number of nodes $l_1 + l_2 + \cdots + l_H$, can be exponential in H.

Problem: OPTIMAL-RING-PLACEMENT
Input: A height $H \in \mathbb{Z}^+$, a number $M \in \mathbb{Z}^+$, and M binary trees of height H.
Output: Can these trees be scheduled on a *ring* of H cells such that each cell contains exactly W/H nodes, where W is the total weight (number of nodes) of all the M trees?

OPTIMAL-RING-PLACEMENT is therefore a decision problem (outputs "yes" or "no") taking an input of size $O(MH^2)$.

Our goal is to show that OPTIMAL-RING-PLACEMENT is an *NP-complete* problem, thus motivating our focus on searching good practical subtree allocation heuristics, rather than on finding provably optimal solutions. We therefore need to show that the problem is in the class of *NP* problems and that it is *NP-hard*. While the first task is almost immediate, the NP-hardness is not trivial. We will use a reduction to a modified version of a known NP-complete problem. The following partition problem is a well-known NP-complete problem, even in the strong sense, as shown in Garey and Johnson [12]:

Problem: 3-PARTITION
Input: A finite set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies the relation $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.
Output: Can A be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that $\sum_{a \in S_i} s(a) = B$ for each $1 \leq i \leq m$?[2]

Unfortunately, the 3-PARTITION problem lets the relationship between m and B unspecified, so one may wrongly assume that the hardness of this problem comes from certain *bad* close relationships between m and B. To avoid this kind of wrong assumption and to settle the ground for our main theorem, we consider a more general version of this problem. Given any arbitrary but fixed "ratio" r, we define the following problem.

Problem: 3-PARTITION$[r]$[3]
Input: A finite set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$ with $m/B \geq r$ and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies the relation $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.
Output: Can A be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ such that $\sum_{a \in S_i} s(a) = B$ for each $1 \leq i \leq m$?

Lemma 1 *For any given r,* 3-PARTITION$[r]$ *is NP-complete.*

Proof: Since 3-PARTITION$[r]$ differes from 3-PARTITION by just a more constrained input, 3-PARTITION$[r]$ is also in NP. We show that 3-PARTITION$[r]$ is NP-hard by reducing it to 3-PARTITION. Let us consider an input of 3-PARTITION: a set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and a size function $s : A \to \mathbb{Z}^+$. We need to construct an input for 3-PARTITION$[r]$, consisting of a set A' of $3m'$ elements, bound $B' \in \mathbb{Z}^+$ such that $m'/B' \geq r$, and size function $s' : A' \to \mathbb{Z}^+$ with $B'/4 < s(a') < B'/2$ for all $a' \in A'$, and then show that 3-PARTITION has a positive answer on the input m, A, B, s if and only if 3-PARTITION$[r]$ has a positive answer on the input m', A', B', s'. If $m/B \geq r$ then one can clearly take m', A', B', s' to be just m, A, B, s, respectively. Suppose now the difficult case, namely that $m/B < r$.

[2]Notice that these constraints on the item sizes imply that every such S_i must contain *exactly* three elements from A.

[3]3-PARTION$[r]$ defines a class of problems, one for each r.

Let B' be precisely B. One can build the set A' by adding enough "fresh" elements to A, so that their total number, m', has the property $m'/B \geq r$: let us first take m' to be $\lceil rB \rceil$, i.e., the smallest natural number larger than or equal to rB, and then lets us take A' to be the set $A \bigcup \{x_1, x_2, \ldots, x_{3(m'-m)}\}$, for some arbitrary elements $x_1, x_2, \ldots, x_{3(m'-m)}$ which do not occur in A. We now need to construct an appropriate size function $s' : A' \to \mathbb{Z}^+$. The crucial idea here is to build it in such a way that all the elements of A have sizes very close to $B/3$ while the elements x_i have sizes far enough from $B/3$, so that the only way to get a positive answer to 3-PARTITION$[r]$ is to actually get a solution to 3-PARTITION and group the elements x_i among themselves. For example, let $s'(a)$ be defined as $B/3+(s(a)-B/3)/1000$ for each $a \in A$, and $s'(x_{3k+1}) = s'(x_{3k+2}) = B/3 - B/100$ and $s'(x_{3k+3}) = B/3 + B/50$. It is now easy to see that 3-PARTITION$[r]$ admits a solution on the input A', B, s' if and only if 3-PARTITION admits a solution on the input A, B, s. That happens because $x_1, x_2, \ldots, x_{3(m'-m)}$ can only be grouped with themselves in any solution of 3-PARTITION$[r]$. □

We can now prove our main theorem.

Theorem 2 OPTIMAL-RING-PLACEMENT *is NP-complete.*

Proof: Let us first note that OPTIMAL-RING-PLACEMENT is in NP. Indeed, if one is given an input and a placement, that is a map $\{1, \ldots, M\} \to \{1, \ldots, H\}$, assigning each tree to a pipeline(ring) stage from where it starts being allocated, then the only thing one has to do is to check whether each pipeline stage has exactly W/H nodes. This can be obviously accomplished in polynomial time.

We next show that OPTIMAL-RING-PLACEMENT is *NP-hard* by reducing it to 3-PARTITION$[r]$ for some appropiate r. Let us consider an input of 3-PARTITION$[r]$: some set A of $3m$ elements, some bound $B \in \mathbb{Z}^+$ with $m/B \geq r$, and a size function $s : A \to \mathbb{Z}^+$ such that $B/4 < s(a) < B/2$ for each $a \in A$ and such that $\sum_{a \in A} s(a) = mB$. We can then build an input of the OPTIMAL-RING-PLACEMENT problem as follows. Let H be m, let M be $3m$, and let us consider one tree, t_a, for each element $a \in A$, having 1 node on the first level, 2 on the second level, 4 on the third, ..., 2^{m-2} on level $m-1$, and $s(a)$ on the last level. In other words, each tree t_a has height $H = m$, is complete on the first $H - 1$ levels and has $s(a)$ nodes on the last level. Our trees are binary, therefore this can happen only if m is large enough so that $s(a) \leq 2^{m-1}$. Since $s(a) < B/2$, we can take r large enough so that $B \leq 2^m$. Note that, with the abstract view of a tree as a list of numbers symbolizing the nodes on each level, $t_a = (2^0, 2^1, 2^2, \ldots, 2^{m-2}, s(a))$ for each $a \in A$. Let us now calculate the total weight of all the trees:

$$\begin{aligned} W &= \sum_{a \in A} (2^0 + 2^1 + 2^2 + \ldots + 2^{m-2} + s(a)) \\ &= 3m(2^{m-1} - 1) + mB. \end{aligned}$$

Thus we created in polynomial time an instance of the problem OPTIMAL-RING-PLACEMENT – the time needed to create

all the trees t_a is indeed polynomial, because 2^k can be represented on $m-2$ bits for all $0 \leq k \leq m-2$. The only thing left is to show that the original input A, B, s of 3-PARTITION$[r]$ admits a solution *if and only if* the created input of OPTIMAL-RING-PLACEMENT admits a solution. The *"only if"* part is easy. Indeed, if there is some partition of A into m disjoint sets $S_1, S_2, \ldots, S_m$, each of 3 elements, such that $\sum_{a \in S_i} = B$, then one can allocate the corresponding trees $\{t_a\}_{a \in S_i}$ of each partition starting with the same cell. Then each cell will contain 3 groups of $1, 2, \ldots, 2^{m-2}$ elements (respectively) plus $\sum_{a \in S_i} s(a)$ for some $1 \geq i \geq m$, that is, $3(2^{m-1}-1)+B$ elements. The ring is balanced with this allocation.

Let us next consider the *"if"* part, that is, let us assume that the input of OPTIMAL-RING-PLACEMENT created above admits one solution and let us prove that the original input of 3-PATRITION$[r]$, A, B, s, also admits a solution. Note first that each stage in the balanced pipeline will contain exactly $3(2^{m-1}-1)+B$ nodes. It is enough then to show that any allocation of the $3m$ trees on the ring requires precisely three trees to be allocated starting with each stage in the ring. If that is the case, then we can group together the items in A corresponding to each of these trees and obtain a partition satisfying the input of the 3-PARTITION$[r]$ problem.

Let us assume that the solution to the input of OPTIMAL-RING-PLACEMENT allocates the trees $T_i = \{t_{a_i^1}, t_{a_i^2}, \ldots, t_{a_i^{k_i}}\}$ starting with the stage i, for $0 \geq i \geq m$. Then clearly $\sum_{i=1}^{m} k_i = 3m$; all what we need to show is that $k_1 = k_2 = \ldots = k_m = 3$. Since each stage contains precisely $3(2^{m-1}-1)+B$ nodes, we can write the following equations:

Stage 1: $k_1 + 2k_2 + 2^2 k_3 + \ldots + 2^{m-2}k_{m-1} + \sum_{j=1}^{k_m} s(a_m^j) = 3(2^{m-1}-1)+B$

Stage 2: $\sum_{j=1}^{k_1} s(a_1^j) + k_2 + 2k_3 + \ldots + 2^{m-3}k_{m-1} + 2^{m-2}k_m = 3(2^{m-1}-1)+B$

Stage 3: ...

...

Multiplying the equation **Stage 2** by 2 and then subtracting the equation **Stage 1** from it, we get:

$$(2\sum_{j=1}^{k_1} s(a_1^j) - k_1) + 2^{m-1}k_m - \sum_{j=1}^{k_m} s(a_m^j) = 3(2^{m-1}-1)+B.$$

Since r is chosen such that B is much smaller then 2^m, it follows that the dominant terms in the two sides of the above equality are $2^{m-1}k_m$ and $3 \cdot 2^{m-1}$, respectively. This directly implies that $k_m = 3$. Iterating the previous steps over the different stages, we get $k_1 = k_2 = \ldots = k_m = 3$. □