Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines

Fan Chung *† Ronald Graham * George Varghese *

ABSTRACT

A crucial problem that needs to be solved is the allocation of memory to processors in a pipeline. Ideally, the processor memories should be totally separate (i.e., one port memories) in order to minimize contention; however, this minimizes memory sharing. Idealized sharing occurs by using a single shared memory for all processors but this maximizes contention. Instead, in this paper we show that perfect memory sharing of shared memory can be achieved with a collection of *two*-port memories, as long as the number of processors is less than the number of memories. We show that the problem of allocation is NP-complete in general, but has a fast approximation algorithm that comes within a factor of 3/2. The proof utilizes a new bin packing model, which is interesting in its own right. Further, for important special cases that arise in practice the approximation algorithm is indeed optimal. We also describe an incremental memory allocation algorithm that provides good memory utilization while allowing fast updates.

Categories and Subject Descriptors
C. 1. 4. Parallel Architechtures
F. 2. 2. Nonnumerical Algorithms and Problems
H.3. Information Storage and Retrieval

General Terms
Algorithms, Performance, Theory.

Keywords
Memory allocation, approximation algorithm

1. INTRODUCTION

Parallel processors are often used to solve time-consuming problems. Typically, each processor has some memory where it stores computation data. To minimize contention and maximize speed, each memory should be read by exactly one process. Unfortunately, if the tasks assigned to processors vary wildly in memory usage, this is not an efficient use of memory, for some tasks one processor's memory may be unused while anothers is exhausted.

The interaction between parallelism (the desire to minimize contention) and memory allocation (the desire to maximize memory sharing) is a general phenomenon that has been largely unexplored in the literature. We encountered this problem in the context of networking while trying to design fast IP lookup schemes. In IP lookup, the time-consuming task at hand is prefix lookup, and the processors are arranged (often within a custom chip) as a pipeline.

Almost all known IP lookup schemes [11] traverse some form of tree (e.g., trie, binary tree) using the destination 32-bit IP address in a received packet as a key. The leaves provide information required to forward the packet. Lookup time is proportional to tree height, and storage required is the sum of the storage required for each node.

Observe that any tree can easily be pipelined by height: all nodes at height i are placed in memory i which is accessible only to processor i. Such a design is simple because there is no memory contention. However, it is extremely wasteful of memory. Since the shape of the tree can vary from database to database, and the trees are in general unbalanced, trees can change their memory needs from database to database. More precisely, the number of nodes at height i can vary for different databases by large factors.

Thus, statically deciding the size of each memory is a bad idea because there will be at least some databases where the total amount of memory required is less than the sum of the sizes of all memories, but the database still cannot fit because memory i is underutilized while say memory j is full. How then should memory be allocated to processors? This problem was left as an open problem in [12].

An approximate solution to the problem of trie memory allocation across pipeline stages is described in [1]. It tries to choose the tree to minimize memory imbalance. Their results show a reduction in the maximum allocation by approximately one-half. Unfortunately these results do not help for worst-case designs. Their worst-case bound is close to the naive bound of requiring each stage memory equal to the total required memory.

*University of California, San Diego
†Research supported in part by NSF Grants DMS 0100472 and ITR 0205061

SPAA'04, June 27–30, 2004, Barcelona, Spain

Given that minimizing memory is required to minimize cost and that pipelining is required for speed, one way out of the dilemma is to *change the underlying model*. In some sense, the rest of this paper can be considered to be the proposal of a new memory model for pipelined engines and its implications. To motivate our final model (multiple two-port memories connected by a partial crossbar), we first consider a series of simpler models, which however have drawbacks.

Our second model (the first is partitioned memory) is *shared memory* which is ideal for memory sharing. Unfortunately, large, fast shared memories are currently infeasible to build. In practice, most large n-port memory is (underneath the covers) time-multiplexed. Every processor is given one memory access for every n memory accesses done to the memory (in the worst case). Unfortunately, multiplexing n-ways causes the effective memory access time to grow by a factor of n. The tradeoff between these two extremes is shown in Figure 1.

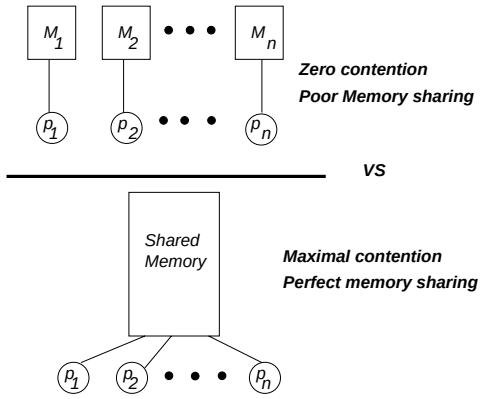

Figure 1: Models 1 and 2 have problems: Strictly partioned memories have poor memory sharing while a single shared memory has poor contention.

When faced with two unacceptable extremes, it is natural to consider intermediate forms. Thus, strictly partitioned 1-port memories have good access speeds and memory densities but have poor memory utilization. On the other hand, n-port memories have the opposite problem. Hence, it is natural to consider a collection of Y-port memories, where $Y < n$. A natural starting point is to consider $Y = 1$ memories. Thus, imagine for our second model that we have a collection of b 1-port memories that are shared among the n processors (see Figure 2).

This can be modeled by a set of n processors (shown on the bottom of Figure 2) and a set of b memories (shown on the top of Figure 2) that are connected by an interconnection network. The interconnection network allows parallel connections to be made between processors and memories, and allows each processor to be connected to multiple memories, but allows at most one processor to be connected to a single memory (because the memories have only 1 port). Such interconnection networks are commonly used in parallel computers[4] and are called crossbar switches.

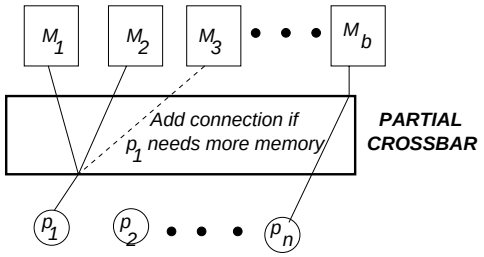

Figure 2: Model 3: Allowing memory sharing by connecting a *large* number of *one ported* memory banks to the set of n processors via a partial crossbar.

Figure 2 shows processor p_1 connected to two memories M_1 and M_2. Suppose that is all that has been allocated to p_1, and p_1 wants more memory. The idea is that the memory allocation system keeps track of the free memories, realizes that, say M_3, is free and (see dashed line in Figure 2) reconfigures the crossbar to allocate M_3 to p_1. Notice that the crossbar need only be reconfigured at allocation time, which is generally orders of magnitude less stringent than lookup times.

At first glance, this looks very attractive, because if b is large, then each processor can waste at most 1 memory, which is small in size for large b. Thus the percentage of wasted memory is at most $\frac{n-1}{b}$. For example, for $n = 16$, if $b = 32$ this can incur a worst case memory wastage of around 50%. While this is quite large, it can be reduced to essentially 0 by increasing b.

While this looks superficially attractive, in practice one does not want to waste even 12.5% of an expensive SRAM memory system, especially if it is on chip. This implies the use of even higher values of b. Unfortunately, practical constraints limit the values of b that can be used. The larger the number of memory banks, the larger the load that must be driven on the data busses that make up the interconnection network, and hence the larger the delay. It is difficult today to imagine a very high speed design with more than say $b = 100$ banks of memory connected via the crossbar. It would be far simpler and faster (important for higher speeds) to use a smaller number of banks, such as $b = 32$, and still get good memory utilization.

Because of the bus capacitance issues of dealing with a large number of memories caused by using a large number of shared 1-port memories, we consider the next natural progression in our model (Figure 3). Thus we consider increasing the number of ports on the memories to $Y = 2$ from $Y = 1$. A collection of 2-port memories will only slow down access speeds (using say time multiplexing) by a factor of at most two. But what kind of memory utilization would such 2-port memories provide?

To understand the model, imagine a collection of n processors that have access to a network (e.g., a crossbar switch) that allows them access to a collection of b 2-port memories. Each memory has 2 ports that can be allocated to any two processors. Thus each memory can be read by at most 2

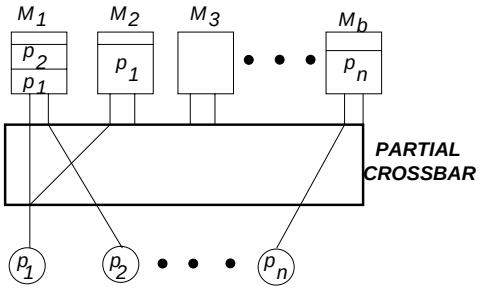

Figure 3: Our final model: Allowing memory sharing by connecting a *small* number of *two-ported* memory banks to the set of n processors via a partial crossbar.

processors at a time. Of course, a processor that needs a large amount of memory could be assigned a port on $X > 1$ memories. Each of the b memories has a fixed amount of memory, say Max memory words.

Notice in Figure 3 that memory M_1 is not completely full and is allocated partially to processor p_1 and partially to processor p_2. Notice also of the two memory ports allocated to each processor in Figure 3, M_1 has both ports allocated, M_2 and M_b have one port allocated and one port free, and M_3 has two ports free. Thus, if say processor p_3 wants even one word of memory, p_3 cannot use M_1 (both of M_1's ports are already allocated to other processors even though it has free memory). However, if p_2 wants more memory it can get more allocation in M_1.

Thus, it should be clear that besides allocating memory, the allocator has to be frugal in allocating ports in order not to waste memory. Consider, for example, a scenario where processors p_1 and p_2 are allocated one word of memory each in all of the b memories. If $Max \gg 1$, then no other processor can then get any memory because all ports are allocated, and the resulting utilization (measured when some processor cannot satisfy a memory allocation request) is nearly zero. Of course, the memory allocator could finesse this particular issue by compacting all of p_1 and p_2's requests to fit in as few memory banks as possible. But this example should indicate that it is unclear whether perfect memory allocation is possible while respecting the 2 port constraint at every memory.

Now consider the offline problem of memory allocation. Imagine that the input is a collection of memory requests per processor (e.g., 5 words for processor 1, 10 for processor 2, etc.). We say that an allocation is feasible if every processor's request is satisfied and the no more than two processors are allocated to any one memory. Ideally, we want a fast algorithm that will guarantee a feasible allocation as long as the input is feasible (i.e., the sum of processor requests is less than total memory size).

We will show that a very fast $O(n)$ algorithm exists for optimal memory allocation for feasible inputs as long as $b > n$. This algorithm is sufficient for practical implementations because one can constrain the design to use more smaller memories (often called *memory banks*) than the number of processors. (As n grows, there is an increased interconnect cost as b grows, but this is not a problem for $n < 64$). While the speed of allocation is usually not as important as reads and writes to memory, fast allocation algorithms allow faster reconfiguration of data structures in this memory structure and are important in their own right.

Even practical problems give rise to theoretical problems that have a life of their own. The practical problem can be abstracted as a theoretical problem of bin packing with an additional constraint. We show that for the general case of arbitrary b and n, the problem of finding a feasible allocation is NP-complete (it should not surprise the reader than an NP-complete problem is efficiently solvable in a special case; consider the case of computing a Hamiltonian cycle, which is trivial if the graph has only a small number of cycles.)

We deal with the NP-completeness by presenting an approximate algorithm that produces memory utilization that is within a factor of 3/2 of optimal. Practically, this means that if the designer wishes to use a smaller number of memory banks than the number of processors, he or she should overdesign the total memory capacity by a factor of 3/2. Fortunately, the approximation algorithm is exactly optimal in the case of $b > n$, so we describe only one algorithm for both cases.

In the rest of this paper we abstract the problem as a bin packing problem with the 2-port constraint abstracted as a "two type" constraint. We also normalize the memory sizes to 1 (instead of Max) without loss of generality by allowing fractional inputs (called weights) for each processor.

While this version of the paper mostly focuses on the offline problem, in practice the set of processors will keep getting new memory requests. When a new memory request occurs that causes the assignment of processors to memories to change, one has to reconfigure the crossbar and possibly move data around between memories. Thus the online problem becomes one of minimizing data movement to deal with allocation (e.g., weight) changes while maintaining good memory utilization. There appears to be a tradeoff here as well. At the end of the paper we briefly describe a very simple algorithm for the dynamic case that works well in practice.

We are unaware of any related work in architecture that relates buffer allocation and pipelining. A result that can be made applicable is the use of *randomization* [9] in storing memory words so that with high probability memory words are evenly distributed across b memory banks. Similar notions of randomizing accesses to memory date back to Valiant [13] and Ranade [8], as well as some recent work [3]. The use of randomization has several problems: first, randomization prevents the use of synchronous pipelines that rely on tight timing guarantees; second, randomization leads to poor contention bounds. For example, using MAPLE, we calculated that for 16 processors making random requests to 16 memories, the probability that at least 3 memory accesses go to the same memory is > 0.805. In other words, there is an 80 % chance that at least 3 memory accesses go to at least one memory.

Thus while randomization is an interesting option, in this paper we examine deterministically layouts that limit contention to at most 2 processors per memory.

2. ABSTRACTING THE PROBLEM

Here is the formulation the bin packing problem that is motivated by the above memory allocation problem.

Suppose we have an unlimited number of bins each of capacity 1. We are given a list of weights, say, $W = (w_1, w_2, w_3, \ldots, w_n)$, where w_i is nonnegative and can be greater than 1 in general. We say W can be packed into b bins if there is a way to partition "items" I_j of type j with weight w_j, for $1 \leq j \leq n$, such that all parts fit into b bins. In other words, for each k, the parts that are grouped into the k-th bin have total weight at most 1.

In this paper, we focus on the following constrained bin packing problem:
Problem: For a given list W, find a way to pack W into a minimum number of bins such that each bin can have parts of at most *two* types.

An immediate question is to decide if this problem is easy or hard to solve. In the next section, we will show that the above problem is indeed NP-complete and thus is probably computationally intractable (see [6], for a survey).

Then we proceed to discuss approximation algorithms. We will consider a fast and robust algorithm that gives approximate solutions in linear time (in n). The solution this algorithm gives is optimum if the total sum of the weights is no smaller than the number of types. In general, the solutions are always within a factor of 3/2 of the optimum. Several examples are given to indicate the sharpness of this worst case performance ratio.

We also examine bin packing problems with more general constraints. For example, for a fixed integer $r > 2$, we consider the bin packing problem such that each bin can have parts of at most r different types. This problem (and others) will be discussed in the final section.

3. OUR BIN PACKING PROBLEM IS NP-COMPLETE

We will prove the NP-completeness of the bin packing problem with the constraint that each bin can have at most two types. The transformation is from the 3-partition problem which can be stated as follows (see [6]):

3-PARTITION

Instance: A 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 $B/4 < s(a) < B/2$ and $\sum_{a \in A} s(a) = mB$.
Question: Can A be partitioned into m disjoint sets A_1, $A_2, \ldots, A_m$ such that for $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$ (note that each A_i must therefore contain exactly 3 elements from A)?

Garey and Johnson [7] showed the 3-PARTITION problem is NP-complete by using transformation from the problem of 3-dimensional matching. In fact, they showed that the 3-PARTITION problem is NP-complete in the strong sense (see [6]).

For a given instance of the 3-PARTITION problem as described above, we consider the following bin packing problem::
(*) We are given a list W of $3m$ weights

$$w_a = \frac{1}{2} + \frac{s(a)}{2B}$$

Determine if W can be packed into $2m$ bins such that no bin contains more than two types.

It suffices to show that the 3-PARTITION problem has an affirmative solution if and only if the above problem (*) has a solution.

First we consider the easy direction. Suppose the 3-PARTITION problem has a solution $A_1, A_2, \ldots, A_m$. For each i, we can pack the weights $w(a)$, for $a \in A_i$ into two bins since

$$\sum_{a \in A_i} w_a = \frac{3}{2} + \sum_{a \in A_i} \frac{s(a)}{2B} = 2$$

and w_i satisfies

$$\frac{3}{8} < w_i = \frac{1}{2} + \frac{s(a)}{2B} < \frac{3}{4}.$$

So, W can be packed into $2m$ bins when each bin has two types and thus problem (*) is solved.

Now suppose problem (*) has a solution with a packing into $2m$ bins. Clearly, each bin contains parts summing up to 1 since $\sum_a w_a = 2m$.

First, we observe that a weight type can not be partitioned into more than 2 parts. Suppose the contrary. There is a weight type, say w_1, that is partitioned into k parts which are contained in k bins where $k \geq 3$. One of the parts is less than 1/4 since $w_1 < 3/4$. This bin that contains this small part can contain another part with weight at most 3/4. Thus, this bin can not have parts summing up to 1, which is impossible.

Second, we claim that the number t of types of weights that are packed in two bins is exactly m. Suppose t is more than m. Then, the total number of parts is more than $4m$. Since at most two parts can be packed into one bin, we need more than $2m$ bins, which is a contradiction. Now, suppose that t is less than m. Since there are at most $2t$ bins that can contain parts of two types, there are at least two bins that can contain at most one type. Those two bins can not have parts summing up to 1, which is again a contradiction.

Hence, there are exactly m weights S that are each partitioned into two parts. We write

$$S = \{a_{j_1}, a_{j_2}, \ldots, a_{j_m}\}$$

We consider A_i consisting of a_{j_i} and the types w_a that are contained in bins containing parts of $w_{a_{j_i}}$. Clearly A_i, $i = 1, \ldots, m$, is a partition of A. Furthermore, we have

$$\sum_{a \in A_i} w_a = \frac{3}{2} + \sum_{a \in A_i} \frac{s(a)}{2B} = 2$$

This implies that

$$\sum_{a \in A_i} s(a) = B$$

Thus, this gives a solution to the 3-PARTITION problem. Hence, we have shown the following:

THEOREM 1. *The bin packing problem with the constraint that each bin contains at most two types is NP-complete.*

4. A GRAPH REPRESENTATION

Before we discuss approximation algorithms for our bin packing problem and their worst case analyses, we consider a graph representation of a packing.

Suppose a list of weights $W = (w_1, w_2, \ldots, w_n)$ is packed into unit bins so that no bin holds more than two types of weights. Let P denote such a packing. We associate a graph G_P with P defined as follows:

(1) G_P has n vertices, each of which represents a type.

(2) The edges of G_p correspond to the bins in one-to-one fashion. If the bin contains only one type, it corresponds to a loop on that type. If the bin contains two types, it corresponds to an (ordinary) edge between the two types.

(3) If the bin is partially filled, we say the corresponding edge is a *weak* edge (or loop).

For example, suppose that $W = (1/2, 2/3, 1/4)$ has three types as shown in Figure 4. One packing configuration P is given in Figure 5 and the associated graph G_P appears in Figure 6.

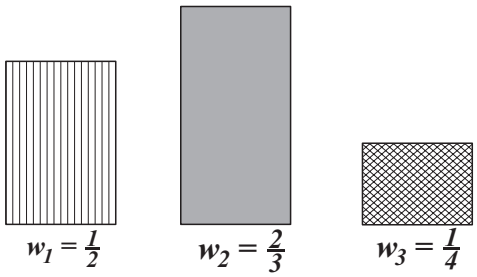

Figure 4: Weights of three types

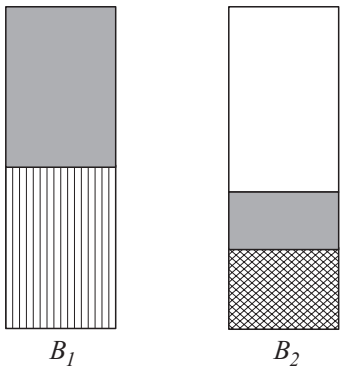

Figure 5: *A packing P*

There are, of course, different ways to pack W into two bins so that each bin contains at most two types. Another packing configuration Q is given in Figure 7 and its associated graph G_Q is shown in Figure 8.

5. SOME BASIC PROPERTIES OF THE ASSOCIATED GRAPHS

Here we examine several basic properties of the associated graphs of bin packings for a given list of weights W. These properties provide the foundation for the reduction steps in the approximation algorithms to be discussed in the next section.

In this paper, we use the convention that a cycle must have at least two vertices. So, by definition, a loop is not a cycle. An edge is either a loop or an ordinary edge with two distinct endpoints.

LEMMA 1. *Suppose that P is a packing of a list of weights $W = (w_1, w_2, \ldots, w_n)$ into b bins, where no bin contains weights of more than two types. If the associated graph G_P contains a cycle, we can find another packing P' which uses no more than b bins with its associated graph containing no cycle.*

Proof: Suppose G_P contains a cycle C with edges $B_1, B_2, \ldots, B_t$. (Here we use B_i to denote both an edge and a bin, if there is no confusion.) We may assume that B_i contains w_i'', w_{i+1}' for $1 \le i \le t-1$ and B_t contains w_t'', w_1' where w_i' and w_i'' are parts of weights of type i. We also may assume that all w_i' and w_i'' are positive since C is a cycle.

Now, without loss of generality, assume that w_1' has the smallest size among all w_i' and w_i''. We consider a new packing P' of W such that P' is the same as P except for the bins B_i, $1 \le i \le t$. P' contains the following bin configuration B_i' instead of B_i. We reorganize the parts of type i weights. Instead of two parts of sizes w_i' and w_i'', we have two new parts of type i of sizes *new* $w_i' = w_i' - w_1'$ and *new* $w_i'' = w_i'' + w_1'$. B_i' consists of *new* w_i'' and *new* w_{i+1}' for $1 \le i \le t-1$ while *new* B_t consists of only *new* w_t''. Clearly, P' is still a valid packing and $G_{P'}$ does not contain C. In fact, $G_{P'}$ contains the same edges as G_P except for B_t since B_t' is either a loop or empty.

By the above procedure, we can eliminate one cycle at a time while the number of bins used stays the same or decreases. Eventually, we will reach a packing with its associated graph containing no cycle and the number of bins is no more than what we started with. □

A graph that contains no cycle is a forest plus some possible loops. We recall that a weak edge in G_P corresponds to a partially filled bin in P.

LEMMA 2. *Suppose that P is a packing of a list of weights $W = (w_1, w_2, \ldots, w_n)$ into b bins, where no bin contains*

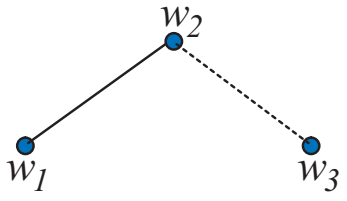

Figure 6: *The graph G_P*

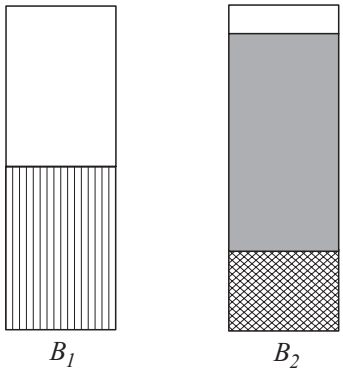

Figure 7: *Another packing Q*

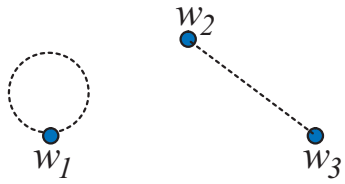

Figure 8: *The graph G_Q*

weights of more than two types. If the associated graph G_P contains two weak edges in the same connected component, we can find another packing P' which uses no more than b bins with its associated graph satisfying the property that every connected component contains at most one weak edge.

Proof: First we apply Lemma 1 so that there is no cycle in the associated graph of the packing. Suppose a connected component of G_P contains two weak edges A_1 and A_2 (which might be loops). The two weak edges cannot have the same vertices, since this would form a 2-cycle, contradicting our initial hypothesis. There must be a path (that is, a sequence edges so that two consecutive edges share a common vertex), say, with edges $A_1 = B_1, B_2, \ldots, B_t = A_2$. (Here we allow the case that A_1 and/or A_2 are loops while all other B_i's are ordinary edges.) We may assume that B_i contains w_i'', w_{i+1}' for $1 \leq i \leq t$, where w_i' and w_i'' are parts of weights of type i. We also may assume that B_1 and B_t are weak edges so that $g_1 = 1 - w_1'' - w_2' > 0$. and $g_1 = 1 - w_t'' - w_{t+1}' > 0$. Without loss of generality, we assume $g_0 \leq g_1$. We consider two cases:

Case 1: Suppose $g_0 \leq w_i''$, for all $1 \leq i \leq t-1$.

We consider a new packing P' of W such that P' is the same as P except for the bins B_i, $1 \leq i \leq t$. P' contains B_i' (instead of B_i) which is defined as follows:. We reorganize the parts of type i weights for $2 \leq i \leq t-1$. Instead of two parts of sizes w_i' and w_i'', we have two new parts of type i of sizes *new* $w_i' = w_i' + g_0$ and *new* $w_i'' = w_i'' - g_0$. B_i' contains *new* w_i'' and *new* w_{i+1}' for $2 \leq i \leq t-1$, and *new* B_1 contains w_1'' and *new* w_2'. Also, bin B_t' contains *new* w_t'' and old w_{t+1}'. Again, P' is still a valid packing and, in $G_{P'}$, the edge B_1' is not a weak edge. In fact, $G_{P'}$ contains one fewer weak edge than G_P does.

Case 2: Suppose $g_0 > w_j''$, for some j, $1 \leq j \leq t-1$.

We consider a new packing P' of W such that P' is the same as P except for the bins B_i, $1 \leq i \leq j$. P' contains B_i' (instead of B_i) which is defined as follows:. We reorganize the parts of type i weights for $2 \leq i \leq j$. Instead of two parts of sizes w_i' and w_i'', we have two new parts of type i of sizes *new* $w_i' = w_i' + w_j''$ and *new* $w_i'' = w_i'' - w_j''$, for $1 \leq i \leq j$. B_i' contains *new* w_i'' and *new* w_{i+1}' for $2 \leq i \leq j$, and *new* B_1 contains w_1'' and *new* w_2'. Clearly, P' is still a valid packing and $G_{P'}$ has one more connected component since B_j' is a loop.

By repeating the above process, either the number of weak edges decreases or the number of connected components increases while we never use more bins. This process must stop after a finite number of steps. At that point, in the final packing no component has two weak edges. □

6. APPROXIMATION ALGORITHMS

We now describe a simple algorithm for bin packing subject to the constraint that no bin contains weights of more than two types.

Algorithm A:

For a given list of weights $W = (w_1, w_2, \ldots, w_n)$, we pack greedily. Put w_1 or part of it of weight 1 into the first bin. For each i, we do the following:

(1) Place the maximum possible part of w_i (or the remainder of w_{i-1}) into a partially filled bin if it has only weights of no more than one type. Otherwise, put it into a new bin.
(2) Check if a cycle is formed in the associated graph. If it does, use the steps as described in the proof of Lemma 1 to transform the packing into one without any cycle.
(3) Check if there is more than one weak edge in a connected component. If there is, use the steps as described in the proof of Lemma 2 to transform the packing into one containing at most one weak edge in each connected component.
(4) Check if there are two weak loops. If there are, repack so we get either on partial bin with two types or one full bin with two types and a partial filled bin of one type.

After we repeatedly use the above procedure, the resulting bin packing can have at most two types in each bin and satisfies the property that there is no cycle and there is at most one weak edge in any connected component of its associated graph.

We want to show that the packing that is generated by the above algorithm has a worst case upper bound given in the following:

THEOREM 2. *We are given a list of weights and a packing P of a list W in which no bin contains more than two types of weights. Suppose that the associated graph G_P contains no cycle and each connected component has at most one weak edge. Then the number of bins in P is within a factor of* 3/2 *of the optimum.*

Proof: Suppose the list $W = (w_1, w_2, \ldots, w_n)$ has total sum of weights $w = \sum_{i=1}^{n} w_i$. Let OPT denote the number of bins needed in the optimum packing. Clearly, we have

$$OPT \geq \max\{w, n/2\} \quad (1)$$

Our proof needs the following strengthening of the above inequality:
Claim 1:

$$OPT \geq \max\{w, w^*/2\} \quad (2)$$

where $w^* = \sum_i \lceil w_i/2 \rceil$. Clearly, $w^* \geq n$.

In the other direction, we want to show that the number of bins in P, denoted by $|P|$ satisfies the following:
Claim 2:

$$|P| \leq \lceil \frac{w + w^*}{2} \rceil.$$

Furthermore, we claim
Claim 3:

$$\lceil \frac{w + w^*}{2} \rceil \leq \frac{3}{2} \max\{w, w^*/2\}$$

If all three claims hold, we have

$$|P| \leq \lceil \frac{w + w^*}{2} \rceil \leq \frac{3}{2} \max\{w, w^*/2\} \leq \frac{3}{2} OPT$$

as desired. It remains to prove these three claims.
Proof of Claim 1:
It is enough to show that $OPT \geq w^*/2$ (since it is straightforward to see that $OPT \geq w$). For each i, any packing contains at least $\lceil w_i \rceil$ parts of type i weight. Since each bin can have at most two parts of different types, the number of parts is at most $2 \cdot OPT$. Thus we have $2 \cdot OPT \geq \lceil w_i \rceil$ and Claim 1 is proved.
Proof of Claim 2:
Suppose P has a bin which is filled with just one type, say w_1. (That is, G_P has a loop which is not weak). Let P' denote the packing of the list of weights W' which is the same as W except that $w'_1 = w_1 - 1$. By the induction hypothesis, it is true for P' (which has a smaller number of bins). This implies

$$|P'| \leq \lceil \frac{w' + (w')^*}{2} \rceil$$

Since $|P| = 1 + |P'|$, $w' = w - 1, (w')^* = w^* - 1$, we have

$$|P| \leq \lceil \frac{w + w^*}{2} \rceil$$

We may assume that each filled bin involves weights of two types. We consider a connected component A of G_P. We denote $w_A = \sum_{i \in A} w_i$ and $w^*_A = \sum_{i \in A} \lceil w_i \rceil$. It is enough to show that the number of edges in A is at most $(w_A + w^*_A)/2$. This is true if there is no weak edge. We consider the remaining two possibilities:
Case 1: There is one weak (ordinary) edge.
Let $\nu(A)$ denote the number of vertices in A. The number of edges is exactly $\nu(A) - 1$. On the other hand, $w_A \geq \nu(A) - 2$ since there are at least $\nu(A) - 2$ filled bins. We then have

$$w_A + w^*_A \geq w_A + \nu(A) \geq 2\nu(A) - 2$$

which is greater than twice the number of edges in A, and this case is finished.
Case 2: There is one weak loop.
Let $\nu(A)$ denote the number of vertices in A. The number of edges is exactly $\nu(A)$. On the other hand, $w_A \geq \nu(A) - 1$ since there are at least $\nu(A) - 1$ filled bins. We have

$$w_A + w^*_A \geq w_A + \nu(A) \geq 2\nu(A) - 1$$

Thus

$$\lceil \frac{w_A + w^*_A}{2} \rceil = \nu(A)$$

which is equal to the number of edges in A. So, Claim 2 is proved.
Proof of Claim 3:
We want to show that

$$\lceil \frac{w + w^*}{2} \rceil \leq \frac{3}{2} \max\{w, w^*/2\}$$

We consider two cases:
Case a: $\lceil w^*/2 \rceil \leq w$.
We then have

$$\lceil \frac{w + w^*}{2} \rceil \leq \frac{3}{2} w \leq \frac{3}{2} OPT$$

Case b: $\lceil w^*/2 \rceil \geq w$.
It follows that

$$\lceil \frac{w + w^*}{2} \rceil \leq \frac{3}{2} \lceil w^*/2 \rceil \leq \frac{3}{2} OPT$$

This completes the proof of Theorem 2 □

As an immediate consequence, we have

THEOREM 3. *Algorithm A always generates a bin packing which has size within a factor of* 3/2 *of the optimum.*

To consider the complexity of Algorithm A, we examine the following:

- From (1), the associated graph G_P has every vertex with degree at most 2. Namely, for each i, every bin that contains some part of type i, with the exception of at most two bins, contains part of type i with weight exactly of value 1.
- There are only a linear number of steps involving (2) since all cycles are vertex-disjoint.
- To check (3), we note that each connected component is a path. There are at most a linear number of steps needed to transform the packing into one containing at most one weak edge in each connected component.
- There are at most $n/2$ steps used in checking (4).

Thus, Algorithm A runs in time $O(n)$, where n is the number of types.

7. AN IMPROVED ALGORITHM

In this section, we consider a modified version of the approximation algorithm given in Section 6. We will show that the modified algorithm gives the optimum solution when the

total weight is greater than or equal to the number of types. In general, the modified algorithm gives an approximation solution within a factor of 3/2 of the optimum.

The modified algorithm, which we call Algorithm B, is exactly the same as Algorithm A in Section 6 except that for each i, in addition to (1)-(4), we add the following:
(5) Suppose there is a loop that is not weak (associated with a filled bin, say B_1, in one type j) and suppose that there is weak (ordinary) edge $\{k, l\}$ (associated with a partially filled bin, say B_2) not in the same connected component as B_1. We reconfigure the two bins as follows:
Suppose B_2 contains parts of weights w'_k and w''_l. We partition the weight of type j in B_1 into two parts w'_j (of size the same as w'_k) and w''_j of size $1 - w'_k$.
Check if there is more than one weak edge in the (updated) connected component that contains j, k and l. If it does, repeat (3) until every component contains at most one weak edge. The resulting packing has its associated graph containing one fewer loops (that are not weak). This process will stop after a finite number of steps.

The resulting bin packing using Algorithm B has a associated graph G with no cycle and each connected component having at most one weak edge. In addition, if there is a loop which is not weak, then all other components have no weak edges.

Suppose the total weight $w = \sum_i w_i$ is greater than or equal to n, the number of types. From the reduction steps in the algorithm, G can have at most $n - 1$ ordinary edges and there is at most one weak loop. Since the total weight is at least n, there is at least one loop that is not weak. Thus there is no weak edge outside of the connected component A that contains the loop. In A, there is at most one weak edge. So altogether, there is at most one weak edge. This implies that the number of bins is exactly $\lceil w \rceil$ which is optimum. When $w < n$, we can still use Theorem 2 to show the resulting packing is within a factor of 3/2 of the optimum.

We have proved the following:

THEOREM 4. *Algorithm B generates bin packing that is optimum if the total weight is at least as large as the number of types. In general, the bin packing using Algorithm B has size within a factor of* 3/2 *of the optimum.*

Here we will given an example which shows that Algorithm A and B can generate bin packing with the number of bins off by a factor $(3/2 + o(1))$ of the optimum.

Suppose that k is an integers. We are given a list W of weights where the first $2(k+1)$ weights are of size $k/(k+1)$ and then the next $2(k+1)$ weights are of size $1/(k+1)$.

Using Algorithm A or B, we will end up with a packing which uses the first $2k$ bins to pack the first $k+1$ weights fully without any waste. Then the next group of bins each contain two weights of size $1/(k+1)$. Altogether, $3k+1$ bins are used. Nevertheless, the optimum packing consists of $2(k+1)$ bins each contain one weight of size $k/(k+1)$ and one weight of size $1/(k+1)$. Thus we have the ratio

$$\frac{\#bins\ by\ Alg\ A}{Opt} = \frac{3k+1}{2(k+1)} = \frac{3}{2} - \frac{1}{k+1}$$

which is arbitrarily close to 3/2 when k is large.

8. DYNAMIC MEMORY ALLOCATION

So far we have only dealt with approximation and exact algorithms for static memory allocation. How can we get good dynamic memory allocation algorithms that maintain overall efficiency? First, assume that all nodes in the data structure are of the same size and that we can do compaction [12] so that we move nodes around without impacting the correctness of the search process. For example, this can be done by storing backpointers to all other nodes that refer to the node in slow memory maintained by software and by automatically adjusting pointers without affecting search [12].

We simply maintain an invariant such that each stage uses at most one partially filled (PF) bin, where a PF bin is one which is not completely full. This prevents the case when a single stage is allocated a lot of memory at all stages and then proceeds to deallocate all but a little memory at all stages, thereby "wasting" a large number of read ports and causing poor efficiency. This invariant can easily be maintained by compaction. For example, when a deallocate is done that violates this invariant, the stage that has a PF bin must have another PF bin from which we can relocate (compact) the newly created PF bin.

Maintaining this invariant requires at most one node copy for each node allocate or deallocate and at most doubles the update time. More importantly, it guarantees that each stage wastes at most one Read Port and so the overall number of read ports is at most n. Since each empty or near-empty bin must waste both its Read Ports in order not be unsable, the number of near empty stages when memory runs out is at most $n/2$. Thus the overall efficiency is at least $1 - n/2b$ where b is the number of bins. For example, if $n = b$, the efficiency is at least 50%. It can be improved by further increasing the number of bins.

These ideas can be extended to nodes that have sizes that are powers of two or even arbitrary sizes by using a compacting version of the buddy system together with the invariant we just described. For node sizes that are powers of two we obtain the same efficiency as above (this is an important case for router tries that use power of two node size). For arbitrary node sizes, there is a further factor of two loss for rounding up a node size to the closest power of two.

While there are clearly deeper algorithms for this purpose that we are studying (akin to the optimal algorithms above), we hope that these simple dynamic algorithms convince the reader that 2-port memory algorithms can be extended easily to the dynamic case albeit with some loss in efficiency below 100%.

9. CONCLUSIONS

When all the theory is said and done, what are the practical lessons? The most important is that it is possible to share memory across parallel stages in an almost perfect manner (regardless of individual demands) if we use *two-port* instead of one-port memories, each of which can be assigned to a stage using some form of partial crossbar switch. In practice, one would simply choose the parameters such that the number of memories is larger than the number of processor stages. In that case, the approximation algorithm we presented will provide 100% efficiency.

In essence, we are finessing a difficult problem (allocating across 1-port memories) by changing the model. The new models are practical. We know at least one implementation of Model 4 (in fact, this paper was abstracted from this second design) that scales to multiple OC-768 speeds. On the theoretical front, we can consider the general case of packing bins so that each bin contains at most r bins for some fixed integer r. In this case, we can formulate the associated *hypergraphs* of a packing instead of just associated graphs.

10. ACKNOWLEDGEMENT

The authors would like to thank John Holst of Procket Corporation who built hardware to implement the model described in this paper, and whose initial ideas about memory allocation was the genesis of this paper.

11. REFERENCES

[1] A. Basu and G. Narlikar, Fast Incremental Updates for Pipeline Forwarding Engines, InfoCom 2003.

[2] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1976.

[3] Guy E. Blelloch Phillip B. Gibbons Yossi Matias *Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, volume 8, 1997.

[4] D. Culler, J, Singh, and A. Gupta. *Parallel Computer Architecture, A Hardware/Software Approach*, Morgan Kaufman , 1999.

[5] M. Degermark, A. Brodnik, S. Carlsson, and Stephen Pink, Small forwarding tables for fast routing lookups, *Proc. SIGCOMM*, (1997), 3-14.

[6] M. R. Garey and D. S. Johnson, *Computer and Intractability, A Guide to the Theory of NP-completeness*, W. H. Freeman and Co., San Francisco, 1979.

[7] M. R. Garey and D. S. Johnson, *Complexity results for multiprocessor scheduling under resource constraints*, SIAM J. Comput., (1975), 397-411.

[8] A. Ranade, *How to emulate shared memory.* Journal of Computer and System Sciences, 42:307-326, 1991.

[9] B. Rau, *Pseudo-randomly interleaved memory.* In Proceedings Int. Symp. on Computer Architecture, 1991.

[10] T.V. Lakshman and D. Staliadis, *High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching*, Proc. ACM SIGCOMM '98, 1998.

[11] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, *Survey and Taxonomy of IP Address Lookup Algorithms*, IEEE Network, March/April 2001.

[12] S. Sikka and G. Varghese, *Memory Efficient State Lookups with Fast Updates*, in Proceedings of SIGCOMM 2000, August 2000.

[13] L. Valiant, *A bridging model for parallel computation.* Communications of the ACM, 33(8), 1990.