Theory Comput. Systems **39**, 829–849 (2006)
DOI: 10.1007/s00224-006-1249-3

Theory of Computing Systems

Parallelism versus Memory Allocation in Pipelined Router Forwarding Engines*

Fan Chung,[1] Ronald Graham,[2] Jia Mao,[2] and George Varghese[2]

[1]Department of Mathematics, University of California, San Diego,
La Jolla, CA 92093, USA
fan@ucsd.edu

[2]Department of Computer Science and Engineering, University of California, San Diego,
La Jolla, CA 92093, USA
graham@ucsd.edu; {jiamao,varghese}@cs.ucsd.edu

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 $\frac{3}{2}$ asymptotically. The proof utilizes a new bin packing model, which is interesting in its own right. Further, for important special cases that arise in practice a more sophisticated modification of this approximation algorithm is in fact optimal. We also discuss the online memory allocation problem and present fast online algorithms that provide good memory utilization while allowing fast updates.

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

* The research of Fan Chung was supported in part by NSF Grants DMS 0100472 and ITR 0205061. The research of Ronald Graham was supported in part by NSF Grant CCR 0310991.

and maximize speed, each memory should be read by exactly one process. Unfortunately, if the tasks assigned to processors vary widely in memory usage, this is not an efficient use of memory, since for some tasks memory of one processor may be unused while memory of another processor 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 [6], [11]. 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 [13] 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 they 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? To our best knowledge, this problem was first raised and left as an open problem in [14].

An approximate solution to the problem of trie memory allocation across pipeline stages is described in [1]. Basu and Narlikov try 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 worst-case designs. Their worst-case bound is close to the naive bound of requiring each stage memory equal to the total required memory.

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 memories are (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.

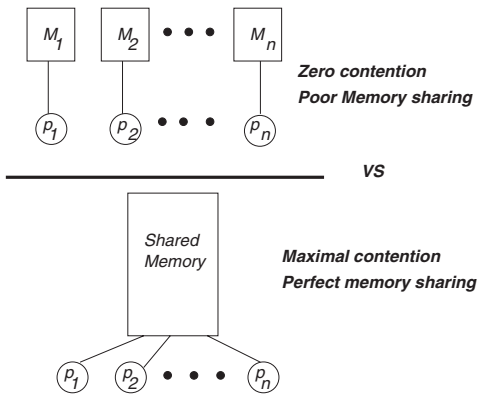

Fig. 1. Models 1 and 2 have problems: strictly partitioned memories have poor memory sharing while a single shared memory has poor contention.

When faced with two unacceptable extremes, it is natural to consider intermediate forms. Thus, strictly partitioned one-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 one-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 one port). Such interconnection networks are commonly used in parallel computers [5] and are called crossbar switches.

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 the dashed line in Figure 2) reconfigures the crossbar to allocate M_3 to p_1.

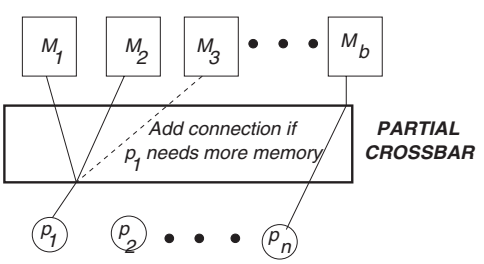

Fig. 2. Model 3: allowing memory sharing by connecting a *large* number of *one-ported* memory banks to the set of n processors via a partial crossbar.

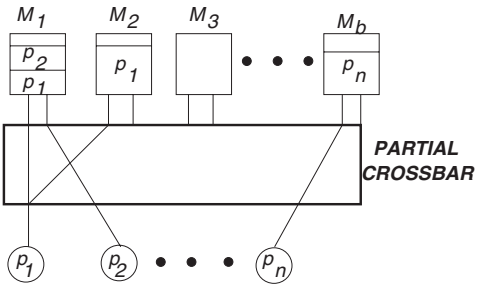

Fig. 3. Our final model: allowing memory sharing by connecting a *small* number of *two-ported* memory banks to the set of n processors via a partial crossbar.

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 one memory, which is negligible for large b. Thus the percentage of wasted memory is at most $(n-1)/b$. 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 zero by increasing b.

While this looks superficially attractive, in practice one does not want to waste even 10% 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 one-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 two-port memories will only slow down access speeds (using say time multiplexing) by a factor of at most 2. However, what kind of memory utilization would such two-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 two-port memories. Each memory has two ports that can be allocated to any two processors. Thus each memory can be read by at most two 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 that 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 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. However, this example should indicate that it is unclear whether perfect memory allocation is possible while respecting the two-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., five words for processor 1, ten for processor 2, etc.). We say that an allocation is feasible if every processor's request is satisfied and 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 the 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.

As often happens, 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 that 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 $\frac{3}{2}$ of optimal asymptotically. 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 $\frac{3}{2}$. 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 two-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 the former part of this 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. In the latter part of this paper we formalize this tradeoff and describe online algorithms for the dynamic case that work 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* [10] 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 [15] and Ranade [9], 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 three memory accesses go to the same memory is > 0.805. In other words, there is an 80% chance that at least three memory accesses go to at least one memory.

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

2. Abstracting the Problem

Here is the formulation of 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 positive 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 kth 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 show that the above problem is indeed NP-complete and thus is probably computationally intractable [4], [12].

Then we proceed to discuss approximation algorithms. We consider a fast and robust algorithm that gives approximate solutions in linear time (in n). The solution this algorithm gives is optimal 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 $\frac{3}{2}$ of the optimum. Several examples are given to indicate the sharpness of this worst-case performance ratio.

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 [8]):

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 three elements from A)?

Garey and Johnson [7] showed the 3-PARTITION problem is NP-complete by using transformation from the problem of three-dimensional matching. In fact, they showed that the 3-PARTITION problem is NP-complete in the strong sense (see [8]).

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 cannot be partitioned into more than two 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 $\frac{1}{4}$ since $w_1 < \frac{3}{4}$. The bin that contains this small part can contain another part with weight at most $\frac{3}{4}$. Thus, this bin cannot 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 cannot 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 analysis, 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 arcs of G_p correspond to the bins in one-to-one fashion, where an arc can either be an edge or a loop. 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 edge between the two types.
(3) If the bin is partially filled with only one type, we say the corresponding loop is *weak*. If the bin is partially filled with two types, we say the corresponding edge is *weak*.
(4) If the bin is completely filled with only one type, we say the corresponding loop is *strong*. If the bin is completely filled with two types, we say the corresponding edge is *strong*.

For example, suppose that $W = (\frac{1}{2}, \frac{2}{3}, \frac{1}{4})$ 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. 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.

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 edge with two distinct endpoints. A graph that contains no cycle is a forest plus some possible loops.

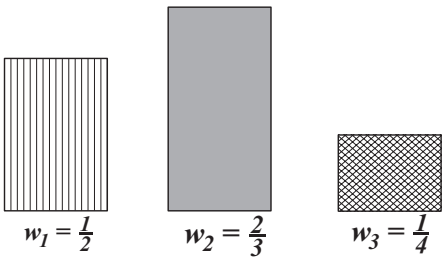

Fig. 4. Weights of three types.

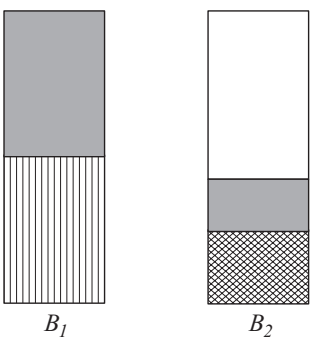

Fig. 5. A packing P.

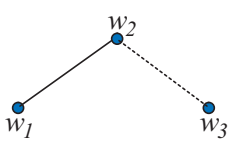

Fig. 6. The graph G_P.

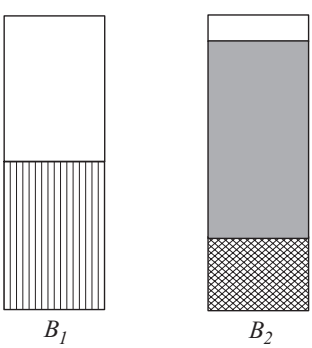

Fig. 7. Another packing Q.

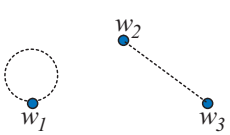

Fig. 8. The graph G_Q.

5. 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. We call an empty bin a *new bin*. We call a partially filled bin with only one type a *live bin* (i.e., a weak loop in the associated graph).

Algorithm A

For a given list of weights $W = (w_1, w_2, \ldots, w_n)$, we pack greedily as follows:
For each $i = 1, 2, \ldots, n$, we place the maximum possible part of w_i into a *live bin* if possible; otherwise, put it into one or more *new bins*.

After we have processed all n weights, the resulting bin packing is valid and satisfies the following properties:

(1) Each connected component is a path with possibly some loops.
(2) There is altogether at most one weak loop (i.e., one *live bin*) which possibly appears at the end of the last connected component formed during the algorithm.
(3) Each connected component except for the last one has at most one weak edge which can only appear at the end of the component.

These properties are intuitive and easy to verify. We omit the proofs. Let OPT denote the number of bins used in an optimal packing. We will show that the packing generated by the above algorithm has an asymptotic worst-case approximation ratio of 3/2.

Theorem 2. *We are given a list of weights W and a packing P of list W in which no bin contains more than two types of weights. Suppose that the associated graph G_P satisfies the three properties above. Then*

$$|P| \leq \tfrac{3}{2}(1 + o(1))OPT$$

as $w \to \infty$.

Proof. Suppose the list $W = (w_1, w_2, \ldots, w_n)$ has a 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

$$OPT \geq \max\{w, n/2\}. \tag{1}$$

Our proof needs the following strengthening of the above inequality:

Claim 1.

$$OPT \geq \max\{w, w^*/2\}, \tag{2}$$

where $w^* = \sum_i \lceil w_i \rceil$. 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.

$$|P| \leq \frac{w + w^* + 1}{2}.$$

Furthermore, we claim

Claim 3.

$$\frac{w + w^* + 1}{2} \leq \frac{3}{2}(1 + o(1)) \max\left\{w, \frac{w^*}{2}\right\}.$$

If all three claims hold, we have

$$|P| \leq \tfrac{3}{2}(1 + o(1))OPT,$$

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 $2 \cdot OPT \geq \sum_i \lceil w_i \rceil$ 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 strong loop.) Let P' denote the packing of the list of weights W' which is the same as W except that $w'_1 = w_1 - 1$. We proceed by induction and suppose it is true for P' (which has a smaller number of bins), i.e.:

$$|P'| \leq \frac{w' + (w')^* + 1}{2}.$$

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

$$|P| \leq \frac{w + w^* + 1}{2}.$$

We may now assume that each filled bin involves weights of two types.

Consider any connected component X of G_P. We let $w_X = \sum_{i \in X} w_i$ and $w^*_X = \sum_{i \in X} \lceil w_i \rceil$. Let $\nu(X)$ denote the number of vertices in X and $|X|$ the number of bins used in X.

Case a: *There is no weak loop in* X. We have $|X| = \nu(X) - 1$. On the other hand, $w_X > \nu(X) - 2$ since the sum of weights is more than the capacity of $\nu(X) - 2$ filled

bins. We also have $w_X^* \geq \nu(X)$. Therefore

$$w_X + w_X^* \geq w_X + \nu(X) > 2\nu(X) - 2 = 2|X|.$$

Case b: *There is one weak loop in X*. We have $|X| = \nu(X)$. On the other hand, $w_X > \nu(X) - 1$ since the sum of weights is more than the capacity of $\nu(X) - 1$ filled bins. We also have $w_X^* \geq \nu(X)$. Therefore

$$w_X + w_X^* \geq w_X + \nu(X) > 2\nu(X) - 1 = 2|X| - 1.$$

Now sum up the inequalities for all connected components. Because there is at most one component with one weak loop, we have

$$|P| \leq \frac{w + w^* + 1}{2}.$$

So, Claim 2 is proved.[1] □

Proof of Claim 3. We want to show that

$$\frac{w + w^* + 1}{2} \leq \frac{3}{2}(1 + o(1)) \max\left\{w, \frac{w^*}{2}\right\}.$$

We consider two cases:

Case a: $w^*/2 \leq w$. We then have

$$\frac{w + w^* + 1}{2} \leq \frac{3w + 1}{2} \leq \frac{3}{2}(1 + o(1))w$$

as $w \to \infty$.

Case b: $w^*/2 > w$. It follows that

$$\frac{w + w^* + 1}{2} < \frac{3w^* + 2}{4} \leq \frac{3}{2}(1 + o(1))\frac{w^*}{2}.$$

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* $\frac{3}{2}$ *of the optimum asymptotically.*

[1] Strict inequality holds here but we do not need it for this theorem.

It is worth noting that if the associated graph of the resulted packing does not have any weak loop, Algorithm A is exactly at most $\frac{3}{2}$ from optimal. It is also clear that Algorithm A runs in time $O(n)$, where n is the number of types.

6. Some 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 algorithm to be discussed in the next section.

An associated graph G is is **stable** if and only if all of the following conditions hold:

- G has no cycle.
- Each connected component has at most one weak arc.
- G has at most one weak loop in total.

Given that G has no cycle, each connected component is a tree with some possible loops. A crucial observation is that if there is a weak edge in this connected component, we can move it freely within this component. During the moving process, we might split the original component into two, but the total number of bins will never increase.

We describe a few atomic repacking operations here as follows. None of them requires more bins than originally given in the packing. None of them creates cycles or strong loops in the associated graph.

Operation 1. If a strong edge $e_1 = (i, j)$ and a weak edge $e_2 = (j, k)$ are adjacent (sharing one type j) in a component, we can repack weights so that e_1 becomes weak and e_2 becomes strong, or split the component into two with one having a weak loop e_1 and the other having an edge e_2 which can be strong or weak.

Operation 2. If two weak edges $e_1 = (i, j)$ and $e_2 = (j, k)$ are adjacent in one component, we can repack weights so that e_2 becomes strong and e_1 stays weak, or split the component into two with one having a weak loop e_1 and the other having an edge e_2 which can be strong or weak.

Operation 3. If a weak loop $e_1 = (i, i)$ and a weak edge $e_2 = (i, j)$ are adjacent in a component, we can repack weights so that we only have a single edge e_2 (i.e., we eliminate one bin e_1), or a strong edge e_2 and a weak loop e_1.

Operation 4. If two weak loops $e_1 = (i, i)$ and $e_2 = (j, j)$ are in two separate components C_1 and C_2, we can repack weights to merge them into one component C such that we only have a single edge e_2 (i.e., we eliminate one bin e_1), or a strong edge e_2 and a weak loop e_1.

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 has a connected component C which contains two weak edges and the rest of the*

graph is stable, we can find another packing P' which uses no more than b bins with its entire associated graph stable.

Proof. Suppose a connected component of G_P contains two weak edges e_1 and e_2. The two weak arcs cannot have the same vertices, since this would form a 2-cycle, contradicting our initial hypothesis. There must be a unique path with no loops (that is, a sequence of edges so that two consecutive edges share a common vertex), say, with edges $e_1 = f_1, f_2, \ldots, f_t = e_2$. Here e_1 and e_2 are weak edges while all other f_i's are strong edges.

Select either weak edge to repack, say e_1, and proceed with Operation 1 one step at a time in order to bring the two weak edges closer together. Now we have two cases:

If we successfully carry this on until the two weak edges become adjacent, we then use Operation 2 to eliminate one weak edge or split the component into two. In the latter case, we need to check whether there are two weak loops in the entire graph. Operation 4 is needed if this is true. Now the graph is stable.

If during the moving process the component splits into two, one component only has at most one weak edge while the other component actually has one weak edge and one newly formed weak loop. For the latter component C', we need to check if there are two weak loops in the entire graph:

Case a. If this is true, Operation 4 is needed first to eliminate the extra weak loop. This will possibly result in another weak edge in this smaller component and now it contains two weak edges. Notice, however, now these two weak edges are closer compared with the original two weak edges in C. We carry out Operation 1 recursively in this case.

Case b. Otherwise, we select the weak edge in C' and move it towards the weak loop the same way as Operation 1 recursively. Use Operation 2 or 3 when the two partially filled bins become adjacent in the graph.

This process will stop in a finite number of steps and the graph will become stable. □

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 weights of more than two types. If the associated graph G_P contains a strong loop in one connected component X and a weak edge in another connected component Y and is stable, we can find another packing P' which uses no more than b bins with its associated graph with one fewer strong loops than packing P and also stable.*

Proof. Suppose in X there is a loop that is strong (associated with a filled bin, say e_1, in one type j) and suppose that there is weak edge $\{k, l\}$ (associated with a partially filled bin, say e_2) in another component Y. We reconfigure the two bins as follows:

Suppose e_2 contains parts of weights w'_k and w''_l. We partition the weight of type j in e_1 into two parts w'_j (of size the same as w'_k) and w''_j of size $1 - w'_k$ and switch the parts w'_k and w'_j.

Check if there is more than one weak edge in the newly formed connected component that contains j, k and l. If it does, use the steps as described in Lemma 1 until the graph

is stable. The resulting packing has its associated graph containing one fewer strong loop. □

7. An Improved Algorithm

In this section we consider a modified version of the simple approximation algorithm given in Section 5. We will show that the modified algorithm gives an optimal 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 $\frac{3}{2}$ of the optimum asymptotically.

Before we introduce the improved Algorithm B, we first consider an intermediate form of it, say Algorithm A′, which takes the output packing of Algorithm A in Section 5 as input and processes it using the following steps:

Algorithm A′

For a given list of weights $W = (w_1, w_2, \ldots, w_n)$, we use Algorithm A to generate a valid packing P.

While there exists a component X containing a strong loop and another component Y containing a weak edge, we use the steps as described in the proof of Lemma 2 to merge these two components into one.

The resulting bin packing using Algorithm A′ has an associated graph G with no cycle and each connected component having at most one weak edge. In addition, if there is a strong loop, 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 $n - 1$ 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 strong. Thus there is no weak edge outside of the connected component C that contains the loop. In C, 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 $\frac{3}{2}$ of the optimum.

We have proved the following:

Theorem 4. *Algorithm A′ generates a bin packing that is optimal if the total weight is at least as large as the number of types. In general, the bin packing using Algorithm A′ has size within a factor of $\frac{3}{2}$ of the optimum asymptotically.*

Now we consider the complexity of Algorithm A′. We note the following:

- Algorithm A produces a *stable* packing P in $O(n)$ time.
- During the execution of the *while* loop, the number of *strong* loops is strictly decreasing. Every time we eliminate one strong loop, at most a linear number of atomic operations are involved, each taking constant time.

Therefore, Algorithm A′ runs in time $O(n^2)$ at most, where n is the number of types. In fact, if we are a little more careful about the order of the atomic operations applied, we can achieve a linear time algorithm. To describe it, we need the following lemma:

Lemma 3. *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 is a forest where some or all of the components have $k \geq 2$ weak edges but otherwise stable, we can find another packing P' which uses no more than b bins with its associated graph stable in linear time of n.*

Proof. Let X be a component which has $k \geq 2$ weak edges. Pick an arbitrary vertex v in X to be the root. If we traverse X using DFS, there is a natural order $O = \{v_1, v_2, \ldots, v_n\}$ of the vertices defined by the last visiting time. In other words, the time when v_1 is last visited is earlier than the time when v_2 is last visited and so on.

For $i = 1, \ldots, n$, we consider all the edges that are adjacent to v_i, say $\{e_1, \ldots, e_s\}$ in which e_1 is the only edge that is closer to the root. If two or more of the other $(s-1)$ edges are weak, we can use Operation 2 to merge them in a pairwise manner. This process could result in just one weak edge, say e_j, in which case we use Operation 2 or 1 to push the weak edge towards the root, or it may split this component X into smaller trees but the total number of weak edges will never increase. Whenever there are two or more weak loops in the entire graph, we use Operation 4 to eliminate the extra weak loops immediately.

Any component X that has $k \geq 2$ weak edges needs to be processed in this way. If it splits during the process, any newly formed component that has $k \geq 2$ weak edges will also be processed. The total number of atomic operations needed until no more such component exists is linear in n.

This proves our lemma. □

Such repacking also does not create cycles or more strong loops. Now we have our linear time Algorithm B:

Algorithm B

For a given list of weights $W = (w_1, w_2, \ldots, w_n)$, we use Algorithm A to generate a valid packing P.

While there exists a component X containing a strong loop and another component Y containing a weak edge, we use only the first step as described in the proof of Lemma 2 to merge these two components into one, without taking care of the possible multiple edges in any one connected component.

After the while loop, we use the steps as described in the proof of Lemma 3 to eliminate the extra weak edges in each component.

Algorithm B produces packings as good as Algorithm A′ with the improvement that its running time is linear instead of quadratic. Therefore we have the following theorem:

Theorem 5. *In $O(n)$ time, Algorithm B generates a bin packing that is optimal 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 $\frac{3}{2}$ of the optimum asymptotically.*

Here we give an example which shows that Algorithms A and B can generate bin packings with the number of bins off by a factor $(\frac{3}{2} + o(1))$ of the optimum.

Suppose that k is an integer. 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 $2(k+1)$ 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

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

which is arbitrarily close to $\frac{3}{2}$ when k is large.

8. Dynamic Memory Allocation

So far we have only dealt with approximation and exact algorithms for static memory allocation. On the more practical side, how can we get good dynamic memory allocation algorithms that maintain overall efficiency? Upon each new memory request, allocate, or deallocate, are we allowed to repack previously assigned memory units besides handling the new request? In this situation we have a tradeoff between memory utilization and cost of repacking or compaction [14]. To capture and analyze this tradeoff, we define the following parameter for compaction cost efficiency.

Definition 1. The compaction ratio of any online memory allocation Algorithm A is defined to be

$$\begin{aligned}\gamma_{\mathrm{A}} &= \max_t \frac{M}{W} \\ &= \max_t \left(\frac{\text{total moved memory units up to time } t}{\text{total memory units allocated up to time } t} \right).\end{aligned}$$

Case a: *γ can be arbitrarily large.* If repacking or compaction is assumed to be of negligible cost and we have unlimited computing power, we can just solve the offline allocation problem every time a new memory request comes in using the best approximation algorithm and perform repacking whenever needed. In practice, however, compaction almost certainly has a cost that is not negligible, especially because it has to perform memory operations. Computation cost should also be taken into consideration since we have already seen that the offline allocation problem is NP-hard.

Case b: *γ is bounded from above.* In particular, $\gamma \leq c$ where $c > 0$ is a constant.

We now give an online allocation algorithm subject to the two-port constraint with compaction ratio $\gamma \leq 1$.

Algorithm C

For any memory allocation request w_i, we say it is *large* if $w_i > \frac{1}{2}$, otherwise we say it is *small*. During the process:

- A large allocation request is always put into a new bin.
- A small request is put into a bin with only one small request if possible, otherwise it is put into a new bin.
- Upon a deallocation request, if it results in two bins each with only one small request, move the *smaller* piece to double up the two requests.

Theorem 6. *Algorithm C always generates bin packing of size $|C|$ which satisfies*

$$|C| \leq \tfrac{3}{2}OPT + 1.$$

Proof. At any point of time, we can describe the packing produced by Algorithm C as t bins each with one large request, r bins each with two small requests, with possibly one more bin with only one small request. Given these requests, how much better can the optimal packing strategy be?

First, let us suppose there is no bin with only one small request. Algorithm C uses $(t + r)$ bins.

Claim 4.

$$OPT \geq \begin{cases} r + t/2 & \text{when} \quad r \geq t/2, \\ \frac{2}{3}(r + t) & \text{when} \quad r < t/2. \end{cases}$$

Proof of Claim 4. The first inequality $OPT \geq r + t/2$ always holds because of the two-port constraint, since we have a total of $(2r + t)$ different types of weights. Exact bound could be achieved when each large weight can be paired up with a small weight into one bin and $t = 2r$.

When $r < t/2$, not all large weights can be paired up. The paired up ones would use $2r$ bins and the remaining $(t - 2r)$ large weights have to occupy at least $\frac{2}{3}(t - 2r)$ bins (proved in Claim 5). This adds up to a total of $\frac{2}{3}(r + t)$ bins and proves Claim 4.

Case a: $r \geq t/2$.

$$\begin{aligned} \frac{|C|}{OPT} &\leq \frac{r + t}{r + t/2} \\ &= 1 + \frac{t/2}{r + t/2} \\ &\leq 1 + \tfrac{1}{2} \\ &= \tfrac{3}{2}. \end{aligned}$$

Case b: $r < t/2$.

$$\frac{|C|}{OPT} \leq \frac{r+t}{\frac{2}{3}(r+t)}$$
$$= \tfrac{3}{2}.$$

In the case where there is one bin having a single small request, the previous argument applies replacing $|C|$ by $|C| - 1$. It remains to prove Claim 5.

Claim 5. *Given m large weights, it is impossible to pack them into fewer than* $\lceil \frac{2}{3}m \rceil$ *bins.*

Proof of Claim 5. m bins would be necessary if we do not split any weights. Suppose we break α of the m large weights each into two small pieces. Then we have a total of

$$2\alpha + (m - \alpha) = m + \alpha$$

pieces in which $(m - \alpha)$ pieces are still large and need $(m - \alpha)$ bins. Therefore we have

$$OPT \geq \max\{(m + \alpha)/2, m - \alpha\}.$$

The minimum of the right-hand side occurs when $(m + \alpha)/2 = m - \alpha$ and is 2/3m. This proves our claim.

This completes the proof of Theorem 6. □

Theorem 7. *Algorithm C has compaction ratio* $\gamma_C \leq 1$.

Proof. Call any memory piece that has not been swapped *clean* and otherwise *dirty*. Recall the compaction ratio is defined as $\gamma = M/W$. We claim that

$$W - M \geq \sum \text{size of all clean pieces.}$$

This is because

- Before any deallocation request, our bin packing consists of only *clean* pieces which contribute to W.
- If any clean piece is moved due to a deallocation request, it becomes *dirty* and contributes to the numerator M.
- If a dirty piece is deallocated, there is no effect on $W - M$.
- In a shared bin of one clean piece and one dirty piece, the clean piece always has size no smaller than the dirty piece according to our algorithm. When the clean piece is deallocated, the weight it contributes to W is still there, so instead of throwing it away, we use its weight to *recharge* the dirty piece into a clean piece. This would also guarantee that no dirty piece will ever be moved again.

Note that $\sum$ size of all clean pieces ≥ 0 at all times so $W - M \geq 0$, i.e., $\gamma = M/W \leq 1$.

We also observe that 1 is a tight bound for γ_C using the following memory request sequence. We begin with two allocation requests, each with weight $\alpha < \frac{1}{2}$. We then allocate one more weight of size α, and follow this with a deallocation of one of the paired weights. This will cause the algorithm to move a weight. We repeat this process until it has been performed s times altogether. Thus, we will have allocated $s+2$ weights and moved s weights, all of size α. This clearly gives compaction ratio arbitrarily close to 1 as s goes to infinity. □

Case c: $\gamma = 0$, *i.e., no compaction is allowed.* When the compaction cost is relatively high and we cannot afford to swap any memory bits, we first claim that any online Algorithm D in this case cannot have an approximation ratio better than 2. This can be shown using the following memory request sequence.

For a large integer k, we are first given a list of k memory allocation requests each with size $\frac{1}{3}$. Observing the allocation assignment made by the online Algorithm D, we are then given a list of $\lfloor k/2 \rfloor$ deallocation requests which remove exactly one weight from every shared bin. At the end, the online algorithm will have each bin holding one weight while the optimum packing uses only half as many bins. This would give us the lower bound of 2 on the approximation ratio.

A simple online algorithm that achieves approximation ratio 2 in this case is just never to share any bin. Given the two-port constraint, the bin packing generated by this algorithm is always within a factor of 2 from the optimum.

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 one-port memories) by changing the model. The new models are practical. We know at least one implementation of one of our models that scales to multiple OC-768 speeds. On the theoretical front, our paper also poses an interesting open problem for the general case of packing bins so that each bin contains at most r types for some fixed integer r. In this case we can formulate the associated *hypergraphs* [2] of a packing instead of just associated graphs and our techniques used in this paper can possibly be extended to derive efficient algorithms in those contexts.

Acknowledgments

The authors thank John Holst of Procket Corporation who built the hardware to implement the model described in this paper and whose initial ideas about memory allocation were the genesis of this paper, Nan Zang for useful comments on one of our examples, and the anonymous reviewers for their helpful comments.

References

[1] A. Basu and G. Narlikar, Fast incremental updates for pipeline forwarding engines, *Proc. InfoCom* 2003, vol. 13, issue 3, pp. 690–703.

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

[3] G. E. Blelloch, P. B. Gibbons, Y. Matias, and M. Zagha, Accounting for memory bank contention and delay in high-bandwidth multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, 8:943–958, 1997.

[4] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, Approximation algorithms for bin packing: a survey, in *Approximation Algorithms for NP-Hard Problems*, D. Hochbaum (ed.), PWS, Boston, MA, 1996, pp. 46–93.

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

[6] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *Proc. SIGCOMM*, 1997, pp. 3–14.

[7] M. R. Garey and D. S. Johnson, Complexity results for multiprocessor scheduling under resource constraints, *SIAM Journal on Computing*, 4:397–411, 1975.

[8] M. R. Garey and D. S. Johnson, *Computer and Intractability, A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

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

[10] B. Rau, Pseudo-randomly interleaved memory, *Proc. Int. Symp. on Computer Architecture*, 1991, pp. 74–83.

[11] T. V. Lakshman and D. Staliadis, High speed policy-based packet forwarding using efficient multi-dimensional range matching, *Proc. ACM SIGCOMM* '98, 1998, pp. 203–214.

[12] J. M. Robson, Storage allocation is NP-hard, *Information Processing Letters*, 11(3):119–125, 1980.

[13] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network Magazine*,15(2):8–23, March/April 2001.

[14] S. Sikka and G. Varghese, Memory Efficient State Lookups with Fast Updates, *Proc. SIGCOMM* 2000, pp. 335–347, August 2000.

[15] L. Valiant, A bridging model for parallel computation, *Communications of the ACM*, 33(8):103–111, 1990.

Received September 27, 2004, *and in final form April* 19, 2006. *Online publication September* 27, 2006.