The Multicast Address Allocation Problem: Theory and Practice

Daniel Zappala, Virginia Lo, and Chris GauthierDickey
Computer and Information Science, University of Oregon, Eugene, Oregon 97403-1202
zappala@cs.uoregon.edu, lo@cs.uoregon.edu, chrisg@cs.uoregon.edu

***Abstract*— In this paper, we perform the first comprehensive study of the multicast address allocation problem. We analyze this problem both within its context as a classic resource allocation problem and with respect to its practical use for multicast address assignment. We define a framework for the problem, introduce complexity results, and formulate several new allocation algorithms. Despite the theoretical superiority of these algorithms, our performance evaluation demonstrates that a common, prefix-based algorithm is better under a range of workloads. We conclude by illustrating the conditions under which dynamic address allocation should be used and provide insight into how to further improve the performance of prefix-based allocation.**

I. INTRODUCTION

Multicast address allocation is one of several obstacles that has slowed multicast deployment. The multicast infrastructure built using Deering's original IP multicast model [1] – now referred to as Any Source Multicast (ASM) – requires that applications share a single, global address space. In this model, a multicast address identifies a logical group of members and any source may send data to this dynamic set of members at any time. No two applications may share the same multicast address at the same time, or else the group members may receive unwanted traffic. While the operating system or the application can easily screen out the unwanted traffic, this may result in significant network overhead.

The key problem for ASM multicast address allocation is to assign a unique address to each application from a globally-shared address space. Because the address space is limited, addresses must be re-used over time. We refer to this problem as the multicast address allocation or *malloc* problem.

To help address the malloc problem, Kumar et al. developed the MASC address allocation architecture [2], which dynamically allocates addresses along the provider-subscriber hierarchy that is present in the Internet. In MASC, a domain claims a range of addresses from its parent, then allocates these addresses to hosts within its domain as well as to its child domains. While MASC (and the rest of the hierarchical allocation architecture) has never been deployed, it represents the best current solution for allocating addresses with the current ASM infrastructure.

Separately from these concerns, the malloc problem remains interesting because it is an instance of a well-known general resource allocation problem, in which a block of resources is allocated and de-allocated based on dynamic requests for sub-blocks of varying sizes. At this level, the malloc problem is nearly identical to that of processor allocation in hypercube computing architectures. Other related problems arise in the areas of logic (cube algebras for circuit simplification), memory management (contiguous memory allocation) and disk space management (contiguous file block allocation).

The malloc problem thus merits a deep understanding of its complexity, and potential solutions to this problem require a thorough performance evaluation.

In this paper, we perform the first comprehensive study of the multicast address allocation problem, using insights we have gained from studying processor allocation for hypercubes. There has been very little activity in this area beyond Handley's landmark paper [3] that illustrated the shortcomings of random allocation and laid the foundation for MASC.

In our study, we use a framework for the malloc problem that classifies address allocation algorithms according to their recognition capability: *prefix-based*, *contiguous*, and *non-contiguous* [4]. While a prefix-based algorithm is recommended for MASC, contiguous and non-contiguous algorithms offer a more flexible representation for address blocks and hence provide a greater ability to recognize free blocks in a fragmented space.

First, we derive several complexity results for the malloc problem and explain their implications for address allocation protocols. These results show that address allocation is a subtle and difficult problem, more so than heretofore understood by the networking community. We then develop several new algorithms

for address allocation – one contiguous and one non-contiguous – that use a hypercube model for address aggregation. These algorithms should theoretically outperform prefix-based allocation.

Second, we conduct a performance evaluation to determine whether our non-contiguous allocation algorithm lives up to its theoretical superiority. Our results are quite surprising – prefix-based allocation performs at least as well as, and often better than, non-contiguous allocation. While our non-contiguous algorithm has a greater potential for finding free blocks, in practice it fragments the free space across multiple dimensions, making it difficult to aggregate the free regions into large blocks. The prefix-based algorithm does a better job of consolidating free space as address blocks are released.

Third, we illustrate the conditions under which dynamic allocation algorithms will outperform static allocation. This is an important contribution because providers have considered using static allocation instead of MASC's dynamic mechanism.

Finally, we show how to further improve the performance of prefix-based algorithms by eliminating forced migration and expanding the number of blocks a domain can hold.

II. BACKGROUND

The malloc problem has driven the evolution of multicast address allocation from the current Session Directory tool to the allocation hierarchy defined by MASC and then to several alternative approaches.

A. Session Directory and MASC

Handley and Jacobson developed the first multicast address allocation protocol as a part of their *Session Directory* tool [5]. The basis of this allocation protocol is informed, partitioned, and randomized allocation [3]. The tool listens to the allocations made by other users (informed) and divides the address space according to session scopes (partitioned) that indicate how many hops multicast data should travel. Within a scope, a new address is chosen randomly from among the addresses that have not already been allocated. Reuse over time is enforced by requiring an application to return its addresses when it is done. Handley extensively studied the performance of the session directory address allocation mechanism and concluded that a hierarchical allocation architecture was needed to allocate addresses from a sufficiently large shared space [3].

Because of this work, a group of researchers at USC/ISI and Michigan developed the MASC architecture, which uses the provider-subscriber hierarchy already present in the Internet to dynamically allocate blocks of multicast addresses to domains [2]. A domain running MASC uses a claim-collide protocol to request blocks of addresses from its parent domain and resolve conflicts with any sibling domains trying to claim the same block. A separate set of protocols is used to allocate addresses from these blocks to hosts within the domain. The MASC architecture has been shown to scale to large numbers of domains, with good address utilization [6].

B. Alternative Approaches

In an attempt to solve or perhaps avoid the malloc problem, there has recently been considerable interest in alternative approaches to multicast addressing. The simplest alternative is to use purely static allocation with GLOP [7]. Static allocation is useful when a content provider wants a permanent multicast address assigned to a particular application, such as a long-running television show or stock feed. However GLOP also has a significant limitation – it can allocate only 256 addresses to each domain under IPv4. IPv6 allows both permanent and transient multicast addresses [8]. Because the transient addresses use 112 bits, GLOP can be used to give each domain a large pool of available addresses, and global coordination will not be needed.

A more radical alternative is to switch to an entirely different multicast architecture that avoids the malloc problem altogether. One such proposal is SSM [9], which solves the malloc problem by using source-specific addressing. This essentially gives each host its own multicast address space consisting of 16 million addresses. Finally, pure application-layer approaches [10], [11], [12], [13] offer simpler deployment, but typically have worse performance than ASM with respect to delay, bandwidth, and network stress.

Despite the fact that each of these alternative solutions is readily available, the Internet still uses an ASM infrastructure. One reason for this is a continuing commitment to running native multicast, which is more efficient than application-layer protocols. Another reason why ASM is still run is that there is some resistance from ISPs to what they perceive as constant changes in the IETF's recommended interdomain multicast protocols (having switched from DVMRP [14] to PIM [15], BGMP [16], and now SSM). Finally, changing to SSM means adopting a one-to-many communication model at the network layer, with application-layer protocols required for many-to-many communication. Some users and network administrators may prefer the network-layer many-to-many model defined by ASM.

III. A FRAMEWORK FOR THE MALLOC PROBLEM

Our framework for the malloc problem begins with the definition of an *address expression*, which represents a block, or set, of addresses. We use the standard *don't care* notation of hypercubes for expressions, e.g., the set of four addresses 0000, 0001, 0010, 0011 can be represented as the address expression 00XX, in which the X's represent *don't care* bits. This notation is similar to that of address masks, which are commonly used in Internet routing protocols.

We define the following taxonomy of address expressions, based on the allowable patterns of the *don't care* bits.

- **Prefix-Based**: Address expressions must have all all the *don't care* bits in the rightmost positions.
- **Contiguous**: Address expressions must have contiguous *don't care* bits, with wraparound allowed.
- **Non-Contiguous**: Address expressions may have the *don't care* bits in arbitrary positions.

For example, given a block of 2^4 addresses allocated from a space of 2^8 addresses, 0010XXXX denotes a prefix-based address expression, 01XXXX10 and XX0100XX represent contiguous allocations, and 0X0XX1X0 represents a non-contiguous allocation. Note that each class is contained in the next, with non-contiguous being the most general class.

A. Basic MASC Allocation

The MASC address allocation mechanism uses prefix-based expressions for address blocks and a worst-fit algorithm for new requests. Figure 1 illustrates these basic concepts using a simple two-level hierarchy. The parent domain has been allocated a range of 64 addresses given in dotted-decimal notation as 224.0.0.0/26. Ignoring the first 24 bits, we can represent this as 00XXXXXX, where the X's represent *don't care* bits that can be set to either 0 or 1. Because MASC uses prefix-based expressions, this means that all of the *don't care* bits must be in the rightmost positions. Similarly, child domain A has been allocated 16 addresses, represented as 224.0.0.0/28 or 0000XXXX. Given this situation, we can represent the free addresses in two blocks: 001XXXXX and 0001XXXX. When domain B requests 16 addresses, this request is filled using worst-fit; first, the largest free block (001XXXXX) is chosen, and then the first sub-block of the requested size is selected (0010XXXX). When there are multiple free blocks of the same size, one is chosen at random.

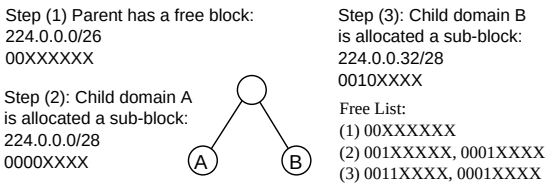


Fig. 1. MASC Allocation Example

B. The Malloc Problem

At the heart of MASC, and indeed any allocation protocol, lies the scheme used for allocation and de-allocation of address blocks. The number of addresses needed by a domain changes over time, and a domain must adjust to this need by allocating and de-allocating addresses to each of its children domains. This fundamental, yet difficult problem, what we refer to as the *malloc problem*, can be defined as follows for a single hierarchy composed of a parent domain and m child domains. The definition is easily extended to a multi-level hierarchy.

The Malloc Problem: A domain is given a contiguous range of 2^n multicast addresses, represented by binary numbers from 0 to $2^n - 1$. Initially, all addresses are available for allocation. Child domains C_0 through C_m request *blocks* of addresses whose sizes are powers of 2. The challenge of the malloc problem is to allocate blocks to child domains under heavy demand, as the address space becomes fragmented over time.

A child domain that requests an additional block of addresses may be satisfied in three different ways:

- **expansion:** A child is given a new block in addition to its current blocks.
- **doubling:** One of the child's blocks is combined with a free *buddy* block, which has the same address expression except for one different instantiated bit. For example, the prefix-based block 00X has one buddy, 01X. Once combined, the block becomes 0XX.
- **migration:** A child exchanges one or more of its blocks for a new block that is as large as all of the old blocks combined. Migration allows a child to move a block to a new area where it can expand by doubling. This helps to keep the total number of blocks assigned to a child within some bound. The old space is released and can be allocated to other children.

A good algorithm can be measured by its ability to successfully allocate addresses while attempting to minimize the number of blocks a child domain holds (to keep routing tables small) and the number of times

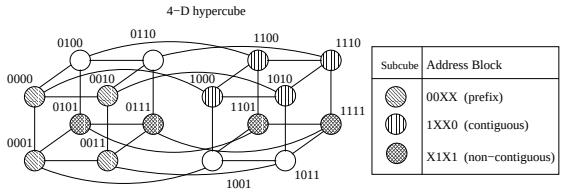


Fig. 2. The correspondence between address allocation and subcube allocation

a child must change addresses (to reduce routing table flux). A good algorithm should also maintain a high level of utilization; that is, it should not waste space by assigning a large number of addresses to a domain that does not need them. Thus, doubling is generally preferable to migration and expansion, but expansion should be used if doubling will result in low utilization.

C. Hypercube Processor Allocation and the Malloc Problem

The hypercube is an elegant recursive mathematical structure that served as the underlying communication network of the Intel iPSC and N-Cube parallel processors back in the late 1980s and early 1990s. In a hypercube, the 2^n processors are each labeled with an n-bit address; processors whose labels differ in exactly one bit position are connected.

A *subcube* is a subset of the nodes and edges of a hypercube that themselves form a smaller hypercube. In a hypercube machine, parallel applications request subcubes, hold them for the runtime of the application, and then release the subcubes back to the operating system scheduler. The algorithm used by the scheduler to handle the requests and releases of the subcubes is the *subcube allocation algorithm* and has been the target of intensive research for many years [17], [18], [19], [20]

Our key observation is the fact that a subcube is equivalent to a block of addresses described by a single address expression. Thus, as shown in Figure 2, a given subcube — or its equivalent block of addresses – can be described using prefix-based, contiguous, or non-contiguous address expressions.

IV. COMPLEXITY OF ADDRESS ALLOCATION

The two crucial operations for an allocation scheme are doubling and migration. Below, we summarize complexity results for the three classes of address allocation schemes – prefix-based, contiguous, and non-contiguous – and discuss their implications for solving the malloc problem.

A. Doubling Complexity

In any prefix-based allocation scheme, there is only one choice for doubling, i.e., doubling can occur only by converting the rightmost instantiated bit to a *don't care* bit. For example, if child domain $C1$ holds address block 000XX, it can only double into the block 00XXX.

In any contiguous allocation scheme, there are two choices for doubling, i.e. by converting either the leftmost or rightmost instantiated bit to a *don't care* bit. For example, if $C1$ holds block 0XX00, it can double into either block XXX00 or block 0XXX0.

The complexity of doubling for prefix and contiguous allocation is $O(C)$, where C is the number of child subdomains associated with a given parent domain. The algorithm simply generates the address expression for the candidate buddy block and then tests whether that block is available by checking for intersection with the other children's blocks via bitwise comparison of address expressions.

In any non-contiguous allocation scheme there are $n - k$ choices for doubling, where n is the total number of bits in the full address space and k is the number of *don't care* bits in the current address expression. Doubling occurs by converting any one of the instantiated bits to a *don't care*.

The complexity of doubling for non-contiguous allocation is $O(C * n)$ since it may have to examine all $n-k$ choices for doubling, testing each for intersection with the other children's blocks.

B. Migration Complexity

The ability of an allocation scheme to migrate to a new block in a highly fragmented address space is a function of its ability to recognize blocks of the desired size in the free address space. We say that an allocation scheme can recognize a block if the scheme can use a single address expression to represent the addresses in the block. We call this ability an algorithm's *recognition capability*.

Table I shows the recognition capacity for a spectrum of subcube allocation schemes all of which can be invoked for the malloc problem [19]. The table gives the general formula for the total number of subcubes/blocks of size 2^k that can be recognized in a hypercube/address space of size 2^n. It is clear that relaxing constraints on the format of the address expression from prefix-based to non-contiguous vastly improves the potential recognition capacity. This potential may not necessarily lead to better migration performance, due to fragmentation. Nevertheless, the

Recognition for n bit address space, k bit subcube/block

Subcube Allocation Scheme	**Total blocks recognized** General Formula	$n = 8, k = 3$
Buddy (prefix)	2^{n-k}	32
Gray (non-contiguous)	2^{n-k+1}	64
Double Gray (non-contiguous)	$\begin{cases} 2^n & \text{if } k = 0 \\ 4 \times 2^{n-1} - n & \text{if } k = 1 \\ 4 \times 2^{n-k} - 1 & \text{if } k = 2 \\ 4 \times 2^{n-k} & \text{if } 2 < k \leq (n-1) \end{cases}$	128
Partners (non-contiguous)	$(n-k+1) \times 2^{n-k}$	192
Cyclic (contiguous)	$n \times 2^{n-k}$	256
Full (non-contiguous)	$\binom{n}{k} \times 2^{n-k}$	1792

TABLE I

RECOGNITION CAPABILITY OF ALLOCATION SCHEMES

increased recognition capability provides strong motivation to explore contiguous and non-contiguous algorithms.

1) Prefix-Based Allocation: Prefix-based allocation was proved to be polynomial time [18]. Under prefix schemes, blocks are allocated and deallocated in a rigid pattern using a free list organized by block size.

2) Contiguous Allocation: We have developed the first known polynomial time algorithm for contiguous allocation. Earlier work with hypercubes under this model focused on parallel algorithms which use an exponential number of processors [21]. Our algorithm, which we call Cyclic, exploits the fact that there are only n classes of cyclic blocks, categorized by the position of the rightmost *don't care* bit. It uses techniques for logic design that are exponential time for logic circuits [22], but polynomial time for cyclic address allocation. In the next section we give an overview of Cyclic; the algorithm is fairly complex and described more thoroughly in [23].

3) Non-Contiguous Allocation: Non-contiguous allocation is not as straightforward because subtly different statements of the problem have been proposed with different complexity results. In the following, a *feasible* set of requests is one in which the sum of the all the requested blocks does not exceed the full address space.

Problem 1 : Single-Request Address Allocation. *Given child domains C_1 through C_m which have already been successfully allocated (disjoint) blocks B_1 through B_m, respectively, does there exist a free block of size 2^k, $k <= n$?*

Theorem 1 : Single-Request Address Allocation is NP-hard. We prove this by reduction from the classic SAT (Satisfiability) problem. We establish a direct correspondence between clauses and subcubes, showing that a set of clauses is satisfied *iff* there is a free subcube of dimension k after the subcubes corresponding to those clauses are allocated to the child domains. The full proof can be found in [23].

Theorem 1 implies that it is not sufficient to know the current allocation state in order to successfully satisfy a new request in polynomial time. For example, a child domain that needs a new block of addresses may want to query its siblings to find out what blocks they hold, or a parent domain may simply track its allocations. In both cases, it is not possible for the child or the parent to find a free block of the desired size in polynomial time.

Problem 2 : Unordered-Requests Address Allocation. *Given a feasible unordered set of requests for blocks of sizes s_1 through s_m, is there an allocation that satisfies this set of requests regardless of the order in which they are issued?*

Theorem 2 : Unordered-Requests Address Allocation is NP-hard. This is identical to a problem known as *offline subcube allocation*, which seeks to satisfy an unordered set of requests for subcubes from a larger complete hypercube. *Offline subcube allocation* was proved NP-hard by Dutt and Hayes [17].

Theorem 2 states that there is no polynomial time algorithm that can satisfy a feasible set of unordered requests. However, we note that in a realistic setting requests for blocks may occur in a fixed, ordered sequence; hence it is not necessary to optimize over all possible orderings.

Problem 3 : Ordered-Requests Address Allocation. *Given an ordered sequence of requests for blocks of sizes s_1 through s_m is there an allocation that assigns a block to each request if a free block exists at the time of the request?*

Problem 3 is a more realistic statement of the malloc problem. We conjecture that it is solvable in polynomial time and outline an algorithm in the following section. The reason this problem may admit a polynomial time solution, while the others do not, lies in the fact that with ordered requests we know which of the past requests have been satisfied and which blocks have been allocated to each child. In other words, past history and current state are known at the time of each given request. In contrast, Problem 1 requires that the algorithm be able to reconstruct the sequence of requests that led to the current situation.

V. MULTICAST ADDRESS ALLOCATION ALGORITHMS

Because non-contiguous allocation has a clear advantage in terms of recognition capability, a focal point of our research has been to design and evaluate the performance of MaxQ, a non-contiguous allocation algorithm. We lead up to this by first describing a prefix-based algorithm and then illustrating the operation of Cyclic, the first known polynomial-time algorithm for contiguous allocation. Due to space limitations, we give only a high level description of our algorithms; details can be found in [23].

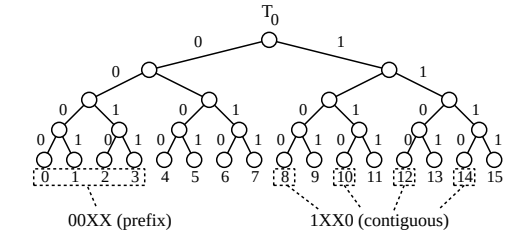
Because doubling is a straightforward operation for all algorithms, we focus here on migration. Recall that a domain tries to migrate when it is unable to double one of its current blocks. Migration algorithms can be characterized by their recognition capacity (prefix, contiguous, non-contiguous) and by their *fit type* (first fit, best fit, and worst fit). Note that we do not discuss best fit, since it is easy to see how it differs from the other two; we do examine its performance later.

Our discussion uses a simple example throughout: a single-level domain hierarchy, an address space of 2^4 addresses, and the following initial sequence of requests for addresses (given as block sizes): 2, 1, 1, 2. We examine the ability of each scheme to allocate more blocks beyond these four initial requests.

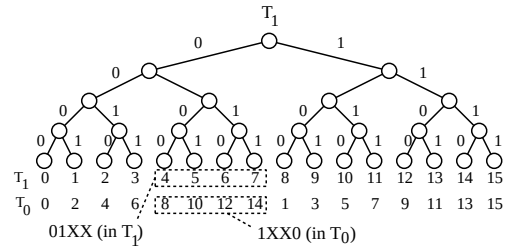
A. Prefix-based algorithms: Prefix-FF and Prefix-WF

Prefix-based algorithms can be best understood through the use of an allocation tree in which the leaf nodes are labeled left to right with the binary addresses 0 through 2^n-1. Left edges are labeled with 0 and right edges labeled with 1. See Figure 3(a).

It is easy to see that the binary sequence on the path from the root to any leaf node is precisely the label of that leaf node. Any interior node in the tree corresponds to a block of addresses contained in the subtree rooted at that node. The expression for this block is the binary sequence on the path from the root to that interior node, followed by *don't cares*.



(a) A prefix-based block and a contiguous block in allocation tree T_0



(b) The contiguous block 1XX0 in T_0 above has been transformed into a prefix block 01XX in T_1

Fig. 3. Allocation trees for prefix and contiguous allocation

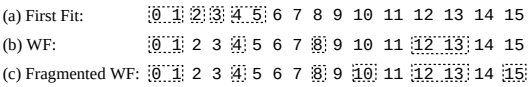


Fig. 4. Allocation for requests 2, 1, 1, 2 under Prefix-FF and WF fits

Prefix-FF allocates addresses using first-fit; it is identical to the Buddy Subcube Algorithm [18]. Prefix-WF allocates blocks using a worst-fit ordering as described in Section III-A. Figures 4(a) and (b) show how Prefix-FF and Prefix-WF would handle the above sequence of addresses. With Prefix-FF, the requests are all packed into the low numbered addresses. As a result, no child block can double into its buddy block, but migration requests for 2, 4, or 8 addresses can be accommodated. Under Prefix-WF, the four initial requests are spaced out so that all children can double. However, no migration requests of size 4 or 8 can be satisfied.

B. Contiguous algorithm: Cyclic

Assuming the initial allocations shown in Figure 4(a) and (b), contiguous allocation improves over prefix-based allocation: all of the children can double and migration requests of sizes 2, 4, and 8 can be satisfied under either FF or WF.

We have developed a polynomial time algorithm for contiguous address allocation called Cyclic. The key features of the algorithm are (1) it inspects only n

allocation trees, (2) it simplifies the task of finding a k-cube into that of finding a single free node in a truncated allocation tree, and (3) it uses binary search and the consensus operation from logic design to locate a single free node and thus a free k-cube.

Cyclic inspects n allocation trees, one corresponding to each of the possible bit positions occupied by the rightmost *don't care* bit. Figure 3(b) shows the representation of block 00XX in T_0 and block 1XX0 as they appear in trees T_0 and T_1.

Within a given allocation tree T_i, Cyclic transforms the task of searching for a free k-cube into the task of finding a single free node in two steps. First, the child holdings are represented as prefix-based holdings in the current allocation tree T_i via wraparound right-shift of i bits. Then, the last k bits are truncated from each child's holding.

Once a tree is transformed, then Cyclic does a binary search of the tree to find a free node. If the search is successful, it yields a free node in tree T_i that can be translated back to the address expression for the corresponding free k-cube. If the search is not successful, there is no free node in the tree, and the operation must be repeated on the next allocation tree.

To determine whether there is a free node in a given subtree, Cyclic uses the *consensus operation* [22]. Consensus is a binary operation from Boolean Algebra that performs a bit-by-bit comparison of block addresses to yield a minimum size block that is consistent with the two original blocks. At the bit level, the consensus of bit b with X is b; the consensus of b and $\neg b$ is X, and the consensus of b with b is b, where $b \in \{0, 1\}$. For example, the consensus of 01X0 and X111 is 011X. Notably, if consensus is applied to two buddies, the result is the combination of the buddies into a larger block, e.g. the consensus of buddies 0XX0 and 1XX0 is XXX0.

Cyclic starts with a list of the child holdings and repeatedly applies consensus to all pairs of adjacent blocks. Any blocks that are covered by a larger block are removed from the list. This procedure is repeated until no buddies remain. At most, a block can be combined with its buddy n times since each buddy changes an instantiated bit to a *don't care*. Thus, we are guaranteed that the algorithm terminates after n iterations. These repeated invocations of the consensus operation will yield the whole subtree *iff* the subtree is covered by the children. This indicates a failure to find a free node in the subtree.

The complexity of Cyclic is $O(C * n^3)$.

C. Non-contiguous algorithm: MaxQ

The advantages of non-contiguous allocation can be seen from the highly fragmented situation in Figure 4(c). Cyclic can only migrate to new blocks of size 2, while a non-contiguous algorithm can migrate to blocks of sizes 2 and 4. For example, a free non-contiguous block is 0X1X.

We have developed a non-contiguous address allocation algorithm for the Ordered-Requests problem called MaxQ. MaxQ uses the consensus operation to maintain a free list that contains a maximal free subcube. This free list is a weaker type of free list than that proposed by [17] which is a maximal free list that is greater than all other maximal free lists. Our free list only attempts to find one of all the maximal free blocks of addresses, of which there may be several, and then the rest of the list contains a sub-optimal list of free address blocks. For example, if the free list contains 000, 001, 110, 100, the algorithm in [17] would be guaranteed to find 00X and 1X0 as the maximal free list. While MaxQ might find this list, it could also find the free list of 001, X00, and 110.

To maintain the free list after an allocation has been made, MaxQ finds the consensus between all pairs of elements in the existing free list. Any new consensus which covers a pair of addresses is kept and the covered pairs are removed. We apply the consensus operation to all pairs in the free list repeatedly until there are no new consensus blocks. As with Cyclic, we are guaranteed that this will execute at most n iterations. Once we have a list of maximal free blocks given from the pairs in the original list, MaxQ chooses one of the largest blocks. The new free list is then composed of this block and the subtractions of this block with all the other blocks.

Using a free list allows us to ensure that if a migration needs a block of size k, then a simple traversal through the list in search of a k-sized block will reveal if one exists. Since we know our free list will contain a maximal free block, then if there is not a k-sized block in the list, there is not a maximal block of that size in the address space.

Proving MaxQ is polynomial time consists of proving that the free list will always remain polynomial in size. This is a difficult problem and remains open.

Note that a non-contiguous model for address expressions called *kampai* was developed for unicast routing [24]. However, the *kampai* algorithm was restricted to growth through doubling only.

VI. MODELING ADDRESS ALLOCATION

In order to evaluate the performance of a variety of allocation algorithms, we have developed a general model of the malloc problem that significantly improves upon the state-of-the-art in this area. Our contributions in this area include:

- Formulating a general allocation algorithm that allows us to test prefix, contiguous, and non-contiguous allocation algorithms.
- Creating a rich set of load scenarios to model shifting demand between domains.
- Specifying a new set of metrics to accurately capture the performance of an allocation algorithm.

Previous work in this area consists of a detailed simulation of the MASC protocol [6]. The focus of that work is to study address allocation latency using a claim-collide mechanism, hence it simulates only prefix-based allocation, with homogeneous load among domains.

A. General Allocation Algorithm

We have developed an abstract model of the malloc problem that isolates the allocation algorithm from the implementation details of the MASC protocol. This enables us to better study allocation performance and then apply those results to problems in both networking and hypercube processor allocation.

In our model, all addresses are initially unclaimed and held by a single parent domain. Child domains may then request blocks of addresses from the parent's space and later release blocks when they are done. A block is a group of contiguous addresses, and the size of a block must be a power of 2.

Address allocation uses blocks (or subcubes) so that a single address expression represents the entire group of addresses. Aggregating addresses is highly desirable for multicast address allocation because it keeps routing tables small; there is one routing table entry per block. Likewise, in hypercube computers, all of the processors must form a subcube so that efficient parallel computation may occur over this set of processors.

As practical considerations (and in keeping with the MASC specification), we place several restrictions on children. First, domains are allowed to hold up to b blocks of addresses. Again, this keeps routing table small, limiting them to at most b entries per child domain. Second, when a domain seeks to double one of its current blocks, it can only double if the resulting utilization of the block remains above a threshold d. We later relax these restrictions in our experiments to determine the performance trade-offs of these parameters.

When a child wants to obtain some addresses, our allocation model takes the following steps:

1) **The child checks to see if the request can be filled internally.** For example, a child may hold a block of 256 addresses. If only 200 addresses are used, then the child can internally allocate 56 more addresses before requesting additional addresses from the parent.
2) **The child attempts to double one of its current blocks.** If a request cannot be satisfied internally, the child requests the buddy of any of the blocks allocated to it. If any of the buddies are available, and if when doubled, the utilization of the child's holdings would remain above d, the parent releases the block to the child and the two buddies are coalesced into a single block. Requests can then be filled from this new address space. For example, consider a child with two filled blocks of 256 addresses. Assuming the buddy of one of the blocks is available, the child may double one of its blocks and obtain 768 total addresses, as long as its utilization ($512/768 = 67\%$) exceeds d.
3) **The child requests a new block from its parent.** If the child has not reached its maximum block count, b, then the child may request a new block large enough to satisfy the request. Children limit the number of blocks they hold to conserve routing table space.
4) **The child attempts to migrate a block to a larger block.** If the request is still not filled, the child will try to migrate one or more blocks to larger blocks from the parent. If successful, the request will be filled from the newly migrated blocks. For example, consider a child with two blocks of 256 addresses, neither of which can be doubled (because the buddies are held by some other child). If space is available, the child can migrate both blocks to a new block of 512 addresses. The child can then gain additional space either by doubling this new block or by requesting an additional block. Migration enables this latter choice if the child already has reached its allotment of b blocks.
5) **The allocation request fails.** The child has failed to allocate new addresses and may not be able to fulfill the requests of all applications within its domain.

There are several aspects of the MASC protocol

$$f(t) = \begin{cases} (g(t)/t_1) * (k_{max} - k_{min}) + k_{min}, & 0 <= g(t) <= t_1 \\ k_{max}, & t_1 < g(t) <= t_2 \\ (t_3 - g(t))/(t_3 - t_2) * (k_{max} - k_{min}) + k_{min}, & t_2 < g(t) <= t_3 \\ k_{min}, & t_3 < g(t) < t_p \end{cases}$$

where

$g(t) = (t + t_o) \bmod t_p$, $0 < t_1 <= t_2 <= t_3 <= t_p$, and $0 <= t_o < t_p$.

Fig. 5. Load Function

that this model explicitly does *not* include, since we are modeling an abstraction of the malloc problem. First, we use a simple request-reply protocol, rather than a claim-collide mechanism. In our model, the parent domain is in charge of allocating addresses to its children. Second, we do not model address lifetimes. Instead, a load function determines when a domain will request or release addresses. Finally, we assume that migration can occur instantaneously when it is successful. In reality, a MASC domain may need to hold its current block of addresses until they have expired. Collectively, these assumptions simplify our model, providing a clean abstraction of the malloc problem. We believe only the migration assumption impacts our results, allowing more aggressive use of the available address space. However, our primary goal is to compare allocation algorithms, and this assumption is the same for all of them.

All of our experiments use a single parent domain with a set of child domains. It is relatively straightforward to generalize the performance of a single level of hierarchy to multiple levels. For example, if a domain is able to allocate on average 90% of its address space, then in a 3-level hierarchy this would result in overall utilization of 73%. For more details on hierarchy as it affects allocation performance, see [6].

B. Load Function

A major focus of our modeling effort has been determining the type of load function to use. The major difficulty we face is that dynamic address allocation has never been deployed on a large scale, and multicast itself has never been widely used by the general public. This means we do not know what kinds of load functions to expect at the domain level.

We explicitly choose *not* to model individual request and releases for addresses at the application level. Based on our previous work [25], we find it is difficult to correlate this type of model to the macroscopic behavior of a domain. For example, a stochastic function may choose in one time step to request one address and then in the next step may choose to release one address. A domain is more likely to have some statistical knowledge of past demand and plan its requests based on its estimate of a load curve.

We choose instead to model demand for addresses at the domain level, using several basic scenarios that generalize common system-level behaviors. One of our primary goals in modeling domain-level workloads is to capture the case where demand shifts between domains. With a limited address space, the job of an address allocation algorithm is to provide addresses to a domain where they are needed, reclaim them when the domain is finished, and re-assign them to a new domain.

Based on this reasoning, we use a simple, periodic load function of the form shown in Fig. 5. This function gives the desired number of address *sets* at time t for a child domain. A set is a group of contiguous addresses, and the size of a set must be a power of 2. All domains use the same set size, and this size is the minimum size of a block. If at any given time a domain has fewer address sets than its desired amount, it requests the additional sets. Likewise, a domain releases address sets if it has more than it needs.

The basic shape of the load function is shown in Figure 6 and is defined by four different regions given by the parameters t_1, t_2, t_3. Between time 0 and time t_1 a domain requests additional addresses until it reaches a maximum value. Between times t_1 and t_2 the domain stays at this level, then releases addresses between times $t2$ and t_3. Finally, the domain stays at its minimum allocation level for the duration of the period, which lasts until t_p. This function can be shifted in time by the offset t_o, with the maximum and minimum values defined by k_{max} and k_{min}.

This basic load function allows us to increase or decrease load by adjusting the parameters of the function. We can also shift demand between domains by changing the offset of each domain's load function, so that one domain is at its maximum allocation while another is at its minimum. Overall, a parent domain will see a request pattern based on the additive combination of its children's request functions.

We model three general scenarios for load across

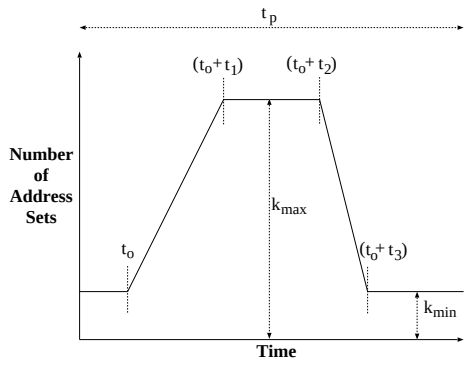


Fig. 6. Illustration of Load Function

a domain by forming a composite of the per-child periodic load functions:

- *Ideal Load.* In this scenario, the load functions of the child domains are shifted and their parameters adjusted so that the overall demand seen by a parent is flat. To accomplish this, we set $t_1 = t_p/4$, $t_2 = t_p/2$, and $t_3 = 3 * t_p/4$, then distribute the offsets of the child load functions across the period t_p. We then adjust the period of the function, as well as k_{min} and k_{max}, until the load is flat. This is an artificial condition and is meant to represent a best-case scenario.
- *Random Load.* For a more realistic scenario, we use the same settings of t_1, t_2, and t_3 as above, but randomly choose an offset for each child. The composite of the load functions under this scenario is a function with random peaks and valleys spread across it over time.
- *Prime-time Load.* For the prime-time load, we roughly model a bell curve, similar to a function that represents peak load for a telephone system. To do this, we set $t_3 = p < t_p$, where p is the duration of prime time. We also set $t_1 = p/3$ and $t_2 = 2 * p/3$. If we suppose that time is measured in hours, we can set the load function period to be 24 hours and then choose prime-time period of p hours. The actual units of time are not relevant here – our intention is simply to model a well-known load function. If multicast addresses are expensive, then domains will want to match their allocation requests to this load curve.

C. Metrics

While address space utilization is typically used to measure the performance of an allocation algorithm, this can be a misleading metric. It is possible to attain nearly any level of utilization if the load is driven high enough. To obtain a more accurate picture of allocation performance, we monitor the *outcome* of each allocation request. When a child makes a request for a set of addresses, there are five possible outcomes:

1) The request is filled internally in one of the child's existing blocks. This occurs when a child owns a block that has unallocated addresses.
2) The request is filled by allocating a new block to the child. This can only happen when the number of blocks currently held by the child domain is less than b.
3) The request is filled by doubling one of the child domain's current blocks. In this case, the buddy of a held block is available in the free-space and the block is given to the child domain.
4) The request is filled by migrating one or more of the child domain's current blocks. The child releases the blocks back to the parent and a new, larger block is allocated to the child.
5) The request fails. If none of the previous results were possible, the request can not be fulfilled.

We distinguish cases 3 and 4 as *growth requests* because these are the requests in which the child's current holdings are insufficient and it must obtain additional addresses. This is the case when the allocation algorithm is exercised, as it needs to meet the demand by doubling or migration in order to satisfy the child. Technically, case 2 is also a growth request, but it happens so rarely that we do not include it in our analysis below.

A good address allocation algorithm is one that can fulfill as many requests as possible (i.e., handle a high load) while migrating and allocating as few new blocks as possible. Thus, high double rates, low migration rates and low failure rates are preferable.

VII. EXPERIMENTS

For our experiments, we use a single parent with a 20-bit address space, equivalent to more than 1 million multicast addresses. Child domains request sets of 256 addresses. For most experiments, the maximum number of blocks b that a domain can hold is 2 and the doubling threshold d is 75%. These numbers are identical to those used in the MASC spec [26], except that our address space is twice as large.

Using our general allocation model, load functions, and metrics, we investigate several important problems that shed new light on the malloc problem, both abstract and practical.

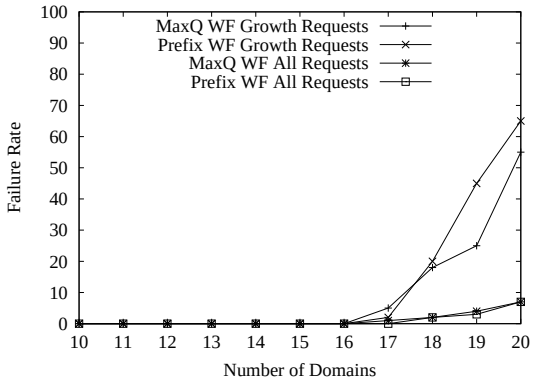


Fig. 7. Failure vs Load for Prefix and MaxQ: Random Load, 75% Doubling Threshold

Algorithm	Ideal	Random
Prefix Worst-Fit	24	18
Prefix Best-Fit	21	18
Prefix First-Fit	22	18
MaxQ Worst-Fit	22	18
MaxQ Best-Fit	21	19
MaxQ First-Fit	20	19

TABLE II

MAXIMUM NUMBER OF DOMAINS SUPPORTED

A. Does non-contiguous allocation outperform standard prefix-based allocation?

The most fundamental question arising from the malloc problem is which type of allocation algorithm yields the best performance. Prefix-based and non-contiguous allocation represent two ends of the spectrum – prefix-based allocation is the simplest scheme yet has the lowest recognition capability, while non-contiguous allocation is the most difficult to achieve yet has complete recognition capability.

In our experiments, we compare the performance of our non-contiguous algorithm, MaxQ, with that of a standard Prefix algorithm such as that used by MASC.

Despite its theoretical advantage, and contrary to both our own intuition and that of the MASC authors [16], our experiments show that MaxQ does not perform significantly better than Prefix [27]. Figure 7 shows an experiment using a random load in which we increase the number of domains in the system and observe the overall success rate for allocation requests. The performance for MaxQ and Prefix is similar; in both cases, the failure rate for growth requests increases sharply as the load increases beyond 17 domains (about 60% of all addresses are requested). The overall failure rate increases gradually at this same point.

This result holds for both the ideal and random loads under a range of operating parameters (varying both the number of blocks held and the doubling threshold). Table II summarizes the results for Prefix and MaxQ with various fit types, listing the number of domains that can be supported with a maximum failure rate of 20% for growth requests. Growth requests are a small fraction of all requests; this corresponds to an overall failure rate of about 2%.

To determine the reason for the strong performance of prefix-based allocation, we developed a visualization tool called *address mapper*. Address mapper can animate an entire sequence of allocation actions, which has helped us to verify in an ad-hoc manner that our algorithms are allocating addresses correctly. More importantly, we are able to examine the allocated and free blocks at any point in the simulation and see patterns in how space is allocated.

Using address mapper, we see that *MaxQ fragments the free space across multiple dimensions, making it difficult to aggregate the free regions into large blocks*. Prefix-based allocation, on the other hand, keeps the free space organized into regular regions. This can be seen from Figures 8 and 9, which show free blocks as white space and allocated blocks as colors. Both algorithms are running at the same point in a simulation and have the same number of free addresses. However, Prefix holds a larger free block than MaxQ, which has greater fragmentation. At this time in the simulation, one child is requesting more addresses and will not be satisfied by MaxQ because it cannot migrate or allocate another block (having reached its maximal block count). The Prefix algorithm will use the large block to satisfy the request.

In summary, these experiments demonstrate the superiority of a simple prefix-based allocation algorithm. Using prefix-based allocation (e.g. in MASC) is typically done because prefix-based expressions are easy for humans to process and because unicast addresses are assigned in this manner. From our experiments, we are able to see that the additional complexity of a non-contiguous allocation algorithm does not appear to buy a significant performance improvement.

B. When should dynamic allocation be used?

Dynamic allocation (with MASC) and static allocation (with GLOP) can be seen as complementary approaches, with static allocation providing permanent addresses to providers with long-lived applications and dynamic allocation providing addresses for short-lived applications. Both approaches are needed in IPv4

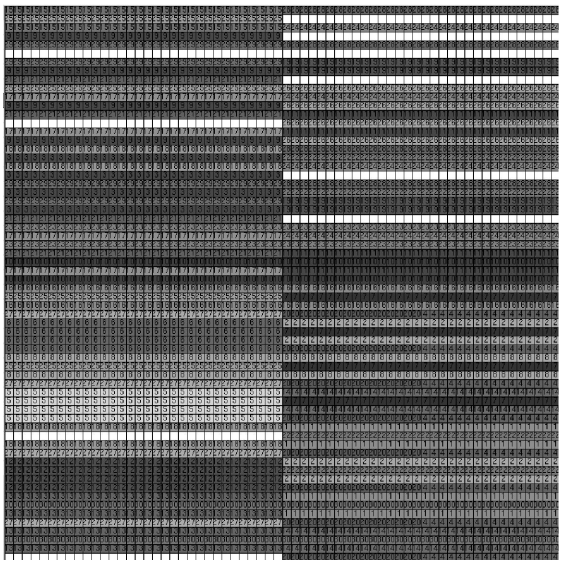


Fig. 8. Visualization of MaxQ First Fit: Allocated Blocks

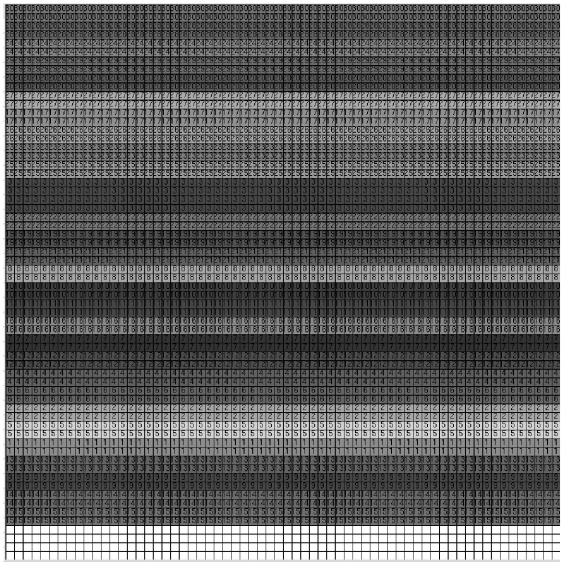


Fig. 9. Visualization of Prefix First-Fit: Allocated Blocks

because GLOP allocates only 256 addresses to each domain.

We perform both a theoretical and simulation-based analysis of these two approaches to illustrate the loads under which dynamic allocation will be necessary.

1) Theoretical Analysis: Under some basic assumptions, we show there is a theoretical maximum number of domains that a dynamic allocation algorithm can support using a prime-time load[1]. Assume the following:

[1]Note that the random load is a special case of the prime-time load so our results apply to both load functions.

- n is the number of domains that dynamic address allocation can scale to
- k is the size of the address space
- p is the duration of the prime-time peak, where $0 < p <= t_p$
- k_{min} is the minimum number of address sets a single domain will hold during its life-time
- k_{max} is the maximum number of address sets a single domain will request during its peak demand, where $k_{max} > k_{min}$

We begin by giving the equation for the number of addresses remaining after the minimum number held by each domain has been allocated.

$$r = k - n * k_{min} \tag{1}$$

Next, we derive the number of domains that can peak during period t_p:

$$n = \frac{r}{k_{max} - k_{min}} * \frac{t_p}{p} \tag{2}$$

This equation takes the remaining addresses r and divides by the amount of addresses that a domain will peak to (which is $max - min$ since they have already been allocated the minimum number of blocks). This gives the number of domains that can peak in a single period. Then we multiply by the factor t_p/p to find the number that will peak during the period t_p. Note that this factor is a gross estimate and does not take into account the startup and slowdown periods surrounding the peak hours.

By solving for n and reducing, we find that the number of domains that dynamic address allocation can scale to is:

$$n = \frac{t_p * k}{p(k_{max} - k_{min}) + t_p * k_{min}} \tag{3}$$

The worst case for dynamic address allocation is when all domains peak simultaneously. This occurs, for example, when the peak time is equal to t_p or:

$$n = \frac{k}{k_{max}} \tag{4}$$

Since this is the same value as the number of domains that GLOP can scale to, we know that dynamic allocation should theoretically always be able to handle more domains than static allocation.

2) Simulation Analysis: The theoretical result we derive for the scalability of dynamic allocation does not hold in practice because it assumes that domains can be allocated any number of blocks of any size. In effect, this assumes a perfect allocation algorithm that knows all requests ahead of time and can optimally place them.

GLOP = maximum 32 domains

Prime-Time Duration (hours)	Minimum Number of Sets		
	8	16	32
16	36	37	36
8	44	43	43
5	49	48	47
3	52	48	48
2	52	55	52

TABLE III

MAXIMUM NUMBER OF DOMAINS SUPPORTED AS A FUNCTION OF PRIME-TIME DURATION, MAXIMUM 128 SETS PER DOMAIN.

GLOP = maximum 16 domains

Prime-Time Duration (hours)	Minimum Number of Sets		
	8	16	32
16	17	17	17
8	20	21	22
5	20	23	23
3	25	25	23
2	25	26	26

TABLE IV

MAXIMUM NUMBER OF DOMAINS SUPPORTED AS A FUNCTION OF PRIME-TIME DURATION, MAXIMUM 256 SETS PER DOMAIN.

These types of allocation algorithms have been shown to be NP-Complete [4], [17] and hence do not match what can be expected from practical algorithms.

As a result of this limitation, we use a set of simulations to compare dynamic and static allocation, showing that under the simulated conditions dynamic allocation outperforms static allocation in most situations involving a prime-time load function. In this set of simulations we use a dynamic allocation model that closely follows the MASC protocol. Specifically, this means that our dynamic algorithm uses prefix-based allocation with worst-fit placement, that each domain can hold at most two blocks, and that the doubling threshold is 75%.

We examine dynamic allocation under 30 different load scenarios by varying the minimum and maximum values of the staircase function and considering a range of durations (in terms of hours) for a prime-time load function (with a period of 24 hours). For each scenario, we increase the number of domains (and hence the load) in the system and determine the maximum number of child domains a single parent can handle without any allocation failures. Typically, the failure rate increases sharply once too many children are present, since each one requires a minimum allocation of addresses. The results are shown in Table III and Table IV.

Not surprisingly, the greatest improvements for dynamic allocation (63% to 72%) occur under the conditions of smaller allocations per domain and short periods of peak demand. The only conditions under which dynamic allocation performs marginally better than static allocation (6% to 13%) are when the peak time is considerably long (16 hours). The minimum number of addresses allocated to each domain has little effect on performance – where dynamic allocation makes sense is when there is a large spread between the minimum and maximum allocation to a domain and a relatively short period of time when the peak allocation is required.

C. How can we improve prefix allocation?

Having illustrated the conditions under which dynamic address allocation outperforms static allocation, we now consider several ways to improve the performance of Prefix allocation, and hence MASC.

1) Eliminating Migration: Migrating a domain from one set of multicast address blocks to another is expensive – applications may need to change addresses while transmitting data, and routers will need to update routing tables. In addition, our simulations indicate that migrating all blocks to a single new block, as recommended by the MASC authors [16], [26], results in poor performance. (The model we use in this paper primarily migrates individual blocks, hence Prefix performance is not as bad as it otherwise would be.)

Our simulations show that we can achieve better performance for Prefix by eliminating migration altogether. We do this by reducing the doubling threshold from 75% to 50%, essentially allowing doubling to occur whenever it is possible. Figure 10 shows Prefix with the threshold set to 50%, while Figure 11 shows Prefix with the threshold set to 75%. As these figures show, the improved Prefix maintains the same failure threshold, but handles all growth requests with doubling rather than migration.

MASC's original motivation for setting the doubling threshold to 75% was to increase utilization. Theoretically, a domain holding 1024 addresses and needing 256 more could double its holdings to 2048 addresses, resulting in a utilization of $(1024 + 256)/2048 = 62.5\%$. However, unless load is very high, these unused addresses are not needed elsewhere. Under most conditions, it is preferable to hold these unused addresses for additional growth in the future. If load curves follow any kind of cyclic pattern, similar to what we are modeling, then this future growth is likely to occur. Restricting a domain to a doubling threshold of 75%

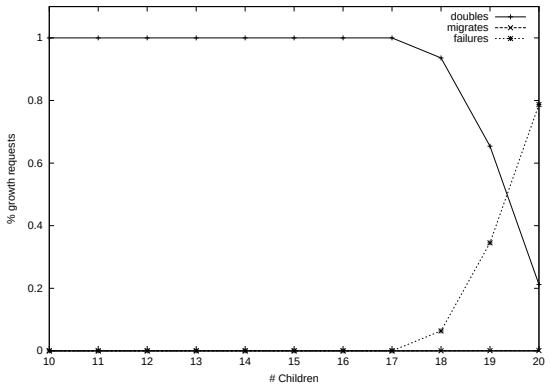


Fig. 10. Outcome of Growth Requests for Prefix: Random Load, 50% Doubling Threshold

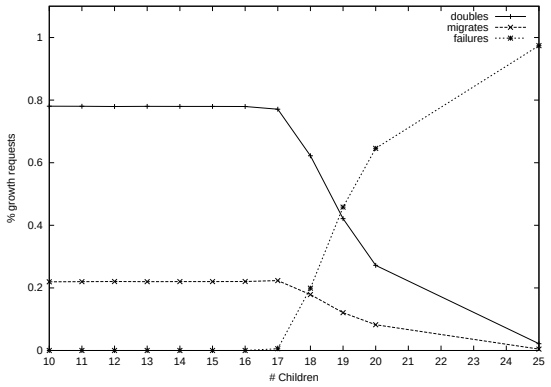


Fig. 11. Outcome of Growth Requests for Prefix: Random Load, 75% Doubling Threshold

GLOP = maximum 32 domains

Maximum Number of Blocks	Minimum Blocks Held		
	8	16	32
1	32	32	32
2	48	48	44
3	48	52	48
4	64	72	68
5	68	72	68
10	92	90	72
Theoretical Max	107	93	73

TABLE V

MAXIMUM NUMBER OF DOMAINS SUPPORTED AS A FUNCTION OF THE NUMBER OF BLOCKS ALLOWED, PRIME-TIME DURATION 5 HOURS, MAXIMUM 128 SETS PER DOMAIN.

GLOP = maximum 16 domains

Maximum Number of Blocks	Minimum Blocks Held		
	8	16	32
1	16	16	16
2	21	23	23
3	23	24	24
4	28	30	32
5	30	31	32
10	37	42	40
Theoretical Max	58	54	46

TABLE VI

MAXIMUM NUMBER OF DOMAINS SUPPORTED AS A FUNCTION OF THE NUMBER OF BLOCKS ALLOWED, PRIME-TIME DURATION 5 HOURS, MAXIMUM 256 SETS PER DOMAIN.

simply forces the domain to migrate earlier than it should need to.

Eliminating migration does not mean that domain holdings do not change over time. Instead of forcing migrations to occur because of utilization requirements, domains naturally change their holdings as they request and release addresses. Addresses held by one domain are released when they are no longer needed and later claimed by a different domain. This natural style of migration helps to improve utilization compared to a purely static algorithm like GLOP.

2) Expanding the Number of Blocks Held: The original MASC paper [16] indicates that each domain should hold at most two blocks of addresses, while the MASC specification [26] states that the recommended number is three. For a given load, increasing the number of blocks a domain may hold reduces the failure rate, while increasing the routing table overhead. An additional benefit is that more domains can be supported. For these simulations we use Prefix with a doubling threshold of 50%.

Table V demonstrates that, given a 5 hour prime-time and 10 blocks per domain, Prefix can scale to 92 domains, very close to the theoretical maximum of 107. At this level, Prefix performs 186% better than GLOP, an advantage that must be weighed against a 10-fold increase in routing table size. At a more reasonable level of 4 blocks per domain, Prefix can handle an additional 16-24 domains, or 38% to 57% beyond what it can do with 2 blocks. Interestingly, there is very little difference between holding 2 or 3 blocks. Similar results are seen in Table VI, which allows up to 256 sets of addresses per domain, and with other prime time durations.

VIII. CONCLUSIONS

Studying the malloc problem has given us insight into the performance of several basic classes of address allocation algorithms. We expect the results of our work to have an impact both in networking and in parallel computing. In particular, our results indicate that a simple prefix-based allocation algorithm performs at least as well as our non-contiguous algorithm. While our

non-contiguous algorithm should theoretically benefit from greater recognition capability, it causes greater fragmentation of the free space, making it difficult to achieve a performance gain. It is possible that a different non-contiguous algorithm may achieve better performance, but no such algorithm has been proposed, and our experience indicates that this will be difficult to accomplish.

We have demonstrated that dynamic allocation outperforms static allocation whenever demand for addresses varies significantly over time. We also show how to improve the performance of prefix-based algorithms, such as those used in the existing multicast infrastructure, by eliminating forced migration and expanding the number of blocks a domain may hold.

Finally, our results indicate that allocation performance drops sharply as more domains are added to a parent. The exact threshold for this drop is difficult to predict as it depends on the cumulative load functions of the domains. For parallel computing applications, the load may be somewhat more predictable or at least controllable. However, this problem is unavoidable with ASM multicast address allocation under IPv4, as the address space is small and must be shared globally. This lends support to alternative multicast architectures, such as SSM, GLOP (for IPv6 where address space is plentiful), or application-layer multicasting. In all of these cases, the malloc problem is no longer an issue.

IX. ACKNOWLEDGMENTS

We would like to thank NSF REU students Tim Singer and Joannie Humphries for their contributions to this work. We are also grateful to the reviewers for their detailed readings and their help in improving this paper.

REFERENCES

[1] S. Deering, "Multicast Routing in a Datagram Internetwork," Ph.D. dissertation, Stanford University, 1991.

[2] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D.Estrin, and M. Handley, "The MASC/BGMP Architecture for Interdomain Multicast Routing," in *ACM SIGCOMM*, August 1998.

[3] M. Handley, "Session Directories and Scalable Internet Multicast Address Allocation," in *ACM SIGCOMM*, August 1998.

[4] V. Lo, D. Zappala, and C. GauthierDickey, "A Theoretical Framework for Multicast Address Allocation," in *IEEE Globecom, Global Internet Symposium*, November 2002.

[5] M. Handley and V. Jacobson, "sdr," now maintained at http://www-mice.cs.ucl.ac.uk/multimedia/software/sdr.

[6] R. G. Pavlin Ivanov Radoslavov, Deborah Estrin, "A Claim-Collide Mechanism for Robust Distributed Resource Allocation," Computer Science Department, University of Southern California, Tech. Rep. USC-CS-99-711, 1999.

[7] D. Meyer and P. Lothberg, "GLOP Addressing in 233/8," RFC 2770, February 2000.

[8] R. Hinden and S. Deering, "IP Version 6 Addressing Architecture," RFC 2373, July 1998.

[9] H. Holbrook and B. Cain, "Source-Specific Multicast for IP," Internet Draft: work in progress, November 2001.

[10] Y. hua Chu, S. G. Rao, and H. Zhang, "A Case For End System Multicast," in *ACM Sigmetrics*, June 2000.

[11] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "ALMI: An Application Level Multicast Infrastructure," in *3rd USENIX Symposium on Internet Technologies*, March 2001.

[12] J. Liebeherr and M. Nahas, "Application-layer Multicast with Delaunay Triangulations," in *Global Internet Symposium, IEEE Globecom*, November 2001.

[13] Y. Chawathe, S. McCanne, and E. A. Brewer, "RMX: Reliable Multicast for Heterogeneous Networks," in *IEEE INFOCOM*, 2000.

[14] D. Waitzman, C. Partridge, and S. Deering, "Distance Vector Multicast Routing Protocol," RFC 1075, November 1988.

[15] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei, "An Architecture for Wide-Area Multicast Routing," in *ACM SIGCOMM*, August 1994.

[16] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoglu, D.Estrin, and M. Handley, "The MASC/BGMP Architecture for Interdomain Multicast Routing," in *ACM SIGCOMM*, August 1998.

[17] S. Dutt and J. P. Hayes, "Subcube Allocation in Hypercube Computers," *IEEE Transactions on Computers*, vol. 40, no. 3, March 1991.

[18] M. Chen and K. G. Shin, "Process Allocation in an N-Cube Multiprocessor Using Gray Code," *IEEE Transactions on Computers*, vol. 36, no. 12, December 1987.

[19] A. AlDhelaan and B. Bose, "A New Strategy for Processor Allocation in an n-Cube Multiprocessor," in *International Phoenix Conf. on Computers and Comm.*, March 1989.

[20] V. M. Lo, W. Liu, B. Nitzberg, and K. Windisch, "Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers," *IEEE Trans. on Par. and Dist. Sys.*, vol. 8, no. 7, July 1997.

[21] M. Livingston and Q. F. Stout, "Fault Tolerance of the Cyclic Buddy Subcube Location Scheme in Hypercubes," in *6th Distributed Memory Computing Conference*, 1991.

[22] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin, "McBOOLE: A New Procedure for Exact Logic Minimization," *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, no. 1, January 1986.

[23] V. Lo, D. Zappala, C. GauthierDickey, and T. Singer, "A Theoretical Framework for Multicast Address Allocation," University of Oregon, Tech. Rep. UO-TR-2002-01, 2002.

[24] P. Tsuchiya, "Efficient and Flexible Hierarchical Address Allocation," in *INET92*, June 1992.

[25] M. Livingston, V. Lo, K. Windisch, and D. Zappala, "Cyclic Block Allocation: A New Scheme for Hierarchical Multicast Address Allocation," in *First International Workshop on Networked Group Communication*, November 1999.

[26] P. Radoslavov, D. Estrin, R. Govindan, M. Handley, S. Kumar, and D. Thaler, "The Multicast Address-Set Claim (MASC) Protocol," RFC 2909, September 2000.

[27] D. Zappala, C. GauthierDickey, and V. Lo, "Modeling the Multicast Address Allocation Problem," in *IEEE Globecom, Global Internet Symposium*, November 2002.