The selfish Nix allocator

Francisco J. Ballesteros

ABSTRACT

On many-core machines, resource allocation usually leads to high contention on a few allocators. To address this issue, Nix includes an allocator conceived for many-core and NUMA machines, *nalloc*, also known as the selfish allocator. In this paper we describe the allocator and some of its preliminary evaluation.

Introduction

Most kernel activities lead to high contention on a few allocators:

- The page allocator supplies pages to user processes (and to the kernel memory allocator and the kernel MMU code).
- The *Chan* allocator supplies structures that represent files in use by the kernel.
- The *Path* allocator supplies *Path* structures that represent file names used by the mount table, descriptors for files in use, etc.
- The *Name* allocator supplies buffers for (flat) file names kept as strings.
- The *Block* allocator supplies I/O buffers which are the building blocks of kernel queues used for most I/O in the system.

These are the most significant allocators. Other important allocators like the memory allocation, the *malloc* implementation, and the mount driver RPC allocator have specific implementations (a variant of ''quick fit'' the former, and a central allocator the later), and are not described here.

The problem is that most of the system activity requires going to one or several of them and contend with other processes in the system. Usually, activity happens in bursts, which means that when they are most needed, is when contention becomes a problem. The larger the number of cores in the system, the bigger the problem caused by high contention.

In fact, in the standard system before Nix, using more than four processors usually implied a slow down of the system and not a speed up. At least for applications such as compiling programs and serving files on the network.

The selfish allocator

To address the problem stated so far, we built a new general purpose allocator along the following design guidelines:

- The allocator focuses on a single data structure (which might keep resources of different sizes, but still pointed to by a single data structure).
- It is a general purpose allocator in the sense that it can be used for different data

structures (using one allocator per data structure).

- It leverages the underlying memory allocator (i.e., the *malloc* implementation) to acquire new structures as demand increases, and adjusts to actual needs.
- It never releases the structures already acquired, but caches them on a free list for later.

Nothing new so far. However, here comes the difference with respect to the previous state of affairs:

- There is a per-allocator and per-process cache for free structures. Thus, a few of the free data structures are kept on the *Proc.* descriptor and not on the actual allocator free list.
- Such free structure cache is *not* released when a process dies, and survives the process existence.

Like in other previous allocators, e.g. the Slab allocator [1], the allocator keeps structures semi-initialized while on the free list. That is, most of the price of initializing a data structure is avoided when structures are taken from the free list and reused.

In particular, like the system has always done with kernel process stacks, when structures require buffering or arrays of different sizes, such buffers are allocated but never released. They are kept pointed to from the main structure considered, which is kept in the free list while unused. Further usage of the structure finds the buffer(s) needed already allocated and saves the price of re-allocating it.

Using a per data structure allocator and retaining buffering as required by each structure is important because

1 It reduces contention on the central memory allocator, and

2 it distribues the contention among all the allocators (one per data structure instead of a single one).

Other NUMA allocators place allocators on a per core or per NUMA domain, but, the Nix allocator keeps a central free list. However, it uses a per-process cache of free resources that survive process existence, which makes it different from them.

This idea makes it able to do a self-allocation of the resource considered, in most cases, without interference to other processes or cores in the system. Moreover, it avoids the need for heavy interlocking with other cores, while not needing any complex data structure for such purpose.

Implementation

An example of an implementation of *nalloc* is depicted in the figure. The figure shows three processes (above) and two Nix allocators, one for *Path* structures and another for *Chan* structures.

Each allocator maintains two main lists: one list chaining all the data structures previously allocated (be they in use or not), and a free list. The former is most useful for debugging.

When the free list is exhausted and more structures are needed, an allocator resorts to *malloc* to allocate more ones.

The interesting point is shown in the tiny arrays shown beneath the left and right processes in the figure. Such arrays are selfish allocators (we call them so). Each allocator in the kernel is assigned an index in the selfish array, and the corresponding entry is a tiny free list for the data structure allocated in each case.

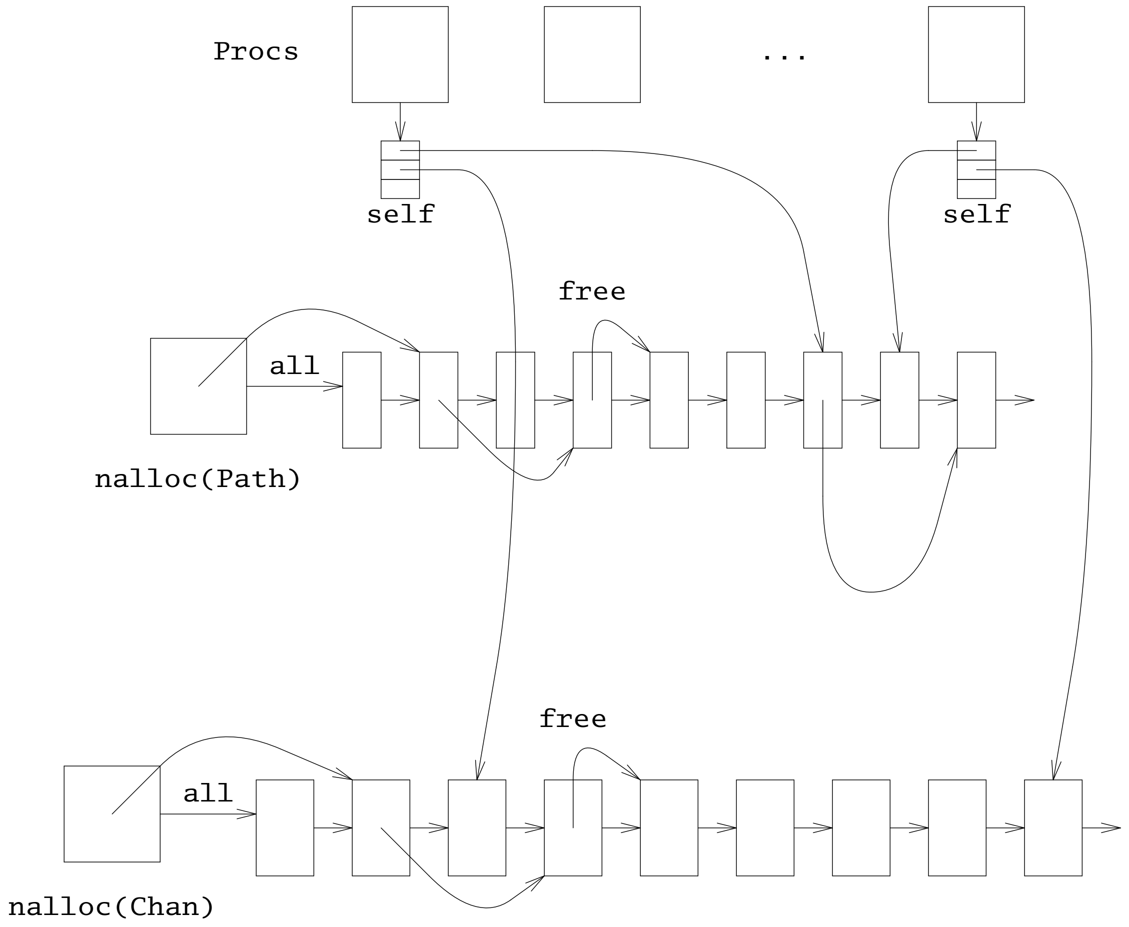

Figure 1 The nix allocator keeps a per-data structure free list of resources along with a per-data structure and per-process cache of a few free resources, which survives the process lifetime.

For example, in the figure, the process on the left has a selfish allocator with two *Path* structures and another with just one *Chan* structure. Should that process need to allocate a *Chan*, it could self-service it without disturbing any other component or process in the system.

In the same way, the process on the right in the figure has one selfish *Path* and one selfish *Chan* as it can be seen.

When the selfish allocators are exhausted, processes would rely on the allocators shown below the processes. Of course, they just call a single function to allocate the data structure, without being aware of how the structure has been allocated.

When a data structure is released, if there are not too many structures in the selfish allocator for it, it is silently placed in there. Otherwise, the structure is handed to its allocator, where it is kept on the central free list.

The Nix allocator includes also some support for debugging and accounting. Each allocator chains all the data structures allocated so far (although some might be in the free list), and a function can be called to dump all of their state to the system console. This is used along with a new feature in Nix that permits any system component to register a ‘‘C-t’’ function to be called by pressing the ‘‘control-t’’ and then the key assigned to the allocator. Thus, it is trivial to let the system dump new allocators added to the system to the console without having to modify anything to do so.

Another feature is the ability to report via */dev/alloc* of the status of the allocator. In this case, the allocator relies on statistics it maintains, including the number of structures free and in use, the number of times a selfish allocation could be done, and the number of total allocations performed. Like before, each allocator register a *summary* function with the console device, so it can report the status for all allocations without having to know which ones have been configured into the system.

Data structures

This structure represents a Nix allocator:

```
struct Nalloc
{
    char    *tag;
    int elsz;
    int selfishid;
    int nselfish;
    void    (*init)(void*);
    void    (*term)(void*);
    void*   (*alloc)(void);
    void    (*dump)(char*, void*);
    Lock;
    Nlink   *free;
    Nlink   *all;
    uint    nused;
    uint    nfree;
    uint    nallocs;
    uint    nfrees;
    uint    nselfallocs;
    uint    nselffrees;
};
```

The *tag* is a string describing the data structure; *elsz* indicates the element size in bytes, so it could rely on *malloc* to allocate more entries; the next two entries indicate the index in the selfish process array and the maximum number of structures to be kept there.

The function pointer permit the allocator to (re)initialize, terminate (i.e., clean up), allocate, and dump to the console each of the structures kept. The last fields are the statistics kept.

Each structure allocated with the Nix allocator simply keeps a *Nlink* field as its first field, which simply contains the pointers to chain the allocation list and the free list.

This is the complete implementation for allocation and deallocation:

```
void*
nalloc(Nalloc *na)
{
	Nlink *n;
	int id;

	id = na->selfishid;
	if(id-- != 0)
		if(up != nil && up->selfish[id] != nil && canlock(&up->selfishlk)){
			n = nil;
			if(up->selfish[id] != nil){
				n = up->selfish[id];
				up->selfish[id] = n->nnext;
				up->nselfish[id]--;
				na->nselfallocs++;	/* race; but stat only */
			}
			unlock(&up->selfishlk);
			if(n != nil)
				goto found;
		}
	lock(na);
	n = na->free;
	if(n != nil){
		na->free = n->nnext;
		na->nfree--;
		na->nused++;
		na->nallocs++;
	}
	unlock(na);
	if(n == nil){
		if(na->alloc != nil)
			n = na->alloc();
		else
			n = malloc(na->elsz);
		if(n == nil)
			panic("nalloc: no memory");
		lock(na);
		n->nlist = na->all;
		na->all = n;
		na->nused++;
		na->nallocs++;
		unlock(na);
	}
found:
	if(na->init != nil)
		na->init(n);
	n->nnext = nil;
	return n;
}
```

```
void
nfree(Nalloc *na, Nlink *n)
{
	int id;

	id = na->selfishid;
	if(na->term != nil)
		na->term(n);
	n->nnext = nil;

	if(id-- != 0 && up->nselfish[id] < na->nselfish && canlock(&up->selfishlk)){
		n->nnext = up->selfish[id];
		up->selfish[id] = n;
		up->nselfish[id]++;
		na->nselffrees++;	/* race; but stat only */
		unlock(&up->selfishlk);
		return;
	}
	lock(na);
	n->nnext = na->free;
	na->free = n;
	na->nused--;
	na->nfree++;
	na->nfrees++;
	unlock(na);
}
```

For safety, we still acquire a lock when using the selfish allocation, so that the allocator could be used also by interrupt handlers. When resources are not acquired/released by interrupt handlers, no locks would be necessary. We plan to add a flag to each allocator indicating if it can operate in selfish mode without locks, but have not done so yet.

Usage and evaluation

This is an example usage for the allocator:

```
Nalloc pathalloc =
{
	"path",
	sizeof(Path),
	Selfpath,	/* selfish per-proc alloc id */
	10,		/* up to 10 paths in selfish */
	nil,		/* no init */
	paterm,
};
```

In this case, it allocates *Path* structures and keeps up to 10 free elements in the selfish list. There is no initialization and the termination function cleans up the deallocated *Path* to prepare it for reusing. Allocating a path can be done as in

```
	path = nalloc(&pathalloc);
```

and deallocation is imply

```
	nfree(&pathalloc, path);
```

It is interesting to mention a bit of the evaluation done so far. We refer to the output from */dev/alloc* reporting usage statistics:

```
% cat /dev/alloc
3473408/67108864 cache bytes
72/100 cache segs 0/0 reclaims 1 procs
0/7 rpcs
98/102 segs
12/100 text segs 0/0 reclaims
1071104000 memory
15663104 kernel
0/0 1G pages 0 user 0 kernel 0 bundled 0 split
0/496 2M pages 0 user 0 kernel 0 bundled 6 split
423/968 16K pages 229 user 194 kernel 0 bundled 0 split
82/99 4K pages 0 user 82 kernel 3792 bundled 0 split
6/6 pgas
1447/1509 pgs
147/167 name 4372/4593 self allocs 4384/4458 self frees
8/8 walkqid 5965/5973 self allocs 5973/5973 self frees
169/181 path 5535/5716 self allocs 5563/5575 self frees
109/130 chan 6595/7392 self allocs 6629/7317 self frees
13928704/14192640 malloc 1 segs
60/66 block 13866/14414 self allocs 13924/14412 self frees
0/8388608 ialloc bytes
61/82 mmu 4096 pages
```

This output was collected after booting the system and recompiling *acme* from sources. The interesting lines are those near the end including ‘‘self allocs’’ statistics.

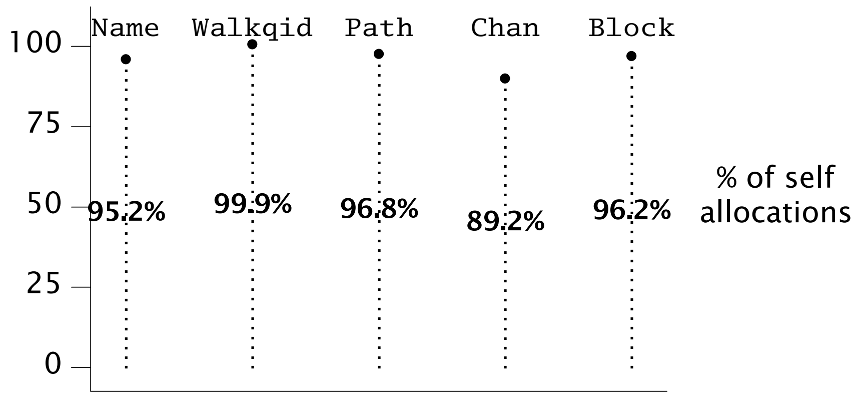

Figure 2 Allocation statistics for several data structures. The higher the percentage, the better. Optimal would be 100%. The selfish allocator permits a process to self-service the allocation most of the times, reducing the contention in the same proportion.

In particular, note how for *Name* data structures, 4372 times (out of 4593) allocation could proceed using the selfish allocators. In total, there are 147 structures in use, including those still kept in the per-process free lists. Thus, only a few of them had to go to the central *Name* allocator. Even so, this does not interlock with other allocators.

For *Walkqid* data structures, 5965 are selfish out of 5973. For *Path* structures, 5535 are self served out of 5716. For *Chan* structures, 6595 out of 7392.

Last, but not least, for I/O *Block* structures, those not larger than 128 bytes are kept on a small *Block* allocator. These are the most popular I/O block sizes, three orders of magnitude more popular than larger sizes (mostly because large RPC buffers are kept by the mount driver and reused by means of an RPC allocator).

Looking at */dev/alloc* output, we can see how 13866 times, out of 14414, blocks could be allocated by the process itself without disturbing others or the central *Block* allocator. Yet there are only 61 *Blocks* in use out of a total of 82 *Blocks* allocated.

It is important to repeat that this happens because such per-process free lists are not released when a process dies. Instead, they are kept on the *Proc* structure for the next process using the same process descriptor. As descriptors are allocated by locating always the first entry available in the process table, they are very likely to be reused soon, and the donations from the previous incarnation are usually an actual benefit for the system.

The next figure shows how the maximum size of the selfish allocator affects the number of self-allocations. We tried with different maximum sizes, from 1 to 50, and it can be seen how a small number of per-process free structures suffices in the case of *Path* allocation. Of course, this depends much on the type of system load, and on how many units of a resources are usually required at the same time by a single process.

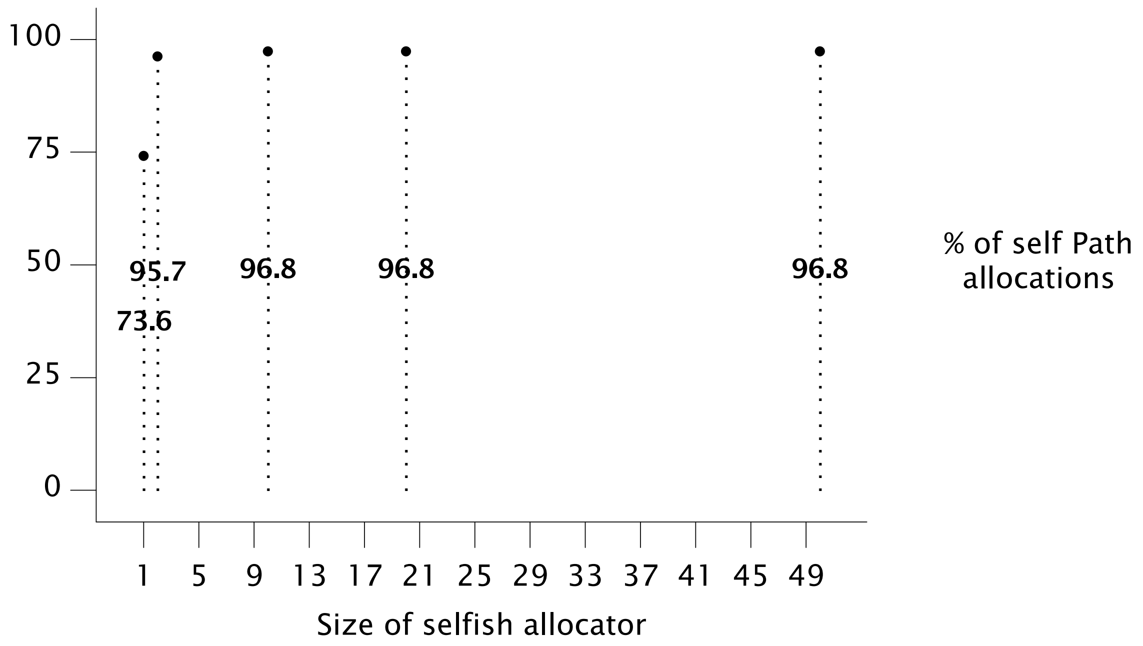

Figure 3 Allocations and self allocations for Path structures depending on the size of the selfish allocator (per process free structures).

As another experiment, we look at how the number of concurrent processes (compilers, in this case) affect the number of self allocations. Increasing the number of processes degrade the number of self allocations but only by an insignificant bit. It seems most processes have enough structures to satisfy most of the allocations required.

All the previous experiments were performed by rebooting the system and the performing the compilation used as a benchmark, and then gathering the allocator diagnostics.

In the next one, we repeated the same benchmark multiple times without rebooting the system, which shows how the age of the system affects the number of self allocations. The longer the system has been used, the more likely a self allocation could be performed. However, for the structure considered, the increase is insignificant.

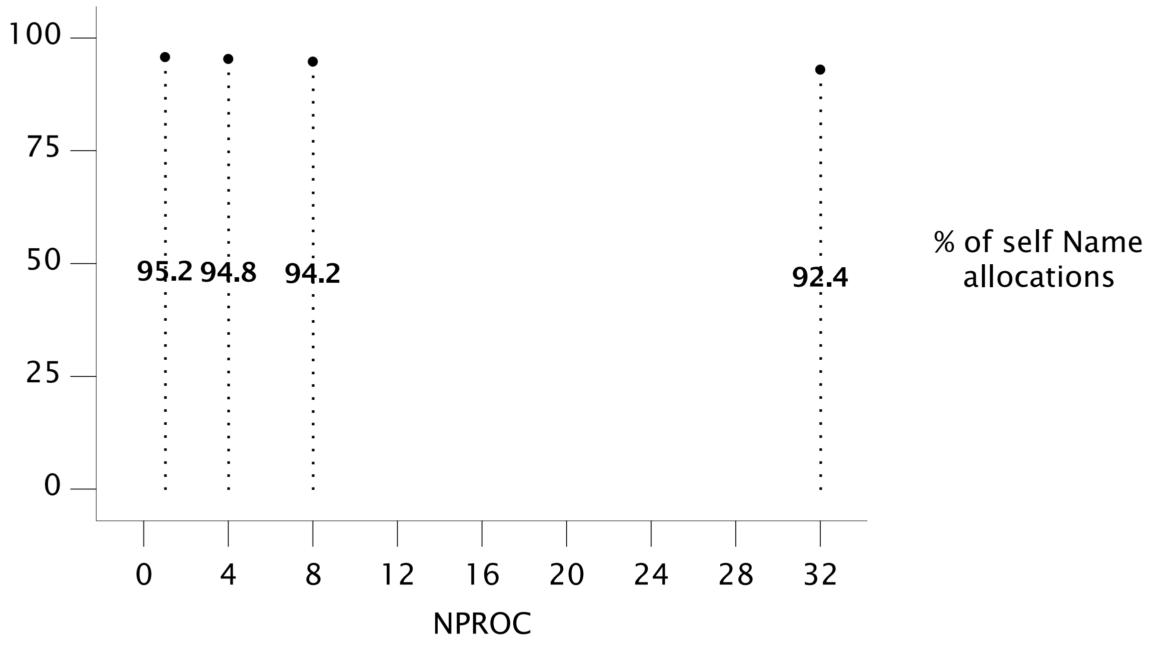

Figure 4 Allocations and self allocations for Name structures depending on the number of concurrent compilations (cold caches).

This seems to suggest that the number of successful self allocations may depend more on the pattern of the system load (eg., the number of structures required by a single process, and for how long are they acquired, etc.) than on the number of processes, processors and repetitions. The reason is that the system usually tries to load the first cores first, and tries to reuse the first structures (eg., process descriptors) first.

We repeated this experiment but looking at *Block* allocations instead. That is shown in the following figure.

Conclusions

The Nix selfish allocator can be considered a success. It makes almost every allocation a self-service process, such that the process making allocation can satisfy it without any synchronization with other processes or cores. For large and NUMA machines this is utterly important.

Only a few structures per process are enough to achieve a high rate of self-allocations, which means that resources are not wasted and are evenly distributed among those needing them.

Future work

More exhaustive evaluation should take place. In particular, we should try to determine which system load patterns lead to higher self-allocation rates and which ones do not.

Also, it would be interesting to let the allocators determine at run time their maximum size depending on the rate of self allocations for each data structure.

Last, we should explore which other data structures might benefit from selfish allocators besides the ones shown here. In particular, applying the same idea to other parts of the system seems to be a good idea.

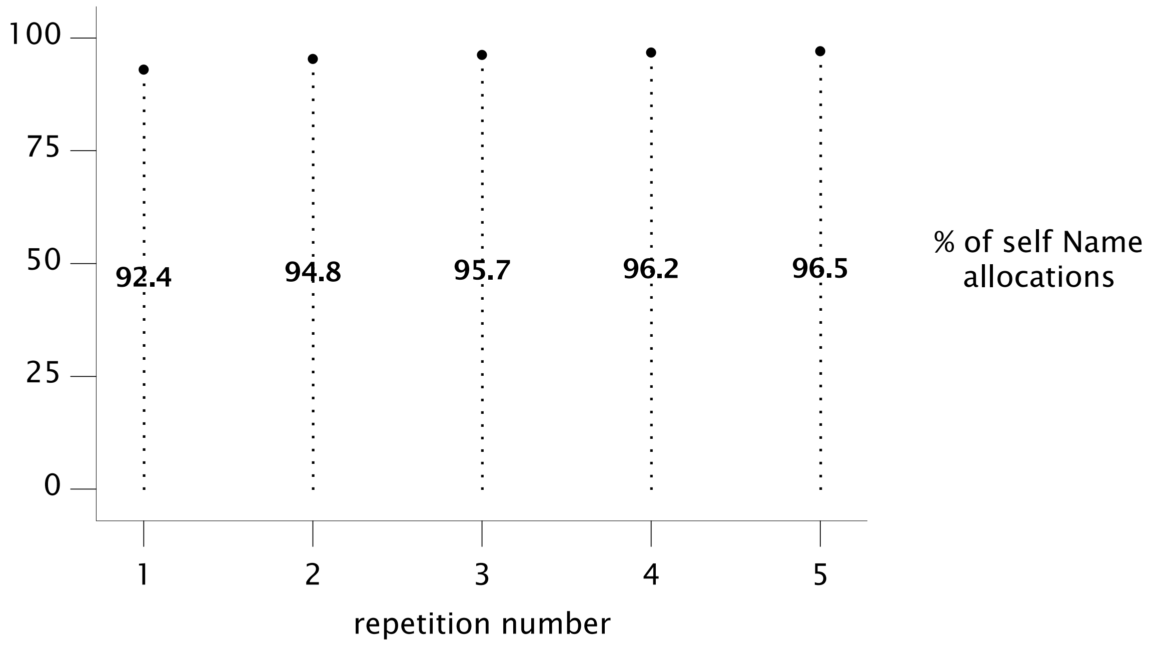

Figure 5 Allocations and self allocations for Name structures for repeated measurements (warm caches).

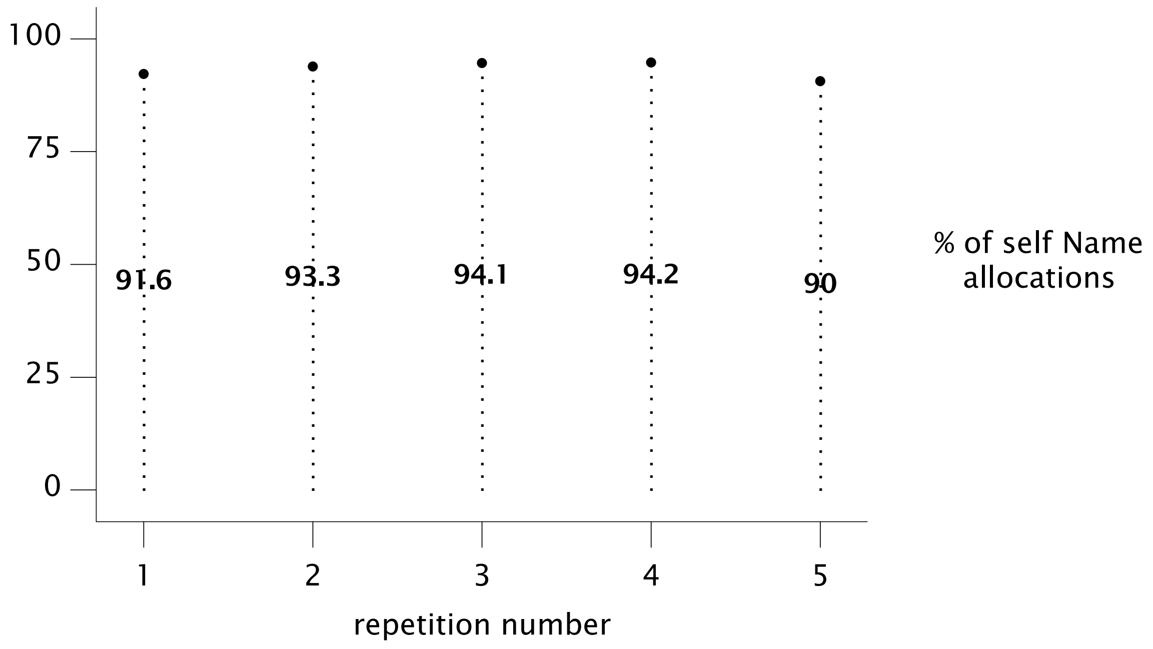

Figure 6 Allocations and self allocations for Block structures for repeated measurements (warm caches).

References

1. J. Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, *USENIX Technical Conference*, 1994.