A WAIT-FREE DYNAMIC STORAGE ALLOCATOR BY ADOPTING THE HELPING QUEUE PATTERN

Philippe Stellwag, Jakob Krainz, Wolfgang Schröder-Preikschat
Friedrich-Alexander University
Erlangen-Nuremberg
Computer Science 4
Martensstr. 1, 91058 Erlangen, Germany
{stellwag,krainz,wosch}@cs.fau.de

ABSTRACT
Most of the real-time applicable dynamic storage allocators rely on conventional locking strategies for protecting globally accessible data. But it is common that lock compositions do not scale well under high allocation and deallocation rates in parallel scenarios, as they lead to convoy effects. Furthermore, lock compositions lead to jitter, which is often a critical factor in real-time systems. Additionally, it is often desirable to guarantee progress of threads in order to be able to determine the worst-case execution time.

This led us designing a wait-free dynamic storage allocator (DSA), which can guarantee progress of threads and does not influence other threads to make progress. Our DSA implementation relies on a kind of buddy strategy with approximate best-fit. Hence, it ensures for this kind of allocation strategy typical memory wastage as a result of internal fragmentation. Preliminary tests show that we can outperform established DSA implementations in terms of predictability, like the famous TLSF memory allocator. To the best of our knowledge, our DSA is the first known approach using a scalable and bounded nonblocking synchronization strategy.

Our approach towards a wait-free DSA algorithm is applicable in real-time applications where adequate a priori knowledge about the memory requirements is available because it uses a statically allocated heap. We think that most real-time systems — especially ones with hard timing constraints — fulfill this precondition.

KEY WORDS
Storage Allocator, Malloc, Wait-Free, Real-Time Systems

1 Introduction

DSA is well studied in the field of general-purpose applications. Most of the DSA algorithms offer good average response times and a good overall performance [10]. But in the sector of real-time systems DSA algorithms are rarely used because of several major drawbacks, such as wastage of memory (fragmentation problem), nondeterministic and unbounded worst-case response times as well as performance issues [12]. There are some DSA algorithms, which try to satisfy the needs of real-time systems, such as the TLSF memory allocator [10]. There are also some real-time aware kernels (e.g., RTAI or MaRTE OS), which offer the functionality of TLSF to their applications.

In parallel real-time systems, running on shared-memory multi-core processors, the use of lock-based DSA algorithms can be very dangerous and inefficient. As an example, the TLSF memory allocator uses one lock to bring concurrent allocate, reallocate and free operations in a sequential order. Without care, this can lead to missing of deadlines of tasks [13] and inefficient exploitation of parallel cpu performance. Furthermore, the use of locks induces jitter, which can be a critical factor, as it can lead to timing anomalies [13]. As Lever et al. wrote in [9], most DSA algorithms only scale up to quad-core processors.

Nonblocking DSA algorithms, such as the malloc from Dice and Garthwaite [3], can be a good remedy for these drawbacks. But nonblocking memory allocators with a stronger progress condition than lock-freedom, called wait-freedom, do not yet exist. Wait-freedom guarantees that each thread completes its operation within a bounded number of steps [4] and hence, satisfies the temporal needs of real-time systems. Therefore, we present a wait-free DSA algorithm by adopting the helping queue pattern, which was firstly described in [14].

The helping queue pattern was originally used to improve performance of a wait-free queue. The original queue was implemented by an array. The queue operations traverse over this array and try to atomically allocate an element. The helping queue was introduced to speed this up. It consists of wait-free linearizable counter operations[1]. Each counter indicated the number of free elements in some part of the array; enqueue operations traverse the helping queue and try to decrement the counter. If the decrementing was successful, the corresponding array part is guaranteed to contain at least one free array element. The performance increases from $O(array\_size)$ to $O(\sqrt{array\_size})$, provided the helping queue is dimensioned appropriately. We exploit one side effect of this approach: If one array part is completely free, atomically decrementing the counter by its total content, that is to zero, reserves the total array part. If the array part is contiguous, it could be used for data structures that require more memory than one element can

[1] as described in Sec. 3.1

deliver. We generalized this approach to binary trees and implemented a DSA, which is able to satisfy arbitrary concurrent requests.

Our approach uses a statically allocated fixed-size array of 512-bit aligned memory blocks, which are organized in a balanced binary tree. Hence, sizes of memory blocks are proportional to some power of two. We use an approximate best-fit strategy [10]. The state of free memory blocks is managed on the basis of linearizable [5] wait-free counters, which are atomically decremented and incremented using the fetch-and-add instruction (FAA) available on, for example, the x86 architecture.

The usable size of memory per allocated chunk of k blocks is restricted to $k \cdot 512 - 96$ bits, since we use three integer variable of 32-bit length for internal state administration of memory blocks. The maximal size of memory, which can be allocated, is restricted to 4 GB minus the size of the mentioned state administration of 96 bits, because our DSA approach is currently optimized to 32-bit architecture only. Possible values for k are $2^d - 1$ for some d.

1.1 Motivation

The use of DSA algorithms can have significant benefits compared to the common static storage management [10] in real-time applications.

A central class implementation of a DSA — with maybe different behavior variants — can breed smaller code size compared to the common practice of object-specific, use case individual implementations with a rather substantial portion of redundant code.

Additionally, using unique interfaces for memory allocation, reallocation and deallocation is rather straightforward compared to using different memory pools with different semantics and behavior for different use cases.

Moreover, the possibility of using DSA offers some kind of flexibility at runtime [6], even if the heaps are allocated statically during the initialization of the application.

Up-and-coming multi-core processors partly force developers to rethink their memory management implementations in terms of scalability. The use of an appropriate DSA can satisfy not only these scalability requirements, but also leads to a more well-defined program structure [6].

1.2 Outline

The paper is organized as follows: Sec. 2 describes the requirements of a real-time applicable DSA algorithm. In Sec. 3 we present the idea, algorithm, worst-case execution time, verification, evaluation and conclusion of our wait-free malloc approach. Related work is shown in Sec. 4, in Sec. 5 we summarize our results and experiences, and give an overview over further work.

2 Requirements

The following points have to be considered for an applicable wait-free DSA. They are the basis of our solution presented in the following section.

Correctness. Conventional locking strategies with spinlocks or semaphores can suffer from multiple problems. The incorrect use of locks can lead to bounded as well as unbounded priority inversion, starvation, deadlocks, livelocks and race conditions; additionally, locks can lead to convoy effects and introduce jitter [14]. These effects have to be avoided as far as possible. Furthermore, a correct DSA implementation has to satisfy the following conditions: (1) Allocated memory locations have to be valid. (2) Allocated memory locations have to be at least as big as requested. (3) Allocation requests may not influence already allocated memory areas. (4) And allocated memory locations may not overlap.

Minimal fragmentation. Memory is often a very limited resource, in particular in the cost-sensitive sector of real-time systems. Moreover, such systems run over a long period of time without reboot. The resulting requirement is to waste as little memory as possible.

Bounded response times. In real-time systems it is essential to fulfill the timing specifications. This makes the worst-case execution time (WCET) of any operation an important parameter.

Fast response times. It is essential to guarantee fast response time of memory operations to satisfy the needs of computationally intensive applications.

Minimal jitter. As described above, jitter is often a critical factor in real-time systems. Hence, we have to minimize the jitter introduced by our protocols.

Scalability. In the future we expect a continuous increase in the number of available processor cores. In order to take advantage of this development, the number of threads has to grow as well. This requires an approach that is able to deal with a growing number of parallel operations, which allocate, reallocate and free memory.

No assumptions about timing. Allocation, reallocation as well as deallocation of memory can occur at any time. This means that we cannot make any assumptions regarding when operations will actually occur.

3 Our Wait-Free DSA

We succeeded in designing a wait-free DSA using a kind of buddy allocation strategy with approximate best-fit to the principle of atomic reservation of a memory area.

3.1 Idea

The available memory is divided into predefined nodes or memory blocks that form a balanced binary tree. The tree has a predefined depth d and is completely filled, that is it has $2^d - 1$ nodes. The structure of the tree does not change at runtime, as it is allocated statically.

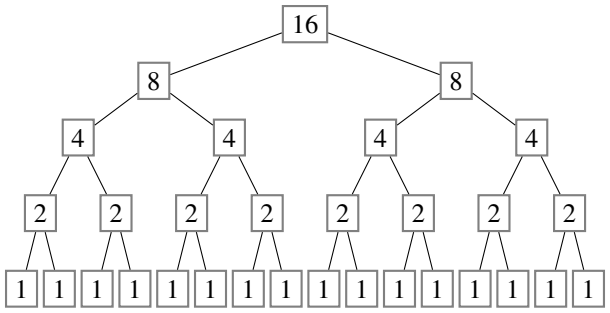


Figure 1. The initial state of the counters inside the tree of depth 5.

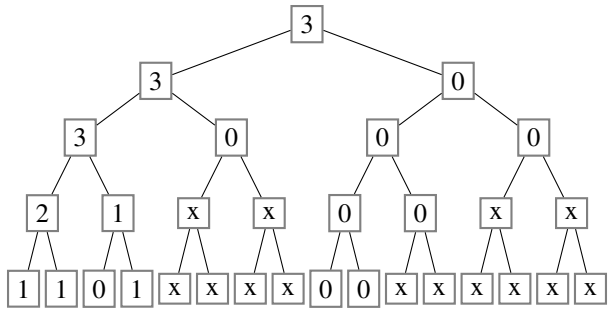


Figure 2. The tree of Fig. 1 with six allocations of memory — three of size 1, one of size 3 and two of size 7.

The algorithm only allocates complete subtrees; that is a successful allocation request allocates a node and the complete subtree that starts at this node. This implies that possible allocation sizes are of the form $(2^n - 1) \cdot (size\_of\_one\_node)$ for some n smaller or equal to d.

Each node has a counter, which contains information on the number of free nodes in the node's subtree. The counter is required to support the following two operations:

$atomic\_incr(c, S)$ This operation atomically increments the counter c by an given amount S and returns. There is no return value. Therefor we used the $xadd$ instruction available on, e.g., the Intel architecture.

$atomic\_decr(c, S)$ This operation atomically decrements the counter c by an given amount S, iff the counter is at least as great as S. It returns $true$ on a successful decrement. Otherwise the operations returns $false$ and leaves the counter unchanged. To realize the conditional part of this operation, we use two consecutive and interruptible $xadd$ operations. The impact of this approach is discussed in Sec. 3.4.

A subtree with depth d has $2^d - 1$ nodes and 2^{d-1} leaves. The counter of the subtree's root node is initialized to the number of leaves in the subtree. After the initialization the counter contains at most the number of unallocated leaves in this subtree. In Fig. 1 the initial state of an example tree of depth 5 is shown.

If a subtree is allocated, its counter is zero; the values of the potential other counters in this subtree are undefined. Note that a counter of zero does not always mean that the subtree starting at the node is itself allocated. For example, Fig. 2 shows the state of this tree after several successful allocations. The character 'X' indicates undefined counters. Note that the right child node of the root node has a counter of zero, even if it has not been allocated itself. Instead of that all of its sub-subtrees have been allocated.

An allocation request that needs a subtree of depth d, i.e. with $2^d - 1$ nodes, has the number of leaves of this subtree, that is 2^{d-1}, as size. Correspondingly, a deallocation request that frees a subtree of depth d with $2^d - 1$ also has size 2^{d-1}.

3.2 Algorithm

3.2.1 Allocation

The allocation algorithm starts at the root node and traverses the tree recursively in depth-first order. It tries to allocate memory by atomically decrementing the counter of a root node to zero.

A simplified pseudo-code version of the algorithm is illustrated in Fig. 3. The recursive method is called with the root node and the number of leaves in the wanted subtree as initial arguments.

It tries to decrement the counter of the current node by the size of the allocation request[2]. If this fails, it returns an indication of failure; in our case $NULL$. The counter of the current node contains at most the number of unallocated leaves in the subtree[3]. Thus, the algorithm will not continue if the subtree does not contain enough memory.

It then checks whether the subtree starting at the current node already has the requested size, in which the node has been successfully allocated and its address is returned.

If the size of the subtree is greater than the requested size, the algorithm tries a recursion call for each of the two child nodes of the current node. If one of these was successful, the allocated node is returned. If neither of the two recursive calls was successful, the algorithm re-increments the counter of the current node and returns an indication of failure.

3.2.2 Deallocation

The deallocation algorithm is even simpler: Firstly, it resets all previously undefined counters of the subtree to be deallocated to their initial values. Then, it starts at the root node of the subtree that is to deallocate, walks up the tree towards the root node of the whole tree and increments every passing node by the number of leaves of the subtree that

[2] i.e. by the number of leaves that a subtree of sufficient size has to have

[3] See Sec. 3.4 for an explanation of the difference; it is sufficient to know that the counter is never greater than the number of unallocated leaves.

```
try_allocate(node N, size S) {
    if(atomic_decr(N.counter, S)) {
        if(S == size of subtree
            starting at N) return N;
        for C in both childnodes {
            node tmp = try_allocate(C, S);
            if(tmp != NULL) return tmp;
        }
        atomic_incr(N.counter, S);
    }
    return NULL;
}
```

Figure 3. Pseudo-code of the allocation algorithm

```
release(node N) {
    reinitialize_subtree(N);
    S := number of leaves of subtree
        starting at N;
    atomic_incr(N.counter, S);
    while(N != root node) {
        N := parent of N;
        atomic_incr(N.counter, S);
    }
}
```

Figure 4. Pseudo-code of the deallocation algorithm

is to be deallocated. A simplified pseudo-code version of the deallocation algorithm is shown in Fig. 4.

3.2.3 Transformation of the Tree to an Array

The tree structure can be easily transformed to an array. This transformation has to fulfill the condition that nodes in each subtree have to occupy a contiguous slice of the array.

This transformation can be defined recursively: A tree consisting of one leaf is equal to one array element. Every other tree consists of one root node and two subtrees of the same size. It is arranged as follows: Firstly, both subtrees are recursively converted to arrays. The array that represents the whole tree is then constructed by concatenating the array of the left subtree, then the array of the right subtree and at last one array element for the root node.

The result is shown in Fig. 5, where a tree of depth 4 is translated to an array of size 15.

Each node is represented as a struct as illustrated in Fig. 6.

For n consecutive allocated nodes, the allocation and deallocation algorithms use the member `length` of the first node and the members `padding` and `counter` of the last node. Everything between them is usable as user data by the respective application. This is illustrated in Fig. 7.

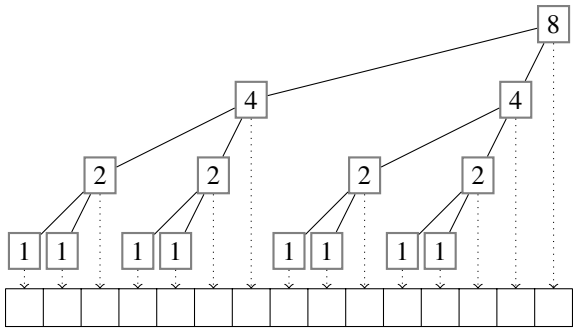


Figure 5. Translation of the tree from Fig. 1 to an array.

```
struct node {
    uint32_t length;
    uint32_t data[13];
    uint32_t padding;
    uint32_t counter;
};
```

Figure 6. Representation of a node in memory

The address returned by our malloc is the start address of the `data` array in the first of the consecutive allocated nodes. The `length` member is required to find the root node of a subtree[4]. The `padding` member is used to detect buffer overruns; it is not strictly neccesary and can be omitted at the risk of not detecting heap corruption.

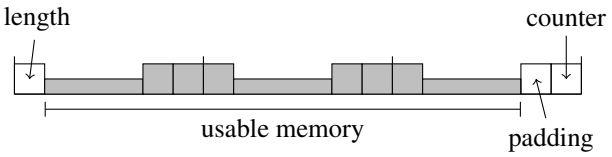


Figure 7. Three consecutive allocated nodes represented by structs

Every consecutive array of allocated nodes is equivalent to a subtree. The last node is the root node of the subtree. Its counter is accessible by the allocation and deallocation routines.

3.3 Worst-Case Execution Time

We determine the WCET for our memory allocator in this section.

Firstly, we examine the execution time of one single call of the allocation function, ignoring recursion calls. All actions apart from the recursion call have, by definition,

[4]The address passed to "'free"' belongs to the first node of the subtree (according to the layout of the tree in the array), while the root node is the last node.

upper execution bounds and the function does not contain loops, so the execution time required for one call itself is bounded.

As the function recursively traverses parts of the binary tree in depth-first search order, it is called only a finite number of times — namely at most once for each node.

Therefor, an upper bound for the execution time of the allocation can be easily established: The recursive function is executed at most

$$O(\frac{available\_mem}{size\_of\_one\_node}) = O(\frac{available\_mem}{64\ bytes}) \quad (1)$$

times.

Deallocation also has bounded response time. Firstly, the re-initialization of the nodes in the subtree is bounded. This re-initialization is done with a depth-first traversal of the subtree, which visits each node of the subtree at most once. Worst-case execution time for this is

$$O(\frac{size\_of\_mem\_chunk}{size\_of\_one\_node}) = O(size\_of\_subtree). \quad (2)$$

Secondly, the loop that makes up the rest of the function is executed a bounded number of times. This loop is executed once for each node on the path from the root node of the subtree that is freed to the root node of the total tree. The number of nodes on this path is obviously bounded; an upper bound is $\log_2(size\_of\_total\_tree)$. Therefor, deallocation completes in at most $O(size\_of\_subtree + \log_2(size\_of\_total\_tree))$ steps.

3.4 Verification

3.4.1 Correctness

As mentioned in Sec. 2, a correct implementation of a DSA needs to satisfy the following conditions to be correct:

1. Allocated memory locations have to be valid.
2. Allocated memory locations have to be at least as big as requested.
3. Allocation requests may not influence already allocated memory areas.
4. Allocated memory locations may not overlap.

Our algorithm works on a predefined array, hence the system designer is responsible to dimension the array so that all allocation requests can be satisfied. Additionally, if he or she ensures that the array is located in valid memory, we also guarantee that returned addresses are valid. Only if there is not enough free memory in the array available, the allocation operation returns $NULL$.

The second condition is also satisfied: Our algorithm returns a node only if the depending subtree is as big as needed. This is guaranteed by correctly designing the recursion of the allocation routine: If the size of the tree starting at the current node is known, the size of the two subtrees is also known. As we know the size of the total array and hence the total size of the tree, we can keep track of whether the current node and its subtrees have the correct size.

As the algorithm only modifies the counters of the nodes, the third condition is satisfied, if the algorithm only modifies the counters that do not lie inside allocated memory. As shown in Sec. 3.2.3, of all the counters in an allocated subtree, only the counter of the root node's subtree is not inside the allocated memory, every other counter in the subtree may not be touched.

The subtree is allocated, so the counter of the subtree's root node is zero. Thus, no allocation or deallocation request will descend into the node's subtree. Every allocation request will first try to decrement the node's counter and will fail; it then will not descend into the subtree. Deallocation requests do not descend into subtrees anyway. The only way a deallocation request gets to change the subtree's root node's counter is, if it tries to deallocate the same subtree, which is valid. Thus, counters inside the array remain untouched by the algorithm.

The fourth condition is equivalent to the following condition, which is somewhat easier to prove: From the moment of allocation until the deallocation of an area of memory, every other concurrently allocated area will not overlap this area.

Our allocation algorithm deals with trees and only allocates complete subtrees. Thus the only possible way two areas can overlap is, if one area is contained in the other, that is if one subtree is a sub-subtree of the other subtree. To show that our algorithm also satisfies the third condition, it is sufficient to show that if a node is allocated, there is no other node in its subtree that is already allocated and there is no other pending allocation request inside the subtree.

To prove this, we introduce the following condition, which is always true, if the implementations of the two counter operations are atomic as mentioned in Sec. 3.1:

Let n_i be any node in the tree; let L_i be the set of leaves of the subtree that starts at n_i. Let $c(n)$ be the value of the counter of the node n; let $a(n)$ be true, if and only if the node n is currently allocated. Let furthermore R_i be the set of requests inside the tree starting at n_i, meaning the set consisting of every pending allocation request that has decremented the counter of n_i and every deallocation request that has not yet incremented the counter of n_i. For each request r, the function $s(r)$ shall return the size of the request.

For each node n_i and its counter $c(n_i)$ the following relation holds: The value of the counter plus the number of allocated nodes in the corresponding subtree plus the sum of the sizes of all pending allocation and deallocation requests that are currently in the subtree is equal to the number of nodes in the subtree.

Formally speaking, the equation 3 holds for every node n_i.

$$c(n_i) + |\{n_j \in L_i, a(n_j)\}| + \sum_{r \in R_i} s(r) = |L_i| \quad (3)$$

Equation 3 is true for a freshly initialized tree. By examining the following five possible changes that can be caused by allocation as well as deallocation requests, it can be proven that equation 3 always holds:

1. Any allocation request might enter the subtree. This is achieved by atomically decrementing the counter of the node; hence the equation 3 holds.
2. Any allocation request might leave the subtree unsuccessfully. It then atomically increments the counter by the size it tries to allocate, so the equation 3 stays true.
3. Any allocation request might leave the subtree successfully. This means it succeeds in allocating a sub-subtree of the subtree. Thus the number of allocated nodes in the subtree atomically increases by the size of the allocation request, so the equation 3 stays true.
4. Any deallocation request might enter the subtree. Deallocation requests enter at the root nodes of allocated sub-subtrees when they deallocate the sub-subtree by atomically incrementing the counter of the sub-subtree's root node. Therefor the number of allocated nodes atomically decreases corresponding to the increasing sum of sizes of pending requests, so the equation 3 holds.
5. Any deallocation request might leave the subtree. This happens when it atomically increments the counter of the subtree's root node, before walking further up the tree. In this case, the counter atomically increases correspondingly to the decrease of the sum of sizes of pending requests, so the equation 3 stays true.

From this point of view it is easy to prove that if an allocation request successfully allocates a node, then there are neither allocated sub-subtrees of the subtree of the node, nor are there any pending requests inside the subtree.

This is a trivial consequence of the employed algorithm: If an allocation request tries to allocate a node's subtree, it tries to atomically decrement its counter by the size of the subtree. If this succeeds, the number of allocated nodes in the corresponding subtree plus the sum of the sizes of all pending allocation and deallocation requests that are currently in the subtree must have been zero.

An allocation request of size n that has successfully decremented the counter of a node whose subtree consists of n nodes has allocated this subtree. There can be no other pending allocation requests in this subtree; they would have decremented the counter of the node, so it would have been less than n and the previously mentioned allocation request would have failed.

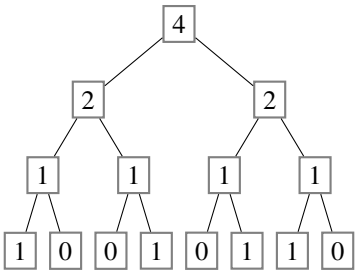
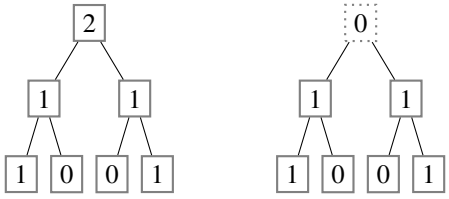


Figure 8. A highly fragmented tree with four allocations of memory, each of size 1.



(a) State A (b) State B

Figure 9. An example tree of depth 3, to show why our DSA is not linearizable.

3.4.2 Linearizability

It might be a disadvantage of the employed algorithm that it is not linearizable [5], that is a parallel execution exists that cannot be formed into a respective sequential execution.

This can be shown with an simple example: Consider the tree in Fig. 9(a). If two allocation requests, one with size 2 and another with size 1 were executed in parallel, then both requests potentially failed, iff the request with size 1 is issued within the time frame where the request with size 2 has already decremented the counter of the root node. This scenario is illustrated in Fig. 9(b). Note that there are no consecutive nodes to satisfy this allocation request of size 2. Hence this request will re-increment the counter of the root node when it recognizes that there is not enough contiguous memory available. Within this time frame the allocation request of size 1 will be rejected, because the counter of the root node is 0.

This implies that our DSA algorithm is not linearizable, because the parallel execution history cannot be formed into a sequential one, as the sequential execution of both requests guarantees that the requests with size 1 will always succeed. This does not pose to be a problem, but within this time frame memory cannot be allocated even if there is enough to satisfy incoming requests.

We can make a weaker guarantee though: For each concurrent history of allocations without allocation failures there is an equivalent sequential history without failures. Furthermore, for each concurrent history of allocations, where only the last allocation fails, there is either a equivalent sequential history where the last allocation also

fails, or there is a sequential history which is equivalent except for the last allocation and whose last allocation does not fail.

Unfortunately, this does not extend to arbitrary concurrent histories. As some early failing allocation in the concurrent history might succeed in the sequential history, it is possible that later on there is not enough memory available in the sequential history, so later on allocations that did succeed in the concurrent history might fail in the sequential history. Note that this is only possible, if there was an allocation failure in the concurrent history. Basically, transformation of the concurrent history to a sequential history makes allocation failures happen later and possibly less often, but not more often.

3.5 Evaluation

In this section we define our testbed and some micro benchmarks to briefly[5] evaluate our wait-free DSA implementation to two famous DSA implementations.

3.5.1 Test Setup

As a highly parallel hardware environment we used a 16-core machine, consisting of four Intel Xeon E7340 quad-core processors, running at a 2.40 GHz clock frequency. Each CPU had 256 KB L1 cache for data and instructions as well as 4 MB L2 cache per core pair (i.e. 8 MB per CPU) and 1,066 MHz FSB. Additionally, this machine had 32 GB main memory and was running a installation of Debian Linux 5.0.2 with a 2.6.29.3 vanilla kernel.

3.5.2 Test Method and Scenarios

For experimental results we implemented four test cases and evaluated the results of our approach to the TLSF DSA. Additionally, we also measured the times of the famous DSA from Michael [11] found in the atomic_ops project[6], even if it is not fully comparable with TLSF and our approach in terms of temporal requirements.

In each test case, we started 16 threads with the highest real-time priority on different idle cores. Furthermore, we dimensioned the heap to 64 megabytes.

Test case I: Here we measured the time required to allocate 948 bytes of memory. The allocated memory is not freed. The test case terminates if no more memory is available.

Test case II: Here we measured the time required to allocate 948 bytes of memory and immediately free them again. The test case terminates after a predefined number of 20,000 attempts.

Test case III: Here we measured the time it takes to allocate memory in exponentially increasing sizes, starting with 10 bytes and growing with a factor of approximately 1.25. Similar to test case I, the allocated memory is not freed. The test case terminates, if no more memory is available.

Test case IV: Here we measured only the time it takes to allocate memory in exponentially increasing sizes, as mentioned in test case III. After allocation the memory is freed immediately. The test case terminates, if no thread can allocate as much as it wants to.

[5] with respect to the already exploited space restriction of this paper

[6] Atomic_ops project: www.hpl.hp.com/research/linux/atomic_ops

	I	II	III	IV
$min : our$	5,006	2,252	1,114	467
$tlsf$	6,485	612	459	416
$atops$	629	901	629	246
$med : our$	19,975	27,557	29,444	12,019
$tlsf$	14,475	19,975	30,600	22,788
$atops$	13,846	20,799	13,642	1,802
$max : our$	29,707	37,111	57,264	36,788
$tlsf$	6e+05	3e+05	536,698	362,585
$atops$	367,438	372,699	1.1e+08	2.0e+07
$avg : our$	20,046	27,469	30,551	11,662
$tlsf$	54,301	46,791	74,580	47,387
$atops$	48,500	53,998	777,530	131,865
$c_v : our$	0.15	0.14	0.35	0.6
$tlsf$	1.7	1.3	1.3	1.3
$atops$	1.4	1.4	8.5	9.3
$util. : our$	92.58%	./.	66.79%	./.
$tlsf$	97.92%	./.	98.99%	./.
$atops$	92.49%	./.	17.32%	./.

Figure 10. Results in cpu clock cycles of our test cases

3.5.3 Results

Our results are shown in Fig. 10. We use the acronyms our for our implementation, $tlsf$ for the real-time memory allocator TLSF and $atops$ for Michael's lockfree DSA. Additionally, we are using c_v for the coefficient of variation.

As shown in Fig. 10, our DSA achieves the least dispersion about the mean, as the coefficient of variation c_v is very small. Hence, the times for allocation and deallocation of our memory allocator are highly deterministic. TLSF uses one global lock to bring concurrent allocate, reallocate and free operations in a sequential order; lock contention is clearly expensive in terms of performance and determinism. In contrast, Michael's DSA suffers from starvation, which is much more expensive in our test cases. Our approach does not use any blocking locks and guarantees progress, which leads to a much smaller WCET.

Without taking the outliers into account, all DSA implementations show a good overall performance, as shown in the according median med values. However, the average response times avg of our approach are much smaller than the others, if we take the outliers into account.

Furthermore, our tests show that memory consumption is typical of buddy allocation; if the requested sizes

fit well, the memory wastage is acceptable. If the sizes do not match the possible tree sizes, internal fragmentation increases, which is typical for buddy allocation strategies.

3.6 Conclusion

Referring to the requirements mentioned in Sec. 2, we can conclude that most of them are indeed satisfied.

We evaluated our DSA in terms of correctness in Sec. 3.4.1. Indeed our protocol is not linearizable as shown in Sec. 3.4.2 (i.e. an allocation request will sometimes fail to allocate memory even there is enough memory available), but from a practical perspective this situation happens only if we are running very close to an out-of-memory scenario with high fragmentation. If we enlarge the memory in Fig. 9 to size 8, there is no possible execution history, where one of the four allocation requests of sizes 1, 1, 2 and 1 can be rejected. Therefore our weaker correctness condition may not be a problem from a practical point of view.

Moreover, our algorithm has bounded response times as mentioned in Sec. 3.3, which — ignoring the cache effects — is dependent on the external fragmentation of the available memory. The internal fragmentation is typical for buddy allocation strategies. It is up to the application design to prevent a high fragmentation. This also leaves room for further research efforts.

Additionally, our algorithm is completely interrupt transparent, i.e. it can also be used in an interrupt handler and does not introduce bounded priority inversion scenarios like approaches with conventional locking strategies. Moreover, it is highly scalable, as the wait-free property ensures that DSA operations cannot influence other threads to make progress.

Compared to other DSA, the response times of our approach are highly predictable, as a result of minimal jitter. Especially conventional lock contention of memory allocators, e.g., TLSF, lead to highly unpredictable response times and respectively to a very pessimistic WCET.

4 Related Work

There is a lot of published work on dynamic storage allocation. For brevity, we only present the most important work for the scope of this paper.

Dice and Garthwaite found out that typical DSA implementations did not scale well in high-order multiprocessor systems. They present in [3] a mostly lock-free malloc by using multi-processor restart-able critical sections (MP-RCS). Their MP-RCS implementation needs a kernel driver to modify the decisions of the scheduler where a thread will run (either in a notification routine or at the original interrupted instruction). Without a priori knowledge about the application and the blocking system calls, it is not possible to determine an upper bound for their malloc operations.

Michael presents a scalable lock-free DSA in [11]. He claims that his DSA can be used even in real-time applications. Nonetheless, lock-freedom only guarantees that at least one thread makes progress at each step and hence can lead to starvation of some other threads. Without further a priori knowledge about the application, we cannot determine an upper bound for the response times of his DSA operations. Anyway, we used his DSA for our evaluation as it offers the best performance on lockfree DSA.

As mentioned in Sec. 1, Masmane et al. present in [10] a memory allocator for real-time systems, called two-level segregate fit (TLSF). It offers fast and bounded response times with an average fragmentation lower than 15 percent. However under stress situations TLSF did not scale well, as it uses one global lock to bring concurrent allocate, reallocate and free operations in a sequential order. Furthermore, in preemption scenarios it suffers from bounded priority inversion, which introduces jitter.

Puaut presents in [12] a case study of performance measurements of different DSA implementations (e.g., sequential fits, indexed fits, segregated fits and buddy systems). She evaluated their results under real and synthetic workloads in terms of suitability in real-time systems. She found out that the best technique for DSA highly depends on the use case. As an example, in scenarios with low allocation/deallocation rates buddy systems and quick-fit strategies are the best-qualified techniques in terms of predictability as well as performance.

Also Kriemann presents in [7, Chapter 8] performance measurements of a variety of DSA implementations investigated under real and synthetic use cases. Unfortunately, the scope of his PhD thesis does not cover temporal requirements.

Herter et al. present in their work-in-progress paper [6] an approach of a DSA that makes cache performance predictable and allocates as well as deallocates memory in constant time. Their DSA is a cache-conscious modification of TLSF. Their approach allocates/deallocates from segregated lists to achieve a constant time. Additionally, they added a new parameter to their malloc, which offers the possibility to specify a dedicated cache set. This induces the requirement that every developer must be familiar with the cache of the used processor. Investigations for multi-core processors were not made.

There are also several approaches for multi-core aware DSA for general-purpose systems. A few of them are: Hoard [1], LKMalloc [8], Vee and Hus's allocator [15] and Vmem [2]. All approaches use locks to protect the heap data for concurrent accesses.

Our work is primary inspired by [14], where we present a wait-free linearizable helping queue mechanism for an unsorted queue. In [14] 'local preferences' were introduced to heavily minimize contention on the queue elements. Locality is a desirable property in a parallel environment, as it ensures the avoidance of interferences with other threads. Additionally, a helping queue mechanism was introduced to heavily reduce the time needed to traverse the

queue in order to find a free queue element. In this paper we have adopted this pattern of the helping queue mechanism for our DSA implementation.

5 Conclusion and Further Work

To our knowledge, we designed the first wait-free approach of a dynamic storage allocator, that guarantees highly predictable response times and hence, can satisfy the temporal requirements of real-time applications.

There are several optimizations to the presented algorithm we did not evaluate. Firstly, it could be possible to improve the memory efficiency of the algorithm by changing the size of a node from 64 bytes to a different value. This might improve the memory utilization under some conditions.

Another possible optimization would be to change the behavior of the recursive traversal function in order to decrease the number of write operations, which are more expensive than respective read operations. It should be feasible to traverse the tree without changing the counters and, once we have found a 'suitable' subtree, walk down from the root node along the direct path. This change would likely increase performance under low concurrency and high fragmentation conditions. It would likely be detrimental to performance in a high concurrency case, even though execution time would still remain bounded.

Yet another possible optimization would be lazy re-initialization of nodes that are to be released. Currently deallocation takes time proportional to the size of the freed memory chunk, by re-initializing every node in the subtree. This is not necessary, if the next allocation request needs memory of the same size. One possible solution would be marking the root node of a freed piece of memory with a flag and omitting the re-initialization of the subtree. A subsequent allocation request that encounters the flag and needs to descend into the subtree can take over as much re-initialization as needed. This optimization would most likely increase the performance of deallocation, at a slight cost for the allocation performance.

References

[1] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *in Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1997.

[2] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab allocator to many cpus and arbitrary resources. *in Proc. of the USENIX Technical Conference*, 2001.

[3] D. Dice and A. Garthwaite. Mostly lock-free malloc. *in Proc. of the 3rd International Symposium on Memory Management*, 38(2):163–174, 2002.

[4] M. P. Herlihy. Wait-free synchronization. *in ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[5] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *in ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[6] J. Herter, J. Reineke, and R. Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. *in Proc. of the Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, Juny 2008.

[7] R. Kriemann. Parallele Algorithmen für $\mathcal{H}$-Matrizen. *Dissertation, Christian-Albrechts-Univ., Kiel*, 2004.

[8] P. Larson and M. Krishnan. Malloc() performance in a multithreaded Linux environment. *International Symp. on Memory Management (ISMM '98)*, 1998.

[9] C. Lever and D. Boreham. malloc() performance in multithreaded linux environment. *in Proc. of the USENIX Annual Technical Conference, San Diego, CA*, June 2000.

[10] M. Masmane, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. *in Proc. of 16th IEEE Euromicro Conference on Real-Time Systems, Catania, Italy*, pages 79–88, July 2004.

[11] M. M. Michael. Scalable lock-free dynamic memory allocation. *in Proc. of the ACM SIGPLAN Conference on Programming Laguage Design and Implementation, Washington DC, USA*, pages 35–46, 2004.

[12] I. Puaut. Real-time performance of dynamic memory allocation algorithms. *in Proc. of the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria*, pages 41–49, June 2002.

[13] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. *in Proc. of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[14] P. Stellwag, A. Ditter, and W. Schröder-Preikschat. A wait-free queue for multiple enqueuers and multiple dequeuers using local preferences and pragmatic extensions. *in Proc. of the IEEE Symposium on Industrial Embedded Systems 2009*, July 2009.

[15] V.-Y. Vee and W.-J. Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. *in Proc. of the International Symp. of Parallel Architectures Algorithms and Networks*, 1999.