A Method for Automatic Optimization of Dynamic Memory Management in C++

Daniel Häggander, Per Lidén and Lars Lundberg
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
{Daniel.Haggander, Per.Liden, Lars.Lundberg}@bth.se

Abstract

In C++, the memory allocator is often a bottleneck that severely limits performance and scalability on multiprocessor systems. The traditional solution is to optimize the C library memory allocation routines. An alternative is to attack the problem on the source code level, i.e. modify the applications source code. Such an approach makes it possible to achieve more efficient and customized memory management.. To implement and maintain such source code optimizations is however both laborious and costly, since it is a manual procedure.

Applications developed using object-oriented techniques, such as frameworks and design patterns, tend to use a great deal of dynamic memory to offer dynamic features. These features are mainly used for maintainability reasons, and temporal locality often characterizes the run-time behavior of the dynamic memory operations.

We have implemented a pre-processor based method, named Amplify, which in a completely automated procedure optimizes (object-oriented) C++ applications to exploit the temporal locality in dynamic memory usage. Test results show that Amplify can obtain significant speed-up for synthetic applications and that it was useful for a commercial product.

1. Introduction

Most applications use a great deal of dynamic memory these days. As the trend towards larger applications continues, there will be even greater demands on the heap manager. Another trend that increases the importance of efficient heap management is the increasing popularity of C++ [12]. C++ programs by their nature use dynamic memory much more heavily than their C counterparts, and often for many very small, short-lived allocations [8]. Previous studies show that C++ programs invoke one order of magnitude more dynamic memory management than comparable C applications [3]. Modern software systems also tend to use various frameworks and design patterns. A main reason for this is to make the source code easy to maintain. Frameworks and design patterns introduce dynamic features. These features are to a large extent based on abstract object types and the introduction of new object types. In C++ this increases the use of dynamic memory even more. However, the dynamic features present in many frameworks and design patterns are mainly used to make the system adaptable to future changes. Thus, temporal locality will characterize the run-time behavior, where the same object structures tend to be created and used over and over again [9].

On multiprocessors, dynamic memory is an exclusive resource that must be protected against concurrent accesses by multiple threads running in the same multithreaded (C++) program. As a result of this, it is often not useful to add more computation power to a system by using multiple CPU:s, since frequent allocation and deallocation of dynamic memory becomes a bottleneck [11]. Several methods for solving this problem have been evaluated [10]. A common approach to make the memory management routines reentrant. This is, usually, solved by creating a dynamic memory manager that handles multiple heaps internally [1][4][5]. This way, the allocation and deallocation routines can work in parallel with a minimum need for mutual exclusion. Nevertheless, evaluations have shown that application specific design and implementation techniques, such as introducing handmade structure pools, is more beneficial in terms of performance [11]. However, this method suffers from some major drawbacks, which makes it a less attractive option. Using pools for reuse of object structures requires the programmer to incorporate these into every class by hand. This means that the programmer must be very familiar with the code. Further, these pools tend to be very tailored and optimized for a specific application, thus making them less reusable. As the system grows the pools must be maintained to reflect changes in the system. In the end, this all adds up to a time consuming activity requiring extensive knowledge of the system's internal structure. Also, since the pools are handmade there is a potential source for errors.

It would be beneficial if the process for incorporating structure pools in a system could be automated. Most of the identified drawbacks of this method could then be eliminated, or at least substantially reduced. We have approached this problem by implementing a pre-processor that inserts the optimizations into the code before it is compiled. The problems with high costs for maintaining a handmade version are eliminated, since it is only a matter of recompiling the code. Further, the potential errors caused by the programmer creating the pools are eliminated (as long as we assume that the pre-processor implementation is correct). Finally, there is no need for special expertise in this area among the developers in a team. Instead they can go on using the traditional programming and design methods and use the pre-processor when compiling the system.

Our results show that the performance speedup gained by using handmade structure pools can be preserved, however not completely, by using a pre-processor based automation. On synthetic programs, Amplify (our method) has shown to be up to six times more efficient, compared to the available, state of the art, C library allocators we have tested. This study also includes tests on a commercial product developed by the Ericsson Telecommunication Company, the Billing Gateway (BGw). BGw is a system for collecting billing information about calls from mobile phones. The application is written in C++ and is an example of a product dependent on a parallel memory management to scale-up on multiprocessors. The tests show a 17% performance increase when the BGw source code was pre-processed by a slightly modified version of Amplify.

The remaining pages are organized as follows. Section 2 describes in further detail problems surrounding dynamic memory. In Section 3 we describe the Amplify method. The test method is presented in Section 4. Our results can be found in Section 5, while Section 6 is a short presentation of related work. Finally, Section 7 concludes the paper.

2. Problems with dynamic memory

In C++ [12], dynamic memory is allocated using the operator `new`, which normally is an encapsulation of the C-library function `malloc()`. Allocated memory is deallocated using operator `delete`, which normally is encapsulating the C-library function `free()`. Many implementations of `malloc()` and `free()` have very simple support for parallel entrance, e.g. using a mutex for the function code. Such implementations of dynamic memory result in a serialization bottleneck. Moreover, the system overhead generated by the contention of entrance can be substantial [11]. The bottleneck can be reduced by using an optimized reentrant implementation of `malloc()` and `free()`, e.g. ptmalloc [4] or Hoard [1]. However, even better performance can be achieved by decreasing the number of heap allocations, e.g. by implement structure pools [9]. A lower number of heap allocations will also improve the performance on uni-processors, whereas using a reentrant implementation of `malloc()` and `free()` will only improve performance on multiprocessors.

In an object-oriented design, especially with design patterns and frameworks, a large number of objects are used. Creating an object often requires more than one dynamic memory allocation (call to operator `new`). The reason for this is that each object often consists of a number of aggregated or nested objects. For example, a car can be represented as a number of wheel objects, an engine object and a chassis object. An engine object may use a string object, usually from a third-party library, for its name representation and so on (see Figure 1).

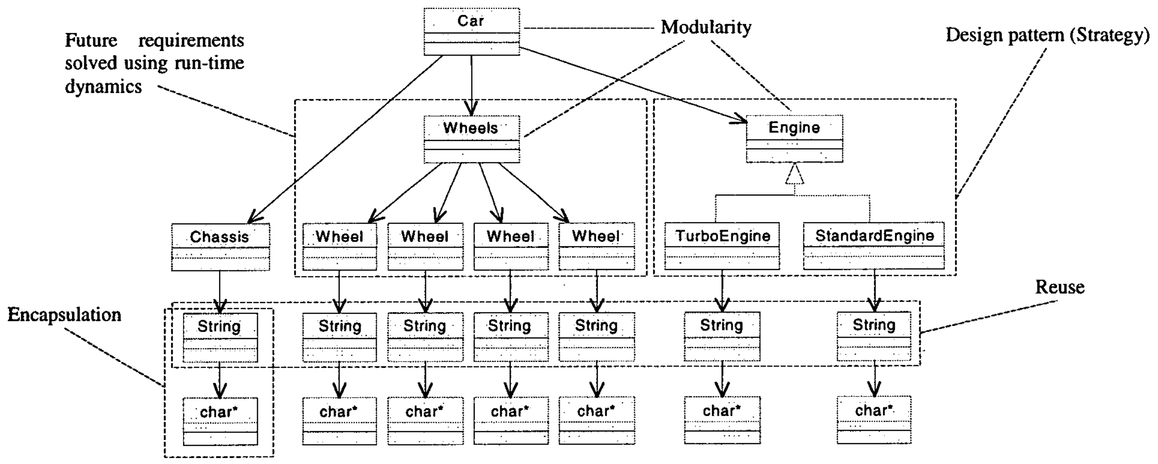

Figure 1. An object representation of a car

It is not rare that these objects are combined at runtime in order to be as adaptable as possible. For example, a car could have an unspecified number of wheels. Such a design requires each sub-object (Wheel) to be created separately. As a result of this, dynamic memory is allocated and deallocated separately for each sub-object. The total number of allocations is, according to the discussion above, dependent on the composition of the objects. Every design decision, which affects the composition of an object, will therefore also affect the number of memory allocations and deallocations during program execution. Large numbers of allocations and deallocations often become a big performance problem, especially on multiprocessor hardware. Actions must therefor often be taken, such as redesigning the application or using an alternative heap manager.

2.1. Structure pools

The concept of object pools, also known as memory pools, is a well-known memory management technique. An object pool acts as a layer between the application and the dynamic memory management subsystem. When allocating memory for an object, a call to an object pool is made instead of making direct calls to the memory manager, i.e. `malloc()`. An object pool holds a free list containing objects of a specific type. When an object is requested an object is extracted from the free list and returned to the caller. Thus, no call to the memory manager is needed, instead an already allocated, but not currently used, object is reused. Further, when deallocating memory a call to the pool is made instead of directly calling the memory manager, i.e. `free()`. The pool will then insert the object into its free list for later reuse. This kind of strategy reduces the number of calls made to the memory management subsystem. However, in a multithreaded environment there is still a need for mutual exclusion during operations on the pool's free list.

Using object structures, rather than single objects, as the reusable units in a free list leads to what we call *structure pools*. Each object that contains references to other objects is a potential root node in an object structure. When an object is deallocated it is placed in the free list, keeping its references to other objects. This means that if we would like to allocate a Car object (see Figure 1) we only need to access the Car's pool once to get the complete car with an engine, wheels and chassis. If we were using traditional object pools we would have had to access the pools of each and every class used by Car to build the same structure. By using pools to store object structures we not only reduce the number of calls made to the memory management subsystem, but we also reduce the number of calls made to pools.

The details on how these pools are implemented are discussed further in Section 3.

3. Amplify

When generalizing and automating the process of using structure pools we started by writing and analyzing handmade versions. During this process we found several points where the programmer makes decisions concerning the application source code or design. These decisions had to be made by the pre-processor when the process was automated. However, a pre-processor can never know as much about the source code as the programmer. Design decisions, rationale, unwritten coding rules, etc, are things a pre-processor can not comprehend. We therefore designed *Amplify* (which is the name of our method) to solve these problems in a general way, not depending on information unavailable to the pre-processor. By inspecting the input source code using a pattern matching approach we were able to modify the code to make use of structure pools.

3.1. Building structure pools by hand

In the case where a structure pool is handmade, the programmer selects the root object of the structure (i.e. Car in Figure 1). The programmer then creates a new class that handles a pool of such root objects. The new pool class typically has three static member functions, `init()`, `alloc()` and `free()` (see Figure 2). These functions manage a free list. Whenever the programmer wants to create or destroy a new structure he/she must not use operator `new` or `delete`, but instead make calls to the pool's `alloc()` and `free()` functions, which removes respectively inserts an object into the free list. It is not only up to the programmer to do the right thing when allocating and deallocating a structure, he/she is also responsible for writing the pool's `alloc()` and `free()` member functions. Further, before starting to make calls to `alloc()`, the pool has to be initialized by a call to `init()`. This is done in order to let the pool pre-allocate a number of structures that is inserted into the free list.

Since structures may not be identical every time they are used, the pool can only pre-allocate the objects that are common to every case (from now on referred to as *template*). In the case with the Car structure, a template would consist of a Car, a Chassis, and a Wheels object. Everything else is dependent on what kind of car is to be created. However, there are cases where the programmer knows that he will only create cars with a maximum of eight wheels. In these cases eight wheels are allocated and inserted into the template. Later, when a template is used, there are always eight wheels available. However, not all of them are used if the car only has four wheels. This memory overhead might be acceptable if it yields better performance. At this point the object structure has everything except an engine. The type of engine used can

```
class Car {
public:
   virtual void init(int numberOfWheels, ...); // Initialize
   virtual void destroy();                     // Clean up
...

class CarPool {
public:
   static void init(int numberOfCars); // Pre-allocate a number of structures
   static Car* alloc();                // Get a structure from the pool
   static void free(Car* car);         // Return a structure to the pool
...
```

Figure 2. Example of handmade pool

vary from time to time and must thus be allocated separately when the car is created.

Moreover, since the traditional memory management functions are bypassed when using these kinds of pools another addition is often made to the classes that compose the structures. It must be possible to initialize and destroy objects without calling operator `new` and operator `delete`. Thus, each class in a structure gets two new member functions, `init()` and `destroy()`. These functions act as replacements for the traditional constructor and destructor (see Figure 2).

Finally, the pools must be usable in a parallel environment. A solution to this would be to protect parts of the pools' code with mutual exclusion. However, this is often not a desirable solution since global locks are potential bottlenecks (and what causes `malloc()` and `free()` to be inefficient). Instead the programmer keeps track of which pools are used by which threads and manually avoids simultaneous allocations/deallocations from the same pool.

3.2. The Amplify method

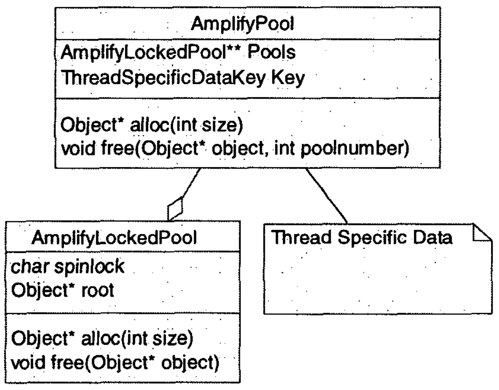

Figure 3. Structure of an Amplify pool

Automatically modifying the source code of an existing program requires the decisions made by the programmer to be generalized to a level comprehensible to the pre-processor. Therefor, we introduce a general structure pool (see Figure 3). The general structure pool must be flexible enough to fit every case and work completely transparent to the programmer. The following sections will describe the generalization.

When building a structure pool by hand, the programmer selects the root object of the structure. The programmer knows which structures are frequently used and can thus decide to only use structure pools in cases where he/she knows it will be beneficial. However, by just looking at the class structure in the source code we can not decide which objects are roots and which are not during execution. Since each object is a potential root node in a structure we can not during pre-processing treat some classes differently from others. Instead we treat every class as if it was a root in a structure, i.e. letting each class have its own pool.

The common parts of a structure were selected by the programmer and composed into a template. The characteristics of a template is based on design decisions and assumptions, e.g. "we will not create a car with more than eight wheels". It is obviously impossible for a pre-processor to make such assumptions and thus the contents of a template must be based on some other information.

One fundamental idea behind Amplify is to exploit temporal locality, which often characterizes object-oriented programs using frameworks and design patterns. In this case, it means that when the program creates a new structure we assume that it will be identical to the structure last deleted (which had the same type of root object). If we later find that the structures were not identical we will then take the overhead of reorganizing the structure to fit this specific case.

Before the programmer started to use a handmade pool he/she called `init()` to pre-allocate a number of structures. The programmer knows that he/she will never use more structures simultaneously than present in the pool after the initialization.

Instead of pre-allocating a number of structures, Amplify will start with empty pools. If a pool is empty when the application makes a memory request, the pool itself will simply allocate more memory using the normal dynamic memory manager, i.e. `malloc()`.

To allocate a structure using handmade pools, all traditional allocations (using operator new) are exchanged with a call to the member function `alloc()` of the corresponding pool. Further, to be able to initialize an already allocated structure the member function `init()` is called, instead of the constructor.

Amplify solves this by overloading operator `new` of each class that is associated with a pool. Operator `new` redirects all memory requests to the pool's member function `alloc()`. This function will extract a structure from the pool's free list and return it. Only if the free list is empty a new piece of memory is allocated on the heap.

If the pre-processed class already has a `new` operator defined, the pre-processor will respect that and not generate another one.

One problem with the standard `malloc()` and `free()` is the critical code regions that must be protected by mutual exclusion. A pool has the same fundamental problem. Only one thread can manipulate the pool's free list at the same time, else there would be a potential source for race conditions that could cause a program to fail. When developing handmade pools, the programmer often avoids this problem by simply not sharing a pool between two threads. A pool could of course be reused, but only when the thread currently using it has been terminated.

One goal when developing Amplify was to avoid expensive locking schemes. In particular, we would like to minimize the situations where a thread has to wait for a lock that is already occupied. This was done in several ways. First, having one pool for each class leads to a more parallel behavior since two threads will only compete with each other if they both try to allocate an object of the same type. Secondly, we used known strategies, mainly from ptmalloc, to spread the threads over a number of pools to avoid lock contention on a multiprocessor.

Freeing a structure when using handmade pools is done by calling the member function `destroy()` of the root object. This function acts as the destructor for the object and will in turn call the `destroy()` function of its children. When this is completed all objects in the structure have done their necessary clean up actions (releasing resources, closing files, etc) and the structure can be considered dead. The programmer now calls the `free()` function of the pool associated with the root. This will insert the structure into the pool's free list for later reuse. It is important to note that the relation between objects in the structure is kept intact, i.e. no pointers are deleted or changed.

The Amplify pre-processor is faced with two problems here. First, when deleting a structure the root object must be placed in the corresponding pool's free list. Second, to be able to reuse the structure later, the object structure must be preserved. Preserving the structure means that child objects of the root must not be deleted or placed in other pools.

These problems are solved by the following two actions. First, operator `delete` is overloaded on all classes associated with a pool. The `delete` operator will redirect all deletion requests to the pool of the class. This is done to avoid letting the memory be passed back to the heap manager. Instead `delete` adds the object to the corresponding pool's free list. Doing this, we allow the memory occupied by the object to be reused the next time such an object is to be created.

The second step is to preserve the object structure, e.g. avoid deletion of child objects of the root. Traditionally when an object structure is deleted, references (pointers) to other objects are deleted as well. Thus, we have to store information about the structure that outlives a deletion of its objects. This information must make it possible to rebuild the same object structure again (preferably with no or little effort). To solve this we introduce what we call *shadow pointers*. Shadow pointers are additional fields in a class, replicating all pointer fields in a class. These additional fields are added by the pre-processor, thus they are completely invisible to the programmer.

```
class Root {
public:
   // Methods
private:
   Child* left;
   Child* right;
   int data;
   Child* leftShadow;
   Child* rightShadow;
};
```

The shadow pointers are used in the following way. When a new `Root` object is allocated on the heap (i.e. the pool was empty) all shadows are set to 0 (null). In `Root`'s methods where the following code appears

```
   delete left;
```

the pre-processor will change it to

```
   if (left) {
      left->~Child();
      leftShadow = left;
   }
```

This is done to avoid deletion of `left`. Instead the object's destructor is called and then the address to the memory occupied by the object is saved in its shadow pointer. By doing this we can keep track of the object structure after it has been deleted. The next time the same structure is created we can reuse the previous one by just reading the shadow pointers. Thus, in `Root`'s methods where the following code appears

```
left = new Child(...);
```

the pre-processor will change it to

```
left = new(leftShadow) Child(...);
```

By also overloading operator new we can add type checking to ensure that there is enough space for the new object.

4. Method

As a first step, we developed three synthetic test programs to evaluate the potential of Amplify. All test programs used dynamic memory extensively and had a run-time behavior characterized by temporal locality, i.e. the program frequently created the same object structures over and over again. The three programs differed from each other by how deep object structures they where creating. The test programs were executed and compared using three alternative memory management solutions, ptmalloc [4], Hoard [1], and Amplify.

The test suite used was based on a program with 100% temporal locality behavior, i.e. creating the same structure over and over again. This was done by creating a number of threads, which allocates, initializes and then destroys and deallocates binary trees. Each node was 20 bytes (28 bytes when "amplified") in size, holding two pointers to its children and some "dummy" data.

The program was set up to vary its behavior with respect to the number of threads created, the number of binary trees created and destroyed, as well as the depth of the binary trees. No system calls were made during execution, thus making it theoretically possible for ideal scalability.

Three test cases (see Table 1) were chosen based on the following:

Test case	Tree depth	Number of objects
1	1	3
2	3	15
3	5	63

Table 1. Size of data structures in test cases.

- Test case one was chosen as a worst case, to see if the Amplify-approach would cause a large overhead when data structures are shallow. Since Amplify tries to reuse data structures it is clearly of interest to see how well it performs if there are no large structure to reuse.
- Test case two can be mapped to the Car-paradigm (see Figure 1) and thus represents something that could, based on the discussion in sections 1 and 2, be considered as a normal case for an application.
- Test case three could be thought of as the best case, where deep structures could be reused.

The tests were executed under Solaris 2.6 on a Sun Enterprise 4000 equipped with 8 processors. Solaris default memory manager implementation was used as the baseline for calculating speedup. In a second step, we applied Amplify on a real application, Billing Gateway (BGw). By doing this we identified a number of necessary adaptations (see Section 5.2).

The Billing Gateway (BGw) is a system for collecting billing information about calls from mobile phones. The system is a commercial product and has been developed by the Ericsson Telecommunication Company. BGw is written in C++ (approximately 100,000 lines of code) using object-oriented design, and the parallel execution has been implemented using Solaris threads. The system architecture is parallel and Ericsson therefore expected good speedup when using a multiprocessor. However, the actual scale-up of BGw was very disappointing.

In the first version of BGw, the sequential implementation of dynamic memory allocation and false memory sharing resulted in performance degradation when using more than one processor. The major reason was the standard sequential dynamic memory management combined with a large number of dynamic memory allocations. The large number of allocations is a result of how the object-oriented design was used. The number of heap allocations in BGw was reduced by introducing object-pools for commonly used object types, and by replacing heap variables with stack variables. The reduction of allocations made the application to scale linear.

However, after a few years of continuous development, BGw once again encountered scalability problems. The reason was the same as the last time. New functionality and program flows had made the object-pools inefficient. At this time, resources for updating the pools were lacking. The BGw is thus nowadays depending on a parallel allocator (Smart Heap for SMP) to scale on SMP:s.

The problem with dynamic memory is in the BGw mainly located in a certain component. This component, approximately 45,000 LOC, was extracted from the rest of the application source code. A test program design to behave identical to the original application was implemented. The test program was executed on a SUN Enterprise 10000 equipped with 8 processors. The time it took to process 5,000 CDR:s was measured.

5. Results

5.1. Synthetic program results

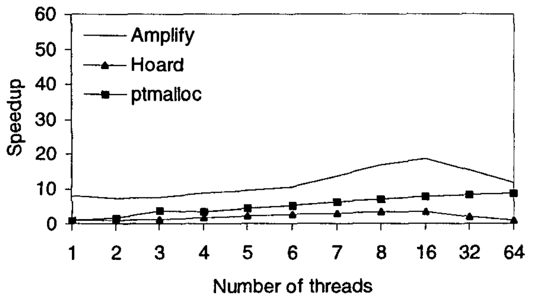

Figure 4. Speedup graph for test case 1

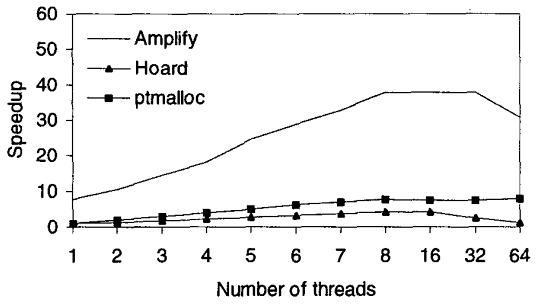

Figure 5. Speedup graph for test case 2

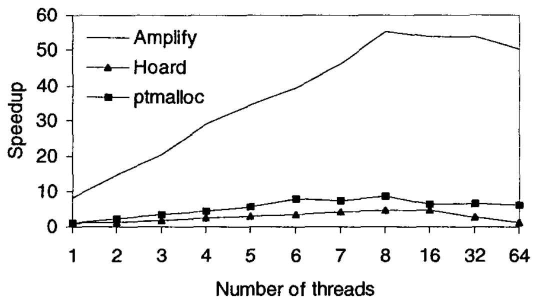

Figure 6. Speedup graph for test case 3

Figures 4, 5, and 6 show the speedup of each test case using three different methods: ptmalloc, Hoard and Amplify. As discussed above, the definition of a speedup value of 1 is the execution time using one thread and the standard Solaris heap manager. In all our tests Amplify outperforms both Hoard and ptmalloc, even when the data structure is shallow. The drop when using 2 threads compared to 1 thread in Figure 4 is caused by the fact that Amplify can operate more efficiently in a non-threaded environment. This since the pre-processor automatically removes all unnecessary locks.

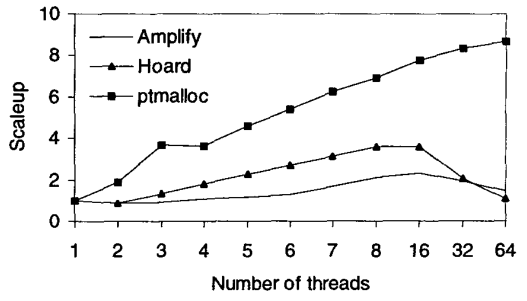

Figure 7. Scaleup graph for test case 1

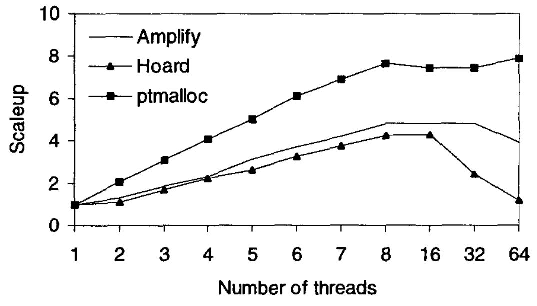

Figure 8. Scaleup graph for test case 2

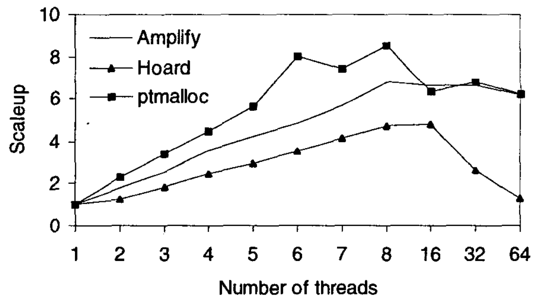

Figure 9. Scaleup graph for test case 3

Figures 7, 8 and 9 show the scaleup for our three test cases. The scaleup is defined as the speedup where the speedup value for one thread is normalized to one. Interesting to note is the fact that Amplify does not scale very well in test case 1 (see Figure 7). The main focus for Amplify is to avoid the contention caused by mutual exclusion during memory allocation. From this point of view the results from test case 1 are successful. When monitoring the locks protecting critical code regions within the pools we noticed a very low number of failed lock attempts. This result has lead us to believe that the ptmalloc approach (i.e. choose another heap/pool if a thread becomes blocked too often) no longer is a favorable solution. Using several heaps/pools has two advantages: it reduces lock contention and it improves cache hit ratio, provided that the threads that share a heap/pool run on the same processor. The time to lock, insert/remove an object into a free list, and then unlock is very short in Amplify (compared to ptmalloc). Thus, threads will seldom or never be blocked when trying to obtain a lock. Since the frequency of failed lock attempts is the axiom for choosing another pool this can lead to a situation where there are no failed locks, but undesirable cache effects, e.g., false memory sharing. Thus, our conclusion is that the poor scalability in test case 1 is not caused by a poor locking mechanism, but rather caused by other factors, such as false memory sharing. The test programs are constructed in such a way that the memory consumption is kept low also when using Amplify. However, the intention in this test was not to stress the memory consumption. There are two potential sources for memory consumption overhead in Amplify.

The first, and most obvious, case is when there are a lot of unused objects structures in the pools. This problem can be handled by returning memory from the pools to the operating system on demand, or when the pools exceed a certain limit.

The second case is when we reuse an unnecessary large data structure for a smaller structure. The Car in Figure 1 may for instance, in a later instantiation, chose not to create an Engine object at all. In this situation the memory, used for the Engine object in the former Car, will be unused. However, the memory can be reused in later Car instantiations. The designer may, chose not to "amplify" objects that can cause this type of overhead. Another alternative is to use the same strategy as the one described for the BGw version of Amplify (see below), i.e., and not reusing unnecessary large memory blocks. Consequently, if memory consumption overhead is critical, one can take precautions by limiting the number of root objects in the pools and/or by (statically) not applying the technique on root objects that can cause serious memory overhead and/or by (dynamically) not reusing unnecessary large memory blocks. The shadow pointers also introduce memory overhead. However, each shadow pointer could be replaced with one bit, which indicates if the original pointer is logically deleted or not. If the original pointer is logically deleted it has the role of the shadow pointer, and if it is not deleted the shadow pointer has no role. This strategy would, however, make the pre-processor somewhat more complex, and we have therefore not implemented it in this prototype.

Figure 10 shows a comparison between ptmalloc, Hoard, Amplify, and a handmade structure pool when executing test case 2. Note that Hoard does not scale when the number of threads is larger then the number of processors, which is a common case for server applications (as discussed before, we use 8 processors). A possible explanation for this can be that when threads start to migrate from one processor to another Hoard runs into lock contention. The publicly available

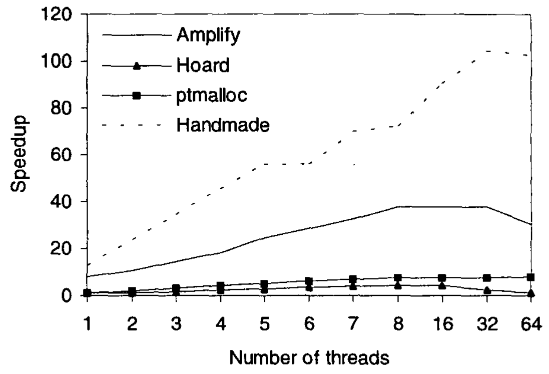

Figure 10. Speedup graph for test case 2 (including handmade structure pool)

implementation of Hoard uses a modulation based on thread id to assign threads to heaps. The handmade version could be seen as the theoretical maximum of what an optimizing pre-processor could do. As discussed before, the programmer knows things about the source code that are not available to the pre-processor. We believe, however, that it would be possible to improve the pre-processor even further by making a more sophisticated source code analysis, causing the speedup to come closer to the handmade version.

5.2. Result from tests on BGw

When Amplify was applied on BGw we encountered some (expected) problems. The first problem was that only half of the allocations in BGw are made from the application source code. The other half comes from tool libraries, such as Tools.h++. Since Amplify must be able to pre-process the code in order to insert the optimizations, Amplify's effect on BGw was limited. We can thus identify this problem to be a major drawback of our method and conclude that it will still be necessary to use some kind of parallel allocator, at least as long as we do not have access to the source code for the libraries.

The second problem encountered was that Amplify only handles objects. In BGw the large number of allocations were made for data type arrays, e.g. char[] and int[]. In order to deal with this problem we extended Amplify to also include data types. If the parent object is allocated from the object pool the following code

```
buffer = new char[length];
```

is replaced with

```
buffer = realloc(bufferShadow, length);
```

and

```
delete buffer;
```

with

```
bufferShadow = buffer;
```

where buffer is an attribute in the parent object.

Amplify works with the standard `realloc()`. However, by overloading the `realloc()` function it is possible to gain performance, and also to better control the memory consumption. We implemented a `realloc()` of our own in order to avoid unnecessary locks and it works in the following way:

If the allocated memory is smaller than the shadow memory but not smaller than half the shadow memory, then the shadow memory is reused. The function will guarantee that, if an allocation is made repeatedly, the maximum memory consumption is twice the normal.

We also introduced a maximum size for shadowed memory, i.e., if memory that should be shadowed is larger than the defined maximum, the memory is deleted (as normal). This function will prevent large single allocations to waste large memory chunks. We also introduced a maximum number of objects for each pool.

Consequently, for repeatedly created objects the memory consumption is limited to twice the amount of memory. However, there can still be memory overhead consumption if an object is created one or more times and after that not re-created. There are a number of solutions that reduce this problem, e.g. garbage collection. We have, however, not yet implemented any of these.

The results from the BGw show that the application is scaleable with SmartHeap. Amplify alone, i.e. without help from Smartheap, did not make BGw scalable. However, when Amplify was used in combination with SmartHeap, BGw was capable of processing CDR:s 17% faster (see Figure 11). The same result was measured if only data type arrays were shadowed or if all objects were shadowed, i.e., the shadowing of data types contributed with the major part of the allocations. The result shows that Amplify speeds up parts of the memory management where SmartHeap for SMP:s fails.

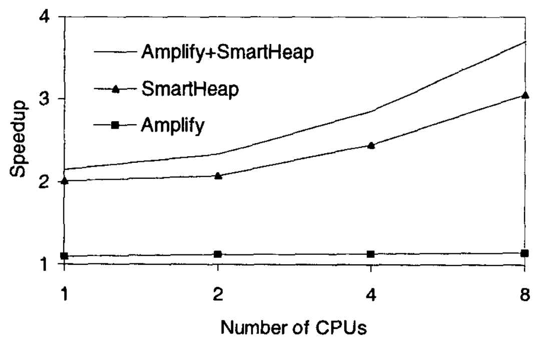

Figure 11. Speedup graph for BGw

6. Related work

The allocator designed by Doug Lea [6] is fast and efficient. However, the original version is neither thread-safe nor scalable on SMP:s. Wolfram Gloger created a thread-safe and scalable version called ptmalloc [4]. The allocator is based on a multiple number of sub-heaps. When a threads is about to make an allocation it "spins" over a number of heaps until it finds an unlocked heap. The thread will use this heap for the allocation and for allocations to come. If an allocation fails, the thread "spins" for a new heap. Since the operating system normally keeps the number of thread migrations low this implementation works fine for many applications. However, ptmalloc can when the number of threads is larger than the number of processors be unpredictable, in terms of efficiency. We have successfully used this heap implementation in our studies. For a more detailed description of ptmalloc see [11].

There is also an SMP version of SmartHeap available [7]. We have not had the opportunity to evaluate this implementation on our synthetic programs since the software is not freely accessible. During our test on BGw, however, we were given the opportunity to make some benchmark tests against Amplify.

Hoard [1] is another allocator optimized for SMPs, which we successfully used in our study. This allocator focuses on avoiding false memory sharing and blowup in memory consumption. The allocator is scalable. However, we have found that Hoard has problems when threads frequently migrate between processors. The Hoard allocator has been evaluated in a number of benchmarks. However, none of the benchmarks have the characteristics found in object-oriented software. Thus, to run these benchmarks on Amplify would be pointless.

LKmalloc [5], developed by Larsson and Krishnan, is yet another parallel memory allocator. (Not investigated by us). It focuses on being fast and scalable in both traditional applications and long-running server applications, which are executed on multiprocessor hardware.

Before ptmalloc, Hoard, LKmalloc etc., which were presented relatively recently, there is not much work available on parallel memory allocators. A survey on prior work in dynamic memory management can be found in Larson and Krishan's paper "Memory Allocation for Long-Running Server Applications " [5] There are also reports on hardware support for dynamic memory management [2]. The temporal locality characteristics exploited by Amplify are not specific to applications written in C++, but rather the behavior of object-oriented applications using frameworks and design patterns. Thus, using Amplify together with other object-oriented languages than C++ would most likely be beneficial. We are currently evaluating Amplify for Java.

7. Conclusion

The traditional solution to performance problems caused by heavy use of dynamic memory is to optimize the C library memory allocation routines. However, attacking these problems on source code level (i.e. modify the application code) makes it possible to achieve more efficient, application specific, memory management.

Optimizing the application by integrating customized structure pools into the source code has shown to be very beneficial in terms of performance. It is, however, a manual procedure and thus carries maintainability drawbacks, which makes it less attractive. By automating the procedure of adding these structure pools, most of these drawbacks are eliminated, making it possible to combine maintainability and performance. We have implemented a pre-processor based method, named Amplify, which in a completely automated procedure integrates structure pools into applications written in C++. By exploiting temporal locality, often characterizing applications developed using object-oriented techniques, Amplify can make very efficient optimizations resulting in low overhead. Amplify has shown to be up to six times more efficient in speeding up synthetic applications, compared to the available, state-of-the-art, C library allocators we have tested. It is also interesting to note that Amplify increases the performance of sequential as well as parallel programs.

Our tests on BGw verify the existents of temporal locality in a real commercial application. However, Amplify's possibility to make optimizations was reduced by the limited access to application source code. Nevertheless, a modified version of Amplify improved the performance of BGw with 17%. It was interesting to note that the relative performance increase due to Amplify was (more or less) the same with and without a parallel heap manager, i.e., the performance improvements of Amplify seem to be orthogonal to the performance improvements of parallel heap managers.

Neither the BGw nor the synthetic test programs suffered from the increased memory consumption. Amplify can, however, in some situations suffer from this problem. We have therefor discussed a number of techniques to limit the memory consumption overhead of Amplify.

Sine all source code may not be available, the Amplify method may have to be complemented with parallel heap managers to be useful in a realistic situation. However, the results show that Amplify has high potential and that it was useful on a commercial product.

8. References

[1] E. Berger K. McKinley, R. Blumofe, and P. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications", *in Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November 2000.

[2] J.M. Chang, W. Srisa-an, C.D. Lo, and E.F. Gehringer, "Hardware support for dynamic memory management", *in Proc. of the Workshop for Solving the Memory-Wall Problem, at the 27th International Symposium on Computer Architecture*, Vancouver, BC, June 2000.

[3] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large C and C++ programs", *Software – Practice and Experience*, pp. 527-542, June 1994.

[4] W. Gloger, "Dynamic memory allocator implementations in Linux", *http://www.dent.med. uni-muenchen.de/~wmglo/malloc-slides.html (site visited January 2001).*

[5] P. Larson and M. Krishan, "Memory Allocation for Long-Running Server Applications", *in Proc. of the International Symposium on Memory Management, ISMM '98, Vancouver,* British Columbia, Canada, October, 1998.

[6] D. Lea, "A memory allocator", *http://g.oswego.edu /dl/html/malloc.html (site visited January 2001).*

[7] MicroQuill Software Puplishing, Inc., "SmartHeap for SMP", *http://www.microquill.com/smp, (site visited January 2001).*

[8] MicroQuill Software Publishing, Inc., "*SmartHeap – Programmer's Guide*", March 1999.

[9] D. Häggander, PO. Bengtsson, J. Bosch, and L. Lundberg, "Maintainability Myth Causes Performance Problems in Parallel Applications", *in Proc. of the 3rd International Conf. on Software Engineering and Applications,* Scottsdale, USA, pp. 288-294, October 1999.

[10] D. Häggander and L. Lundberg, "Attacking the Dynamic Memory Problem for SMPs", *in Proc. of the 13th International Conf. on Parallel and Distributed Computing Systems,* Las Vegas, Nevada, USA, August 8-10 2000.

[11] D. Häggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", *in Proc. of the 27th International Conf. on Parallel Processing,* Minneapolis, USA, August 1998.

[12] B. Stroustrup, "*The C++ Programming Language*", Addison-Wesley, 1997.