Type-preserving Heap Profiler for C++

József Mihalicza
Eötvos Loránd University
Faculty of Informatics
Dept of Programming Languages and Compilers
jmihalicza@gmail.com

Zoltán Porkoláb
Eötvos Loránd University
Faculty of Informatics
Dept of Programming Languages and Compilers
gsd@elte.hu

Ábel Gábor
NNG Ltd.
abel@abel.hu

***Abstract*—Memory profilers are essential tools to understand the dynamic behaviour of complex modern programs. They help to reveal memory handling details: the wheres, the whens and the whats of memory allocations. Most heap profilers provide sufficient information about which part of the source code is responsible for the memory allocations by showing us the relevant call stacks. The sequence of allocations inform us about their order. However, in case of some strongly typed programming languages, like C++, the question what has been allocated is not trivial. Reporting the actual allocation size gives minimal or no information about the structure or type of the allocated objects. Though this information can be retrieved from the location and time of allocation, it cannot be easily automated, if at all. Therefore in large software systems programmers do not have an overall picture of which data structures are responsible for bottlenecks and have too few clues for pinpointing enhancement possibilities. In this paper we present a type-preserving heap profiler for C++. On top of the usual heap profiler features our allocation entries, including those of template constructs, contain exact type information about the allocated objects. We can extract information on individual memory operations as well as supply aggregated overview. Having such a type information in hand programmers can identify critical classes more easily and can perform optimizations based on evidence rather than speculations.**

I. Introduction

Most object-oriented programs heavily use heap operations. Objects with dynamic lifetime are created on the heap. There are also numerous design patterns [1] which require one or more of its components to be located on the heap. Heap operations are inherently slow compared to other in-memory activities. They usually go through several layers and manipulate a complex data structure. Sometimes an allocation has to perform garbage collection and/or even file operations (swapping). Thread-safety is another bottleneck regarding effective heap usage. Because of their high performance footprint, the optimal usage of heap operations is crucial when optimizing applications for speed.

According to the Pareto principle [2] in most cases only a relatively small amount of code is responsible for the majority of application running time. Therefore it is very important to know what parts of the code these are. Profilers can help us identifying them.

There are many good heap profilers for different platforms. They typically hook heap allocation routines (`malloc` and `free` on many platforms) and monitor what happens. At the end we can freely explore what call stacks belong to the allocations. One property, however, is lost in almost all C++ memory profiler tools we know: what type of object the allocation has created.

Type related information on heap usage can be very useful in modern object-oriented languages. As types, classes are the main language construct, this information can reveal hot-spots, critical bottlenecks on a higher level.

For some languages, where a reflection model is built into the language, retrieving heap related type information is easy. This is the case in Objective C[3] and Java[4] for example. The C++ programming language is infamous about its very poor introspection capabilities. C++ does not give much aid for preserving type information for heap operations.

Tools operating on binaries have minimal chance to recover such data in C++ since the very same code is generated for different kinds of objects with same size and they are not differentiable afterwards. Also, in C++ only minimal run-time information are kept about the individual objects. A solution, therefore, probably requires a separate build configuration where these information live longer.

Once we retrieved profiling related data we want to use them to better understand program behaviour. Contrary to the low level *infrastructure* types, like strings, data structures, thread synchronisation primitives, many classes can be directly connected to certain program features. Therefore type information can naturally group memory allocations to features or program components. On resource-limited platforms it is important to know the memory footprints of each individual feature when composing a new product. In some cases features can be easily measured using only call stack information. Large systems, however, often contain complex cooperative components where the objects have shared ownership. In those situations we cannot select the exact code area which is responsible for handling these objects.

In this paper we present a heap profiler framework for C++ that is capable of preserving the type of the allocated objects. Our framework is based on macros, operator overloading, and instrument functions. To apply it on a project, some constructs in the source code have to be modified, but fortunately this transformation is straightforward and seldom requires significant time. The data retrieved by using the profiler can be analysed in our interactive visualiser tool, where with user defined filter graphs we can focus on arbitrary aspects.

The rest of the paper is organized as follows. Section II

walks through the subtleties of the implementation of our profiler framework. In section III we get familiar with the usage patterns of the toolset. Section IV describes what kinds of analysis and visualisation is possible once we have the profiler output. Section V presents a case study, showing a few typical use cases while Section VI presents some similar works and focuses on the differences. Finally we conclude in section VII.

II. Implementation

The C++ language is well known of its limited (almost nothing) introspection features compared to Java or C# languages, where the run time system supports reflection. Therefore among the numerous C++ heap profilers we can hardly find any that provides sufficiently precise type information. Our target is implementing a type preserving heap profiler for C++.

First we enlist our main requirements against the desired tool. Then we systematically narrow the space of possible solutions, and at the same time present some details of our concrete implementation.

An ideal type preserving heap profiler has the following essential features:

- work on real, industrial sized C++ systems
- be as platform/compiler independent as possible
- have acceptable overhead both in the means of memory and speed
- provide a full overview of the allocations and deallocations with call stacks and types
- preserve exact type information, including template constructs like `std::vector<std::string>` or even more complex ones
- ease of use
- have minimal syntactic overhead
- no influence on performance if the profiler is not enabled (do not pay for what you did not ask)
- support multithreading

A. Capturing memory operations

Many C++ heap profilers work on binary code using techniques like dll injection or preloading. A tool that operates on compiled binaries would be very useful, but unfortunately the C++ binaries do not preserve type information (RTTI for example can only be applied to polymorphic types). Though we can identify calls to `malloc` and `free`, the original types are lost. Attempts to recover type information from memory layout is hopeless. Having two structures with identical member layouts, the generated code for object allocation of one or the other will not differ. The debug information, if exists, could help to identify the type, but that would require big efforts for each supported compiler (more precisely each supported debug information format).

Due to the issues listed above it is more promising to address the problem at the time of compilation when type information is still available. Our main targets are the typed `new` and `delete` expressions and the typeless `malloc` calls with its fellows: `free`, `realloc`, `calloc`.

In C++ the `new` *expression* is the main language feature for creating objects on the heap. The `new` *expression* is not equivalent to the `new` *operator*, it has a much wider role. The `new` *expression* is responsible for allocating space for objects on the heap by calling one of the overloaded `new` *operator*s. Then it calls the constructor to ensure proper initialisation of the raw memory area. It is also responsible for catching possible exceptions caused by the constructor and clean up the already allocated memory in such cases to avoid memory leaks. The `new` expression therefore has exact type information, on the contrary `new` *operator*s are typeless.

Though the *operator*s will not help us identifying the types, they are perfect hook points of memory operations. Let us start with redefining `operator new`. The signatures are the following:

```
void* operator new (size_t);
void* operator new[] (size_t);
void operator delete (void*);
void operator delete[] (void*);
```

For simplicity we omit the discussion of other overloads like the `nothrow` versions, they can be added similarly. We left out the placement versions as they have nothing to do with actual (de)allocation.

Apart from calling the actual allocation routines (the standard `malloc`, or any already existing custom implementation), we decorate [1] these functions with an extra administration step. With this added administration step we can keep track of the address and size of each allocation and the corresponding deallocation.

The recommended data structure for storing allocation entries is a hash table, using the address as key. In multithreaded environments concurrent access should be considered. The hash table contains allocation entries for all heap memory actually being used by the application. This is ideal for creating snapshots at arbitrary time. If we are interested also in the freed entries, they can be added to a circular vector at deallocation. This vector is dumped when it gets full, producing a series of dumps describing the past. It is important not to use the redefined allocation routines from within themselves. This can be accomplished by using custom allocators or static memory areas. The custom allocators should use the low level allocation primitives not garnished with the administration.

Caveat: Heap allocations can happen during static initialisation:

```
A* globalVarA = new A;
```

which means our administration should not rely on static initialisation order. An on demand initialisation technique is recommended.

B. Retrieving type information

Although we are able to monitor allocations and deallocations through the `new` and `delete` operators, the types are still missing. The expression `new T` has a static type of `T*`, but we cannot reach this type information inside the

new operator, which only gets a size parameter and returns the allocated area as a typeless void* pointer. Note that the allocation may pass arguments to one of the constructors, and also the type name can contain commas: new std::pair<int,int>(3,4). This is important if we are considering using macros, which is a usual technique to minimize syntactic overhead. To demonstrate the difficulties of seizing type information, consider the following attempts:

- New((std::pair<int,int>),(3,4))
  where New is a macro with two parameters
- New((std::pair<int,int>)(3,4))
  where New is a macro with a sequence [5] parameter
- New(new std::pair<int,int>(3,4))
  where New is a template function
- new std::pair<int,int>(3,4))
  where new is a macro, substituting to New(new, resulting in the previous line

The last one already has minimal syntactic overhead, though in a complex expression the extra parenthesis can be very confusing. We should eliminate that. Fortunately C++ has the possibility to call a function without writing any parenthesis with its infix operator syntax. Operators are ranked by well defined precedence categories and can be templates as well.

Our solution is new std::pair<int,int>(3,4), with zero syntactic overhead, where the new macro is defined as follows:

```
#define new NewTrick() * new
```

Here NewTrick is a helper type, with an overloaded * operator for each T:

```
template<class T>
struct TYPE_IDENTIFIER {
    static char mDummy;
};
struct NewTrick {
  template<class T>
  T* operator* (T* Ptr) {
    RegisterAllocation(
        Ptr, &TYPE_IDENTIFIER<T>::mDummy);
    return Ptr;
  }
};
```

The key feature of our construction is the mDummy static member of the TYPE_IDENTIFIER<T> helper template. In NewTrick() * new we call the default constructor of NewTrick to create a temporary object on which we can perform our overloaded operator*. This *template* operator* takes a pointer of arbitrary type as right hand side operand. In our case this operand will be provided by the new expression. Inside the operator the second parameter of the call to RegisterAllocation instantiates the TYPE_IDENTIFIER template class for each type which happens to appear on the right hand side of the new keyword.

For each such type there will be a unique global variable for type identification. It is important to understand that these variables are not holding any specific value for identifying the corresponding types like std::type_info does. It is the existence of the variables with their separate addresses which holds the information we are looking for. These addresses with the appropriate type names, however, are listed in the *map file* generated by the linker. This file contains the addresses of entities in the binary, including our mDummy variables. This functionality is available on all platforms, although the format of the file may vary. Having the *map file* at hand, we can associate type names to the type identifier addresses. Below are some fragments of a name demangled map file:

```
public: static char TYPE_IDENTIFIER<unsign
 ed short>::mDummy 010f29d4
public: static char TYPE_IDENTIFIER<unsign
 ed char>::mDummy 010f29d5
...
public: static char TYPE_IDENTIFIER<class
 std::set<int,struct std::less<int>,class
 std::allocator<int> > >::mDummy 01118dec
...
public: static char TYPE_IDENTIFIER<struct
  std::pair<unsigned long,bool> >::mDummy
```

This way we do not have to generate a string from a type, which would be a difficult task, if not impossible without syntactic overhead. This has also a positive impact on the memory overhead of our solution, as the growth of the executable size and static memory footprint is minimal. In the optimized version of a concrete mid-sized real application the map file had 3675 TYPE_IDENTIFIER entries. This equals to the number of types involved in new expressions. With our method the memory footprint is therefore 3675 bytes (1 byte per each mDummy instance). Having the mangled or demangled names stored in memory, however, it would be 191412 or 243222 bytes respectively.

We chose operator* because that was one of the suitable binary operators with strongest precedence. Thus we avoided unwanted side effects if someone mixed allocation and pointer arithmetics:

```
ObjWithRefCount* obj = (ObjWithRefCount*)(
 sizeof(RefCount) + new char
 [sizeof(RefCount) + sizeof(Obj)]);
```

If we had chosen operator= for example, then in this case the preprocessed code would have been:

```
ObjWithRefCount* obj = (ObjWithRefCount*)(
 sizeof(RefCount) + NewTrick() = new char
 [sizeof(RefCount) + sizeof(Obj)]);
```

and the compiler would have complaint about missing operator+ overload between size_t and NewTrick.

In certain cases a two phase approach is preferable, where the lowest level memory manager routines (called from a redefined malloc and/or operator new) call RegisterAllocation without type information and NewTrick::operator* calls a SetAllocationType

type modifier to the already registered entry afterwards. This implementation has the advantage that if the type information is not needed, the whole `NewTrick` trick can be disabled, still keeping track of the allocations via the typeless mechanism.

C. Pitfalls

We have just walked into a trap. In case of `new[]` the item count is sometimes stored at the beginning of the allocated area. More precisely it is when the type has a non-trivial destructor to be called for each allocated element at deallocation. The following C++ code:

```
struct A {
   ~A() {}
   int member;
} *fiveAs = new A[5];
```

is compiled as:

```
size_t* allocated = (size_t*)::operator
   new(sizeof(size_t) + 5 * sizeof(A));
*allocatedArea = 5; // store size
A* fiveAs = (A*)(allocated + 1);
```

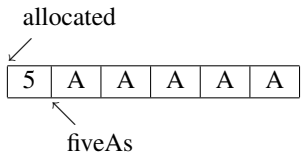

Fig. 1. Example of the typical memory layout of an array with non-trivial destructor

Having the `new` macro defined as described, we get this:

```
size_t* allocated = (size_t*)::operator
   new(sizeof(size_t) + 5 * sizeof(A));
*allocatedArea = 5; // store size
NewTrick Temporary;
A* fiveAs = NewTrick::operator*<A>(
    &Temporary, (A*)(allocated + 1));
```

That means in `NewTrick::operator*` a different address will be passed to `SetAllocationType` than has been to `RegisterAllocation` somewhere under `::operator new[]`. Let us consider our possibilities:

- Adjust the pointer in `::operator new[]` when calling `RegisterAllocation`. Without type information we have no chance to determine if the adjustment is really necessary. Arrays of built-in types, like `int`, will not be prefixed with size.
- Do the necessary pointer adjustment when calling `SetAllocationType`. Using the described syntax we are not aware of whether it was a single object or an array allocation.

The safe solution is either having `fiveAs` (see Fig. 1.) available in `::operator new[]`, or having `allocated` available in `NewTrick::operator*`. The first is impossible because `fiveAs` is calculated only after `::operator new[]` returned. With the current syntax we can only pass the value via global variables. That pops up multithreading issues, and it is also not guaranteed that there are no subsequent calls to `::operator new[]` before the corresponding `NewTrick::operator*` gets executed. We should probably use a stack on each thread. Do not forget that `NewTrick::operator*` cannot differentiate between `new` and `new[]`, therefore we should do this extra pointer administration also in the more frequent single object allocation case. Though this solution could probably work, we rather chose a simpler one and gave up the nice syntax. Instead of

```
A* fiveAs = new A[5];
```

one will have to write

```
A* fiveAs = New_<A>(5);
```

for array allocations. The underscore is only added to reduce the possible name clash with existing symbols. `New_` passes a pointer to a local stack variable (named `allocated` in the previous example), to `::operator new`. The operator is supposed to write the actual allocation address into this variable:

```
template<class T>
T* New_(int Count) {
    void* allocated;
    T* result = new (&allocated) T[Count];
    SetAllocationTypeAndCount(
        allocated,
        /* result, */ // optional
        &TYPE_IDENTIFIER<T>::mDummy,
        Count);
    return result;
}
void* operator new[](size_t size,
                     void** pAllocated) {
    return *pAllocated =
        RegisterAllocation(malloc(size));
}
```

Here `RegisterAllocation` returns its argument unchanged allowing the elimination of a temporary variable. Note that passing `allocated` as `void*&` would have conflict with placement new. Be careful to add this code to a place where the `new` macro has not yet been defined. We also pass `Count`, which eliminates the need of knowing `sizeof(T)` when the allocations are analysed. In fact, having `Count` at hand, we will use it for the opposite purpose, to calculate `sizeof(T)` based on the allocation sizes.

Fortunately array allocations are less frequent, hence the introduced syntactic overhead hurts less. Regardless of how frequently the new form of array allocation is advertised among the coders, likely they will sometimes forget not to use the standard form. In those cases `SetAllocationType` will get an address that is not present among the allocation entries. It is a silent killer, because it either produces a run time error if the execution once wanders there, or just

simply distorts the measure. We should somehow protect the framework against misuse. Using the standard form of `new[]` should result in a broken build. Though we cannot delete the always existing `void* operator new[](size_t)` overload, we already have a define for `new`. Let us add an extra parameter to it:

```
struct UseNew_ { static UseNew_ st; };
#define new NewTrick() * new (UseNew_::st)
```

Of course this will affect the single object allocations, we should modify the declaration of `operator new` accordingly:

```
void* operator new(size_t size,
  const UseNew_&) { ... }
```

For the `new[]` case let us declare an overload that does not match:

```
void* operator new(size_t size,
  const UseNew_*);
```

With some compilers we will get compilation errors wherever someone tried to use the standard `operator new[]` syntax. With other compilers, however, we will not get any errors, they just silently fall back on using the standard one. On these compilers it is better using the proper overload, *undefined*:

```
void* operator new(size_t size,
  const UseNew_*);
```

This will result in an unresolved external at link time. The exact place of misuse is not revealed, but at least the build gets broken. In most cases a clever regular expression suffices to find the problematic line.

Array `typedef`s can make our lives harder too. Consider the following example:

```
typedef A HideArrayA[10];
A* Array = new HideArrayA;
```

Despite the allocation seems correct, it is actually a call to `operator new[]`. In these cases we have to substitute the typedef, which unfortunately hurts the DRY (Do not Repeat Yourself) principle [6].

We have discussed the two most common usages of `new`, the single object and the array forms. There, however, can be arbitrary additional overloads with their special custom behaviour. An important example of that is placement new, which is the official way of explicitly calling the constructor on a previously allocated memory area:

```
char Mem[sizeof(std::vector<int>)];
new (Mem) std::vector<int>();
((std::vector<int>*)Mem)->push_back(42);
```

With the first version of our `new` macro, this code would compile, but the construct would be treated as an allocation and would silently result in corrupt allocation administration. The current version, however, produces the following code:

```
new NewTrick() * new (UseNew_::st) (Mem)
std::vector<int>();
```

which is ill-formed [7]. In fact in these cases we should not use our macro. The proper solution is turning it off for these special lines:

```
#pragma push_macro("new")
#undef new
new (Mem) std::vector<int>();
#pragma pop_macro("new")
```

The undefinition of `new` must be as limited as possible. Even undefining it only for that single line can lead to administration errors if the parameter expressions of the constructor call contain another `new`. The recommended style breaks these calls into two lines:

```
#pragma push_macro("new")
#undef new
new (Mem2)
#pragma pop_macro("new")
    std::auto_ptr<int>(new int(42));
```

`push_macro` and `pop_macro` are not part of the C++ standard [7], but most compilers support them. The simplest standard compliant solution would simply undefine the macro and redefine it afterwards. That is, however, against the DRY principle as the definition of `new` would be repeated at each location. We recommend putting its definition and the corresponding undefinition to dedicated headers:

```
#include "undef_new.h"
new (Mem2)
#include "define_new.h"
    std::auto_ptr<int>(new int(5));
```

It is important for these files not to contain header guard as they can be used multiple times in the same compilation unit.

Having all these implemented, we started using our heap profiler and the type information seemed correct. Sometimes, however, the occupied memory was not matching our knowledge about the software. It turned out that allocations of STL containers [8] were all typeless, because they internally used `malloc`. To solve the problem either `std::allocator` should be modified, or a custom allocator should be used. The first is simpler if a custom STL implementation is already part of the code base, but sometimes not possible. The custom allocator probably produces another syntactic overhead when using STL containers. STL is only an example, in fact each generic data structure probably uses `malloc` for allocation.

Similarly, even at places outside of templates, we can encounter usage of raw allocation routines, typically together with placement new. Such behaviour is mainly driven by efficiency claims. In these situations our framework will not be able to produce relevant type information. We can get around this problem by adding explicit calls to `SetAllocationType`/`SetAllocationTypeCount` manually directly after the raw memory allocation. Moreover, even defining new pseudo types only for type identification

makes sense. That way symbolic names can be used at places where buffers of built-in types are allocated:

```
struct MEM_TYPE_IMGBUF {};
int* ImgBuf =
 New_<int,MEM_TYPE_IMGBUF>(w*h);
// or
int* ImgBuf = LogMemoryType
 ((int*)malloc(w*h*sizeof(int)));
```

where `LogMemoryType` is a template function:

```
template<class T>
T* LogMemoryType(T* Ptr) {
 SetAllocType(Ptr,
  &TYPE_IDENTIFIER<T>::mDummy);
 return Ptr;
}
```

D. Call stack

With the method we described in the previous section we are able to keep track of all the heap operations of the software and maintain a database of the actually used memory areas. The entries by now can contain type, size, count and time stamp. In this subsection we describe how we can collect information about the call stacks. Unfortunately there is no standard way in the C++ language to obtain the actual call stack. Thus we have only the following suboptimal solutions:

- Use a platform dependent system routine. Quite often such a function simply does not exist. If we have luck and it does, usually its slowness makes it unusable for our purposes.
- Write an own version of such a function. This is not impossible, but it is very sensible for special optimisation techniques, and requires deep knowledge of the actual architecture. This solution is definitely against our platform independent intentions as it introduces a huge number of platform dependent techniques.
- Continuously maintain a vector of code addresses representing the call stack utilizing instrumentation functions.

The last option can be implemented so as to fulfil the necessary speed requirements while keeping the compiler dependent parts on minimum.

Many compilers provide options for generating instrumented code for profiling purposes. If the given compilation option is set then dedicated *enter* and *exit* functions are called at the beginning and end of each function respectively. Table I shows the signatures of these special functions in different compilers [9]. We should avoid calling any instrumented function from within these hooks to avoid recursion. Some compilers provide syntactic elements for disabling the instrumentation of particular functions.

In case of Visual C++ we can choose from two instrumentation methods: to place the special calls inside the called functions (`callcap`), or around the call instruction at call site (`fastcap`).

We can maintain the call stack of each thread by adding or removing an entry to/from a thread local array in the `enter` and `exit` functions respectively. In normal operation simple push and pop procedures are performed, but there are some rare scenarios when special handling is needed. In case of `longjmp` or exceptions the caller address passed to the `exit` function may differ from the one passed to `enter`. In case of exceptions we will find the address in lower positions of our array and should pop the unwound elements.

Utilizing the instrumentation technique we can have an always up to date call stack information for each thread. The next question is how these call stacks are associated to the allocation entries. Copying the whole array into the entry would consume too much memory. Once entered into a function, all allocations from within that call will have the same call stack prefix. We can optimize our data structures to exploit this property. It is a good trade-off between speed and storage if we store the unified call graph (call trees of each individual thread) instead of independent arrays. Each node of the graph represents a function that is called on a specific path starting from the thread's main entry. The relation is not injective, there can be multiple graph entries referring to the same function, provided that the function has been called on different paths. The path to the thread root clearly identifies the call stack, thus it is enough for an allocation entry to refer to the corresponding graph node of the given call stack. For simplicity we use an ever-growing data structure, the unused nodes are not removed from the graph. With a reference count technique this can be easily added though. When a new allocation entry is registered, the functions of the actual call stack array are looked up in the call graph and the missing nodes are created if needed.

A reference to the node representing the last function of the call stack is stored in the newly registered allocation entry. The operation that finds the graph nodes matching the call stack is invoked at each heap allocation, therefore it has to be fast. Once a graph node is found for a call stack element, it is worth memorizing it in the given call stack element. That data is valid until the entry is removed from the stack. This way the lookup can remain relatively fast even when the call paths tend to be long.

The described structure is called calling context tree [10]. A leftmost-child-right-sibling representation can keep memory footprint low, while remaining fast.

Our framework has been implemented along the above lines, and has been successfully compiled and is actually being used on the following compiler/platform pairs:

- Visual C++ 8.0, 9.0 / Windows XP, 7
- Visual C++ 8.0 / Windows CE
- g++ 4.2 / iPhone OS[1]
- g++ / QNX

[1]Though the applications for iOS are usually written in Objective C, in our case the bulk of the code is in C++, only a thin platform layer deals with the native API.

Compiler	enter function exit function how to disable

g++
Intel C++ on IA-32 and Intel®64
PathScale
PGI
	`void __cyg_profile_func_enter(void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit (void *this_fn, void *call_site);`
	`__attribute__((__no_instrument_function__))`

Intel C++ on IA-64	`void __cyg_profile_func_enter(void **this_fn, void *call_site);`
	`void __cyg_profile_func_exit (void **this_fn, void *call_site);`
	`__attribute__((__no_instrument_function__))`

Visual C++	`void _CAP_Start_Profiling(void *call_site, void* this_fn);`
	`void _CAP_End_Profiling  (void *call_site);`
	`N/A`

TABLE I
SIGNATURES OF *enter* AND *exit* FUNCTIONS IN DIFFERENT COMPILERS

III. USING OUR FRAMEWORK

In the previous section we discussed our method for collecting stack trace and detailed type information on heap operations. We also got familiar with the data structures our framework uses. Now let us see how we can extract the gathered data from the running application.

The profiling process consists of the following steps:

1) Profiler integration
2) Profiler configuration
3) Compile the profiling enabled application
4) Run the application, test different scenarios
5) Convert resulting profiler output dump files
6) Analyse/visualize results

We have discussed the tasks of step 1 in section II.

In most cases we do not want to measure the whole run of the application, instead we may want to restrict the scope of profiling into a time interval. To be able to define this range either at compilation time, or initiated by user actions at run time, we have to add necessary support code. There are three different approaches:

- Profile the whole run.
- Start and end profiling programmatically with explicit calls to the framework from client code. The free operations should handle the cases when the allocation is not yet present in the administration.
- Start and end profiling at run time. For this we have to add the trigger functionality to the client code. It can be a keyboard shortcut, a special mouse gesture, the presence of a file with a specific name (non-gui applications), a special http request (network server application) etc.

Every feature of the profiler has its own cost. We pay for them in seconds and/or kilobytes. It is rare that for our specific measurement we need all the data the profiler can collect. We recommended implementing them as orthogonal as possible to let the user freely select the desired components. Sometimes different measurements require different categorisations of object types. In a software environment that uses garbage collection techniques one may want to perform garbage collection right before creating the dump not to be misled by already (logically, not physically) freed elements in the dump. On the contrary, one may specifically want to dump a state right at the point when garbage collection is to be performed to see what memory state has triggered it. These alternatives are usually controlled by preprocessor directives and/or configuration files which are read typically at program startup. In the configuration step these defines and configuration files are set up according to the needs of the given measurements. Do not forget to enable map file generation for the build process, we will need it later for the types and call stacks.

Once we have our application armed with the configured profiler code we can perform the test scenarios. They vary from simple ad hoc operations driven by curiosity to well defined test cases of scalability, a concrete bug of out of memory condition etc. Sometimes dumps of separate runs are brought under comparative analysis. Depending on the components chosen the following types of dumps are generated:

- Regular dumps of freed elements. Whenever the framework exceeds its limit of storing freed allocation entries, it flushes them to a file.
- Full dump of the allocation entries, the freed elements and the call graph at exit. Non-freed elements of this dump correspond to leaked memory areas.
- On demand dump of the call graph, the allocation entries and optionally the freed elements.

The naming of profiler output files follow a numbering convention to handle multiple dumps properly. In our implementation all these files are plain text files. A fragment of a freed dump:

```
HeapLog|version=4|address|size|count|
 type_id|tick|freed_tick|callgraph_node
22A19EC8,32,8,0073D411,132,132,022B7864
0212DF10,30,30,0073D3AD,132,132,022C68C4
298D0B58,30,30,0073D42E,132,132,022C68DC
22A19E30,124,31,0073D411,132,132,022C68F4
...
```

Once we went through our test cases we have dozens of these dump files. Their format is suitable for being dumped from the application easily, but they are not appropriate for direct analysis. We have to apply our converter tool to generate one or more .muar file(s) from these dumps. The extension

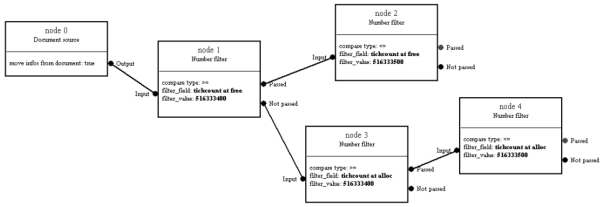

Fig. 2. A complete filter graph, each box has its inputs on the left, and its outputs on the right. Straight lines between input and output pins denote connections.

stands for *memory usage analyser result*, it is the base format of our visualiser application, see section IV.

allocation entry dump
freed entry dumps
call graph
$\xrightarrow{\text{convert}}$ `.muar` file

IV. Visualisation

Profiling industrial sized applications can produce enormous amounts of data. Understanding this raw information is often very time consuming and can leave crucial connections unrevealed. In the last decade visualisation tools proved to be key tools for program comprehension. These tools can provide an extremely good overall picture about the internal processes of applications. Quite often, however, we want to examine only a special subset of the data. Such subset can be the memory usage associated to a certain data structure, type or thread for example. Users would prefer to define their own specific views of profiling information. To support this request we provide elementary processing steps (*filter*s) on the allocation entries and a way to combine these elements into complex queries as *filter graph*s.

Our visualiser tool (memory usage analyser) works with an ordered sequence of coloured allocation entries. These entries can have the following properties: address, size, count, type (address of `TYPE_IDENTIFIER<`*type*`>::mDummy`), call stack (function addresses), time at allocation, time at free, freed flag, thread id, colour.

Some of the fields may be missing from our input data depending on the configuration. For the type and call stack to appear properly we always need the corresponding map file.

To construct the filter graph we provide a visual language, where boxes represent filters, and edges represent data flow. The user can define complex queries using graphical interface via drag and drop technique. These graphs represent different measurement patterns and can be stored and reused later. Figure 2 shows how a filter graph appears in our tool.

Boxes in these filter graphs can execute input or output actions, filter on the values of given fields (including time stamps, call stack, type etc.). Usual functions like averaging or summarizing field values can also be performed with boxes. The *colouring* box changes the colour property of the passing entries to customize visual appearance.

With this general approach we can build the following queries, for example: (1) keep allocation entries initiated from some `std::vector` instance, (2) colour entries of which type begins with `std::` to blue, (3) keep the allocations which lived (no) longer than a given time, (4) keep allocations only of a given thread.

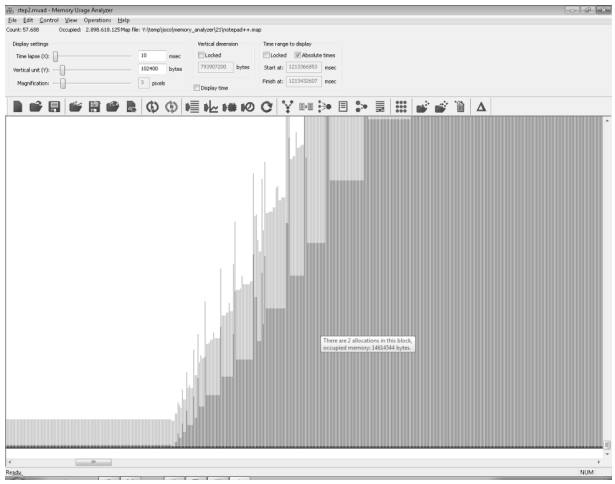

Fig. 3. Timeline view

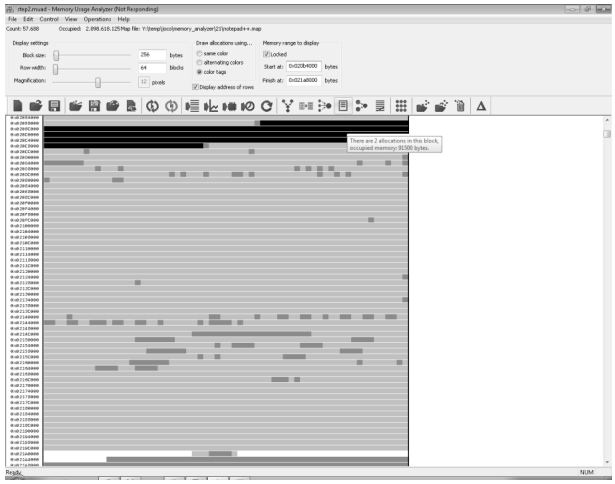

Fig. 4. Memory view in Memory Usage Analyser

Allocation entry sequences (possibly already processed with filter graphs) can be visualized graphically in different views.

The *timeline* view can give a quick overview on how total memory usage changes over time, see figure 3. If the entries have different colours, we see them stacked. By hovering over the chart the details of the actual time slot are presented.

In memory view we can explore a grid representation of the physical memory. By choosing alternating colours we can easily differentiate separate entries, otherwise the colour tags of the entries determine pixel colours. We can define how many bytes one pixel represents. Hovering over the pixels the details of the actual allocation entries appear.

Text view is useful for going through the allocations one by one, typically after a filter procedure. Some additional operations we can perform on our data set:

- join entries of same type/call stack
- unify the leafs of the call graph until the sum of allocation size reaches a specified limit
- sort entries by one of their fields
- calculate the differences and common parts of two sets

Some of the frequently used operations:

- keep entries allocated or deallocated in a given time range (sometimes as the difference of two separate memory snapshots)
- join by type, then sort descending by occupied memory
- colour entries of a given type, or instances of a given template, then consult timeline view
- switch to memory view with alternating colours to check fragmentation and pool behaviour

V. Usage examples

One could say type information is not that necessary, we have been working with our good old profilers for decades. Let us show an example where type had an important role. On PDAs and mobile phones there are much stricter memory limitations than on PCs. However, even these platforms have complex applications, like a real time navigation software for example. In our case the software had some special interactive visual elements that tracked mouse movements (actually touch movements) for processing the sequence at the end of or during the motion. These controls used `std::vector` to store touch coordinates, then called `clear` on them when the data was no longer needed. At first glance this seems correct, but the `clear` operation of `std::vector` did not free its allocated buffer, it remained reserved. After using the software for a while these buffers could accumulate hundreds of mouse events for each such interactive component. When grouped the allocations by type, this event type was among the top 10 most memory consuming types of the whole application. It was responsible for ∼400kB of the 12MB total memory consumption (∼3.3%), which was too much for being an unused reserve area. If we had not had type information about the allocations, the memory wasted by these events could have been easily treated as necessary costs of the visual component.

For demonstration purposes we have selected an open source C++ application, the Notepad++ free text editor and profiled it. This editor integrates Scintilla, a complex text editor component.

After integrating our framework to the Scintilla library first, then to Notepad++, we could get dumps at the end of each run. We opened some smaller and bigger files in Noteped++, and loaded the generated dumps in the visualiser. The timeline view showed that the memory was allocated in bigger blocks as the file was being loaded. To determine what types are involved in the loading process and in storing the contents of the file, we applied the following operations: join by type, then sort descending by occupied memory. The result immediately showed that `SplitVector<char>`, `char` and `SplitVector<int>` are the major ones. To see the dynamic behaviour, we assigned different colours to these types with a filter graph. Figure 5 shows the resulting timeline

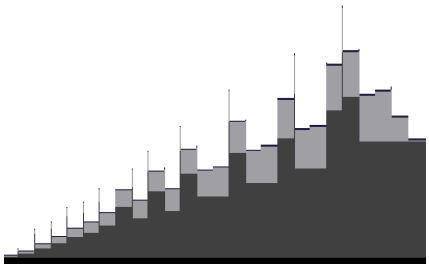

Fig. 5. Timeline view of loading a text file in Notepad++. One pixel represents 1 msec time horizontally and 320kB memory consumption vertically.

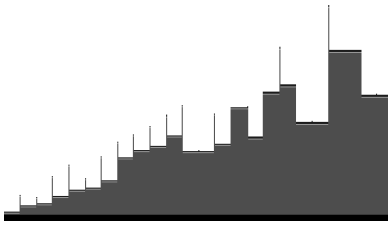

Fig. 6. Timeline view of loading a text file in Notepad++ after optimisation.

view. The light gray area represents allocations of type `char`. It is strange that they are present during the loading process, but then they are deallocated. We restricted the view to a small time slice and from the call stack information we found which part of the code was responsible for these allocations. It turned out that this was the undo history. During loading each added buffer creates an insert operation, that copies the string to the undo queue. This is done internally in Scintilla. After loading the file, the Notepad++ component clears the undo history. This is why the light gray area disappears. It is clear that maintaining the undo history when the insertions come from file loading wastes speed and memory. As a quick check we introduced a global flag that is on during file loading and turned off undo handling for this case. The resulting timeline view (Figure 6) shows that we could eliminate the unnecessary allocations. Without knowing the internal details of the code, we could easily pinpoint a performance issue and could reduce peak memory usage by 15%. This peak usage determines how big files we can open in Notepad++. With this little modification we raised that limit by 18%.

VI. Related work

The idea of adding extra arguments to `new` by defining it as a macro appears in [11]. There `__FILE__` and `__LINE__` are passed to the custom overload, allowing for associating source code locations to allocations. With this technique a useful memory leak finder tool can be easily implemented. Note that the file and line information does not necessarily identifies an allocation precisely:

```
vecA.AddA(new A(new B(a,b), new B(c,d)));
```

There are many heap profilers on the market, providing a wide variety of features from bound checking, memory leak hunting, call graph analysis etc. Most of them are designed to

be used on desktop computers, for middle sized applications. When applied one of them to a strategic game with around 1GB memory usage, the profiled version failed to launch. There were many heap allocations in the software and their administration took more memory than the heap the program used. Exceeding the 2GB heap allocation limit on the given platform heap analysis was not possible. Though this limit could be raised to 3GB, one of the 3rd party components did not handle pointers with highest bit set correctly. We had to give up using that tool for profiling.

For languages, where a reflection model is built into the language, retrieving heap related type information is easier. Sometimes such profilers are part of the platform SDK [12], [13], [14], [3]. [4]). We have also a wide range of third-party solutions, like [15], [16], [17], [18].

The *Memory Validator* tool [19] of Software Verification Ltd. provides some type information corresponding to C++ memory allocations. However, it prints the type information looked up from source code, rather than utilizing the type system of the language itself. This approach requires debug information during runtime execution and may fail in case of complex preprocessor macro definitions or intense code optimisations.

Valgrind is an instrumentation framework [20] for building dynamic analysis tools. It has a heap profiler module called *Massif* which does not provide type information on allocations. As the Valgrind framework is widely used an alternative approach could be based on modifying Massif. However, Valgrind officially supports only the Linux and Mac OS X platforms.

GPTL [9] utilizes instrumentation functions to perform performance analysis on different platforms. In [21] different visualisation methods of dynamic memory allocations are presented, including a similar one to our timeline view. In [14] stacked timeline views differentiating allocations by type are presented as an everyday profiling technique. *Heapviz* [4] visualises heap usage of Java programs as a graph. The idea of defining queries is envisioned in the future work section of that paper. The *Leaks* component in the *Instruments* profiler tool of the iPhone SDK[3] provides type preserving heap profiling for Objective C.

VII. Conclusion

Memory profilers are key tools to understand modern object-oriented programs. Although languages like C++ are strongly rely on types and classes, most memory profilers fail to provide sufficient information on actual types of memory allocations. We implemented a type preserving heap profiler for C++. The main contribution of our framework is the ability of providing detailed type information of each heap related operation additionally to usual profiling features. To support program comprehension we defined a visual language (filter graph) to allow the user to construct arbitrary queries in a fairly convenient way. Various visualisation possibilities are available to examine profiling result. Our solution is highly platform independent and has moderate memory and speed footprint. As a case study we used our framework to find and fix a performance issue in a code base what was new and unknown to us.

Acknowledgment

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003). Thanks for Gergely Herendi for designing and working out the details of the generic multithreaded call graph monitoring engine.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[2] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Optimization (Engineering Design and Automation)*. Wiley-Interscience, Dec. 1999.

[3] I. Piper, *Learn Xcode Tools for Mac OS X and iPhone Development*, 1st ed. Berkely, CA, USA: Apress, 2010.

[4] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, "Heapviz: interactive heap visualization for program understanding and debugging," in *Proceedings of the 5th international symposium on Software visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 53–62.

[5] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, December 2004.

[6] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] International Standards Organization, "Information technology – Programming languages – C++," ISO/IEC 14882:2003, 2003.

[8] N. M. Josuttis, *The C++ standard library: a tutorial and reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] J. Rosinski. (2008) Gptl timing library. [Online]. Available: http://www.burningserver.net/rosinski/gptl/

[10] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, pp. 85–96, May 1997.

[11] P. DiLascia, "Performance monitoring, managed extensions, and lock toolbars," *MSDN Magazine*, September 2004. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163931.aspx

[12] S. Ramaswamy and V. Morrison, "Profiling the .net garbage-collected heap," *MSDN Magazine*, October 2009. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/ee309515.aspx

[13] K. O'Hair, "Hprof: A heap/cpu profiling tool in j2se 5.0," *Sun Developer Network*, vol. Developer, 2004. [Online]. Available: http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[14] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[15] G. Barnett, "Review: Ants profiler 4," *dotnetslackers*, November 2008. [Online]. Available: http://dotnetslackers.com/articles/net/Review-Ants-Profiler-4.aspx

[16] "Jprofiler from ej-technologies gmbh." [Online]. Available: http://www.ejtechnologies.com/products/jprofiler/overview.html

[17] "Yourkit profiler from yourkit, llc." [Online]. Available: http://www.yourkit.com

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of java profilers," *SIGPLAN Not.*, vol. 45, pp. 187–197, June 2010.

[19] "Memory validator from software verification limited." [Online]. Available: http://www.softwareverify.com

[20] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100.

[21] S. Moreta and A. Telea, "Visualizing dynamic memory allocations," in *VISSOFT*, 2007, pp. 31–38.