COMPREHENSIVE PERFORMANCE ANALYSIS OF C++ SMART POINTERS

[1]Bence BABATI, [2]Norbert PATAKI

Department of Programming Languages and Compilers, Faculty of Informatics
Eötvös Loránd University, Pázmány Péter st. 1/C, H-1117 Budapest, Hungary
e-mail: [1]babati@caesar.elte.hu, [2]patakino@elte.hu

Abstract: Smart pointers play an important role in bypassing memory leaks in C++. Since C++11 standard the smart pointers have become widely-used tools because they let the programmers not to deal with memory deallocation. However, abstraction penalty occurs because of this convenience. Overhead is related to runtime, memory usage and compilation time. There are many different smart pointers in the standard library. However, the performance difference between the smart pointers and raw pointers is not measured before. This paper presents an analysis of their effectiveness. An alternative approach to the C++17's optional construct is searched for.

Keywords: C++, Smart pointers, Performance

1. Introduction

The programmers are supposed to deallocate of dynamically allocated memory in C and C++ programming languages. Compilers typically do not support the programmers' work with error and warning diagnostics, so it is easy to write code with memory leaks [1]. Still many static analysis techniques have been invented for detecting memory leaks [2]. Smart pointers play an important role in avoidance of memory leaks in C++ because they enable the programmers to unfocus on memory deallocation. However, abstraction penalty may occur because of the convenience [3], thus the usage of smart pointers is not priceless. Overhead appears when smart pointers are in-use. Abstraction penalty is related to runtime, memory usage and compilation time as well. There are many different smart pointers (std::unique_ptr, std::shared_ptr, etc.) in the standard library

that work in different ways [4]. C++ is a performance-oriented programming language, so high-level constructs should be as fast as possible [5]. However, the performance difference between the smart pointers is not measured before. Safety of pointer-like iterator objects are discussed [6].

This paper presents a comprehensive analysis of effectiveness of smart pointers and their overhead related to the raw pointers. An alternative approach for the C++17's optional construct is searched based on the available smart pointer constructs. Performance differences of smart pointers are analyzed based on test programs.

This paper is organized as follows: in the following section the background of smart pointers is presented. In the same section the C++17's optional construct is described, as well. Performance measurements and analysis of smart pointers are presented in section 3. Finally, this paper concludes in section 4.

2. Smart pointers

The standard smart pointers are able to deallocate memory when the smart pointer objects go out of scope. Smart pointers take advantage of the C++ template construct, so they are independent of the type of the managed memory. C++ template construction is very important feature from the view of performance [7]. Effectiveness of C++ template constructs is still evaluated [8]. The basic operations of smart pointers are those of the raw pointers but smart pointers offer some convenience methods. Different standard smart pointer types are available. However, dealing with memory usage optimization in concurrent execution is still problematic [9].

The smart pointers are based on the RAII (resource acquisition is initialization) principle: constructors and destructors are automatically executed in a well-defined moment. Invocation of these operations is based on the smart pointer objects' lifetime. The standard smart pointers are: std::auto_ptr<T>, std::unique_ptr<T>, std::shared_ptr<T> and std::weak_ptr<T>.

The std::auto_ptr has been defined as deprecated type because it was not able to work together with C++ Standard Template Library (STL) [10]. Previously [11], the unstandard boost::shared_ptr<T> was suggested to use instead of. The std::auto_ptr guarantees that only one auto_ptr object is responsible for the deallocation. However, std::auto_ptr is copyable object and when an auto_ptr is copied into another one, the original becomes null [12]. The std::unique_ptr is introduced in C++11 as a replacement of std::auto_ptr. This smart pointer type also guarantees the unique ownership but it is not capable. This pointer type offers move semantics and operations [13]. The std::shared_ptr offers a reference-counted approach. The std::shared_ptr keeps track of how many shared_ptr objects refer to an object that is allocated dynamically. When a shared_ptr object is copied the counter is incremented and when an object is destructed the counter is decremented. Deallocation happens when the counter becomes 0. The std::weak_ptr is a smart pointer that holds a non-owning ('weak') reference to an object that is managed by std::shared_ptr, thus it models a temporary ownership. It can be used when one observes an object but does not guarantee the object's survival. Nevertheless, the std::weak_ptr can be used to break circular references of std::shared_ptr. The

modern smart pointers (e.g. std::shared_ptr) have implemented firstly in the Boost libraries that offer further smart pointer types [14].

C++17's planned feature, the std::optional manages on optional value. This is a value that may or may not be present. It can be used when an operation may fail. The public interface of the std::optional is similar to pointers. However, pointers are able to simulate optional objects.

3. Performance evaluation

In this section, the purpose of analysis and the methodology are described, what and why is measured during tests. Defining exact methodology is necessary to evaluate and compare tools [15]. For test cases, code parts are provided at the end of the appreciate subsection.

3.1. Purpose

All the times, the recommendation is that smart pointers should be used wherever one can since the C++11 standard. Although, these template classes have different behavior and working logic that impacts the performance.

Therefore, in this paper, the answers for two questions are seeking, first of them is about smart pointer performance, how much performance overhead comes to play with smart pointers?

Another one is about the optionality problem, which is best representation for optional values. Pointers can be null pointers, but what other possibilities are available to represent optional values in C++ and how they perform. For instance, when software is developed, in some cases, there can be variables that can be null, e.g. create structures in the program from externally provided data (e.g. database or file) but some fields are not mandatory, so sometimes they are not presented. In this case, you should implement these structures to be able to handle null values. There are many choices in this case, for instance raw pointers, smart pointers, maybe optional (from C++17 standard). In this paper, these are the questions addressed and comprehensive performance analysis of smart pointers and optional answers. It is also examined how they perform against each other. Also this is good comparison for smart and raw pointers.

3.2. Test environment

The tests have been performed on one machine, which has an Intel Core i5-4200U 1.6 GHz cpu and 8 GB ram. There are two operation systems; one of them is an Arch Linux 64bit with grsec kernel 4.7.10 and the other one is Ubuntu 16.04.01LTS with kernel 4.4.

Two kinds of compiler were used with different STL implementations:

- gcc 6.2.1 compiler with libstdc++ 6.2.1 STL implementation;
- clang 3.9.0 compiler with libc++ 3.8.0 STL implementation.

The presented results are aggregated from many runs, but on different platform and different hardware the results could be different from ours.

To profile runtime performance of test cases, two different profiler tools were used.

- Google perftools [16], CPU profiler 2.5;
    - o   Google perftools is an utility collection to analyze C++ programs;
    - o   CPU profiler is part of perftools, used to profile runtime of programs. It uses profiling events with predefined frequency;

- Valgrind [17], Callgrind 3.12.0;
    - o   Callgrind is dynamic software analyzer that collects calls during the execution by recording CPU instructions [18].

In the test cases, Google perftools has been used to measure runtimes, because it gives much better result in time measurement. Valgrind is only used for profiling CPU instructions and got an overview of distribution. However, Valgrind is widely-used in many open-source projects and it is a quite mature technology [19].

3.3. Test cases

In the analysis, four different types are participated and their properties were analyzed through different test cases. For each types, large number of samples have been performed (about 100 000 000 - 1 000 000 000) by test cases. These tests are profiled with previously mentioned tools and their results are analyzed at the end.

For each test case try to measure one important property, for example the cost of copy of object. Two different example code snippets are provided for all test cases because we have checked them with different optimization levels of compiler.

So, to avoid from the compiler optimize out fully the case, a bit more complex test cases are needed for maximum optimization level because nowadays efficient algorithms for optimizing are available [20].

Two optimization levels were used:

- no optimization (-O0);
- maximum optimization (-O3).

The test cases:

- Construction and destruction;
    - o   Create an object and destroy it. The reason of empty object is that, the stored object's construction cost does not affect the result of test case;
    - o   Test code for O0:

```
for (std::size_t i = 0; i != count; ++i) {
  std::shared_ptr<int> ptr;
}
```

    - o   Test code for O3:

```
for (std::size_t i = 0; i != count; ++i) {
  std::shared_ptr<int> ptr =
    std::make_shared<int>(i);
  globalX = ptr.get();
}
```

- Copy;
  - o Create an object by copying another one;
  - o unique_ptr excluded from this test case, because it is not copyable;
  - o Test code for O0:

```
int* ptr = new int(9);
for (std::size_t i = 0; i != count; ++i) {
  int* copyOfPtr = ptr;
}
delete ptr;
```

  - o Test code for O3:

```
int* origPtr = new int(9);
for (std::size_t i = 0; i != count; ++i) {
  int* ptr = origPtr;
  ++*ptr;
  globalX = ptr;
}
delete origPtr;
```

- Assignment;
  - o Assign an object to another one;
  - o unique_ptr excluded from this test case too, because it is not assignable;
  - o Test code for O0:

```
std::experimental::optional<int> opt;
std::experimental::optional<int> copyOfOpt;
for (std::size_t i = 0; i != count; ++i) {
  copyOfOpt = opt;
}
```

  - o Test code for O3:

```
std::experimental::optional<int> opt(0);
std::experimental::optional<int> copyOfOpt;
for (std::size_t i = 0; i != count; ++i) {
  copyOfOpt = opt;
  ++*copyOfOpt;
  globalX = &*ptr;
}
```

- Dereference
  - Access to the stored object;
  - Test code for O0:

```
std::unique_ptr<int> ptr =
  std::make_unique<int>(0);
for (std::size_t i = 0; i != count; ++i) {
  ++*ptr;
}
```

  - Test code for O3:

```
std::unique_ptr<int> ptr =
  std::make_unique<int>(0);
globalX = ptr.get();
for (std::size_t i = 0; i != count; ++i) {
  ++*ptr;
}
```

3.4. Results

In this section, the measured results are provided for each test case that has been defined in 3.3. Two optimization levels have been involved. Runtime is depicted for test cases.

With zero optimization, try to determine the real performance, how these types perform with no aggressive optimization done by compiler, calling all functions, etc.

With maximum optimization level, the performance in practice can be measured. These measurements are trickier caused by aggressive optimizations such as in-lining, compilation evaluations, etc.

Construction and destruction

- Using no optimization (O0): Smart pointers have higher cost of creating and destroying empty objects than others. Also optional almost has the same cost of raw pointers, (see *Fig. 1*);
- Using full optimization (O3): Using full optimization, construction times are balanced very well. Smart pointers almost have no disadvantage, (see *Fig. 2*).

Copy

- Using no optimization (O0): Shared pointers have much higher cost caused by incrementing and decrementing the atomic reference counter. On the other hand, the experimental implementation of optional performs very well compared to raw pointers too, (see *Fig. 3*);
- Using full optimization (O3): Same as in construction case, times are balanced mainly, but shared_ptr is slower a bit than others, caused by reference counting handling (see *Fig. 4*).
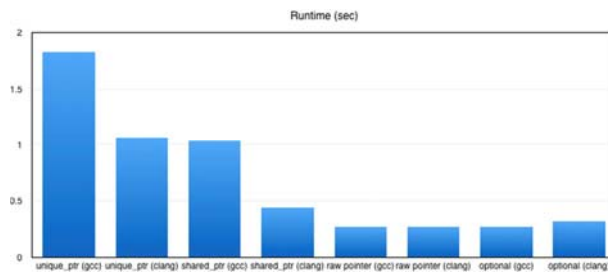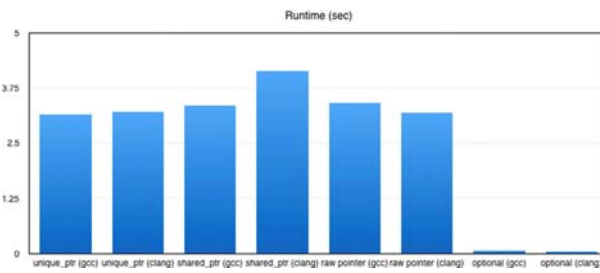
Fig. 1. The runtime of construction with - O0

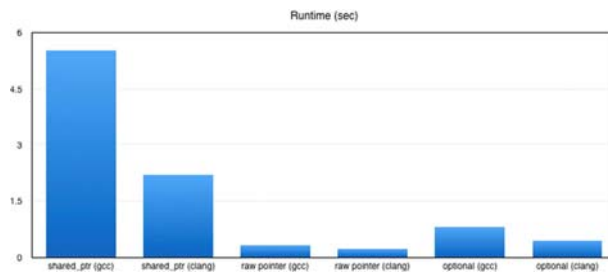*Fig. 2.* The runtime of construction with - O3
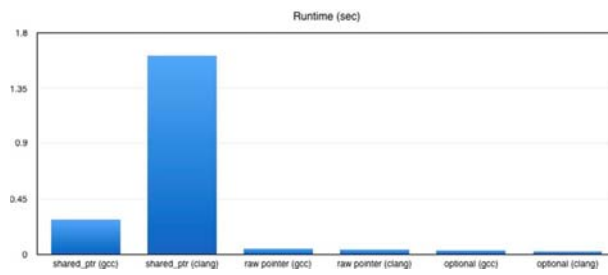

Fig. 3. Runtimes of copy with O0

Fig. 4. Runtimes of copy with O3

Assignment

- Using no optimization (O0): Same situation as previously, shared_ptr's cost is greater than others (see *Fig. 5*);
- Using full optimization (O3): Exactly the same results to the copy tests (see *Fig. 6*).

Dereference

- Using no optimization (O0): Accessing the stored object is balanced without optimization too. Smart pointers almost have no overhead compared to others (see *Fig. 7*);
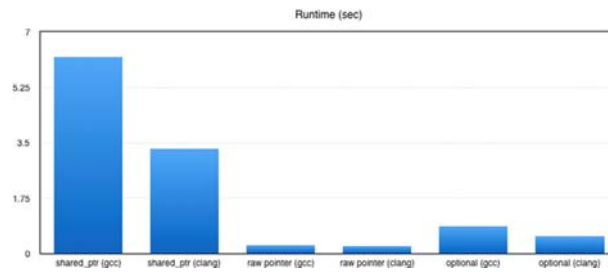- Using full optimization (O3) (see *Fig. 8*).

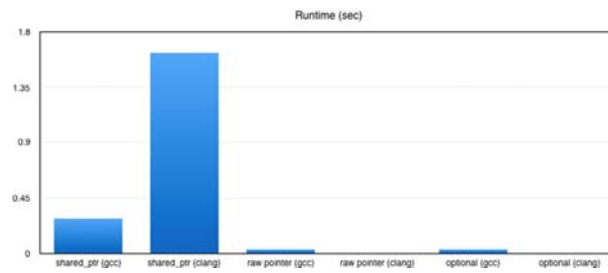*Fig. 5.* Assignment cost with O0

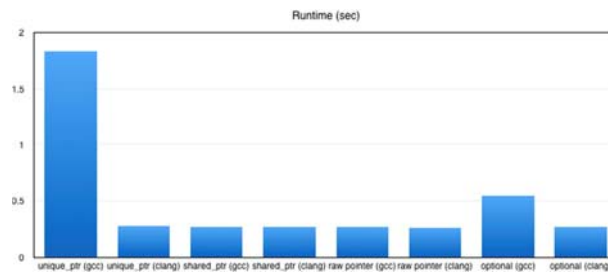*Fig. 6.* Runtimes of assignment with O3
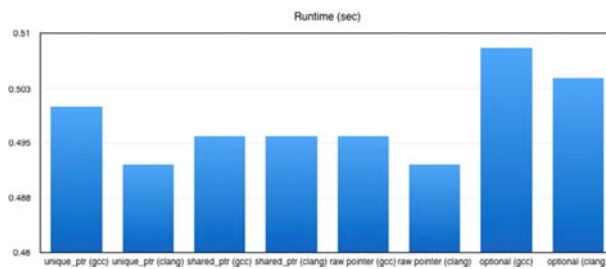

Fig. 7. Dereference cost with O0

Fig. 8. Runtimes of dereference with O3

4. Conclusion

In this section the result is concluded that is relevant to the two different questions: how effectively the smart pointers can be used as optional values and how they perform related to each other.

Without optimization, smart pointers have very large overhead construction, destruction, copy and assignment against raw pointers with both implementations. In this case, using smart pointers is good idea for holding the objects ownership, because it manages the objects' lifetime, but do not overuse them. Also, passing smart pointers across the program could be very expensive (e.g.: shared_ptr by value). At performance critic parts of program, raw pointers are much better.

With full optimization, raw pointers almost have no advantage from the view of performance. Smart pointer implementations are well optimized by the compiler and their performance is not significantly lower than others. In case when a type is needed that can represent null values, std::optional from the C++17 standard is the best among the analyzed types, they performed well in tests, both cases, with and without optimization.

As a short summary different pointer types have been analyzed from raw pointers to modern smart pointers. It is also measured, which type performs well in case of value can be null. Two different performance profilers were used for the analysis, and Google perftools time measurements are depicted in results.

References

[1]   Staiger-Stöhr S. Practical integrated analysis of pointers, dataflow and control flow, *ACM Transactions on Programming Languages and Systems* (TOPLAS), Vol. 35, No. 1, 2013, Article 5.

[2]   Madhavan R., Ramalingam G., Vaswani K. A framework for efficient modular heap analysis, *Foundations and Trends in Programming Languages*, Vol. 1, No. 4. 2015, pp. 269−381.

[3]   Pataki N., Szűgyi Z., Dévai G. Measuring the overhead of C++ standard template library safe variants, Electronic notes in theoret, *Comput. Sci*, Vol. 264, No. 5, 2011, pp. 71−83.

[4]   Stroustrup B. *The C++ programming language*, 4th Edition, Addison-Wesley, 2013.

[5]   Eddelbuettel D., Sanderson C. RcppArmadillo: Accelerating R with high-performance C++ linear algebra, *Computational Statistics & Data Analysis*, Vol. 71, 2014, pp. 1054−1063.

[6] Pataki N. Safe iterator framework for the C++ standard template library, *Acta Electrotechnica et Informatica*, Vol. 12, No. 1, 2012, pp. 17−24.

[7] Estérie P., Falcou J., Gaunard M., Lapresté J., Lacassagne L. The numerical template toolbox: A modern C++ design for scientific computing, *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3240−3253.

[8] Iglberger K., Hager G., Treibig J., Rüde U. Expression templates revisited: A performance analysis of current methodologies, *Journal on Scientific Computing*, Vol. 34, No. 2, 2012, pp. C42−C69.

[9] Carvalho F. M., Cachopo J. Optimizing memory transactions for large-scale programs, *Journal of Parallel and Distributed Computing*, Vol. 89, 2016, pp. 13−24.

[10] Horváth G., Pataki N. Clang matchers for verified usage of the C++ standard template library, *Annales Mathematicae et Informaticae*, Vol. 44, 2015, pp. 99−109.

[11] Meyers S. *Effective STL* (Standard Template Library), Addison-Wesley, 2003.

[12] Pataki N. C++ standard template library by template specialized containers, *Acta Universitatis Sapientiae, Informatica*, Vol. 3, No. 2, 2011, pp. 141−157.

[13] Baráth Á., Porkoláb Z. Automatic checking of the usage of C++11 move semantics, *Acta Cybernetica*, Vol. 22, 2015, pp. 5−20.

[14] Karlsson B. *Beyond the C++ standard library,* An introduction to Boost, Addison-Wesley, 2006.

[15] Kovacs G, Kuczmann M. Comparison of the different design software tools for the brushless DC motor designing, *Pollack Periodica*, Vol. 8, No. 1, 2013, pp. 179−188.

[16] Google perftools, https://github.com/gperftools/gperftools, (last visited 28 December 2016).

[17] Valgrind, http://valgrind.org/, (last visited 29 December 2016).

[18] Nethercote N., Seward J. Valgrind: A program supervision framework, *Electronic Notes in Theoret, Comput. Sci,* Vol. 89, No. 2, 2003, pp. 44−66.

[19] Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation, *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI' 07), San Diego, California, USA, 10-13 June 2007, pp. 89−100.

[20] Kota L., Jarmai K. Efficient algorithms for optimization of objects and systems *Pollack Periodica*, Vol. 9, No. 1, 2013, pp. 121−132.