C++ und Performance

William Mendat, Angewandte Informatik, Hochschule Offenburg

***Abstract*—Schon seit mehreren Jahrzehnten begleitet uns Software und stellt somit einen signifikanten Bestandteil unseres Lebens dar. In beinahe jeder Lebenssituation heutzutage findet sich Software wieder, sei es nun in den unzähligen Mikrocontrollern eines modernen Autos, welches ein hightech Computer auf rädern darstellt oder ein Videospiel, um von einem langen Arbeitstag abzuschalten. Die Anforderungen an Applikationen wachsen dabei stetig, weshalb neben logischer Korrektheit vermehrt auch Geschwindigkeit von übergeordneter Bedeutung ist. Vor allem im Embedded Bereich bekommt die Geschwindigkeit eines Programms noch mal eine viel größere Bedeutung, da hier auch meist in Lebens gefährdeten Bereichen hantiert wird. Werden die letzten Jahrzehnte betrachtet, zeigt sich, dass die vorherrschende Programmiersprache im Embedded Bereich C ist. Jedoch zeigt sich auch, dass ein Trend zur Programmiersprache C++ entwickelt wurde.**
Dieses Paper befasst sich mit C++, einer Programmiersprache, die ursprünglich von Bjarne Stroustrup im Jahr 1979 als eine Erweiterung von C entwickelt wurde. Genauer soll darauf eingegangen werden, wie mit C++ sehr performant programmiert werden kann. Dabei soll gezeigt werden, wie mit einfach Ticks die Performance verbessert werden kann.

***Index Terms*—C++, Performance**

I. Einleitung

Bei dem Gedanken an die Programmiersprache C++ erschauern viele Programmierer, sei es nun der blutige Anfänger, der noch nie etwas mit Programmierung zu tun hatte, oder der langjährig erfahrene Embedded Programmierer, der lieber auf seine bewährte Programmiersprache C zurückgreift. Dabei stellt sich die Frage, woher die Abneigung gegen diese Sprache kommt, wenn Sie doch Perfekt scheint mit Ihrer nähe zur Hardware und den Vorteilen einer hohen Programmiersprache. Gut, die Antwort darauf ist simpel, denn C++ ist ein Fluch und ein Segen zugleich. C++ ist die momentan mächtigste Programmiersprache und in keiner anderen Programmiersprache bekommt der Programmierer so viele Freiheiten, jedoch ist dies wie alles im Leben nicht um sonst, denn mit vielen Freiheiten kommt auch viel Verantwortung.

Dieses Paper soll sich jedoch nicht mit dem sicheren Programmieren von C++ beschäftigen, sondern eher damit, wie C++ ausgenutzt werden kann, um sehr performante Software zu schreiben. Tatsache ist nämlich, dass genau so Performanter, wenn nicht sogar noch performanteren Code geschrieben werden kann wie in C, jedoch um dies zu erreichen, sollte sich an die ursprünglichen Prinzipien von C++ gehalten werden. Eines der ersten und zudem auch wichtigsten Prinzipien ist das *zero overhead abstraction* Prinzip beziehungsweise, nicht für das zu Bezahlen, was nicht gebraucht wird [5][vgl.]. Unterstrichen wird dies noch mit dem Zitat:

> Technically, C++ rests on two pillars. A direct map to hardware (initially from C) and zero-overhead abstraction in production code (initially from Simula where it wasn´t zero overhead). Depart from those and the language is no longer C++ [4].

II. Take Four Problem

Kontrovers soll dieses Paper mit der Thematik beginnen, wann es akzeptabel ist, nicht grundsätzlich auf Performance zu achten. Dazu soll das *Take Four Problem* genauer betrachtet werden. Was ist das *Take Four Problem*? Nun das *Take Four Problem* zeigt sechs Qualitätsmerkmale an Software, die folgenden Schaubild dargestellt werden:

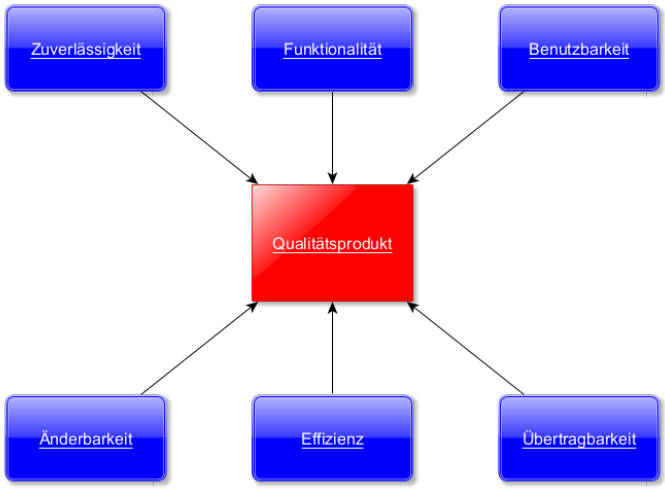

Abbildung 1: Take Four Problem

Das Problem, welches hier dargestellt werden soll, ist es, dass bei der Entwicklung von Software sich auf vier der sechs Qualitätsmerkmale geeinigt werden muss, da sich manche Qualitätsmerkmale gegenseitig ausschließen. Um ein konkretes Beispiel eines solchen Konfliktes zu nennen: Effizienz und Zuverlässigkeit schließen sich gegenseitig aus, da Zuverlässigkeit viel auf Fehlertoleranz baut und das Programm einfach performanter wird, wenn nicht jeder *Pointer* auf *null* abgefragt wird. Auch die Wartbarkeit leidet meistens unter der Effizienz, was den Code dann zum einen schlechter zu Lesen oder Erweitern ist.

Zu sehen ist also, dass es auf den Anwendungsfall stark

ankommt. Ist das Produkt eine Webseite, die ständig weiterentwickelt wird, dann ist es viel wichtiger, dass die Software leicht erweiterbar geschrieben ist, zumal es bei einer Webseite nicht darauf ankommt, ob sie 20 oder 200 Millisekunden braucht, um zu laden.

Andersherum gesehen ist die Performance von Echtzeit kritischen Systemen von großer Relevanz. Bei einem Airbag beispielsweise kann es lebensentscheidend sein, ob dieser innerhalb von 20 oder 200 Millisekunden rauskommt. Im Endeffekt muss im Vorfeld entschieden werden, welche Kriterien für die Software als wirklich relevant erachtet werden.

III. Speicher Management

Der Speicher ist ganz klar die wichtigste Komponente beim schreiben von einem Programm, denn ohne den Speicher geht nichts. Da diese Komponente so enorm wichtig ist, muss auch besonders darauf acht gegeben werden. Vor allem wenn es um die Performance eines Programms geht, kann beim Speicher Management viel herausgeholt werden. In C++ ist dem Programmierer die Möglichkeit gegeben, zwischen dem *Stack* und dem *Heap* zu Entscheiden, wo dieser seine Objekte speichert.

A. Stack

Dabei ist die Speicherung auf dem *Stack* aus Performanter Sicht sehr lukrativ, da dass Speichern und Verwerfen von Objekten sehr schnell von statten gehen kann. Jedoch ist nichts umsonst und der *Stack* kommt mit einigen Einschränkungen wie zum Beispiel der festen Größe, die in Embedded Systems nicht besonders groß sein muss, oder auch das keine Dynamischen Objekte dort gespeichert werden können [1].

B. Heap

Sollen Objekte nun Dynamisch gespeichert werden, dann kommt der Programmierer nicht drum herum, diese Objekte auf dem *Heap* abzulegen. Dies kann erreicht werden, mit der Verwendung von der Funktion *malloc* oder mit dem C++ Operator *new*. Im Endeffekt wird *malloc* im *new* Operator aufgerufen, der einzige Unterschied zwischen *malloc* und *new* ist, das bei *new* noch der Konstruktor des Objektes aufgerufen wird. Das große Problem, welches entsteht bei der Verwendung vom *Heap*, ist dass diese Operation sehr teuer aus Performanter Sicht ist. Zwei Faktoren weshalb diese Operation so teuer ist spielen dabei eine Rolle. Zum einen ist das Anfordern von frischem Speicher an sich eine sehr teure Angelegenheit und zum anderen ist die Platzierung der Daten komplett willkürlich, was zu sogenannten *cache misses* führen kann.

Cach misses sind unter anderem sehr kostspielig wenn es um Mehrdimensionale Arrays geht. Im folgenden wird als Beispiel ein Dynamisches zweidimensionales Array auf dem *Heap* abgelegt:

```
int main(int argc, char** argv){
 int** array2d = new int*[5];
 for (int i = 0; i < 5; ++i)
  array2d[i] = new int[5];
}
```

Listing 1: Anlegen eines Zweidimensionalem Array auf dem Heap

Nachdem das Array angelegt wurde, hat der Programmierer die Möglichkeit sich die Adressen anzusehen, wie im Folgenden durch eine Konsolen Ausgabe.

```
arr2d[0]: 010B5188
arr2d[1]: 010B51C8
arr2d[2]: 010B52C8
arr2d[3]: 010B5308
arr2d[4]: 010BEE78
```

Listing 2: Mögliche Ausgabe der Adressen

Zu sehen ist, dass die Adressen sehr weit auseinander liegen, was dann sehr Langsam sein kann wenn das Array beschrieben werden soll.

C. Speicher auf dem Heap zuweisen

Wie schon vorher erwähnt sind die *cach misses* nicht die einzigen Performance Probleme, die entstehen wenn Objekte auf dem *Heap* gespeichert werden. Um Speicher vom *Heap* zu bekommen, muss der *memory allocator* folgende Anforderungen erfüllen:

- Es muss *Multithreading* Unterstützen, da alle Threads nur auf den globalen
- *Heap* zugreifen, daher muss es auch eine Art von Synchronisation implementiert
- haben.
- Es muss effizient verschiedene Größen von Objekten zuweisen können.
- Es sollte sich nicht über die Zeit verschlechtern.
- Am Anfang eines Programm wird vom Betriebssystem dem Programm speicher zur Verfügung
- gestellt und wenn dieser Aufgebraucht ist, dann muss es runter zum *kernel* um
- neuen Speicher anzufordern.

All diese Anforderungen und noch einige mehr müssen erfüllt werden! Zu sehen ist also, dass das Speicher auf dem *Heap* keines falls Trivial ist und mit bedacht einzusetzen ist [5][vgl.].

D. Benchmarking Stack vs. Heap

Theorie ist gut, Praxis ist besser! Nach diesem Motto soll in dieser Sektion zwischen den beiden Speicher Möglichkeiten, verglichen werden, wie groß der Performance Unterschied tatsächlich ist. Für diesen Zweck wurde ein Zweidimensionales Array der Größe *10x10* sowohl auf dem *Stack*, als auch auf dem *Heap* gespeichert und dessen Felder auf den Wert 2 gesetzt. Dabei wurde die Compiler Optimierung ausgestellt, damit nichts essentielles vom Compiler bei diesem Versuch wegoptimiert wird. Dieser Vorgang wurde dann *N* Mal wiederholt und resultierte in das folgende Ergebnis:

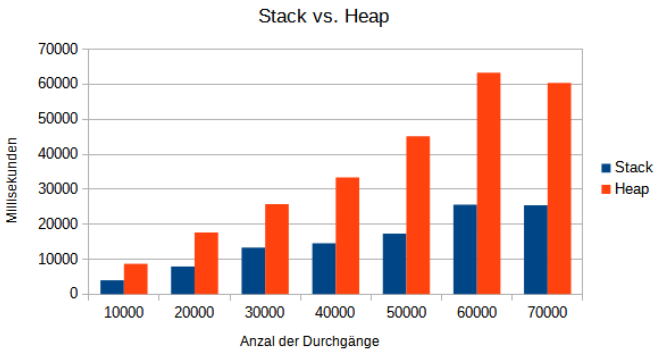

Abbildung 2: *Stack* vs. *Heap*

Wie zu sehen ist, ist die *Heap* Speicherung deutlich langsamer und der Performance unterschied ist keinesfalls zu Unterschätzen.

Ein Interessanter Fakt der zudem noch zu beobachten war, war dass der Compiler auf der höchsten Compiler Optimierungsstufe, in der Lage war, den Stack Code wegzuoptimieren. Dadurch wurde der Komplette Code für die Stack Implementierung verworfen und die gemessene Zeit resultierte in 0 Mikrosekunden.

E. New Operator Überschreiben

> If we need dynamic memory and can't avoid allocations, we may have to write a custom memory manager that performs better for our specific needs [1].

In der Vorhergehenden Sektion wurde für das Speichern auf dem *Heap* die Standard Implementation von C beziehungsweise C++ benutzt, dies ist jedoch nur eine generelle Funktion zum Zuweisen von Speicher, deswegen auch auf nichts spezialisiert. C++ bietet dem Programmierer die Möglichkeit seine eigene Implementation zu schreiben, indem der *new* Operator Überschrieben wird. Zum Beispiel könnte am Anfang des Programms einmalig der gesamte Speicher, welches das ungefähr Programm braucht reserviert werden, um dann vom reservierten Speicher alle Objekte zu verwalten.

Neben der eigen Implementierung existieren natürlich noch externe alternativen zu der Standard Implementierung von *malloc*, wie zum Beispiel die Implementierung von Google, welche die *tcmalloc* Funktion zur Verfügung stellen.

F. Zusammenfassung

Abschließend kann gesagt werden, dass Speicher vom *Heap* extrem kostspielig ist und hinter dem *new* Operator mehr steckt als manch einer vielleicht vermutet. Wie zudem auch zu sehen war, ist die Möglichkeit gegeben, eine eigene Implementierung zu schreiben oder auf externe Implementierungen zurückzugreifen. Alles im allem bleibt die Benutzung des *Heap* sehr teuer und es sollte wann immer möglich mit dem *Stack* gearbeitet werden.

IV. Temporäre Objekte

Um Temporäre Objekte verstehen zu können, sollte zuerst geklärt werden, was *lvalues* und *rvalues* sind und worin diese zu Unterscheiden sind. Die Antwort darauf soll folgendes Zitat liefern:

> In concept (though not always in practice), rvalues correspond to temporary objects returned from functions, while lvalues correspond to objects you can refer to, either by name or by following a pointer or lvalue reference. A useful heuristic to determine whether an expression is an lvalue is to ask if you can take its address. If you can, it typically is. If you can't, it's usually an rvalue [6].

Wie zu sehen ist, resultieren *rvalues* oft in Temporären Objekten. Diese Objekte sind in so fern schlecht, da diese vom Compiler erzeugt werden nur um Daten zu servieren, um dann sofort wieder verworfen zu werden. Solch ein Verhalten frisst dann unnötigerweise Performance .[5]

Um ein solches Verhalten darzustellen, wird im folgenden eine Simple eigen Implementation der *String* klasse, sowie die Klasse *Person*, die eine *const* Referenz übergeben im Konstruktor übergeben bekommt gezeigt:

```
class String {
public:
 String(const char* string) {
  printf("Created\n");
  m_Size = strlen(string);
  m_Data = new char[m_Size + 1];
  memcpy(m_Data, string, m_Size);
  m_Data[m_Size] = 0;
 }
 [...]
 ~String() {
     printf("Deleted\n");
     delete m_Data;
    }
protected:
    uint32_t m_Size;
    char* m_Data;
};

class Person{
public:
 //Konstructor akzeptiert auch rvalues
 Person(const String& name) : m_Name(name) { }
private:
 String m_Name;
}
```

Listing 3: Implementation einer String klasse

Wenn nun eine Instance von Person mit einem *rvalue* erstellt wird, dann wird zuerst ein Objekt vom Typ *String* erzeugt, nur um dann die Daten an *m_Name* zu kopieren. Im späteren Verlauf dieses Papers wird noch auf das Kopieren von Objekten eingegangen. Jedoch kann für den Moment gesagt werden, dass das Verhalten, ein Objekt lediglich zu erzeugen

nur um die Daten dann weiterzureichen sehr zu lasten der Performance gehen.

V. Kopieren von Objekten

Sobald ein Objekt ein anderes Objekt zugewiesen wird, findet ein Kopiervorgang statt. Dabei existieren zwei arten von Kopiermechanismen, einmal die *flache* Kopie und die *tiefe* Kopie. Eine *flache* Kopie ist eine Pseudo-Kopie, das bedeutet, dass hier nicht wirklich Kopiert wird, sondern nur auf die Adresse des Objektes verwiesen wird, währenddessen ist eine *tiefe*, eine echte Kopie, bei der die Daten an eine neue Adresse geschrieben werden. Dies ist insofern relevant, da bei dem Kopieren von einem Objekt der *copy constructor* aufgerufen wird, und dieser Standardmäßig eine *flache* Kopie durchführt. Dieses Verhalten soll im folgenden mit Hilfe der String klasse dargestellt werden.

```
int main(int argc, char** argv){
 String stringA = "Test";
 String stringB = stringA;
 stringB[1] = 'a';

 std::cout << "StringA: " << stringA << std::endl;
 std::cout << "StringB: " << stringB << std::endl;
}

//Output:
//StringA: Tast
//StringB: Tast
```

Listing 4: Demonstration einer *flachen* Kopie

Wie klar zu erkennen ist, wurde durch die Veränderung von *stringB* auch *stringA* verändert. Dieses verhalten ist jedoch sehr Problematisch, da nun zwei Objekte existieren, die auf die gleiche Adresse verweisen und das Programm abstürzt beim versuch beide Objekte zu löschen. Dies Passiert, weil versucht wird, Speicher freizugeben, der schon freigegeben wurde. Um dieses unerwünschte verhalten zu umgehen, ist dem Programmierer die Möglichkeit gegeben, eine eigene Definition für den *copy constructor* zu schreiben. Für die eigen-implementierte String klasse könnte der *copy constructor* wie folgt aussehen:

```
class String {
 [...]
    String(const String& other) {
        printf("Copied\n");
        m_Size = other.m_Size;
        m_Data = new char[m_Size + 1];
        memcpy(m_Data, other.m_Data, m_Size + 1);
    }
 [...]
};
```

Listing 5: String *copy construtor*

Damit wurde erreicht, dass die beiden Objekte nun komplett voneinander unabhängig sind und das Programm nicht abstürzt wenn beide Objekte gelöscht werden.

Jedoch ist gleichzeitig auch zu sehen, dass beim Kopieren neuer Speicher auf dem *Heap* angelegt werden muss, was wie im Kapitel *Speicher Management* sehr ineffizient ist. Das bedeutet, dass jedes mal, wenn das String Objekt kopiert wird zum Beispiel beim Übergeben an eine Funktion per *Value*, dieser Vorgang unnötigerweise Performance in Anspruch nimmt. Im späteren Verlauf dieser Arbeit wird noch ein C++ Feature gezeigt, welches eingesetzt werden kann um unnötige Kopien zu vermeiden, jedoch für den Augenblick sollte gemerkt werden, dass ein Objekt jedes mal per *Referenz* an eine Funktion übergeben werden [8][vgl.].

VI. Runtime Type Identification

Runtime type Indentification kurz *RTTI*, kann genutzt werden um sicher zwischen verschiedenen Typen in einer Klassenhierarchie zu *casten*. Dabei verwendet *RTTI* Metadaten, die einer Klasse hinzugefügt werden, die dann zur Laufzeit verwendet werden können. Mittels dem $dynamic\_cast$ Operator kann überprüft werden, ob zwischen einer Basisklasse und einer Abgeleiteten klasse gecastet werden kann. Im folgenden wird ein klassisches Beispiel für die Verwendung von $dynamic\_cast$ demonstriert:

```
class Stringbuilder : public String {
public:
 Stringbuilder(const char* string)
  :String(string) { }

    void PrintString() override {
     std::cout << *this
        << " printed with Stringbuilder" << std::
    endl;
    }
};

int main(int argc, char** argv){
 String demo = new Stringbuilder("Demo");
 Stringbuilder* demoBuilder =
 dynamic_cast<StringBuilder*>(demo);

 if(!demoBuilder){
  //handle Error
 }
}
```

Listing 6: Demonstration von *dynamic_cast*

Hier kann nun geprüft werden, ob der *cast* erfolgreich war, indem die Variable *demoBuilder* auf *null* geprüft wird und wenn dem der Fall ist, dann war der *cast* nicht erfolgreich und es kann darauf reagiert werden.

Jedoch kommt es nun zu folgendem Problem: *RTTI* fügt, aus Performance Sicht, unnötigen *overhead* hinzu und die Überprüfung bei $dynamic\_cast$ findet zur Laufzeit statt. Wie schon in der Einleitung erwähnt, ist das *zero overhead prinzip* eines der, wenn nicht sogar das wichtigste Prinzip, welches hier klar verletzt wird. In dem Sinne, sollte, sofern Möglich, *RTTI* ausgeschaltet werden und auf alternativen zurückgegriffen werden. Hier existiert kein Wissenschaftlicher Grundsatz, jedoch eine Möglichkeit zumindest $dynamic\_cast$ zu ersetzten, wäre die Verwendung einer *Virtuellen Funktion*.

VII. Fehlerbehandlungen

Fehlerbehandlungen stehen der Performance eines Programms ständig im Weg, dies ist auch Logisch zu erklären,

da abgefangen und überprüft werden muss, ob es zu einem Fehler kam. Dafür muss Programmcode hinzugefügt werden, der nicht essentiell für den eigentlichen Ablauf des Programms ist. Dennoch kommt der Programmierer nicht drum herum Fehlerbehandlungen in seinem System zu integrieren, da es immer zu unvorhergesehenen Fehlern kommen kann.

A. Exceptions

Um einen Fehler zu behandeln, kann der Programmierer einen Status Code zurückgeben und diesen Überprüfen oder eine *Exception* werfen und die Funktion die Potentiell eine *Exception* werfen könnte mit einem *Try-Catch* Block umringen. Die Vorteile von *Exceptions* gegenüber von Status Codes, sind in dem folgenden Zitat schön zusammengefasst:

> The use of exceptions isolates the error handling code from the normal flow of program execution, and unlike the error code approach, it cannot be ignored or forgotten. Also, automatic destruction of stack objects when an exception is thrown renders a program less likely to leak memory or other resources. With exceptions, once a problem is identified, it cannot be ignored – failure to catch and handle an exception results in program termination [2].

Wie zu sehen ist, bieten *Excetions* nennenswerte Vorteile und haben somit auch eine Daseinsberechtigung.

Obwohl klare Vorteile existieren, sind *Exceptions* seit ihrer Einführung ein stark umstrittenes Thema. Dies hat sich auch nach der Tabellen orientierten Optimierung, dass kein Performance Verlust entsteht, wenn keine *Exceptions* geworfen wird, nicht geändert. Der Grund für die Umstrittenheit ist, dass *Exceptions* gegen das *zero-overhead prinzip* verstoßen. Auch bei der Betrachtung der Performance eines Programms zeigen sich *Exceptions* als sehr Problematisch, da diese Dynamisch zur Laufzeit, zum Beispiel auf dem *Heap* erstellt werden müssen, was wie im Kapitel Speicher Management gezeigt, schon sehr teuer ist, und anschließend noch der Dynamische Type der *Exceptions* herausgefunden werden muss, was in einem *dynamic_cast* resultiert. Ein weiteres Problem, welches mit der Verwendung von *Exceptions* kommt, ist dass nicht gesagt werden kann, wie lange es braucht bis eine *Exceptions* vollständig abgearbeitet ist. Dadurch kann die Situation entstehen, dass während eine *Exceptions* verarbeitet wird, eine andere geworfen wird und somit auch eine unbestimmbare menge an Speicher gebraucht wird.[5]

B. Expected

Exceptions sind zwar sehr langsam, jedoch bringen sie wie vorher im Unterkapitel *Exceptions* erwähnt auch nennenswerte Vorteile gegenüber einfachen Status Codes. Deswegen kann auf die Alternative von *Expected* zurückgegriffen werden, welches einen Kompromiss zwischen Status Codes und *Exceptions* darstellt. *Expected* wird durch *Templateklasse* repräsentiert, die Beispielhaft als Pseudocode, wie folgt aussehen könnte:

```
template <class T>
class Expected {
private:
    union {
        T value;
        std::exception_ptr exception;
    };
public:
    Expected(const T& value) { ... }

    Expected(const std::exception_ptr& e) { ... }

    bool hasError() { ... }

    T getValue() { ... }

    std::exception_ptr getException() { ... }
};
```

Listing 7: Pseudocode einer Expected Klass[7]

Wie zu sehen ist, sind die zwei Variablen in einem *union* gespeichert um das *zero overhead prinzip* nicht zu verletzen. Die Idee dahinter ist es, die *Exception* nicht mehr zu werfen, sondern diese wie einen erwarteten Wert zurück zu geben. Sollte die Funktion erfolgreich durchlaufen, dann kann der erwartete Wert mittels der Funktion *getValue()* bekommen werden. Sobald jedoch ein Fehler aufkommt, wird dieser in der gleichen *Instance* gespeichert und mittels *hasError()* kann dann überprüft werden, ob ein Fehler vorliegt. Dadurch wurde erreicht, dass noch Metainformationen, zum Beispiel repräsentiert anhand von einem *String*, im Falle eines Fehlers zurückgegeben werden kann [7][vgl.].

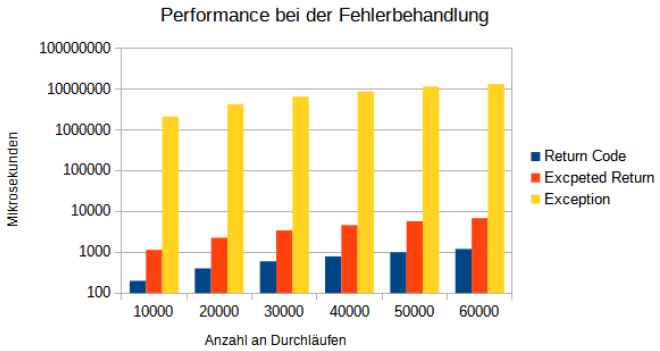

Abbildung 3: Performance bei der Fehlerbehandlung

Abbildung 3 zeigt den Performance unterschied zwischen der vorig besprochenen *Expected* Klasse als *return* Wert und einen *int* als *return* Wert, sowie der Fall, dass eine *Exception* geworfen wird. Klar zu erkennen ist zwar dass der Status Code immer noch wesentlich Performanter ist, jedoch ist der Unterschied zu *Exceptions* deutlich geringer.

C. Zusammenfassung

Trotz dessen, dass *Exceptions* klare Vorteile bieten und durch Optimierungen, die Performance nicht verschlechtern,

wenn diese nicht geworfen werden, ist der Verlust an Performance zu enorm, wenn diese geworfen werden. Große Firmen wie zum Beispiel Google haben *Exceptions* sogar komplett aus Ihren Projekten verboten. Durch die *Templateklasse Expected* existiert eine lukrative Alternative, die wesentlich Performanter ist als *Exceptions*. Dennoch, wenn das Maximum an Performance in einem Programm herausgeholt werden soll, dann sollte mit Status Codes gearbeitet werden.

VIII. Strings

Heutzutage ist es schon fast undenkbar und unmöglich ein Programm ohne *Strings* zu schreiben. Sie sind ein fester Bestandteil, bei so gut wie jeder höheren Programmiersprache, geworden. Viele Programmierer sehen diese deswegen als etwas sehr leichtgewichtiges. Dies kommt daher, dass Strings genauso behandelt werden wie primitive Datentypen, wie zum Beispiel *int* oder *char*. Jedoch wäre es Fatal anzunehmen, dass Operationen mit *Strings* genauso effizient geschehen, wie bei den primitiven Datentypen. Dies zeigt auch das folgende Zitat:

> Strings are convenient because they automatically grow as needed to hold their contents. By contrast, C library functions (strcat(), strcpy(), etc.) act on fixed-size character arrays. To implement this flexibility, strings are allocated dynamically. Dynamic allocation is expensive compared to most other C++ features, so no matter what, strings are going to show up as optimization hot spots [3]

A. Probleme mit Strings

Strings ziehen viele Probleme mit sich, jedoch sind die zwei Größten Probleme die *Strings* mit sich ziehen sind folgende:

- *Strings* werden auf dem *Heap* gespeichert, was, wie im Kapitel *Speicher Management* gezeigt ein sehr Teurer Prozess ist.
- Dadurch dass *Strings* wie primitive Datentypen behandelt werden, verursachen *Strings* viele Kopiervorgänge, was, wie im Kapitel *Kopieren von Objekten* sehr suboptimal ist.

Um zu Demonstrieren, wie Problematisch diese zwei Punkte tatsächlich sind, soll die *String* klasse, die im laufe dieses Papers vorgestellt wurde um die + und = Operatoren erweitert werden, wie im folgenden gezeigt:

```
String& operator=(const String& other) noexcept {
 printf("Copied via =\n");
 if (this == &other)
  return *this;

 delete[] m_Data;
 m_Size = other.m_Size;
 m_Data = new char[m_Size + 1];
 memcpy(m_Data, other.m_Data, m_Size + 1);
 return *this;
}

String& operator+(const String& other) noexcept {
 this->operator+(other.m_Data);
}

String& operator+(const char* other) noexcept {
 printf("Copied via +\n");
 String temp = *this;

 delete[] m_Data;
 int other_Size = strlen(other);
 m_Size = temp.m_Size + other_Size;
 m_Data = new char[m_Size + 1];

 int i = 0;
 for (int j = 0; j < temp.m_Size; ++j, ++i)
  m_Data[i] = temp.m_Data[j];
 for (int j = 0; j < other_Size; ++j, ++i)
  m_Data[i] = other[j];

 m_Data[m_Size] = 0;
 return *this;
}
```

Listing 8: String-Operatoren

Damit die beiden Operatoren so funktionieren, wie sie sollen, muss der Speicher für *m_Data* mittels *new* vergrößert werden. Darauf hin müssen die Daten, entweder mit Standard Funktionen wie *memcpy* oder mit simplen *for-schleifen*, kopiert werden.

Da nun die beiden Operatoren zur Verfügung stehen, kann nun mit der folgenden Sequenz, die Problematik zur Schau gestellt werden. Hierfür wird außerdem noch der *new* Operator überschrieben um später zu überprüfen, wie oft die Funktion aufgerufen wurde.

```
static uint64_t Allocations = 0;

void* operator new(size_t size) {
 ++Allocations;
 return malloc(size);
}

void PrintString(String string) {
 std::cout << string << std::endl;
}

int main(int argc, char** argv)
{
 String demo = String("Hello") + " World" + "!";
 PrintString(demo);
 std::cout << "New called " << Allocations << "
    times" << std::endl;
 std::cin.get();
}
```

Listing 9: String-Operationen

Wird die Sequenz betrachte, dann ist dies nichts Weltbewegendes, jedoch resultiert diese Sequenz schließlich in diesen Output:

```
Created
Copied via +
Copied
Copied via +
Copied
Copied
Copied
Hello World!
New called 7 times
```

Listing 10: Ausgabe des String-Programms

Tatsächlich wurde für dieses doch sehr simple Programm Sieben mal der *new* Operator aufgerufen. Eine Optimierungsansatz für dieses Problem, sind *Move Semantiken*, die im späteren Verlauf dieses Papers noch besprochen werden, jedoch ist dieses Verhalten aus Sicht der Performance inakzeptabel und der Einsatz von *Strings* sollte auf ein nötigstes beschränkt werden.

IX. Fazit

...

X. Schlusswort

References

[1] B. Andrist, V. Sehr, and B. Garney. *C++ High Performance: Master the Art of Optimizing the Functioning of Your C++ Code, 2nd Edition*. Packt Publishing, 2020. ISBN: 9781839216541. URL: https://books.google.de/books?id=U9XczQEACAAJ.

[2] Dave Abrahams. "Technical Report on C++ Performance". In: (). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.

[3] Kurt Guntheroth. *Optimized C++*. First edition. Sebastopol, CA: O'Reilly, 2016. ISBN: 9781491922064. URL: http://proquest.tech.safaribooksonline.de/9781491922057.

[4] H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong. "DIRECTION FOR ISO C++". In: (). URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0939r4.pdf.

[5] M. Krajewski. *Hands-On High Performance Programming with Qt 5: Build cross-platform applications using concurrency, parallel programming, and memory management*. Packt Publishing, 2019. ISBN: 9781789533309. URL: https://books.google.de/books?id=kiWGDwAAQBAJ.

[6] Scott Meyers. *Effective modern C++: 42 specific ways to improve your use of C++11 and C++14*. Online-Ausg. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 9781491903995. URL: http://proquest.tech.safaribooksonline.de/9781491908419.

[7] Amit Nayar. *Investigating the Performance Overhead of C++ Exceptions | PSPDFKit*. 2021. URL: https://pspdfkit.com/blog/2020/performance-overhead-of-exceptions-in-cpp/.

[8] TheCherno. *Copying and Copy Constructors*. Youtube. 2017. URL: https://www.youtube.com/watch?v=BvR1Pgzzr38%5C&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb%5C&index%20=45.