

Attribute-based Checking of C++ Move Semantics

ÁRON BARÁTH and ZOLTÁN PORKOLÁB, Eötvös Loránd University

Worst-case execution time (WCET) analysis is an important research area in various application areas, like real-time and embedded programming as well as a wide range of other high performance applications. However, WCET analysis is a complex and difficult area, because the processor's cache operations, interrupt routines and the operating system impact the execution time. Also, it is very difficult to replicate a specified scenario. Measuring the worst-case execution time as an absolute quantity therefore is still an unsolved problem for the most popular programming languages. In the same time, WCET calculation is also important on the source code level. When evaluating a library which can be compiled on different platforms with different compilers we are interested not in absolute or approximated time but rather certain elementary statements, e.g. number of (expensive) copy instructions of composite types.

In this paper we introduce a new approach of worst-case execution time investigation, which operates at language source level and targets the move semantics of the C++ programming language. The reason of such high-level worst-case execution time investigation is its platform and compiler independence: the various compiler optimizations will less impact the results. The unnecessary expensive copies of C++ objects – like copying containers instead of using move semantics – determine the main time characteristics of the programs. We detect such copies occurring in operations marked as *move operations*, i.e. intended not containing expensive actions.

We implemented a tool prototype to detect copy/move semantics errors in C++ programs. Move operations are marked with generalized attributes – a new feature introduced to C++11 standard. Our prototype is using the open source LLVM/Clang parser infrastructure, therefore highly portable.

Categories and Subject Descriptors: D.3.3 [Programming Languages] Language Constructs and Features; D.2.4 [Software Engineering] Software/Program Verification

Additional Key Words and Phrases: Programming languages, Execution time, Worst-case execution time estimation, C++, C++ generalized attributes, C++ move semantics

1. INTRODUCTION

In this paper we introduce a new approach of worst-case execution time investigation for C++ programs. Worst-case execution time estimation is important in different application areas, for example in embedded programming, and time critical, high responsive server programs. In these areas the "buy a better hardware" philosophy is not an option. The final code must be reviewed, tests and validations must be applied before the program being used. The validation process may contain worst-case execution time estimation (WCET) or measurement. Note that, such measurement are very hard to complete.

The WCET estimation can be defined in various ways, depending on which program features should be estimated. The most obvious but also the most difficult approach is the quantitative *time* estimation (e.g. in seconds) [Wilhelm et al. 2008; Huynh et al. 2011]. Such analysis is a complex and difficult area, because of the processor's cache operations, interrupt routines and the operating system impact the execution time. Also, it is very difficult to replicate a specified scenario. Measuring the worst-case

Author's address: Á. Baráth and Z. Porkoláb, Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary; email: {baratharon, gsd}@caesar.elte.hu

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Z. Budimac, T. Galinac Grbac (eds.): Proceedings of the 3rd Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA), Lovran, Croatia, 19.-22.9.2014, published at <http://ceur-ws.org>.

execution time as an absolute quantity therefore is still an unsolved problem for the most popular programming languages, like C++.

In the other hand, we can make an estimation of the number of executed instructions – this method will ignore any cache accesses and interruptions, but still enough low level. Also, on RISC processors the estimation can be converted into an approximated time. However, these methods must be done for every binary versions and architectures, because the different compilers and optimization strategies.

In the same time, WCET calculation is also important on the source code level. When evaluating a library which can be compiled on different platforms with different compilers we are interested not in absolute or approximated time but rather certain elementary statements, e.g. number of (expensive) copy instructions of certain types.

This problem led to us the high-level WCET investigation. It allows us to process the code at the language source level, which is architecture independent. At the language level the number of assignments, comparisons, and copies can be measured. These discrete values can be capped by a predefined limit and can be checked by static analysis tools. Also, using such tools serious programming errors can be detected, what none of the low-level WCET estimations can reveal.

In this paper, we introduce such a tool for C++ programs, to detect copy/move semantics errors in order to ensure the pertinence of the program. Moreover, the number of copies can be checked, because the unnecessary copies can slow done the code as well.

This paper is organized as follows: in Section 2 we present the necessity of the move semantics in C++, and we present use cases to introduce the move semantics with related code snippets. In Section 3 we presented a new C++11 feature, called generalized attributes, and we used this mechanism to annotate the source code with expected semantics informations. Also, we introduced our Clang-based tool to check semantics and display error messages to the mistakes. Finally, in Section 4 we present our future plans with our tool.

2. C++ MOVE SEMANTICS

The C and C++ languages have value semantics [Stroustrup 1994]. When we use assignment, we copy raw bytes by default. This behavior can cause serious performance issues. In case of concatenating multiple arrays in one assignment, the cost of the temporary objects are high. The used `new` and `delete` operators, the overhead of the extra loops and memory accesses, and the amount of the allocated memory are costly operations. Todd Veldhuizen investigated this issue, and suggested C++ template metaprogramming [Alexandrescu 2001; Czarnecki and Eisenecker 2000; Sinkovics and Porkoláb 2012; Veldhuizen 1995b] and expression templates [Veldhuizen 1995a] as a solution. The base idea is to avoid the creation of temporaries, but "steal" the resources of the operands. Such operation can be implemented library-based [D. Abrahams 2004] almost trivial with overloading, but to distinguish non-destroyable objects from those which can be "wasted" language support is required. Therefore C++11 standard has introduced a new reference type: the *rvalue* reference [Standard 2011; `str`]. In the source code, the rvalue references have a new syntax: `&&`. Using this syntax, the constructor can be overloaded with multiple types; and the *move constructor* has been introduced, as we can see in Figure 1. The move constructor steals the content of the argument object, and set it to an empty but valid (destroyable) state.

Note that, the actual parameter passed in the place of an overloaded constructor may be either an rvalue or an lvalue. When the object passed as parameter is referred by a name, then it is lvalue, otherwise it may be an rvalue. This rule is explained in Figure 2. This is one of the situations where C++ programmers can easy make mistakes, and no errors or warnings will be generated by the compiler.

```

class Base
{
public:
    Base(const Base& rhs); // copy ctor
    Base(Base&& rhs);      // move ctor

    // ...
};

```

Fig. 1. Copy- and move constructors.

```

Base&& f();

void g(Base&& arg_)
{
    Base var1 = arg_; // copy Base::Base(const Base&)
    Base var2 = f();  // move Base::Base(Base&&)
} // arg_ goes out of scope here

```

Fig. 2. Named rvalue explained.

```

class Derived : public Base
{
    Derived(const Derived& rhs); // copy ctor
    Derived(Derived&& rhs);      // move ctor -> move semantics

    // ...
};

```

Fig. 3. Derived class from Base.

```

// wrong
Derived(Derived&& rhs) : Base(rhs) // wrong: rhs is an lvalue
{
    // Derived-specific stuff
}

// good
Derived(Derived&& rhs) : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
    // Derived-specific stuff
}

```

Fig. 4. Wrong and good move constructors.

In Figure 3 we can see a derived class from the Base class. The Derived class has a copy constructor and a move constructor. The implementation of the copy constructor using costly copy operations, while the move constructor uses move semantics.

Because of the rule explained above, if an object has a name, then it behaves like an lvalue. In this case, we must explicitly call the `std::move` function to enforce the move semantics. Without it, the `Base(const Base& rhs)` will be called, i.e. the Base part of the object will be copied instead of using the move semantics. This implies an obvious error in the code – as we can see in Figure 4.

```

template<class T>
void swap(T& a, T& b)
{
    T tmp(std::move(a)); // enforce move semantics
    a = std::move(b);
    b = std::move(tmp);
}

```

Fig. 5. swap with move semantics.

```

template<class T>
void swap(T& a, T& b)
{
    [[move]] T tmp(std::move(a));
    [[move]] a = std::move(b);
    [[move]] b = std::move(tmp);
}

```

Fig. 6. The swap function with annotated statements.

The situation in Figure 5 is similar. It is critical to write the `std::move` call in the first line, because the variable `a` has a name. Without the `std::move` it will be copied to `tmp`, and the code will likely be executed slower than as assumed. Our prototype tool is analyzing such issues to detect the unnecessary copy operations.

3. ATTRIBUTE-BASED ANNOTATIONS AND CHECKING

In C++11 a new feature has been introduced to able the programmers to annotate the code, and to support more sophisticated domain-specific language integration without modifying the C++ compiler [Porkoláb and Sinkovics 2011; Sinkovics and Porkoláb 2012]. The new feature is called *generalized attributes* [Kolpackov 2012; J. Maurer 2008]. Currently this is rarely used, because the lack of standard custom attributes, but it is a great extension opportunity in the language.

Most important C++ compilers, like GNU g++ and the Clang compiler (which is a C++ frontend for the LLVM [Klimek 2013; Lattner et al. 2014]) parses the generalized attributes, binds to the proper Abstract Syntax Tree (AST) node even if Clang displays a warning, because all generalized attributes are ignored for code generation. Even if the attributes are ignored for code generation, they are included in the abstract syntax tree, and they can be used for extension purposes. In our case, we will annotate functions and statements about the expected copy/move semantics. For example, the original code in Figure 5 can be annotated with the `[[move]]` attribute as can be seen in Figure 6.

Though, the statements are validated correctly, the annotations are redundant and every line starts with the same `[[move]]` attribute. To avoid this, the whole function can be annotated as can be seen in Figure 7. Annotating the whole function changes the default semantics of the statements inside the function – the unannotated functions have copy semantics by default.

Our validator tool use two different annotations: the `[[move]]`, and the `[[copy]]`. To determine the expected semantics of a statement needed the default behavior of the function (which defaults to `[[copy]]` in the most cases, but it can be overridden), and the optional statement annotation. If the statement does not have annotations, then the semantics of the statement is the same as the semantics of the function. If the statement contains e.g. a constructor call, then the tool checks whether the proper constructor call is made. When not the proper constructor is called, then the tool displays an error message.

```

template<class T>
[[move]]
void swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}

```

Fig. 7. The annotated version of the swap function.

Furthermore, our tool validates the implementation of the move constructors. As can be seen in Figure 4, easy to make mistakes in the move constructor. As to prevent these errors, the tool sets the default semantics of the move constructors to `[[move]]` instead of `[[copy]]`.

Note that, the builtin types, like `int` and `long` does not have constructors, so the tool will allow to copy primitive types even the `[[move]]` is set.

4. FUTURE WORK

The functionality of the tool can be integrated into the compiler itself, and the 3rd party tool problem can be solved. The compiler traverses the whole AST during the compilation (at least when the parser builds the AST), and our tool traverses the AST too. There is a trivial solution for the issue, because the AST visitor of the compiler may do more work at the same time.

Furthermore, we are investigating the possibilities of the comparison-analysis at language level. The first approach is to measure the depth of nested loops. The second step is to identify definite loops – it is hard to detect loops with "read-only" loop variable (the variable must be incremented/decremented in the *step* part of the loop). Note that, these problems appear in code validation process too, because the understandable loops are important in programming.

All of the features in this paper will be implemented in our experimental programming language, called Welltype. This language is an imperative programming language with strict syntax and strong static type-system. This language supports generalized attributes as well, and the attributes can be used by the programmer – the attributes are read-only at runtime, and the dynamic program loader checks their values at link time, whether are matching with the import- and export specification.

5. CONCLUSION

In this paper we presented the problems in real-time and embedded programming, such as high-performance computing and responsive server programs. For such applications giving an exact execution time estimation is very problematic. Many external factors influences the execution: including internal mechanisms in the processor and in the operating system. Instead of investigating absolute time properties, in our research we focused on measuring copy operations – an operation which highly influences C++ execution times.

We shortly described the C++ move semantics, and their positive effects on the programs. The C++ move semantic is a relatively new language level enhancement for optimizing the program execution avoiding unnecessary copy operations, but keep these optimizations at a safe level. Unfortunately, it is easy to make mistakes when working with move semantics. Many operations intended to use move semantics, in fact, apply expensive copy operations instead. Our method targets such mistakes, making operations planned for move semantic marked explicitly by C++11 annotations and checking their implementations for unwanted copy actions.

We implemented a Clang-based prototype tool to detect copy/move semantic errors in C++ programs. Firstly, our tool helps to avoid the negative effects of the unnecessary copies in execution time. Secondly, the validation of the program: it detects the unfortunate cases when the programmer fails to implement a proper move constructor.

Our tool can deal with both regular functions and constructors. The execution time of our tool is linearly depends only the size of the source code.

We recognised that, the functionality of our tool can be integrated into the compiler, and the overhead of detecting copy/move semantics errors can be greatly decreased. Furthermore, we are intended to implement the same functionality in our experimental programming language, called Welltype.

REFERENCES

The C++ Programming Language, 4th Edition.

- A. Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Boston, MA.
- Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.
- A. Gurtovoy D. Abrahams. 2004. *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, Boston, MA.
- Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-aware data cache analysis for WCET estimation. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 203–212.
- M. Wong J. Maurer. 2008. Towards support for attributes in C++ (Revision 6). Open Standards, N2761. (2008). Retrieved Aug 7, 2014 from www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf
- M. Klimek. 2013. The Clang AST – a Tutorial. LLVM Developers’ Meeting. (April 2013). Retrieved July 28, 2014 from <http://llvm.org/devmtg/2013-04/klimek-slides.pdf>
- B. Kolpackov. 2012. The Clang AST – a Tutorial. (April 2012). Retrieved July 15, 2014 from <http://www.codesynthesis.com/~boris/blog/2012/04/18/cxx11-generalized-attributes/>
- C. Lattner and others. 2014. Clang: a C language family frontend for LLVM. (2014). Retrieved Aug 7, 2014 from <http://clang.llvm.org/>
- Zoltán Porkoláb and Ábel Sinkovics. 2011. Domain-specific language integration with compile-time parser generator library. *ACM SIGPLAN Notices* 46, 2 (2011), 137–146.
- Á Sinkovics and Zoltán Porkoláb. 2012. Domain-specific language integration with c++ template metaprogramming. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments* 32 (2012).
- ISO International Standard. 2011. ISO/IEC 14882:2011(E) Programming Language C++. (2011).
- Bjarne Stroustrup. 1994. *Design and Evolution of C++*. Addison-Wesley.
- Todd Veldhuizen. 1995a. Expression templates. *C++ Report* 7, 5 (1995), 26–31.
- Todd Veldhuizen. 1995b. Using C++ Template Metaprograms. *C++ Report* 7, 4 (1995), 36–43.
- Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, and others. 2008. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 36.