Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 293–300
doi: 10.14794/ICAI.9.2014.2.293

Concurrent object construction in modern object-oriented programming languages*

Viktor Májer[a], Norbert Pataki[b]

[a]EPAM System European Headquarters, Budapest
Viktor_Majer@epam.com

[b]Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
patakino@elte.hu

Abstract

Nowadays concurrency is a key issue in modern object-oriented programming languages. Billions of objects and object graphs are created at runtime, thus parallelization of object construction may result in significant speed-up in applications.

The C++ programming language uses value semantics as basis of object-orientation. Objects can be used as values and C++ offers copy constructors automatically that copy all members of a class. This operation seems to be ideal for parallelization because independent sources need to be copied to independent targets: without locks, without synchronization problems.

In this paper we present scenarios where parallelization seems to be valuable. These scenarios belong to object constructions which is fundamental. We present our approach and tool that makes the copy constructors concurrent. This tool is based on the Clang architecture.

Keywords: C++, concurrency, constructors

MSC: 68N19 Other programming techniques

1. Introduction

Nowadays hardware supports parallel computation. Performance-oriented applications are using manycore processors. The multicore processors are available in smartphones, too. The GPGPU aims execution of general purpose code on the graphic processor. FPGA (Field-programmable gate arrays) architectures can also be bought at fair price.

*This study/research was supported by a special contract No. 18370-8/2013/TUDPOL with the Ministry of Human Recources.

Operating systems and virtualization techniques follow the progress of hardware correctly. They can take advantage of parallel behaviour [1]. On the other hand, the recently widely-used programming languages do not support concurrent programming adequately [10]. Typically, the object-oriented languages are far from ideal approach of parallel execution [11]. These languages support thread-based programming with a base type of thread [5]. The programmers have to create a subtype and deal with synchronization, communication and locks [3]. However, thread is low level construct from the view of the processor's architecture. Hard to write efficient, scalable and maintainable code for multi- and manycore architectures with this approach.

The C++ programming language is famous for taking advantage of the hardware capabilities. It supports thread-based programmming since the C++11 standard [2]. Some elemental types (like `thread`, `future`, `mutex`) can be used but these constructs are not so high-level to deal with current hardware architectures [6]. Unfortunately, they also do not support scalability, thus more convenience tools are required [7, 8, 9].

C++ uses value semantics as basis of object-orientation. Objects can be used as values and C++ offers copy constructors automatically that copy all members of a class. This operation runs sequentially, but it is ideal for parallelization because independent sources need to be copied to independent targets: without locks, without synchronization problems.

In this paper we argue for a new kind of copy constructor execution. Because programmers are eager for easy-to-use and efficient parallel execution, we define copy constructors to run faster than the standard one. We use the Clang architecture for creating custom copy constructors [4]. We have developed a tool that generates concurrent copy constructor under the circumstances. The source code of this tool can be downloaded from the `https://github.com/majerv-articles/ccopy-cpp/tree/master/tool` URL.

This paper is organized as follows. In the section 2 we argue for a new approach for making copy constructors concurrent in C++. After, we present our tool and architecture that makes the execution of constructors parallel in section 4. The evaluation of measurements and the approach is detailed in section 5. Finally, this paper concludes in section 6.

2. Motivation

The object-oriented approach of C++ is based on the value semantics. Let us consider that we have a class that represents complex numbers:

```
class complex {
  double re, im;
  // ...
};
```

The objects that we create are values, not just links. The constructed complex numbers are created on the runtime stack, just like an integer. We can call the copy constructor to create copies from a complex number, even if this constructor is invisible on the public interface. The automatically created copy constructor copies all the members of the class, that is fine in this case:

```
void f() {
  complex a( 3.2, 5.6 );
  complex b = a;
  // ...
}
```

Let us consider a more complicated class that represents students. A student has surname, first name, mother's name and grades grouped by subjects:

```
class student {
  std::string surname, first_name, mother_name;
  std::map<std::string, std::vector<int> > grades;
  // ...
};
```

There is a copy constructor that copies all elements sequentially. However, this operation can be faster if we take advantage of threads. The automatic copying process should be done in a multithreaded way to achieve speed-up. The question is which members are copied on the same thread. We aggregate some tuples that are copied concurrently. Every thread copies one tuple. However, copying every member on a different thread is not reasonable. In this paper we present our approach to make the copy constructor concurrent.

3. Our approach

As we said earlier, the heart of our approach is to copying the data members of an object from a given class in different execution threads. To achieve this, we arrange the data members into tuples, which tuples are disjunct groups of the fields to be copied. The key question is how to form these groups to provide more efficient copy-constructor than its single-threaded version.

To answer this question we concluded performance tests to see which type of fields and data structures are worth to be copied on a separate thread. These tests showed that in case of primitive types the performance gain is negligible, furthermore it is rather a performance overhead.

However these tests also revealed, that

- if a type's copy-constructor is expensive[1] enough, and

[1] in this paper under expensive we always mean expensive in time

- some sort of collection containing objects of this type is stored as a field

then simultaneous copy of these fields can represent a real performance benefit. Based on these experiences our grouping approach for a given class is:

1. Form a group from all primitive-typed or pointer fields
2. Create a group from fields having user declared type
3. Form a group from STL containers containing expensive objects

It is hard to programmatically determine the costs of copying an object[2]. For instance, the compiler cannot know how many elements are stored in a vector when it is copied. Thus we provide "marker interfaces" to extend: `ExpensiveObject` to indicate that the copy of a type is presumably expensive, and `C4Type` to claim a concurrent copy constructor for it.

Having these said, all we have to do is to:

1. look up every class in the compilation units those do not have user-defined copy constructor and marked as `C4Type`
2. generate copy constructor for each of these classes by creating separate copy-groups according to the mentioned grouping rules, let these groups execute on separate threads and finally join these threads

4. Our architecture

To implement the outlined approach we needed a compiler architecture that is flexible and extensible enough to allow this kind of static analysis and source code transformation. We chose the Clang front-end built on the LLVM compiler architecture for this purpose due to its suitable characteristics [4, 12]. The following is an overview of our tool implementation using Clang.

When writing a Clang-based tool, the common entry point is the interface `FrontendAction` that allows execution of user specific actions as part of the compilation. To run tools over the AST (Abstract Syntax Tree) Clang provides the convenience interface `ASTFrontendAction`, which takes care of executing the action. The only part left is to implement the `CreateASTConsumer` method that returns an `ASTConsumer` per translation unit.

```
class FindExpensiveClassAction : public clang::ASTFrontendAction {
public:
  virtual clang::ASTConsumer* CreateASTConsumer(
  clang::CompilerInstance& compiler, llvm::StringRef inFile) {
    return new FindExpensiveClassConsumer(&compiler, inFile);
  }
};
```

[2] it may be addressed as future work

We define `FindExpensiveClassConsumer` that is our implementation of Clang's `ASTConsumer` interface for write generic actions on an AST. `ASTConsumer` provides many different entry points, but for our use case the only one needed is `HandleTranslationUnit`, which is called with `ASTContext` for translation unit.

```
class FindExpensiveClassConsumer : public clang::ASTConsumer {
  CCopyConstructorInjector injector;
  FindExpensiveClassVisitor visitor;
public:
  FindExpensiveClassConsumer(clang::CompilerInstance* compiler,
   llvm::StringRef inFile): inFile(inFile), injector(compiler),
   visitor(&injector) {}

  virtual void HandleTranslationUnit(clang::ASTContext& context) {
    visitor.TraverseDecl(context.getTranslationUnitDecl());
    injector.dump(inFile);
  }
};
```

We traverse each consumed translation unit with a visitor that implements Clang's `RecursiveASTVisitor`. It is called `FindExpensiveClassVisitor` and it holds a reference to our injector that modifies the traversed AST by generating and inserting concurrent copy constructor where applicable. Finally, we write the injector's content back to the parsed source file[3].

The `RecursiveASTVisitor` class provides convenient hooks of the form `bool VisitNodeType(NodeType*)` for most AST nodes. In our case we only need to implement the methods for the relevant node type: `CXXRecordDecl` that represents a C++ struct/union/class.

```
class FindExpensiveClassVisitor
  : public RecursiveASTVisitor<FindExpensiveClassVisitor> {
public:
  bool VisitCXXRecordDecl(CXXRecordDecl* decl) {
    const bool hasCopyCtr = decl->hasUserDeclaredCopyConstructor();
    if (!hasCopyCtr && decl->isCompleteDefinition() && isC4(decl)) {
        injector->inject(recordDecl);
    }
    return true; // we continue visiting elements
  }
private:
  bool isC4(const CXXRecordDecl*const recordDecl) {
    return recordDecl->isDerivedFrom(recordDeclOfC4);
  }
  //..
};
```

[3]practically we dump the content to {inFile}_out.cpp to preserve original code

The code checks for every visited `CXXRecordDecl` whether the requirements are met: a) it is a complete definition, not just a forward declaration; b) it lacks user-defined copy constructor; c) it is marked to have concurrent one.[4]

Visitor uses a `CCopyConstructorInjector` and calls its `inject` method by passing the current `CXXRecordDecl` object being visited. This custom injector class utilizes Clang's `Rewriter` that is the main interface to rewrite buffers (and thus the AST itself). `Rewriter` has an `InsertText(SourceLocation, StringRef)` method that inserts the specified string at the specified location.

For us, the end of a type's definition is adequate as the location when inserting a new copy constructor. This can be determined with Clang's `SourceRange` class available on the visited declaration.

```
void CCopyConstructorInjector::inject(CXXRecordDecl* recordDecl) {
    const SourceRange sourceRange = recordDecl->getSourceRange();
    const SourceLocation sourceLocation= sourceRange.getEnd();
    const std::string typeName = recordDecl->getNameAsString();

    rewriter.InsertText(sourceLocation,
    "public:\n"+ typeName + "(const " + typeName + "&rhs) {\n");
    // insert copy ctr body ...
    rewriter.InsertText(sourceLocation, "}\n");
}
```

For defining the copy constructor's body, the code iterates through the fields of the visited type and arrange them into disjunct groups. For proper grouping we utilize Clang's `Type` and its predicates as demonstrated below:

```
std::string CopyGroups::determineGroupOf(FieldDecl* fieldDecl) {
    const QualType qualType = fieldDecl->getType();
    const Type*const type = qualType.getTypePtr();

    if(type->isUnionType() || type->isPointerType()) // ...
}
```

If more than 1 group is created, injector generates simultaneous copy statements for the groups by utilizing `std::thread` and *lambda* expressions.

```
// copy ctr's body
ccopy::CopyGroups cg;
cg.divideBy(recordDecl->field_begin(), recordDecl->field_end());

for(auto cit = copygroups.begin(); cit != copygroups.end(); ++cit) {
    std::string groupName = cit->first;
    ccopy::CopyGroups::group_type group = cit->second;
    insertCopyStmt(rewriter, groupName, group.getFields());
}
```

[4] it is called C4 check in code that stands for "Candidate for Concurrent Copy Constructor"

A copy statement looks as follows:

```
std::thread default_group([&](){
    i = other.i;
    y = other.y;
});
```

5. Measurements

As we said earlier, it is not easy to determine when copying of data members on separate threads is effective enough to be worth the efforts. Furthermore the application of the approach for wrong scenarios may cause serious performance overhead as well. To see in which use cases it is worth, we concluded some performance tests.

For these tests we used a small program that creates objects from different input classes, calls the copy constructor of them and measures the execution time of the copy operation by utilizing `std::time` and the `ctime` library. For the tests we simulated an expensive object type by having a 20ms sleep in its copy constructor. The test cases and the measured results are summarized by the following table.

Input test class	Exec. time (normal, sec)	Exec. time (concurrent, sec)
Case 1: 2 `std::array` members storing 50k `float`s each	min: 0.000183 max: 0.000499 avg: 0.000442	min: 0.000655 max: 0.001142 avg: 0.000882
Case 2: 2 vector members, both store 50k POD objects (having char and float members)	min: 0.002108 max: 0.003976 avg: 0.002617	min: 0.002673 max: 0.010437 avg: 0.005637
Case 3: 2 vector members storing 100 expensive objects each	min: 3.72 max: 4.41 avg: 4.13	min: 1.76 max: 2.23 avg: 2.10
Case 4: 3 vector members storing 100 expensive objects each	min: 5.94 max: 6.78 avg: 6.34	min: 3.86 max: 4.77 avg: 4.25
Case 5: 3 vector members storing 500 expensive objects each	min: 5.94 max: 6.78 avg: 6.34	min: 3.86 max: 4.77 avg: 4.25

As a conclusion it can be said, that if an object has expensive data members, especially large collections of them, our approach offers a better alternative to copy constructor. The slower the copy constructor, the more speed-up our approach achieves.

6. Conclusion

Nowadays concurrency plays in important role in software engineering. However, object-oriented programming languages do not support it adequatly. In this paper we argue for a new approach of C++'s copy construction. We have developed a tool that generates concurrent copy constructor under the circumstances based on the Clang architecture. We have measured the performance of our approach as well.

References

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I, Warfield, A.: *Xen and the Art of Virtualization*, In Proc. of the 19th ACM SOSP, 2003, pp. 164–173.

[2] Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: *Mathematizing C++ concurrency*, in Proc. of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, (POPL'11), 2011, pp. 55-66

[3] Hughes, C., Hughes, T.: "Object-Oriented Multithreading Using C++", Wiley (1997)

[4] Lattner, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.

[5] Oaks, S., Wong, H.: "Java Threads" (Third Edition), O'Reilly (2004)

[6] Stroustrup, B.: "The C++ Programming Language" (Fourth Edition), Addison-Wesley (2013)

[7] Szűgyi, Z., Pataki N.: *Generative Version of the FastFlow Multicore Library*, Electronic Notes in Theoret. Comput. Sci., **279(3)**, pp. 73–84.

[8] Szűgyi, Z., Török, M., Pataki N., Kozsik, T.: *Multicore C++ Standard Template Library with C++0x*, in Proc. of International Conference on Numerical Analysis and Applied Mathematics (ICNAAM 2011), American Institute of Physics, **1389**, pp. 857–860.

[9] Szűgyi, Z., Török, M., Pataki N.: *Towards a multicore C++ Standard Template Library*, in Proc. of Workshop on Generative Programming 2011 (WGT 2011), 2011, pp. 38–48.

[10] Szűgyi, Z., Török, M., Pataki N., Kozsik, T.: *High-level Multicore Programming with C++11*, Computer Science and Information Systems, **9(3)**, pp. 1187–1202.

[11] Tripathi, A., Berge, E., Aksit, M.: *An implementation of object-oriented concurrent programmming language SINA*, Software Practice and Experience, **19(3)**, pp. 235–256.

[12] Voufo, L., Zalewski, M., Lumsdaine, A.: *ConceptClang: an implementation of C++ concepts in Clang*, in Proc. of the seventh ACM SIGPLAN workshop on Generic programming (WGP '11), 2011, pp. 71–82.