CTwik: Hot Reloading & Quick-Build System

An Efficient Approach for Building Incremental Changes in a Large C++ Project

Abstract

We extend the model of incremental builders (ex: Makefile, SCons etc) by maintaining more granular indexing of project components instead of file level, which is symbol level indexing. We design a persistent data structure, which maintains intermediate build-state, helping our builder to quickly build new changes on top of previous build-state.

Current build systems like Makefile and SCons maintain only last-edit timestamp of a file, as part of intermediate build-state. Hence current incremental builder take time of the order of minutes to compile a project (when recompiling the entire project would take time of the order of hours), by compiling the changed files only. Our approach reduces it to order of seconds.

How Current Build Systems Work

```cpp
// model_details.hpp

class ModelDetails {
public:
    int aa;
    int bb;
    int bestPrice();
    // ...
};
ModelDetails getModelDetails(int model);
```

```cpp
// model_details.cpp

#include "model_details.hpp"
int ModelDetails::bestPrice() {
    return aa;
}
ModelDetails getModelDetails(int model) {
    ModelDetails m = {model*100, 22};
    return m;
}
```

```cpp
// product.hpp

class Product {
public:
    int aa;
    ...
};
```

```cpp
// car.hpp

#include "product.hpp"
class Car: public Product {
public:
    int getPrice(int model);
    int method1(int x);
    int method2(int x);
    ...
    ...
    int method500(int x);
    ...
    ...
    int method1000(int x);
};
```

```cpp
// car.cpp

#include "model_details.hpp"
#include "car.hpp"
int Car::getPrice(int model) {
    ModelDetails m = getModelDetails(model);
    ...
    ...
    float discount = 0.2; // ...
    ...
    return (int)(m.bestPrice()*(1-discount));
}
int Car::method1(int x) {
    ...
}
...
...
int Car::method1000(int x) {
    ...
}
```

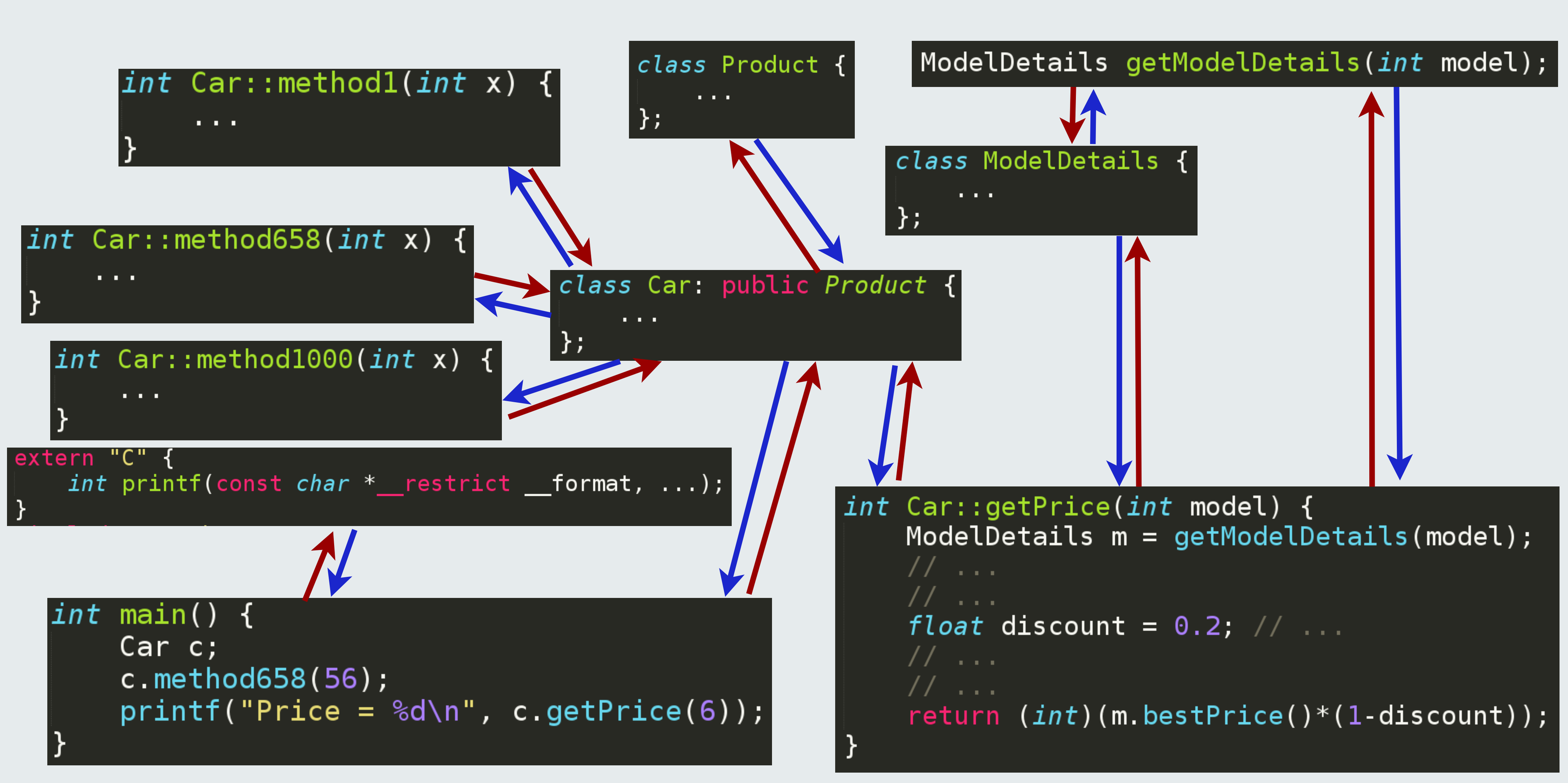
```cpp
// main.cpp

extern "C" {
    int printf(const char *__restrict __format, ...);
}
#include "car.hpp"
int main() {
    Car c;
    c.method658(56);
    printf("Price = %d\n", c.getPrice(6));
}
```

In this sample project, when *Car::getPrice()* method is changed, current incremental builders will compile *car.cpp* to *car.o* and link [*main.o, car.o, model_details.o*] to produce final executable, whereas *main.cpp* and *model_details.cpp* will not be compiled again.

Compiler will recompile all the definitions *method1*, *method2*, ... *method1000*, present in *car.cpp*, which are neither changed nor dependent on definition (may depend on declaration although) of *Car::getPrice()* function. Which is an unnecessary computation and can be prevented.

Approach

CTwik runs as background process, which maintains the symbol dependency graph. Here is graph for above sample project.



Whenever code is edited, dependent and affected components are extracted out and compiled

Dependent and Affected Code Extraction

Definition 1.1 (**Component**). Basic blocks of globally accessible scope excluding 'namespace' and 'extern' blocks. example: function, class, struct, global-statement,.. etc.

Definition 1.2 (**Symbol Dependency Graph**). Graph made of components as vertex, having directed edge $c1 \rightarrow c2$ if component $c2$ is necessary to compile $c1$ i.e. $c2$ is declaring a symbol used in component $c1$. It's refereed as Red colored edge.

Definition 1.3 (**Inverse Symbol Dependency Graph**). Graph made by inverting all the edges of Symbol Dependency Graph. It's edges are referred as **Blue** colored edges. If component $c2$ is necessary to compile $c1$ then $c1$ should be affected by code-edits in $c2$ and should be considered for recompilation.

Definition 1.4. .
$L1$: Set of edited components
$L2$: Set of components reachable from $L1$ via Blue edges.
$L3$: Set of components reachable from $L2$ via Red edges.

Claim: Set $L3$ is necessary and sufficient to be extracted out and recompiled.

Ex: when $Car::getPrice()$ definition is changed, 5 components ($Car::getPrice()$, $ModelDetails$, $getModelDetails$, Car, $Product$) will be extracted out.

Ex: when component $main$ is changed, only 4 components ($main$, Car, $Product$, $printf$) will be extracted out. Note that definition $Car::method658()$ won't be extracted out. Because it's neither necessary to compile $main$ nor $main$ is necessary to compile it. $main$'s compilation depends only on the declaration of $Car::method658()$, which is already picked up as part of class Car.

Linking the Compiled Code

From the previous step, we extracted out minimal set of components to recompile. We sort them topologically, place them in a file '*change.cpp*' and compile it to binary object '*change.o*' using standard compiler.

As we know that compiler converts C++ code into collection of assembly functions. In x86-64 architecture, each of these functions are relocatable, provided that operand of '*call*' instruction is corrected pointed to IP offset of desired assembly-function-symbol. i.e. it's guaranteed that none of '*jump*' instruction points to an IP outside the body of assembly function unless depreciated '*goto*' statement is used in C++ code.

Hence it can be claimed that below defined mechanism to inject new assembly functions (*change.o*) into existing executable (of whole project) produces a logically equivalent executable when all compiled binary object would have linked using standard linker.

```
define Injection(old_code, new_code) {
     updated_symbols = new_code.symbol_map;
     for inst in old_code.instructions {
          if (inst.opcode == CALL &&
               getSymbol(inst.operand) in updated_symbols) {
               inst.oprand = new_IP_offset;
          }
     }
     final_code.instructions = old_code.append(new_code.instructions);
     final_code.symbol_map = old_code.merge(new_code.symbol_map);
     for (symbol, offset) in new_code.relocation_table {
          new_code.instructions[offset].operand = final_code.symbol_map[symbol];
     }
     return final_code;
}
```

Injection process appends new machine code (*change.o*) after machine code of existing executable and override the operand of '*call*' instructions in existing machine code, which are pointing to old IP offset of new symbol.

CTwik: Hot Reloading & Quick-Build System

An Efficient Approach for Building Incremental Changes in a Large C++ Project

Example of Injection Mechanism

In the mentioned sample project, when definition of '*Car::getPrice()*' method is edited, *change.o* will have only one assembly function, labelled '*_ZN3Car8getPriceEi*'. In the already existing executable, all the '*call*' instruction referring to old IP of '*_ZN3Car8getPriceEi*' will be updated to new IP offset, which represents machine code for new definition of *Car::getPrice()* method.

```
00000000004008c4 <_ZN3Car8getPriceEi>:
  55                         # push   %rbp
  ...
  ...
  e8 00 00 00 00             # callq  <_Z15getModelDetailsi>
  ...
  ...
  48 89 c7                   # mov    %rax,%rdi
  e8 00 00 00 00             # callq  <_ZN12ModelDetails9bestPriceEv>
  ...
  ...
  c3                         # retq
```

New Code (*change.o*)

Call operand remains null after compilation. Generally linker overrides it with correct IP offset using symbol table and relocation table.

```
... ...
  ...
  e8 9c 77 e5 6f             # callq  <_ZN3Car8getPriceEi>
... ...
  ...
... ...
  ...
  e8 48 36 6d 1f             # callq  <_ZN3Car8getPriceEi>
... ...
  ...
  e8 a6 1f eb cf             # callq  <_Z15getModelDetailsi>

000000000040078e <_ZN12ModelDetails9bestPriceEv>:
  ...
  ...
  c3                         # retq

000000000040079e <_Z15getModelDetailsi>:
  ...
  ...
  c3                         # retq

00000000004007c4 <_ZN3Car8getPriceEi>:
  55                         # push   %rbp
  ...
  ...
  c3                         # retq

... ...
  ...
  e8 d4 64 fb 2f             # callq  <_ZN3Car8getPriceEi>
```

Existing Executable

```
... ...
  ...
  e8 9c 78 e5 6f             # callq  <_ZN3Car8getPriceEi>
... ...
  ...
... ...
  ...
  e8 48 37 6d 1f             # callq  <_ZN3Car8getPriceEi>
... ...
  ...
  e8 a6 1f eb cf             # callq  <_Z15getModelDetailsi>

000000000040078e <_ZN12ModelDetails9bestPriceEv>:
  ...
  ...
  c3                         # retq

000000000040079e <_Z15getModelDetailsi>:
  ...
  ...
  c3                         # retq

00000000004007c4 <_ZN3Car8getPriceEi>:
  55                         # push   %rbp
  ...
  ...
  c3                         # retq

... ...
  ...
  e8 d4 65 fb 2f             # callq  <_ZN3Car8getPriceEi>
00000000004008c4 <_ZN3Car8getPriceEi>:
  55                         # push   %rbp
  ...
  ...
  e8 c1 ff ff ff             # callq  <_Z15getModelDetailsi>
  ...
  ...
  48 89 c7                   # mov    %rax,%rdi
  e8 98 ff ff ff             # callq  <_ZN12ModelDetails9bestPriceEv>
  ...
  ...
  c3                         # retq
```

Executable after Injection

Call instructions's operand is overridden by new IP offset for symbol '*_ZN3Car8getPriceEi*'

Results

Test Sets	Compilation Time by SCons	Compilation Time by CTwik
1	20 seconds	520 milliseconds
2	47 seconds	1.06 second
3	93 seconds	316 milliseconds
4	150 seconds	876 milliseconds

Test Sets	Time Taken by Linking	Time Taken for Injection
1	6 seconds	100 milliseconds
2	26 seconds	121 milliseconds
3	37 seconds	168 milliseconds
4	72 seconds	300 milliseconds

Hot Reloading

The mechanism to reflect the code changes in already running process is same as injection mechanism on static executable file. Implementation of Injection should be included in program itself, which can be triggered via external input.

Guidelines to Include Injection Mechanism in a Program

- Injection mechanism should execute mutually exclusively with all the running threads in program.
- New changes should not include any function, which is present in current call-stack of any thread.

Most of the time exclusion can be achieved using Read-Write lock having write-priority. Injection mechanism should be wrapped in write-lock and every other thread should be wrapped by read-lock. Every request for injection will wait for all reader threads to finish and block the creation of new reader-thread until injection thread finishes the injection mechanism. However this generalized approach of exclusion doesn't guarantees the deadlock safety and may not work in programs when read-threads are interdependent via other locks.

```
onRequest() {
    Start_New_Thread(Handle_Request, request);
}
```

Adding Support for Injection Mechanism in Existing Program

```
onRequest() {
    Start_New_Thread(Handle_Request_Wrapper, request);
}
lock = Read_Write_Lock_with_Writers_Priority();
Handle_Request_Wrapper(request) {
    if(request.action == Injection_Mechanism) {
        lock.acquire(WRITE);
        Apply_Injection_Mechanism(...);
        lock.release(WRITE);
    } else {
        lock.acquire(READ);
        Handle_Request(request);
        lock.release(READ);
    }
}
```

All threads including injection-thread should be initiated from very top level function in call-stack so that there are very small number of function in call-stack during injection.

Limitations

Limitations

- Compatible only with '*no-optimization*' mode of compiler.
- Not compatible with '*goto*' statement in C++ code.

Implementation Backlogs

- Hot Reloading mechanism is not fully implemented.
- Support for dynamic libraries is not implemented.

Connect with Author for Explanation ! Author is Online Right Now !

WhatsApp
+91 7503-759-053

Google Hangout
mohitsaini1196@gmail.com