Implementing Continuation based language in LLVM and Clang

Kaito TOKUMORI
University of the Ryukyus
Email: kaito@cr.ie.u-ryukyu.ac.jp

Shinji KONO
University of the Ryukyus
Email: kono@ie.u-ryukyu.ac.jp

***Abstract*—The programming paradigm which use data segments and code segments are proposed. This paradigm uses Continuation based C (CbC), which a slight modified C language. Code segments are units of calculation and Data segments are sets of typed data. We use these segments as units of computation and meta computation. In this paper we show the implementation of CbC on LLVM and Clang 3.7.**

I. A Practical Continuation based Language

The proposed units of programming are named code segments and data segments. Code segments are units of calculation which have no state. Data segments are sets of typed data. Code segments are connected to data segments by a meta data segment called a context. After the execution of a code segment and its context, the next code segment (continuation) is executed.

Continuation based C (CbC) [1], hereafter referred to as CbC, is a slight modified C which supports code segments. It is compatible with C and has continuation as a goto statement.

Code segments and data segments are low level enough to represent computational details, and are architecture independent. They can be used as architecture independent assemblers.

CbC was first implemented on micro-C one path compiler. GCC based CbC compiler is developed in 2008[1]. GCC is GNU Compiler Collection [2]. In GCC version, nested function is used to implement goto with environment in 2011[3]. In this study, we report a latest CbC compiler which is implemented in LLVM and Clang 3.7.

`C--` [4] is also known as a lower level language of C. It has precise type specification and goto statement with parameters. CbC introduces `__code+` type for code segment which makes clear separation of functions and code segments.

II. Continuation based C

CbC's basic programming unit is the code segment. These are not subroutines, but they look like functions because they take input and produce output. Both input and output should be data segments. Table III details the definition of the data segment.

In this example, the code segment **f** takes the input data segment **allocate** (allocate is the data segments identifier) and sends f's output to the code segment **g**. The CbC compiler generates the data segment definition automatically, so writing it is unnecessary. There is no return from code segment **g**. **G** should call another continuation using **goto**. Code segments have input data segments and output data segments. Data segments have two kind of dependency with code segments. First, Code segments access the contents of data segments using field names. So data segments should have the named fields. The second dependency is a data dependency, that is all input data segments should be ready before their execution.

```
__code f(Allocate allocate){
  allocate.size = 0;
  goto g(allocate);
}

// data segment definition
// (generated automatically)
union Data {
  struct Allocate {
    long size;
  } allocate;
};
```

TABLE I
CbC Example

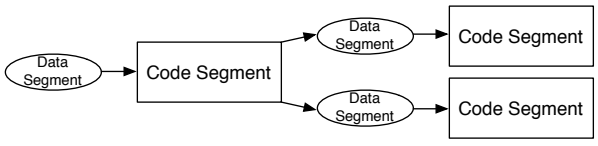

Fig. 1. Code Segments and Data Segments on CbC

Shifting completely, from C to CbC is unnecessary as in CbC we can go to code segments from C functions and call C functions in code segments The latter is straightforward but the former needs further extensions.

```
int main() {
  goto hello("Hello World\n", __return,
    __environment);
}

__code hello(char *s, __code(*ret)(int, void*),
  void *env) {
  printf(s);
  goto (*ret)(123);
}
```

TABLE II
Call C Functions in a Code Segment

In this hello world example, the environment of **main**() and its continuation is kept in the variable **__environment**. The environment and the continuation can be accessed using

__environment and **__return**.The arbitrary mixing of code segments and functions is allowed. The continuation of a **goto** statement never returns to the original function, but goes to the caller or the original function. In that case, it returns the result 123 to the operating system. This continuation is called **goto with environment**.

III. LLVM AND CLANG

The LLVM Project is a collection of modular and reusable compilers and tool chain technologies, and the LLVM Core libraries provide a modern source and target independent optimizer, along with code generation support for many popular CPUs. Clang is an LLVM native C/C++/Objective-C compiler. Figure 2 shows Clang and LLVM's compilation flow.

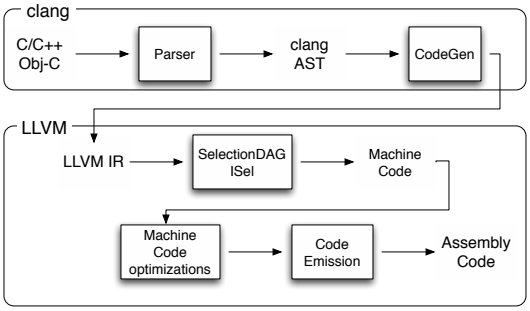

Fig. 2. LLVM and Clang structure

LLVM has an intermediate representation which is called LLVM IR[5]. This part remains unmodified so that the optimization part does not need to be modified.

IV. IMPLEMENTATION IN LLVM AND CLANG

The CbC compiler is implemented in LLVM and Clang using the following ideas.

- Code segments are implemented by C functions.
- Transition is implemented by forced tail call.
- Goto with environment is implemented by setjmp and longjmp.

__code is implemented as a new type keyword in LLVM and Clang. **__code** is similar to an attribute of a function, which means that the function can only be called in tail call elimination. Because of this implementation, code segments can actually be called as C function calls.

Forcing a tail call requires many conditions be met. For example, there should not be a statement after a tail call, the caller and callee's calling conventions must be the same and their types should be cc10, cc11 or fastcc and the callee's return value type has to be the same as the caller's.

All code segments have the void return type and writing statements after continuation is not allowed. As a result, type problems and after statement problems are solved.

Tail call elimination passes are enabled in **BackendUtil.cpp** In Clang, when the optimization level is two or more, tail call elimination passing is enable. Here it has been modified to be enabled anytime, however if the optimization level is one or less, tail call elimination passes only work for code segments. A calling convention problem was also solved. fastcc was selected for a code segment's calling convention. In Clang, calling conventions are managed by the CGFunctionInfo class and its information is set in **CGCall.ccp** (a part of CodeGen), which is where code segments calling conventions were set to fastcc.

Goto with environment is implemented by code rearranging. If the **__environment** or **__return** is declared, the CbC compiler rearranges the code for goto with environment. Setjmp and longjmp are used to do this. setjmp to save the environment before continuation and longjmp to restore it.

V. RESULT

Table III shows the benchmark program.

```
1  int f0(int i) {
2    int k,j;
3    k = 3+i;
4    j = g0(i+3);
5    return k+4+j;
6  }
7
8  int g0(int i) {
9    return h0(i+4)+i;
10 }
11
12 int h0(int i) {
13   return i+4;
14 }
```

TABLE III
BENCHMARK PROGRAM IN C

Fig.III is a normal C program. We can rewrite this program into CbC in several way, `conv1,conv2,conv3`. Basicaly function call is emulated by goto statement with explicit stack. `conv2,conv3` uses extra argument to eliminate these stacks. The CbC `conv3` source is shown in fig.IV Using this benchmark, function call overhead become visible. In order to see the overhead, inline function expansion is prohibited. The benchmark results are shown in TABLE V.

```
1
2  struct cont_interface { // General Return
       Continuation
3      __code (*ret)();
4  };
5
6  __code f2_1(int i,char *sp) {
7      int k,j;
8      k = 3+i;
9      goto g2_1(k,i+3,sp);
10 }
11
12 __code g2_1(int k,int i,char *sp) {
13     goto h2_11(k,i+4,sp);
14 }
15
16 __code h2_1_1(int i,int k,int j,char *sp) {
17     goto f2_0_1(k,i+j,sp);
18 }
19
20 __code h2_11(int i,int k,char *sp) {
21     goto h2_1_1(i,k,i+4,sp);
22 }
23
24 __code f2_0_1(int k,int j,char *sp) {
25     goto (( (struct cont_interface *)sp)->ret)(k+4+j
       ,sp);
26 }
```

TABLE IV
BENCHMARK PROGRAM CONV3 IN CBC

	conv1	conv2	conv3
Micro-C	6.875	2.4562	3.105
GCC -O2	2.9438	0.955	1.265
LLVM/clang -O0	5.835	4.1887	5.0625
LLVM/clang -O2	3.3875	2.29	2.5087

TABLE V
EXECUTION TIME(S)

LLVM and Clang compilers are faster than Micro-C when optimization is enabled. This means CbC get benefits from LLVM optimizations. The LLVM and Clang complier is slower than GCC, but GCC cannot compile safely without optimization. This means LLVM can compile more reliably than GCC.

VI. CONCLUSION

This Continuation based language has been designed and implemented for practical use. CbC has been partially implemented using LLVM and Clang 3.7. CbC can use LLVM's optimization. LLVM IR was not modified to implement CbC's compiler.

In the future, data segments, meta code segments and meta data segments for meta computation will be designed and implemented.

REFERENCES

[1] S. Kono and K. Yogi, "Implementing continuation based language in GCC," *Continuation Festa 2008*, 2008.

[2] Free Software Foundation, Inc., "GCC, the GNU Compiler Collection," March 2008. [Online]. Available: http://gcc.gnu.org/

[3] S. Kono, "Demonstration of continuation based c on gcc," *Continuation Workshop*, 2011.

[4] N. Ramsey and S. P. Jones, "A single intermediate language that supports multiple implementations of exceptions," in *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[5] LLVM Language Reference Manual, *http://llvm.org/docs/LangRef.html*.

[6] LLVM Documentation, *http://llvm.org/docs/index.html*.

[7] clang documentation, *http://clang.llvm.org/docs/index.html*.