A Continuation based Programming Language for Embedded Systems

Shinji Kono E-Mail: kono@ie.u-ryukyu.ac.jp

Information Engineering, University of the Ryukyus,
PRESTO, Japan Science and Technology Corporation

Abstract

To solve the gap between hardware and software, continuation based languages are introduced. C with Continuation is a super set of C, which supports light weight continuation. C based Continuation is a subset of C which has no function call. Using light weight continucation, state machine and stack machines are programmed in uniform.

1 Gap between Programming Language and System Description

Today, we have to develop various kind of things from very small one to very big one. Big or small, the systems require complex functions such as complex user interface or i-mode interface in mobile phones. These systems are combination of hardware description of ASIC and software (assembler or C). Some system features new CPU or modified CPU. It requires assembly language level descriptions or a new compiler. These modifications are of course system dependent.

On the other hands, the state of the art of programming languages such as C++ or Java are Object oriented and featuring complex semantics such as message passing with inheritance and protection. These features are useful for dynamic systems. But embedded system are usually used in stable environment. Even in large systems, most computations are routine works or predicted events. Less than 0.1In this situation, object oriented computation does not work well.

The situation is this. Both hardware and software become complex towards different directions. (fig.1)

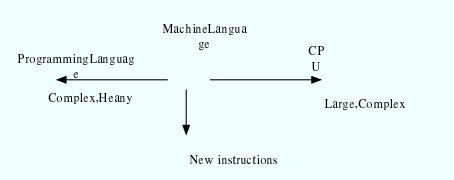

Fig1 Gap between Hardware and Software

In case of C++ or Java, small objects creations and automatic destructions are essential. It requires huge amount of temporal memory such as stacks. This is not allowed in small embedded system. Even in a large system, this causes performance problem both in CPU time and memory space, which affects power consumption and system costs.

Unlike large system in very high performance computer, embedded systems require fine tuning. Java people said, "It's Ok. Computer becomes faster and larger." Or they say complex compiler solves without any directive. This is unlikely true in embedded systems. There is not time to waiting fast JVM or complex com-

piler for special hardware. Manual or automatic fine tunings are sometimes required. But many developers find out the outputs of C++ are too complex.

What we really need in embedded systems are not nice object oriented language, but good small language; something in between C and Verilog.

2 Gap between a State Machine and a Stack Machine

Main difference between Hardware description language and Programming language is a notion of stack. Usually stack is hidden in a syntax of a programming language. But it is completely lacked in Hardware description language. In a hardware description, a stack is an external elements; a register and memories.

An implementation of a Programming languages requires stack machine. Stacks are used to hide local variables in nested function call. This is necessary to make the program readable. But these local variables are visible from a view point of execution. In other words, nested local variables are exists only for human readability.

Stack behavior is very difficult to estimate. If it is not fit within cache or register window, the penalty is very big and it may happens repeatedly. (fig.2)

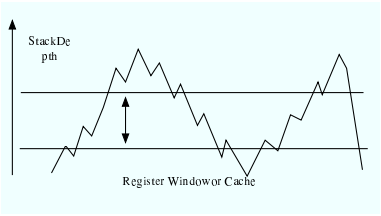

Fig2 Stack Behavior

In normal programming languages, both function calls and message passing are implemented in terms of stacks. But a hardware is usually implemented on a set of finite state machine. Since an embedded system is closer to a hardware than a software, state machine description is preferable. But conventional programming languages are not suitable for state machine descriptions.

In this paper, we introduce a continuation based language, which becomes a bridge between a State Machine and a Stack Machine.

3 Light Weight Continuation

Continuation is introduced in early 70's by Baker [3]. In stead of memorizing return address in a stack, a continuation directly points the address where the answer should be sent. (fig.3)

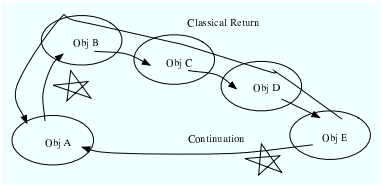

Fig3 Continuation

There are several implementations of Continuation. Scheme uses call/cc function call. It also possible to store a continuation in a global variable. In C, a combination of setjmp and longjmp is a continuation. It is also called catch/throw or try/throw. These are basically used to handle exceptions. This is because it have to handle complex stack structure or so called environment.

An environment are accumulation of locally scoped variables. The size of environments are system dependent but usually 100kbytes to 1 Mbytes. Stack is used to implement an environment. For example, in case of thread library, each thread requires an environment. Implementing Continuation requires handling of these environments. Copying entire registers into stack in order, make it available for other functions. This makes continuations more expensive than function calls.

But the idea of continuation is independent from stack machine. If we don't use stacks,

Continuation is a jump instruction. Everybody knows a jump is faster than a call in an assemble language. Continuation without environment operations is very fast. We can call it light weight continuation.

4 C with Continuation

C with Continuation is an extension of C, which supports light weight continuation.

```
code fact(int n,int result,
      code (*print)()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto *(print)(result);
}
```

`code` is a basic unit of this language. Since it does not handle stack, we cannot call `code` from C functions. But it can be entered by `goto` statement. `code` also has no `return` statement.

This language has two type of goto statements.

- direct goto
- indirect goto

Since this is an extension of C language, `code`s and C functions can be mixed. In order to handle the environment of C language, we also have new builtin variable and a goto statement.

return return address, or light weight continuation of current environment

environment top of the stack, or the environment itself

goto-with-environment equivalent of C `return` statement

```
code target(int n,code (*exit1)(),
   void *exit1env)
{
    ...
        printf("err %d!\n",n);
        goto (*exit1)(0),exit1env;
    ...
}
```

```
int main( int ac, char *av[])
{
    int n;
    n = atoi(av[1]);
    goto target(n,return,environment);
}
```

In this example, return address and environment are passed as arguments of `goto` statement. In the `code target`, `goto` with environment returns to the caller of `main`.

The arguments of `code` are called interface of `code`. These are partially allocated in registers and the rest of arguments are stored in a stack. `goto` with the same interface is guaranteed to compiled into a jump instruction. Unlike function calls, the stack does not glow in `goto` statements. (fig.4)

In a sentence, C with Continuation is a language which has parameterized goto statements. Here after we call it CwC.

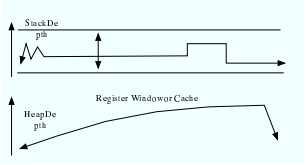

Fig4 Continuation Behavior

5 State Machine description in CwC

It is very easy to implement State Machines in CwC. It is also faster than table driven implementation such as Yacc or lex. Instead of using indirect table jump, we can use simple goto statement.

There are many Virtual Stack Machine implementation in C. These are usually using very large switch statements.

```
for(;;) {
    switch(instruction) {
    case ALOAD:
        aload(); break;
    case GETFIELD:
        getfield(); break;
```

```
            ...
        }
    }
    aload() {
        ...
        return;
    }
```

In this case, we don't have to use auction call for `aload()`. But it is not allowed without very large switch statement. This is not readable and this is not easy to maintenance and it is not easy to compile. In CwC, we can write as follows without any execution penalty.

```
code execute() {
    switch(instruction) {
    case ALOAD:     goto aload();
    case GETFIELD:  goto getfield();
        ...
    }
}
code aload() {
    ...
    goto execute();
}
```

In modern CPUs, a fixed jump statement is pre-fetched and it is executed without clock consumption.

5.1 Small Working Set

`code` element requires very few stack. Of course a programmer can write `int a[1000]` in the interface, but it is very easy to predict the max size. Usually stack size predication is very difficult in C language, but in CwC, the max size of interface is the max stack size.

5.2 Natural Goto Structure

CwC's `code` is considered as a unit of thread. It is not necessary to use thread library such as POSIX thread.

Instead of using thread, we can write simple scheduler in CwC itself. For example, if have several TV game objects, a scheduler perform goto statement with a continuation to the scheduler. (fig.5)

If game objects are connected with links like a list, we can write goto statement to the linked object as a simple scheduler. There is no penalty of function call nor complex thread creation library call.

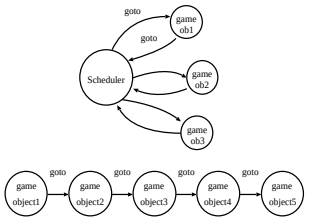

Fig5 Goto Structure

5.3 Interface Consistency

The arguments in parameterized goto in CwC is called interface. The only important language constraint in CwC is consistency of the interfaces. If interfaces are consistent, assembler language or hardware implementation can be mixed. In stead of modifying compiler outputs, the replacement of `code` implementation is enough.

5.4 As a universal intermediate language

`code` can be used as a compiler targets. In compiler technology basic unit is a set of operation between conditional jump or function call. `code` is a basic unit in this sense. If CwC is used as a compiler targets, it is considered as an architecture independent assembler language.

If the compiler outputs no loop or no function call in the `code`, writing `code` compiler is easier then full set of C. The optimization of basic unit level can be done before the CwC compiler.

It is also easy to write tail-optimization base compiler for CwC. Writing LISP compiler or Prolog compiler is very difficult in C, but in CwC, it is straight forward.

5.5 As a lower language of C

We are designing C based Continuation

(CbC); a subset of CwC.

- without function call
- without loop structure

CwC can be compiled into CbC, which is a subset of C. Since stack is easily simulated with in CbC's parameterized goto statements, the conversion is straight forward. (fig.6)

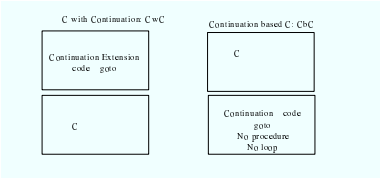

Fig6 CwC and CbC

6 Comparison with other C based languages

Various C based languages are introduced for various area. Ramsey's C - - [5], Synopsys's System C [4] [2] and Spec C [1] by Spec C Open Consortium are compared. CwC is not a language for replacing these languages, but it is designed to implement or simulate these.

6.1 C - -

C - - supports continuations and machine conscious data structure such as 16 bit integer. This is an extension of C. Unlike CwC, it does not use special code structure for continuation. This means C - - is good compatibility with C. CwC is designed to be converted to CbC; the proper subset of C. Adding correct interface definition, C - - can be converted to CbC.

6.2 System C

System C is a set of C++ library for designing or simulating ASIC. It is a C++ application. Large and Complex libraries are suitable for complex circuit design.

CwC is not a language for large Object Oriented Design. Actually it is not an object oriented language. System C can be a higher level description language on top of CwC or CbC.

6.3 Spec C

Spec C is introduced as an executable specification language. It supports various construct for parallel, pipeline or state machine description. Instead of introducing complex parallel syntax, CbC only has goto statements. Parallel executions are described as a state machine. If synchronization mechanisms are required, we can simulate it by a scheduler. Or we can define external temporal logical constraints on state machine descriptions. Then we can compile the constraints to CbC or target hardware such as ASIC with multiple CPU core.

7 Current and Future works

A small integer version of CwC compiler is implemented both in Intel based CPU and MIPS CPU.

Since light weight continuation based programming is rather new notion, we have to accumulate programming examples.

Our implementation targets subset of C unlike other implementation such as long jump or [6].

References

[1] Daniel D. Gajski, Jianwen Zhu, Rainer Dmer, Andreas Gerstlauer, and Shuqing Zhao. *SPEC C:SPECIFICATION LANGUAGE AND METHODOLOGY*. KLUWER ACADEMIC PUBLISHERS, 1999.

[2] Joachim Gerlach and Wolfgang Rosenstiel. System level design using the systemc modeling platform. In *Specification and Description Language 2000*, 2000.

[3] Carl Hewitt and Jr. Henry Baker. Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Dec. 1977.

[4] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware. In *DAC '97*, 1997.

[5] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[6] Y. Ishikawa. Parallel Programming in MPC++ Version 2. In *Future Directions for Parallel C++*,

June 1997.