Clock Driven Programming: A Programming Paradigm which Enables Machine-independent Performance Design

Kenjiro Yamanaka
NTT DATA Corpration
3-3-9 Toyosu, Koto-ku
Tokyo, Japan
yamanakaknj@nttdata.co.jp

ABSTRACT

Cloud computing provides more efficient resource utilization and reduced costs for software systems. However, performance assurance of these systems is difficult because the execution environment cannot be precisely specified and can change dynamically. This paper presents a new programming paradigm, in which software performance is independent from execution environments. The paradigm is called clock driven programming (CDP). The main idea is to introduce the notion of a periodic timer, i.e., clock, for synchronizing program execution timing, just like a clock signal is used in synchronous circuit design. This paper defines CDP and derives the theoretical throughput formula of a CDP program. A CDP program shows the same throughput, even if it runs on different execution environments when its timer period is the same. Therefore, using the CDP enables performance assurance in cloud.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.1.0 [**Programming Techniques**]: General

1. INTRODUCTION

Cloud computing provides dynamic system environments. Using a cloud service like Amazon EC2, system resources can be procured quickly. The pay-as-you-go model can result in more efficient resource utilization and reduced costs. Dynamic system environments make performance assurance difficult. Virtual machine performance varies as a result of other VMs in the same physical machine. There are variations in machines provided in the cloud, even if the same instance type is used. Users cannot specify the exact machine. This is because this restriction enables rapid provisioning and the pay-as-you-go model. As software performance depends on execution environments, system performance can vary.

Performance engineering has been providing performance prediction methods using a model-based approach. If a precise model of the system is available, we can estimate an execution environment which meets performance demand. However, these methods are not suitable for dynamic system environments. In cloud, we cannot obtain some performance parameters required by the performance model. To handle dynamic system environments, we need new approaches. Open-world software [2] seems to be one of such new approaches.

We present a new approach to assure system performance in dynamic system environments. If software performance does not depend on execution environments, system performance assurance in dynamic system environments can be resolved. We present a programming paradigm in which software performance is independent from execution environments. We call it clock driven programming because its model is clock driven, i.e. synchronous, circuit design. In the synchronous design, performance of circuits does not depend on the device technology but on the clock frequency. This feature makes performance assurance easy. Clock driven programming, as presented here, is a transplantation of synchronous circuit design to software. In CDP, a program is driven by just one periodic timer. If the execution time of the program is less than the timer period, software performance is determined not by the execution environment but by the timer period. This is the principle that CDP enables machine-independent performance design.

In Section 2, the definition of clock driven programming is presented. In Section 3, we will consider performance of CDP programs. In Section 4, we present related works and conclusion.

2. CLOCK DRIVEN PROGRAMMING

Clock driven programming is a subset of event driven programming (EDP). Here, C language is used for sample codes, but any programming languages and operating systems that support EDP can be used to implement CDP. In CDP, we write a program in the same procedure as project planning. 1) We list tasks, and then describe each one. 2) We derive a precedence relation between tasks and create a PERT chart. 3) We create a schedule that satisfy the precedence constraints. CDP is explained in this order.

2.1 Task

A task is a state machine which performs the task repeatedly. A task is driven by a periodic timer and the execution time of the task needs to be less than the timer period. This requirement can be divided into the following two requirements.

Requirement 1. A task has an upper bound of execution steps. □

Requirement 2. Each statement in a task has an upper bound of execution time. □

To fulfill these requirements, we define a task as follows.

Definition 1. (Task) A task is a software module that has the following elements.

1) Internal variables: One or more variables holding values. 2) Interface functions: Zero or more global functions for other tasks to refer to and update internal variables. 3) A reset function: A global function for initializing internal variables. 4) A task's main function: A global function called from within the timer event handler.

All functions consist of statements composed by sequence (;) and selection (if then else). □

The task's main function satisfies Requirement 1 because it does not include repetition. To satisfy Requirement 2, we do not use blocking I/O in tasks. Execution of `read()` is blocked by OS until data become available. This means there is no upper bound of the execution time of `read()`. Instead, we use non-blocking I/O in tasks.

Although a task does not include repetition, the computability of a task is equivalent to usual programs. The following theorem assures this property.

THEOREM 1. *(Normal Form Theorem [3] [1])*
Every flowchart is equivalent to a while-program with **one** *occurrence of while-do, provided additional variables are allowed.* □

The normal form theorem is similar to the famous structure theorem. The difference is number of occurrences of while-do: whereas the structure theorem allows any number, the normal form theorem allows just one. In CDP, just one while-do is expressed by a periodic timer execution, and additional variables are expressed as an internal variable, which usually named 'state'. Practically, `for` statements which have a fixed upper bound of iteration count can be used. This type of repetition is considered to be an abbreviation of the combination of sequence and selection.

Listing 1 shows an example of a task description.

Listing 1: Sample task (TaskA)

```
1   typedef enum {free, calc1, calc2, fin} taskA_state_t;
2   typedef struct {
3     taskA_state_t state; int x, y, result;
4   } taskA_data_t;
5   static taskA_data_t taskA_data;
6
7   #define LIMIT (100)
8   const int busy = -1;
9   const int param_err = -2;
10
11  extern int F (int);
12
13  int calc (int x, int y) {
14    int ret = busy;
15    if (y > 0) {
16      if (taskA_data.state == free) {
```

[1]There is no well-known name of this theorem. Harel [3] investigated this theorem and pointed out that its ancestor was Kleene's 1936 normal form theorem for partial recursive functions. Therefore, we call it normal form theorem.

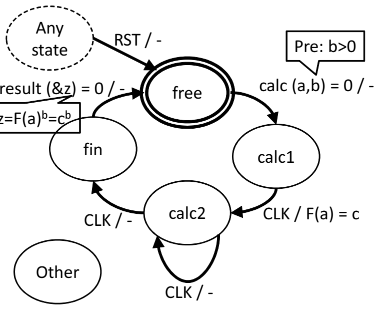

Figure 1: State diagram of TaskA.

```
17          taskA_data.state = calc1;
18          taskA_data.x = x;
19          taskA_data.y = y;
20          ret = 0;
21        }
22      } else {
23        ret = param_err;
24      }
25      return (ret);
26  }
27
28  int get_result (int* y) {
29    int ret = busy;
30    if (taskA_data.state == fin) {
31      *y = taskA_data.result;
32      taskA_data.state = free;
33      ret = 0;
34    }
35    return (ret);
36  }
37
38  void taskA_RST () {
39    taskA_data.state = free;
40  }
41
42  void taskA_CLK () {
43    int i, c;
44
45    switch (taskA_data.state) {
46    case calc1:
47      taskA_data.x = F (taskA_data.x);
48      taskA_data.result = taskA_data.x;
49      taskA_data.y--;
50      taskA_data.state = calc2;
51      break;
52    case calc2:
53      if (taskA_data.y > LIMIT) {
54        c = LIMIT;
55        taskA_data.y -= LIMIT;
56      } else {
57        c = taskA_data.y;
58        taskA_data.y = 0;
59      }
60      for (i = 0; i < c && i < LIMIT; i++) {
61        taskA_data.result *= taskA_data.x;
62      }
63      if (taskA_data.y == 0) {
64        taskA_data.state = fin;
65      }
66      break;
67    default:
68      break;
69    }
70  }
```

TaskA computes $F(x)^y$ repeatedly. Internal variable is defined on Line 5. Two interface functions, `calc()` for requesting a calculation and `get_result()` for getting the result are defined on Line 13 and 28. The reset function which is denoted by name with `_RST` and the main function with `_CLK` are defined on Line 38 and 42.

A state diagram gives a clear view of the task's behavior. A state diagram of taskA is shown in Figure 1. Let S be the whole set of assignment of values to internal variables in a task, and $S_1, \ldots, S_n$ be a finite partition of S. We call each S_i $(1 \le i \le n)$ a state, and denote it as node. In Figure 1, S is divided in five states by the value of `taskA_data.state`.

A transition $S_i \xrightarrow{f/g} S_j$ means that when the function f is executed in the state S_i, f executes an external function g, and the result state becomes S_j. We omit transitions from node S_i to S_i unless the omission leads misunderstanding. A state that includes the result of the reset function is called an accepting state, and denoted by a double line circle. In Figure 1, the state 'free' is the accepting state. On this state, the machine waits for a new request and accepts it. The machine executes the requested task, and returns to an accepting state. In other states, the state machine rejects the new one, so the requester needs to wait.

Listing 2 shows a task (taskB), which uses taskA.

Listing 2: Sample task (TaskB)

```
void taskB_CLK()
{
   switch (taskB_data.state) {
   ...
   case state_10:
      if (calc (taskB_data.a, taskB_data.b) == 0) {
         taskB_data.state = state_11;
      }
      break;
   case state_11:
      if (get_result (&taskB_data.z) == 0) {
         taskB_data.state = state_12;
      }
      break;
   case state_12:
   ...
   }
}
```

Listing 2 describes a part of taskB's main function, to show how to use taskA. TaskB requests taskA to calculate $F(a)^b$ at `state_10`. If the request is accepted, taskB's state is changed to `state_11`. At `state_11`, taskB waits the result by calling `get_result()`. If taskB gets the result, taskB's state is changed to `state_12`.

When taskA is working for taskB, another task should wait for until taskA become free. If there are multiple tasks with the same features, we can handle multiple requests simultaneously. In CDP, we represent multiple tasks with the same features using arrayed task. An arrayed task has internal variables as an array. Each global function of an arrayed task has an additional parameter to select internal variables in the array.

2.2 Creating a PERT Chart

After describing tasks, we derive the precedence relation between tasks. A task's main function calls an interface function of other task to get a processing result. For example, taskB calls `get_result()` to get $F(a)^b$. TaskB can get the result after the execution of `taskA_CLK()`, because processing is performed `taskA_CLK()`. Therefore, `taskA_CLK()` must be executed earlier than `taskB_CLK()`. This means that the precedence relation between tasks can be determined by caller-callee relation of task's interface functions. We can create a PERT chart of a CDP program by reversing arrows in the static call-graph of the program. This procedure is shown in Figure 2. A PERT chart must be a directed acyclic graph (DAG). To keep the created chart DAG, we need to meet the following requirement.

Requirement 3. A call-graph of a CDP program must be acyclic.

The parallel execution of a task's main function and interface function may cause data hazard. If this happens, the assignment of values to internal variables may change non-deterministically. Data hazard between them can be prevented by using the precedence relation. If scheduling is carried out by using this, the task's main function and interface function are not executed in parallel. There is another source of data hazard. This is parallel execution of the task's interface functions. To prevent this hazard, CDP requires the following.

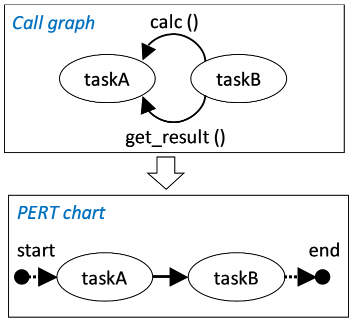

Figure 2: Conversion from call graph to PERT chart

Requirement 4. If interface functions of a task can be executed in parallel, these functions satisfy Bernstein's conditions.

We can determine the parallel executability of interface functions using the following procedure. Let graph $G =< V, A >$ be a PERT chart derived from a CDP program, and $G^+ =< V, A^+ >$ be the transitive closure of G. If two tasks u, w satisfy $< u, w >\notin A^+$ and $< w, u >\notin A^+$, u and w are likely to be scheduled in parallel. If there exists a task v such that $< v, u >\in A, < v, w >\in A$, interface functions of v called in u and ones called in w can be executed in parallel.

THEOREM 2. *(Bernstein's conditions[1])*
Let $R(f)$ be a set of internal variables which are referred in function f, and $W(f)$ be a set of internal variables which are updated in function f.
$R(f) \cap W(g) = \phi$, $R(g) \cap W(f) = \phi$, $W(f) \cap W(g) = \phi$.
If these three conditions are upheld, parallel execution of functions f and g does not cause any data hazard. □

If a program does not satisfy Requirement 3 and 4, the program has static semantic errors. A CDP program has the following property.

PROPERTY 1. *A CDP program is data hazard free if tasks in the program are scheduled to satisfy the precedence constraint given by the PERT chart of the program.* □

This is obvious because module scope rule protects illegal access to internal variables. In CDP, data hazard is prevented by static semantics checks. This is possible because there is no repetition in tasks.

2.3 Scheduling

The last step of CDP is making the schedule of tasks that satisfy precedence constraints. A PERT chart used explaining the scheduling method is shown in Figure 3.

To represent a schedule, we usually assign start times of tasks. If execution times of tasks are known and constant, this method is easy to understand. However, execution times of tasks are neither fixed nor known because these

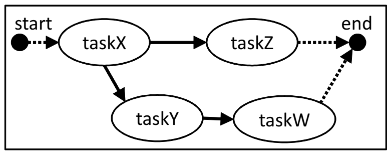

Figure 3: Sample PERT chart

varies with execution environments. In CDP, we represent a schedule using sequential composition. To create a sequential schedule, we use the topological sort of the PERT chart. We can create three schedules from Figure 3.

```
S1 = taskX_CLK(); taskY_CLK(); taskW_CLK(); taskZ_CLK();
S2 = taskX_CLK(); taskY_CLK(); taskZ_CLK(); taskW_CLK();
S3 = taskX_CLK(); taskZ_CLK(); taskY_CLK(); taskW_CLK();
```

The selection of schedules does not affect execution result of the program because of Property 1. Listing 3 shows the timer event handler and the main function. This sample assumes Windows OS. `OnTimer()` is the timer handler that executes the schedule S1 (Line 3). In the main function, we initialize tasks (Line 13, 14), then start timer (Line 19). In this program, the timer period is 100 ms (Line 10, 19). This program terminate when the event `end` is set (Line 20).

Listing 3: Timer handler and main function

```
 1  static Handle end;
 2
 3  void OnTimer () {
 4     /* Schedule S1 */
 5     taskX_CLK (); taskY_CLK ();
 6     taskW_CLK (); taskZ_CLK ();
 7  }
 8
 9  int main () {
10     unsigned int period = 100; /* Unit: ms */
11     MSG msg;
12
13     taskX_RST (); taskY_RST ();
14     taskW_RST (); taskZ_RST ();
15
16     /* Create Events end */
17     end = CreateEvent (NULL, FALSE, FALSE, NULL);
18
19     SetTimer (NULL, 0, period, OnTimer);
20     while (WaitForSingleObject(end, 0) == WAIT_TIMEOUT) {
21        GetMessage (&msg, NULL, 0, 0);
22        DispatchMessage (&msg);
23     }
24  }
```

There is no precedence relation between `task_clk(i)` in an arrayed task. Therefore, we can make a schedule of `task_clk(i)` in any order.

Context switching of multitask OS may cause random delay to the task's execution. Such delay may cause a critical error if the schedule is represented based on time. However, our schedule representation does not depend on the task's execution time. Random delay extends the completion time of the schedule, but the precedence relation between tasks are kept. Therefore, a CDP program works well under multitask OS.

3. PERFORMANCE OF CDP PROGRAMS

This section defines the throughput of a task, and derives the theoretical throughput formula. In general, throughput is represented by transactions per sec. First, we define transaction for a task.

Definition 2. (Transaction) Transaction is a series of state transition which satisfy the following three conditions. First, it starts a state transition from an accepting state to a non-accepting one. Second, it ends at a state transition from a non-accepting state to an accepting one. Third, it does not contain an accepting state in the middle. □

The following is a series of state transition of taskA in section 2. The series of state transition in boldface is a transaction.

$$free \xrightarrow{CLK/-} \mathbf{free} \xrightarrow{calc(a,5)=0/-} \mathbf{calc1} \xrightarrow{CLK/F(a)=c} \mathbf{calc2}$$
$$\xrightarrow{CLK/-} \mathbf{fin} \xrightarrow{get\_result(\&z)=0/-} \mathbf{free} \xrightarrow{CLK/-} free$$

Definition 3. (Clocks per transaction: CPT) The CPT of a transaction is the number of `CLK` contained in the transaction. □

We derive the theoretical throughput formula. If all CLK belongs to transactions, the throughput of a task becomes the maximum. In this case, the throughput is denoted by $1/(cT)$, where c is the CPT and T is the time period. We can increase the throughput using arrayed task. If the number of elements of the array is m, the throughput becomes $m/(cT)$. In order to preserve the timer period, duty ratio D, the ratio of the task's execution time to the timer period, must be kept to less than 1. The reciprocal of T is the frequency f. From the above, the maximum throughput of a task P is obtained using the following formula.

$$P \;=\; \frac{m}{c} f \qquad (if\ D < 1) \tag{1}$$

The performance variance of execution environments does not affect throughput of a CDP program. It affects duty ratio. The duty ratio is proportional to the frequency f. Therefore, if the frequency f is chosen appropriately, we can assure throughput of the program irrespective of execution environments.

4. RELATED WORKS AND CONCLUSION

The predecessor of the clock-driven programming is a programming method that has been used in real-time systems [4]. The main progress of CDP is introducing Requriment 4, Bernstein's conditions, to prevent non-determinism caused by schedule selection.

Clock driven programming was defined and the theoretical throughput formula was derived. A CDP program shows the same throughput even if it runs on different execution environments when its timer period is the same. On the other hand, an adaptation method is required in order to change performance according to environment. The dynamic control of timer period enables such adaptation, but needs to further study.

5. ACKNOWLEDGMENTS

This work was partially supported by Ministry of Internal Affairs and Communications of the Japanese Government.

6. REFERENCES

[1] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, (5):757–763, 1966.
[2] C. Ghezzi. The challenges of open-world software. In *Proc. of WOSP '07*.
[3] D. Harel. On folk theorems. *CACM*, 23:379–389, July 1980.
[4] J. W. Liu. *Real-time systems*. Prentice Hall, 2000.