An Improved Quasi-static Scheduling Algorithm for Mixed Data-Control Embedded Software

EE 249 project report

Cong Liu and Xuanming Dong

12/10/2001

Abstract: An embedded system is defined as a set of concurrent processes that communicate through channels. Described in flow C, each process is a sequential program that may contain data-dependent or synchronization-dependant control structures. A task is a set of sequential operations that the system will perform to respond to the inputs from the environment. The embedded software coordinates these tasks generated for input events. A software synthesis approach includes two correlated parts: scheduling and code generation. Our scheduling algorithm can guarantee that all tasks can be executed within finite memory for arbitrary input streams, and the code generated can have a smaller size than those previously used. Petri net (PN) is used the underlying model of computation (MoC) to do formal analysis and verification.

1. Introduction

A reactive embedded system must respond to the inputs from the environment at the speed and with the delay dictated by the environment. The embedded software coordinates system operations. Scheduling of these operations is subject to satisfy hard time constrains, and make most use of the system resources (minimize CPU idle time).

This report is organized as follow: in Section 2, system definitions, background knowledge, and underlying models are given; in Section 3, the detail about the algorithm is presented; in Section 4, some experiments and results are offered; in Section 5, some open topics are illustrated for future research directions.

2. System, Processes, and Tasks

We define an embedded **system** as a set of concurrent processes. A set of input and out-put **ports** are defined for each process, and point-to-point communication between processes occurs through uni-directional **channels** between ports. We support multi-rate communication, in which the number of objects read or written by a process at any given time may be an arbitrary constant. A process may communicate with the environment in which the system is executed. This is done through input and output ports for which no channel is defined. Such primary input ports can belong to one of two classes, which we call **controllable** and **uncontrollable**. The latter is used to trigger an execution of the system, i.e. when the system receives an object at an uncontrollable port, it reacts to the environment by performing operations. On the other hand, the system may request the environment for further inputs through controllable ports, while this is not allowed for uncontrollable ports. Output ports are always written under system control, and the environment must be ready to accept them at any time (as allowed by the concurrent process specification). We restrict our attention to processes described as sequential programs and whose implementation is mapped as software to be executed on a programmable processor. The sequential program for each process is specified in **Flow C**, which is based on C, but extended to specify communication operations. It may contain both **data-dependent** control statements (e.g., if-then-else or arbitrary while or for loops) and **synchronization-dependant** control statements (e.g. SELECT). A **task** is generated for each input uncontrollable port, which performs the operations that required to react to an event of that port.

2.1 Static, quasi- static, and dynamic scheduling

Static scheduling does most of the work at compile time, so the resulting software behavior is highly predictable, and the overhead due to task context switching is minimized. They may also achieve very high

CPU utilization if the rate of arrival of inputs to be processed from the environment has predictable regular rates that are reasonably known at compile time. Static scheduling, however, is limited to specifications without runtime choice (called Marked Graphs or Static Dataflow [2]).

Researchers have recently started looking into ways of computing a static execution order for operations as much as possible, while leaving data or synchronization-dependant choices at runtime. This body of work is known as **Quasi Static Scheduling** (QSS).

Dynamic scheduling is commonly used for a system which contains real-time controls, and the run time behavior is heavily dependent on the occurrence of external events.

2.2 Petri Net

A **Petri net** (PN) is defined by a tuple of (P,T,F,M_0), where P, T are sets of **places**, and t**ransitions** respectively. F is a function from $(T \times P)$ $(P \times T)$ to non-negative integers. A **marking** is a function from P to non-negative integers, where its output for a place is denoted by M[p], which we call the number of tokens at p in M. If M[P] is positive, the place is said to be marked at M. M_0 is a marking, which we refer to as the initial marking. A Petri net can be represented by a directed bipartite graph, where an edge [u,v] exists if F([u,v]) is positive, which is called the **weight** of the edge. We may call v a **successor** of u and u a **predecessor** of v, respectively. A transition is enabled at a marking M[p], if $M[p] \geq F([p,t])$, for all p of P. In this case, one may fire the transition at the marking, which yields a marking M' given by: $M'[p] = M[p] - F(p,t) + F(t,p)$, for each p of P. In the sequel, $M \xrightarrow{t} M'$ denotes the fact that a transition t is enabled at a marking M and M' is obtained by firing t at M. A sequence of transitions $(t_1,t_2,\ldots t_k)$is said to be **fireable** from a marking M, if there exists a sequence of markings $(M_1, M_2,\ldots M_k)$ such that $M_1 = M$ and $M_i \xrightarrow{t_i} M_{i+1}$' holds for each I=1,2,…k. A transition is said to be a **source**, if F(p,t) = 0 for all p of P.

A marking M' is said to be **reachable** from M, if there is a sequence of transitions fireable from M that leads to M'. The set of markings reachable from the initial marking is denoted by $R(M_0)$. The **reachability graph** of a Petri net is a directed graph in which $R(M_0)$ is the set of nodes and each edge [M,M'] is a transition t with $M \xrightarrow{t} M'$. The reachability tree of a Petri net is a tree in which each node is labeled with a marking of $R(M_0)$, the root node is labeled with M_0, and each edge $[v,v']_t$represents a transition with $M \rightarrow M'$.

A key notion we use in Petri nets for defining schedules is equal conflict sets. A pair of non-source transitions t_i and t_j is said to be in **equal conflict**, if $F([p,t_i]) = F([p,t_j])$, for all p of P. These transitions are in conflict in the sense that t_i is enabled at a given marking, if and only if t_j is enabled, i.e. if the firing of one transition disables t_i, it also disables t_j. The equal conflict is an equivalence relation defined on the set of non-source transitions, and each equivalence class is called **equal conflict set (ECS).** As a special case, we also define as an ECS a set that consists of a single source transition. By definition, if one transition of an ECS is enabled at a given marking, all the other transitions of the ECS are also enabled. Thus, we may say that this ECS is **enabled** at the marking.

A place is said to be a **choice** place if it has more than one successor transition. A choice place is **Equal Choice** (a generalization of free choice [4]) if all the successor transitions are in the same ECS. A Petri net is Equal Choice if all choice places are equal. A choice place is **unique** if no more than one successor transition can be enabled in any of the markings of $R(M_0)$. A **unique-choice Petri net** (UCPN) is that in which all choice places are either equal or unique.

2.3 Overview

A system function is represented as a network of processes. A process is specified as a sequential programwritten in **FlowC**. The network of processes is transformed into a single Petri net, which is built in two steps. In the first step, called **compilation**, a Petri net for each process is constructed and each port is associated to a place of the Petri net. The second step, called **linking**, builds a **Petri net** by "connecting" the Petri nets according to the defined channels. The **scheduling** algorithm takes as input a Petri net and an uncontrollable source transition for which a schedule is sought. The goal of **code generation** is to synthesize a sequential program to be run on a microprocessor which implements the system behavior, based on the schedule computed with the algorithm

3. Scheduling algorithm

A task is generated for each uncontrollable input port, and thus we compute a schedule for each uncontrollable source transition. A schedule for a given uncontrollable source transition is a directed graph.

3.1 Definition and properties of schedules

A **schedule** is a representation of all possible execution flows of the tasks that can be executed with finite memory for arbitrary input streams. It has five properties:

1) Exactly one node (vertex) for M_0
2) The set of transitions associated with the outgoing edges of node v is an ECS enable at M(v)
3) $M(v) \xrightarrow{t\,[v,w]} M(w)$ holds, for each edge [v,w]
4) Each node has at least one path to an **await node**
5) Each await node is on at least one **cycle**

3.2 Scheduling algorithm

The basic steps of the algorithm are given below.

1) Create the root r and set $M(r) = M_0$
2) Create a node v and an edge [r, v]
3) Associate the transition t1 to the edge, and a marking with v, s.t. $M(r) \xrightarrow{t\,[r,v]} M(v)$ holds
4) Call function EP (v,r), if return r, schedule found, then call post-processing and terminate
5) Termination condition: the irrelevance criterion

```
function EP(v, target)
  if(termination conditions hold) return UNDEF;
  if(∃u : u < v and M(u) = M(v)) return u;
  EP ← UNDEF, ECS(v) ← φ;
  for(each ECS E enabled at M(v))
    EP_ECS ← EP_ECS(E, v, target);
    if(EP_ECS ≤ target)
      ECS(v) ← E, return EP_ECS;
    if(EP = UNDEF or EP_ECS < EP)
      ECS(v) ← E, EP ← EP_ECS;
  return EP;
```

```
function EP_ECS(E, v, target)
  EP_ECS ← UNDEF, current_target ← target;
  for(each transition t of E)
    create a node w and an edge [v, w];
    T([v, w]) ← t;
    M(w) ← the marking obtained by firing t at M(v);
    EP ← EP(w, current_target);
    if(EP = UNDEF or v < EP) return UNDEF;
    EP_ECS ← min(EP_ECS, EP);
    if(EP_ECS ≤ target) current_target ← v;
  return EP_ECS;
```

Two basic functions are given above.

3.3 Heuristics using T-invariants

To find an entry point of a node, the algorithm may explore all possible ECS's at the node until it finds a desired one. The number of nodes created in the algorithm depends on the order of ECS's explored. Although the ordering does not influence the worst-case search space or run time of the algorithm, some orderings help finding a desired entry point sooner than others. In this section, we present a heuristic approach of sorting the ECS's for this purpose. The heuristic helps keeping the resulting schedules small. It also provides us with a sufficient non-schedulability condition, and if the condition holds, we can immediately terminate the procedure, reporting no schedule.

The heuristic method tries to find a short sequence of transitions such that if the sequence is fired from the current node, a marking associated with some ancestor of the node can be obtained. such an ancestor becomes a candidate of an entry point returned by the function EP for the node.We use t-invariants in finding such a sequence. A **t-invariant** is a vector of non-negative integers that solves the system of homogeneous marking equations $C \cdot X = 0$, where $C = [C_{i,j}]$, $C_{i,j} = F(t_j, p_i) - F(p_i, t_j)$ .C is the **incidence matrix** of the

Petri net. A t-invariant represents a set of sequences in which the number of occurrences of the ith transition is given by the integer at the ith position of the t-invariant. We say that such a sequence is **contained** in the t-invariant. Each of such sequences has a property that if the sequence can be fired from a marking M, the marking obtained after the firing is also M. We call a non-negative basis of the homogeneous marking equations a base of t-invariants. Such a basis can be obtained by using a Smith Normal Form decomposition. It is known that a schedule does not exist if there is no base of t-invariants. Therefore, if the algorithm identifies this case, it terminates immediately without applying the function EP. If a **base of t-invariants** is found, using the base and a sequence of transitions associated with the path from the root to the current node, the heuristic computes a vector of non-negative integers called the **promising vector** in EP. The positions of the vector correspond to the transitions, and those with positive integers represent transitions to be fired from the current node. The ECS's are sorted using this vector.

If we have several enabled ECSs, which all have transitions appear in the t-invariants, then sorting t-invariants can help to find a better t-invariant as the promising vector which leads to a cycle. Our heuristic sorts t-invariants according to their "length". The "length" of a t-invariant is defined as the total number of firings in the t-invariant. We choose the "shortest" t-invariant, if possible. Because based on experience, it is found that a shorter t-invariant has a better chance to make a cycle and produce a small schedule, thus small code size.

Reusing firing sequence is also an approach to reduce the resulting code size, even though it may not reduce the size of the schedule. Our current algorithm only consider the cyclic firing sequence from and to await nodes, because we can find a necessary and sufficient condition of friability of a given firing sequence at await nodes. Generally, we only have a necessary condition. So first, we record all the cyclic firing sequences into a dictionary, then we come to an await node, we simply check the friability of "previously" fired cyclic firing sequences at that await node. If it is firable, we force the await node fire that sequence by give the associated ECS the highest "priority". If it is not firable, we simply use the t-invariant to choose the associated ECS to be fire. One thing about checking friability is that we check not just the friability of a specific cyclic firing sequence, but also the "parasitic" firing sequences associated to the cyclic firing sequence. This is because of the "atomicity" of ECS. It means that if a transition *t* appears in a schedule, then all the transition in the ECS of *t* will also appear in the schedule. Another consideration is that if we have several await nodes associated with the cyclic firing sequence, then we need to identify all the firable cyclic firing sequences. This is because if we have more than await node then several firable cyclic firing sequences may only have one cycle identified by the schedule.

4. Implementation and experiments

4.1 Introduction to QSS-1.2

The C source code of QSS-1.2 has been modified to implement the ideas proposed in this report. The entire software synthesis flow from specifications to synthesized tasks, which consists of compiler, linker, scheduler, and code generator, are implemented in QSS-1.2 in a set of tools. The complete software implementation of the system, including the real-time scheduling of tasks, is supposed to be managed by an embedded operating system.

QSS-1.2 software package consists mainly four components: FCC, PNL, QSS, and CGEN, as depicted in figure 1. FCC is a FlowC Compiler program. It accepts users' input FlowC program and generates an intermediate petri net representation. PNL is the Petri Net Linker program. It composes the intermediate representations generated by FlowC Compiler and ouput is a petri net file using hpn format. QSS is the Quasi Static Schedule program that finds a subtree of the reachability tree from a given Petri net. QSS output the subtree by using a hpn format and a dot format. CGEN is the Code Generator program that synthesizes code for a given schedule. CGEN output files with Poindexter C format, which is supported by Cadence VCC codesign environment. Therefore the synthesized code can be simulated and estimated in VCC. In addition, the QSS-1.2 software includes some other supporting tool modules, for example, the manipulation tool of hierarchical Petri nets, the T-invariance analysis tool, etc.

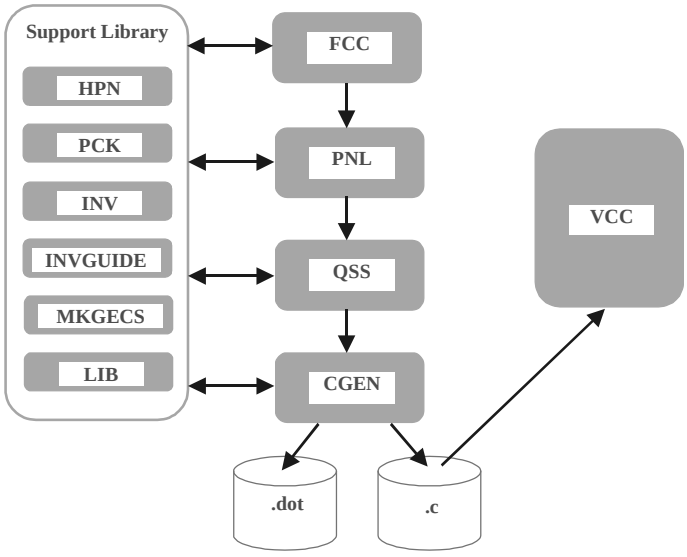

Figure 1: Structure of QSS-1.2

The simulation and verification of the synthesis code can be done in Cadence VCC codesign environment by importing the code into VCC.

4.2 Link List for Fire Sequences

In order to reuse previous successful fire sequences, a dynamic data structure is used to store the fire sequences. The linked list works very well in this case. Now the problem is how to store the fire sequences such that

- Fire sequences can be easily accessed and manipulated, for example, add a new sequence, compare two fire sequences, record the frequency of the fire sequences.
- The important context information about each transition should not be lost.
- In addition, we hope to reuse previous code as much as possible and keep the original structure of QSS-1.2.

After carefully consideration, we define each item in the fire sequences link list as following structure:

```
typedef struct fire_sequence    {
        char *sequence;         // string type of fire sequence
        lsList t_seq;           // transition type of fire sequence
        sm_row inv;             // one of T-invariance vectors
        mkg_t marking;          // the marking of the first node
        short frequency;        // frequency record when scanning the fire sequence
        short priority;         // priority
}       fire_sequence;
```

Actually the first field is the redundant information of second field. However, the separation between these two fields provides great advantages. The operations on fire sequences are classified into link list management function and fire sequence analysis function. The link list management function on fire sequences could be adding a fire sequence, deleting a fire sequence, retrieving a fire sequence, etc., which can be easily implemented through string manipulation. The fire sequence analysis function could be testing whether current node can find a cycle by current sequence, etc, which can be simplified through transitions.

4.3 Experiments and Results

The new implementation have been used to synthesis our demonstration example and examples given in the QSS-1.2 software package. We compare both the schedule size and code size for the system.

	Schedule size		C code Size	
Task name	QSS-1.2	New	QSS-1.2	New
Revised example	1766	963	*	*
Two-loops	2056	1762	2484	2369
Dag1	876	876	*	*
Dac00	3248	3248	3881	3881

* no code generated (the input is Petri Net not Flow C code)

Table 1: Experiment results for code size. All values are in bytes

The preliminary experiment results are presented in table 1. We find that both the schedule size and the code size generated by the new implementation are smaller than or at most equal to those generated by QSS-1.2 for some examples. It shows that our ideas can reduce the code size reasonably in some cases. However, it is only a very limited experiment. More examples have to be tested and more experiments should be done.

5. Conclusions and Future Work

In this report, we propose several ideas to optimize the single source task generation and compile-time scheduling for mixed data-control embedded system. Those ideas can be further formalized with appropriate theoretical analysis. Current implementation is just a preliminary step. It needs to be refined and cleaned. In addition, the experiments and measurement is fairly straightforward and simple with limited examples. More experiments will be done with different kind of examples. Furthermore, the synthesized code should be imported into VCC and functionally simulated to verify the correctness of the algorithm.

Acknowledgement

Dr Yosinori Watanabe proposed some ideas of this project and provided the QSS-1.2 source code. We would like to thank Drs Yosinori Watanabe and Alex Kondratyev for their great help.

References

[1] S. Bhattacharyya, P. Murthy, and E. Lee. “Software synthesis from dataflow graphs” *Kluwer Academic Press*, 1996.

[2] Cortadella, J.; Kondratyev, A.; Lavagno, L.; Massot, M.; Moral, S.; Fasserone, C.; Watanabe, Y.; Sangiovanni-Vincentelli “Task generation and compile-time scheduling for mixed data-control embedded software,” *Design Automation Conference*, 2000. Page(s): 489 –494

[3] T. Murata. “Petri nets: properties, analysis, and applications,” *Proceedings of the IEE*E, 74(4), April, 1989.

[4] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. “Synthesis of embedded software using free-choice Petri nets,” In *36th ACM/IEEE Design Automation Conferenc*e, June 1999.

[5] B. Lin. “Software synthesis of process-based concurrent programs,” in*35th ACM/IEEE Design Automation Conference,* June 1998.

[6] Ha, S. Lee, E.A. “Compile-time scheduling of dynamic constructs in dataflow program graphs,” *Computers, IEEE Transactions,* Volume: 46 Issue: 7 , July 1997, Page(s): 768 -778