Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems *

Pai Chou, Gaetano Borriello
Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

Abstract – Existing software scheduling techniques limit the functions that can be implemented in software to those with a restricted class of timing constraints, in particular those with a coarse-grained, uniform, periodic behavior. In practice, however, many systems change their I/O behavior in response to the inputs from the environment. This paper considers one such class of systems, called reactive real-time systems, where timing requirements can include sequencing, rate, and response time constraints. We present a static, non-preemptive, fine-grained software scheduling algorithm to meet these constraints. This algorithm is suitable for control-dominated embedded systems with hard real-time constraints, and is part of the core of a hardware/software co-synthesis system.

I Introduction

An embedded system is a special purpose computer consisting of one or more controllers and peripheral devices. A *reactive* system is an embedded system that changes its I/O behavior in response to inputs from the environment. This is in contrast to those systems with a uniform periodic behavior that is independent of their input. Many reactive systems must also meet hard timing constraints of various types imposed by the devices' protocols or by the required system behavior. These systems are referred to as *reactive real-time systems.*

Esterel [1] and StateCharts [4] have been used for specifying reactive systems. Reactive behavior can be succinctly and conveniently captured with parallelism and *watchdogs.* A watchdog is a wait-on-signal statement that encloses a statement block, and breaks control flow out of the block upon receiving the signal. Both Esterel and StateCharts assume an idealized timing model, where simple computations are assumed to take zero time to perform. However, lengthy computations that violate this assumption are extracted and treated as external signals, and timing constraints cannot be specified on them for scheduling. While this assumption simplifies semantics, it also restricts the class of applications that can be specified.

*This work was supported by NSF under Grant MIP-8858782 and DARPA under contract N00014-J-91-4041.

The watchdog-concurrency reactive programming model has been augmented with timing constraints in [2]. The system behavior is divided into a number of *modes.* A mode specifies a scope within which a set of timing constraints must be met, until one of the watchdogs detects an event and *disables*, or causes a transition out of, the mode. When such a transition is initiated, each concurrent branch to be disabled is scheduled to run until a *safe exit point* is reached. This enables interleaving while maintaining the integrity of I/O protocols and program state.

A timing constraint specifies a minimum or a maximum time separation between the *start times* of two operations. Elementary operations, such as reading and writing an I/O port or computations, take some *bounded* amount of time to execute and are not preemptable. Other operations, specifically polling loops that wait for an input value, can iterate indefinitely, and are said to have *unbounded delays.* When there are multiple paths between two operations, the maximum timing constraints are defined for those paths with only bounded-delay operations. We classify the constraints into sequencing, rate, and response time.

A *sequencing* constraint specifies the separation between the start times of two operations in the same mode and same iteration without intervening unbounded delay operations. Sequencing constraints are commonly found in I/O protocols of peripheral devices. These protocols consist of a sequence of read and write steps.

A *rate* constraint specifies the separation between the start times of two consecutive iterations of a loop. Note that the rate constraint holds only between successive iterations until the loop exits. A loop with a rate constraint can have a body with sequencing constraints. A maximum rate constraint is well posed only if the loop does not contain operations with unbounded delays.

A *response time* constraint is a constraint on a mode transition. The path is defined to be from the last iteration of the first mode to the first iteration of the next mode. Response time constraints are also referred to as *intermodal* constraints. Sequencing constraints are also called *intramodal* constraints. For the purpose of scheduling, a rate constraint can be formulated as an intramodal constraint.

This paper presents a static scheduling algorithm for producing a sequential program to meet both intramodal and intermodal timing constraints. Static scheduling is necessary because dynamic scheduling cannot guarantee that constraints will always be satisfied [7]. In the next sections, we formulate the scheduling problem in terms of a graph model and then present the scheduling algorithm.

II Scheduling Problem Formulation

The input to this problem is a constraint graph. It is an extended version of the constraint graph used in relative scheduling [6]. The vertices represent operations and the edges represent timing dependencies. Each graph is required to have a single entry point, or *anchor*. Each vertex v has an non-negative integer execution delay $\delta(v)$, shown in Fig. 2 as a value after the '/' in the node. In the basic problem, each graph corresponds to a mode and contains only bounded delay operations.

Timing constraints are represented by a set of directed edges. All edges have integer weights and are categorized as either *forward* edges (those with zero or positive weights) or *backward* edges (those with negative weights). A forward edge from vertex v to vertex w with weight $e_{v,w}$ indicates that the start time of w must be scheduled at least $e_{v,w}$ time units after v's scheduled start time. A backward edge from vertex w to vertex v with weight $e_{w,v}$ indicates that the start time of w must be scheduled no more than $-e_{w,v}$ time units after v's scheduled start time. We call the constraint graph limited to forward edges only the *forward constraint graph* and label it G_f. In all modes, all nodes are required to be reachable from the anchor, or start node, along a path in G_f.

The basic problem is defined as follows. Given a constraint graph G and an anchor vertex a, derive a valid serial schedule. A schedule is a mapping of the vertices to integers representing their start times relative to the anchor a. Serialization requires that operations be assigned nonoverlapping times. That is, if vertex v has duration $\delta(v)$ and is assigned start time $\sigma(v)$ then no other event is assigned a start time between $\sigma(v)$ and $\sigma(v) + \delta(v)$. A schedule is valid if it satisfies all the constraints. *Intramodal* constraints are satisfied if for all vertices v and w in G with edge $e_{v,w}$ between them, $\sigma(w) - \sigma(v) \geq e_{v,w}$.

In the extended scheduling problem, we must also consider the safe exit points, disables, and the intermodal constraints. A set of safe-exit points S include all leaves in G_f and any specified internal safe exit points. The set of disable nodes D is a subset of S. A *legal exit point* is a safe exit point whose peer branches in G_f are at their safe exit points. Formally, let $P(c)$ be the set of vertices scheduled before c (that is, $\{v : \sigma(v) < \sigma(c)\}$). A safe exit point c is legal if every $v \in P(c)$ is *safe* with respect to c and σ. A vertex v is safe with respect to c and σ if $v \in P(c)$ and either (i) v is a safe exit point, or (ii) all children of v in G_f are safe with respect to c and σ.

Intermodal constraint edges are similar to their intramodal counterparts. An intermodal edge (v^x, w^y) with a nonnegative weight e_{v^x,w^y} associated with the $x \rightarrow y$ transition requires that w^y be scheduled no earlier than e_{v^x,w^y} time units after v^x's scheduled start time. An intermodal edge (w^y, v^x) with a negative weight e_{w^y,v^x} associated with the $x \rightarrow y$ transition requires that w^y be scheduled no later than $-e_{w^y,v^x}$ time units after x^v's scheduled start time.

III Scheduling Algorithm

A Intramodal Scheduling

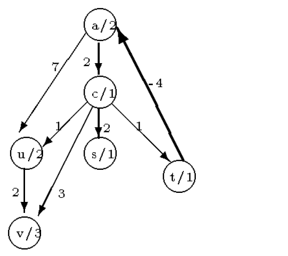
Although intramodal scheduling can be solved using serialization [5] and start time assignment [6], we present here a combined algorithm, that is adaptable to intermodal scheduling (Section B) and can be easily modified to use different heuristics. The input is an intramodal constraint graph, and the output is a schedule for the start times. The algorithm is shown in Fig. 1.

The algorithm is called with three parameters. The first parameter G is a modal constraint graph. The second parameter is the anchor a of the graph. Every vertex must be reachable from a in G along non-negative weight edges, as explained in section II. The third parameter c is the *current vertex* being traversed. It is initially set to a, and it separates the subgraph already serialized from the rest of the graph.

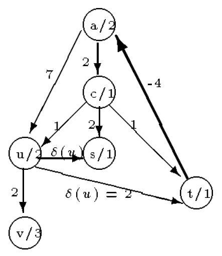
This algorithm performs a variation of topological traversal, starting from a. A vertex is a candidate to be serialized next if all of its predecessors in G_f have been serialized. If a vertex v is chosen from the candidate set to be visited after c, then a forward edge is added from v to all other successor candidates u of c. When adding a forward edge (v, u), we assign the edge weight $e_{vu} = \mathrm{Max}(\delta(v), L_a(u) - L_a(v))$, where L_a is the longest path length from the anchor a to the vertex, as computed by the BellmanFord longest paths algorithm. The justification for the edge weight is that since u is to be ordered

```
Serialize(Graph G, anchor a, candidate c) {
    $L_a$ := SINGLE SOURCE LONGEST PATHS($G$, $a$);
    if positive cycle found,
        return FAIL ;
    $C$ := topological successors of candidate $c$;
    if ($C$ is empty )
        return schedule with $\sigma(v)$ = longest path from $a$;
    $D := C$;
    while ($D$ not empty) {
        $v$ := SelectSuccessor($D$);
B:      foreach $u \in C - \{v\}$ {
                add edge $(v, u)$ to $G$, with weight
                $e_{vu} = \mathrm{Max}(\delta(v), L_a(u) - L_a(v))$;
                /* delay all successors by at least $\delta(v)$ */
        }
        Serialize($G$, $a$, $v$);
        if (schedule found) return schedule;
        /* else - positive cycle or backtrack */
        Undo step B;
    } /* while */
    return FAIL; /* no more candidates */
}
```

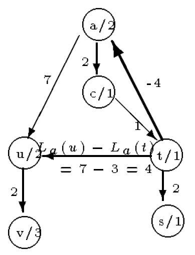
Fig. 1: Intramodal Scheduling Algorithm



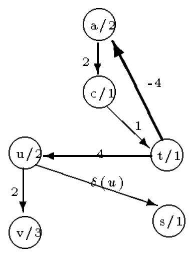
(a) Vertices a, c have been serialized. The successor candidates of c are $\{u, s, t\}$ but not v since v is a successor of u.



(b) Suppose we pick u to serialize next. We add an edge from u to its peers s and t, with the edge weight of $\delta(u) = 2$ in both cases. However, this results in a positive cycle (a, u, t).



(c) Suppose t is selected after c. We add edges (t, s) and (t, u), with edge weights 1 and 4, respectively. No positive cycle is formed. The successor candidates of t are $\{u, s\}$.



(d) Suppose u is chosen to be serialized after t. We add the edge (u, s) and no positive cycle is formed. The successor candidates of u are $\{v, s\}$. The graph can be completely serialized in one more step (not shown).

Fig. 2: Example of Intramodal Serialization

after v, u cannot start until after v completes, or after a longer minimum constraint from the anchor.

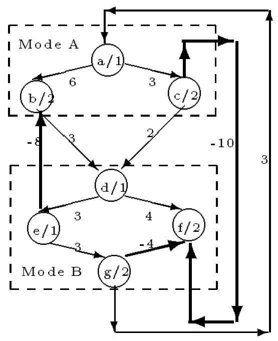
Note that the edge (v, u) could already exist, but it can only be a backward (negative weight) edge representing a maximum constraint from u to v. Since we order v before u, this maximum constraint is always satisfied, and therefore no information is lost by converting (v, u) into a forward edge.

A positive cycle in the constraint graph implies an infeasible constraint, since it requires a node to be scheduled later than its own start time. If the addition of new edges results in positive cycles, then the algorithm backtracks. The next candidate is considered, until a schedule is found or all candidates have been exhausted. Fig. 2 illustrates the algorithm.

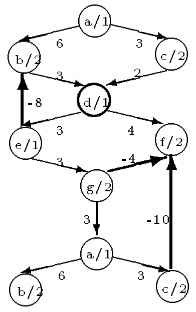
This algorithm is guaranteed to find a feasible ordering if one exists. At any level in the recursion, the algorithm cannot fail unless all possible orderings of the remaining unserialized nodes are infeasible. Since there is a feasible order, this will not happen. The correctness of the algorithm can be proved by induction, and is sketched as follows. Assuming a valid ordering exists, the basis is that the anchor a is correctly ordered. Inductive hypothesis is that everything from the anchor up to the current vertex c is in the correct order. Suppose v is the next vertex in a valid ordering, then $L_a(v)$ is exact as the longest forward path from a to v. A forward edge (v, u) is added for each peer u of v. The edge weight $\delta(v)$ is a necessary minimum constraint by definition of serialization, because no computations can overlap. If $L_a(u) - L_a(v) > \delta(v)$ then it is also a necessary constraint. The weight is exact if u is to be ordered immediately after v. Since the algorithm backtracks to try all possible vertices, it finds a solution if one exists.

Note, however, that in the worst case, an exponential number of orderings may be attempted. The complexity of this scheduling problem is NP-hard. There exists a simple transformation to this problem from the "Sequencing with Release Times and Deadlines" problem, which is NP-complete in the strong sense [3].

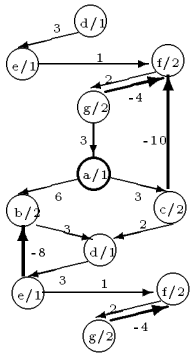
It is possible to substitute the `SelectSuccessor()` function (just above label B) with a heuristic function that selects vertices in a better order and considers the effects of choices on scheduling disables and safe exit points. To select a good candidate to serialize next, we use a "slack" function as a heuristic. Slack is a measure of how urgently a vertex should be serialized. Smaller slack implies higher priority. A heuristic for choosing operations on a path with a disable is to schedule the disable near the safe exit points such that the amount of code remaining before reaching the safe exit points is minimized. This code will need to be executed at the exit to the mode and impacts not only code size but more importantly, the ability to meet response time constraints.

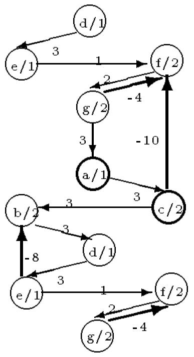


(a) An intermodal constraint graph. The response time edges associated with the A to B transition are $\{(b, d), (c, d), (e, b)\}$; those for the B to A transition are $\{(g, a), (c, f)\}$.



(b) Construct a graph with copies of mode A before and after mode B. Serialization starts on the anchor of B, which is the vertex d.





(c) Another graph is constructed, where the predecessor and successor of A are both the serialized version of B. The anchor for this serialization is a.

(d) After a complete ordering is obtained for mode A, and since mode B is also totally serialized, the intermodal constraints are sufficient for both serialized graphs. The final orderings are (a, c, b) and (d, e, f, g).

Fig. 3: Example of Intermodal Serialization

B Intermodal Scheduling

Intermodal scheduling, the scheduling of modes to meet intermodal constraints, can be viewed as an extension to the intramodal version. Instead of scheduling each mode in isolation, now we must also consider intermodal constraints.

Fig. 3 shows an example of our method of intermodal serialization. Since modes A and B alternate we serialize B by generating a graph consisting of two copies of A, and one of B, with one A before the B and one after. Additional precedence edges are added from all legal exit points of a preceding mode to the anchor of its successor to ensure that all of a mode's nodes are executed before control is passed to the next mode. After B is serialized, we repeat the process for A with two copies of the serialized version of B, one before and one after A. Should no feasible solution be found for serializing A, the algorithm backtracks to find a new feasible solution for B before retrying A. In addition, intermodal constraints can be relaxed by considering different legal exit points as more schedules for the various modes are completed.

IV Conclusion

Software synthesis is an emerging field in the automation of embedded system design. We specifically target the co-synthesis of embedded reactive real-time controllers, where the software is characterized by real-time constraints on control-dominated programs. Our focus is on low-cost systems that exploit microcontrollers or core processors and do not use an operating system to implement dynamic scheduling.

In this paper, we have presented an algorithm for software scheduling based on an extended model of timing constraint specifications as described in [2]. It is more general than earlier work in this area. The concept of safe exit points allows us to consider the effects of watchdog-style constraints used to describe reactive behavior. A new scheduling technique is guaranteed to find a static schedule that meets all the sequencing, rate, and response time constraints. The specification methods and scheduling algorithm are part of the Chinook hardware/software co-synthesis system currently under development at the University of Washington.

References

[1] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9), Sept. 1991.

[2] P. Chou, E. Walkup, and G. Borriello. Scheduling issues in the co-synthesis of reactive real-time systems. Technical report, Univ. of Washington, Dept. of Computer Science, Mar. 1994.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[4] D. Harel. StateCharts: a visual formalism for complex systems. *Science of Programming*, 8, 1987.

[5] D. C. Ku and G. De Micheli. Constrained conflict resolution and resource sharing in Hebe. *Integration – The VLSI Journal*, 12:131–165, Dec. 1991.

[6] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(6), June 1992.

[7] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, Jan. 1993.