
Scheduling for Embedded Real-Time Systems

FELICE BALARIN

Cadence Berkeley
Laboratories

LUCIANO LAVAGNO

Politecnico di Torino and
Cadence Berkeley
Laboratories

PRAVEEN MURTHY

Cadence Design Systems

**ALBERTO SANGIOVANNI-
VINCENTELLI**

University of California at
Berkeley

The authors review several approaches to control-oriented and dataflow-oriented software scheduling to determine whether a given technique can satisfy deadlines, throughput, and other constraints for embedded real-time systems.

REAL-TIME EMBEDDED SYSTEMS are often characterized by the need for running several tasks on a limited set of processing units. Scheduling these tasks on processors so that real-time constraints are met is a difficult problem. However, designers who have to face a difficult trade-off between efficiency and safety choices have several choices available to them.

We review some of these approaches and present the main techniques and their properties, depending on the characteristics of the system and of its environment. In particular, we provide well-developed preruntime (static) scheduling that offers very good optimization and trade-off possibilities for systems with regular input and output streams such as those found in digital signal processing applications. Our approach is very predictable and reliable, and so it can also be a valid choice for control-dominated applications with inputs irregularly distributed in time but with stringent safety requirements. The price to be paid in this case is in terms of processor usage and code size. Runtime (dynamic) scheduling, on the other hand, is the preferred (and sometimes the only) choice for control-dominated systems with very tight timing constraints.

We also review present research directions and point out the need for a more formal analysis of heuristically chosen

scheduling techniques commonly implemented in applications.

The problem

The scheduling problem consists of deciding the order and/or the execution time of a set of tasks with certain known characteristics (periodicity, duration) on a limited set of processing units. These units have a given capability (capacity, processing speed) and are subject to a set of constraints on the completion time of each task and on the use of the processing units. The scheduling problem is quite general and must be solved in many domains. In manufacturing, the tasks may be manufacturing operations and the processing units may be machines. In transportation, the tasks may be flights and the processing units may be airplanes.

We consider a specific subclass of scheduling problems: those that arise from scheduling software tasks on a single processor (or onto a limited number of processors for DSP applications) in reactive, real-time embedded computing applications. Most reactive embedded systems continuously react to environmental stimuli and must follow the speed of the environment. This is in contrast to interactive systems, such as a word-processing system that responds to the user (the environment) as fast as it can but with no hard-time constraints and with batch systems for

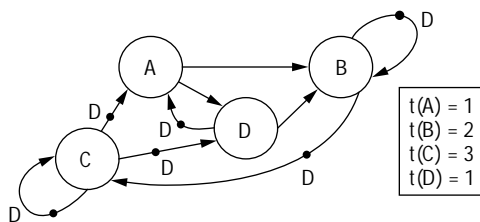


Figure 1. A dataflow graph. t = execution times.

which time is almost irrelevant. “Real time” is slightly more specific than “reactive” because it generally identifies a system composed of several distinct, often cooperating, tasks.

In addition, real-time systems are usually assumed to be nonterminating; the duration for which they operate is usually long enough that it is effectively infinite. Scheduling in this domain consists of finding an execution order for a set of mutually exclusive, sometimes suspendable tasks. These tasks are characterized by various parameters such as expected activation times (maybe periodic, maybe not), maximum required execution times, and deadlines by which they must be completed after activation. Since the programs are nonterminating, issues such as memory usage and deadlock avoidance also become very important. For example, a task might produce data at a faster rate than it is consumed, or circular dependencies might arise in which each task is waiting for another to finish, resulting in early program termination—an error in most cases.

Reactive systems. Designers of reactive systems must carefully resolve conflicts between different enabled tasks at runtime. (Enabled tasks have received enough inputs to start computing.) Resolving conflicts implies a nondeterministic runtime behavior and can lead to very subtle bugs that may manifest themselves a long time after system deployment.

DSP systems. Emphasis on throughput and the interdependence among tasks are the main differences between conventional real-time reactive scheduling problems and DSP scheduling problems. The throughput goal is to execute the DSP algorithm at a rate faster than the incoming sample rate.

Task interdependence in reactive real-time applications involves specifying task readiness externally by giving their frequency of occurrence or expected times. This models the temporal characteristics of the inputs coming from the environment. There may or may not be explicit information on the dependence of the task execution on another task. However in DSP applications the input arrival, or sample, rate is precise, and the completion of other tasks enables the tasks that follow. We specify this dependency as a

dataflow graph in which tasks to be executed, called actors, are nodes and directed edges specify the computation flow and thus the precedence among tasks. See Figure 1.

There are two broad areas in which scheduling problems arise in digital signal processing: high-level synthesis and general-purpose DSPs.

In high-level synthesis, the problem is to synthesize hardware that will perform the operations as specified in the dataflow graph. The scheduling problem is to map the dataflow graph, where tasks are atomic in the sense that they represent very elementary operations, like addition and multiplication, to a number of processing elements such as adders and multipliers. The optimization criteria of interest are the total number of processing elements, the throughput, and the total amount of memory usage (registers). These are all conflicting goals because to increase throughput, we might have to increase pipelining, and this increases the number of registers being used. Reducing the number of processing elements will reduce the chip area but might decrease the throughput because there is less computational power available. While this is an important problem and has been thoroughly investigated in the literature, it is outside the scope of this article.

With general-purpose DSPs, the sequence of operations specified by the dataflow graph may be complex, as in an FFT. These DSPs are quite popular for applications that do not require very high throughput rates such as audio signal processing. Traditionally, to achieve the best throughput and memory usage in DSPs, programmers used assembly language, a tedious and error-prone task at best. We could specify DSP programs in high-level languages (C or Fortran), but compilers for these languages have not been successful at producing optimized code that compares well with handwritten assembly language. Block-diagram languages have become more popular as a specification language because they

- are more intuitive than either assembly or high-level languages
- can be based on computation models well-suited for expressing DSP systems such as dataflow
- are much easier to parallelize than imperative languages like C or Fortran

One of the desirable features required of block diagram languages is efficient code generation. Scheduling plays a key role in meeting the various optimization criteria of throughput, code size, and buffer sizes. Because the computation models (for example, synchronous dataflow) impose restrictions on the overall control flow of the program, compilers for these block diagram languages can perform optimizations not usually possible for more general models or for imperative languages.

Control-dominated systems

We generally classify scheduling policies for real-time systems as

- *static, or preruntime*, when tasks execute in a fixed order determined offline. This order may or may not contain repetitions designed to cope with different expected activation times and/or deadlines.
- *dynamic, or runtime*, when the order of execution is decided online as the tasks present themselves to the processing units ready to be executed.

Generally, a dynamic execution policy is based on priority; that is, one among the set of ready tasks is dynamically chosen according to a priority order. (Priorities may be seen as additional information that helps in determining an execution policy that satisfies all the constraints.) Priority, intuitively, is a measure of the urgency of each task and can be determined either manually or automatically, in turn statically at compile time or dynamically at runtime. Moreover, dynamic scheduling can be preemptive if the currently executing task can be suspended when another task of higher priority becomes ready, and nonpreemptive otherwise.

Verifying whether a given scheduling satisfies all the deadlines (schedulability analysis) is an extremely hard problem, even when the execution times of all tasks are precisely known.

If a schedule satisfies the constraints, its quality often depends on the use of the processing units—the percentage of time spent by the processor when executing tasks. A processing unit may be poorly used either because it spends time in computing the schedule itself or because the schedule requires an execution overhead.

Static scheduling requires almost no CPU power at runtime, implying low overhead. (The time required to compute the schedule before execution may not be negligible but is required only once in a system's lifetime.) However, static scheduling requires that a task be executed whenever it is expected to be ready or at least tested for readiness cyclically. If enabling times are not predictable, too much CPU time is wasted in simply polling events that are unlikely to occur. Thus static scheduling best suits cases in which the time that tasks become enabled is well-known in advance. We discuss this later in the section on dataflow systems.

In general, dynamic scheduling may be necessary to better use the CPU. In fact, processor use that guarantees schedulability in the presence of irregular task activations is much higher for preemptive static priority scheduling¹ than for static scheduling. On the other hand, dynamically scheduled systems are much more difficult to tune and debug than statically scheduled systems. This is because the execution times and execution order are largely unpredictable.

These difficulties can be very serious problems for systems that require a high degree of safety and reliability. Indeed, the choice of a scheduling technique is often the result of a hard compromise between conflicting criteria!

Static scheduling. Probably the simplest and most popular scheduling approach is the round-robin method. Tasks are checked for readiness in a predetermined order, and the tasks that are found to be ready are immediately executed. Round-robin is easy to implement. It is also easy to prove at least some timing guarantees. Every task is checked for readiness once in a cycle, and thus the time between a request for execution and the corresponding execution can be easily bounded by the execution time of other tasks.

The problem with round-robin scheduling is that it provides poor service to urgent tasks. It is possible that even the most urgent task needs to wait for all other tasks to execute before it gets its turn. Thus to satisfy the timing constraints, a very fast processing unit may be necessary, which may not be available. Then round-robin would not produce a feasible schedule. In addition, if some tasks are rarely enabled, the overhead of checking them in every cycle may be significant compared with the actual execution time.

A slight improvement on round-robin scheduling is static cyclic scheduling. Here again tasks are checked for readiness in a predetermined order, and the tasks that are found to be ready are immediately executed. Tasks may appear more than once in the cycle. Hence, more urgent tasks may appear more often and be serviced more frequently, yielding better schedulability and processor use than with round-robin. However, unless the cycle is made very long (incurring a memory penalty), static cyclic scheduling may still suffer an overhead due to the frequent readiness checking of tasks that are rarely executed.

Designers also use static cyclic scheduling for dataflow systems. However, since in that case the inputs are regular data streams, there is no need to check a task for readiness. A task is (statically) scheduled to run only when its inputs are known to be present.

Dynamic scheduling. Static priority dynamic scheduling has received significant interest since the pioneering work of Liu and Layland.¹ In their approach, at any point in time the task with highest priority among the enabled tasks executes; that is, the scheduling follows a preemptive static priority scheme. Strong schedulability and processor use results have been proven. Unfortunately, the theoretical results are valid only when

- each task is assigned a fixed and unique priority. Each task is enabled when its execution is requested and disabled when it completes its execution.

The choice of a scheduling technique is often the result of a hard compromise between conflicting criteria!

- executions of each task are requested periodically, with a constant and known interval between requests. We call this interval a period and use P_i to denote the period of task i .
- tasks are independent. That is, requests for execution of a certain task do not depend on executions of other tasks.
- each task must be completed before the next request for it occurs. If a priority assignment is such that this is always true for any task, we say the priority assignment is feasible. A set of tasks is schedulable when a feasible priority assignment exists.
- each task has a constant and known runtime; that is, the processor time it requires for a single execution, assuming it is not interrupted. We use T_i to denote the runtime of task i .

Designers may assign priorities either by hand or let them be determined by an algorithm. For systems satisfying the above-mentioned assumptions, Liu and Layland proposed a particular algorithm for priority assignment, called rate monotonic scheduling. RMS is a static priority scheduling scheme that assigns priorities according to periods such that tasks with shorter periods get higher priorities.

Liu and Layland showed that RMS is optimal in the sense that if the RMS priority assignment is not feasible, a set of tasks is not schedulable. They also discovered the least upper bound on processor use in a static priority scheme. More precisely, if, for a set of n tasks, the processor use is less than $n(2^{1/n} - 1)$, that set of tasks is schedulable. (In particular, RMS priority assignment is feasible.)

Liu and Layland have also found an even stronger utilization result for a dynamic priority assignment policy called earliest deadline first (EDF). The basic assumption of EDF is that at any point in time the priority of an enabled task is not fixed; it depends on the time until the next request for that task. According to the EDF policy, the executing task must always have the least time remaining until the next request (or equivalently the earliest deadline) among all the enabled tasks. An EDF-executed set of tasks is schedulable if, and only if, its processor use is less than 1. Unfortunately, this ap-

parent improvement over RMS is often offset by the costly runtime overhead of EDF scheduling. Consequently, EDF is not widely used in embedded systems, despite its attractive theoretical properties.

Analysis. Liu and Layland's seminal work has had a wide impact on research in real-time computing and embedded systems. Yet, every assumption of their model is often violated to some extent in the design of embedded systems.

For many high-volume, low-cost embedded systems, task preemption is too expensive in time and space. The runtime penalty is due to the context switching overhead. The memory penalty is due to the unpredictability of stack requirements for storing states of preempted tasks. For these reasons, designers often use nonpreemptive schemes, even though elegant theoretical results do not extend easily to them.

In many systems, requests for task executions do not arrive in regular periods. Still, some constraint on the frequency of requests is usually known. It might be in the form of minimum and maximum times between two requests or in terms of a minimum and maximum number of requests in a given period of time.

Tasks are very rarely independent; much more often they are reactive. They are enabled by either events in the environment (and the same event might enable several tasks) or the execution of other tasks.

The correctness criteria of a task execution are often not quite clear. The designer usually has a reasonable set of requirements for the embedded system as a whole but finds it difficult to relate these overall constraints to requirements on individual tasks.

A task's runtime is almost never constant. It may vary with different input patterns as well as with the state of the task. Modern processor design techniques make things even worse. The runtime in a pipelined processor or a processor with memory caches may vary even for the same inputs and the same internal task state. Considering this lack of predictability and the cost pressure, it is not surprising that embedded systems often include simple processors based on designs that are over a decade old.

Beyond RMS and EDF. Fortunately, the static priority scheduling scheme is amenable to analysis even in more realistic models in which some of the assumptions of Liu and Layland have been relaxed. For example, Audsley et al.² proposed an optimal priority assignment algorithm for static priority scheduling that is valid for any model for which there exists a test satisfying the following conditions:

1. Given a priority assignment, it is possible to check whether some task i satisfies its timing constraints. The

pass or fail test results for task i do not change if two tasks that have a higher priority than i exchange their priorities. In other words, the test results should only depend on the tasks having higher priority than i and not on their exact priorities.

2. If some task i fails the test for some priority assignment in which j has lower priority than i , it also fails in the assignment in which i and j switch their priorities. In other words, if i fails the test, we cannot make it pass by lowering its priority.

Later we show an example of a such test, but at the moment let's just assume that such a test is available and focus on using it to devise a feasible priority assignment. Audsley's algorithm divides a set of tasks into two groups: those that have and have not already been assigned a priority. Similarly, it divides priorities into those that are already assigned and those that are still available. The algorithm always assigns the lowest available priority to any task that satisfies its timing constraint when assigned that priority.

The algorithm may terminate in two ways. It either assigns priorities to all tasks, in which case a feasible assignment has been found, or at some point the lowest available priority (say, p) cannot be assigned to any of the remaining tasks. This last event would occur because every one of the remaining tasks violates its timing constraints if assigned priority p . Then we may conclude that the set of tasks is not schedulable because in every priority assignment at least one of these tasks must have priority p or lower. The algorithm is thus optimal, in the sense that it finds a feasible priority assignment, if such an assignment exists. The number of tests required grows quadratically with the number of tasks. This is a vast improvement over the naive algorithm that checks all priority assignments and requires an exponential number of tests.

Schedulability analysis. We analyze schedulability to determine whether a set of tasks meets its timing constraints. One approach to this problem is to compute the worst-case response time (WCRT) of each task.³ A task's WCRT is the maximum possible length of an interval that begins with the task being enabled and ends with the task completing its execution. It includes both the task's runtime and interference from other tasks.

The WCRT concept is useful regardless of the scheduling approach. It is well defined both for cyclic and priority (static and dynamic) approaches as well as for preemptive and non-preemptive schemas. Once we compute WCRT, checking whether a task meets its deadline is a trivial comparison. Even in more complex models in which timing constraints cannot be expressed simply as deadlines on individual tasks, WCRTs are often very useful information in schedulability analysis.

We analyze schedulability to determine whether a set of tasks meets its timing constraints.

As an example, let's compute WCRTs in a simple case. This system satisfies the Liu and Layland assumptions of preemptive static priority scheduling, and independent periodic tasks with fixed runtimes. In general, the WCRT of a task with priority p is bounded by the length of the longest interval in which the processor is continuously busy, and no task of priority lower than p executes. Let B_p denote the maximum length of such an interval. If B_p is shorter than the period of the task with priority p , the WCRT of that task exactly equals B_p . We can compute B_p as follows:

1. Let initial estimate B_p^0 be the runtime sum of all tasks with priority p or higher, and set the iteration counter k to 0.
2. For each task i of priority p or higher, determine the number of times n_i that it can be enabled in the interval of length B_p^k . Let the new estimate B_p^{k+1} be the sum of $n_i T_i$ over all tasks i of priority p or higher.
3. If $B_p^k = B_p^{k+1}$, stop; B_p is B_p^k . Otherwise, increment k by 1, and go to step 2.

Analyzing a task set's schedulability in a realistic embedded system is not nearly as easy. The analysis needs to deal with varying—possibly state-dependent—runtimes, dependency between tasks, and nonperiodic events in the environment. In addition to these are nonpreemptable task sections, scheduling overhead, and other characteristics of real systems not reflected in the Liu and Layland model. Balarin and Sangiovanni-Vincentelli⁴ proposed a conservative extension of the WCRT analysis that designers can apply to systems in which

- scheduling is either preemptive or based on nonpreemptive static priority
- any number of events in the environment as well as the execution of other tasks can enable tasks
- events in the environment only need to respect the minimum time between two occurrences

However, not even this work addresses the problem of input- and state-dependent runtimes. Any existing method

***Esterel lets us describe
reactive tasks as concurrent
modules communicating
via signals.***

taking those into account suffers from the so-called state explosion problem: the need to examine a state space that grows exponentially with the number of tasks.

Synchronous scheduling

The synchronous approach to embedded system programming originated in France in the 1980s (Halbwachs⁵ provides a good overview). Synchronous languages have unambiguous semantics based on finite-state machines. The languages come with a complete set of programming aids: compilers to “standard” programming languages (for example, C) and formal verification and simulation analysis tools.

The basic assumption for synchronous languages is the so-called synchronous hypothesis.^{5,6} The hypothesis states that, given a set of communicating tasks with each represented by a finite-state machine, computation of next-state functions and of output functions as well as communication among the finite-state machines take exactly zero time.

Taken literally, this assumption is obviously false for physical systems. However, if we sample the system and the task inputs at discrete instants, the system behavior does not depend on the computation time, as long as this time is bounded by the difference between two consecutive input sampling instants. We can regard such time as zero without changing system behavior. In other words, if the reaction time of the system is much smaller than the time resolution (the time constants) of the environment, the synchronous hypothesis is realistic. If task implementation is not homogeneous (for example, some tasks are implemented as hardware and some as software), the synchronous hypothesis is not plausible, and we must use other techniques.

The synchronous hypothesis allows us to abstract away the internal structure of the embedded control software, replacing the user-defined cooperating task model with a single, monolithic reactive block. In finite-state machine lingo, the set of cooperating finite-state machines can be replaced by a single product machine. The interesting point is that this replacement is absolutely deterministic and easy to analyze.

In a sense, the “synchronous revolution” in reactive programming holds the same promise for boosting designer pro-

ductivity as the introduction of the clock in digital hardware design. It basically permits a clear separation of concerns between timing (the analysis of a single, loop-free piece of code, the next-state function, and the output function evaluation code) and functionality (the product of a set of finite-state machines). The structure of the product machine is equivalent to a kind of static scheduling of the component machines. So we can look at the formation of the product machine as a scheduling technique, the synchronous scheduling technique.

There are two points, though, in which the analogy unfortunately breaks down. First, synchronous scheduling can lead to a larger program size or to an increase in the execution time of a single reaction with respect to traditional asynchronous task-based scheduling. The size and complexity of the product machine is often larger than the size and complexity of the component machines.

Second, we can only schedule synchronously when the tasks are represented as extended finite-state machines. This problem is often not too serious, because most embedded system design methodologies generally recommend a finite-state specification style. Moreover, the synchronous skeleton is generally used as a coordination language for tasks that are black boxes. It can then be seen as a structured and powerful mechanism to describe functional dependencies between tasks.

These considerations imply that synchronous languages cannot be used for all cases but should be used with a clear understanding of the trade-offs involved in the final implementation. For example, the synchronous approach may pay off in the design of safety-critical systems. Then, deterministic response and debugging ease offset the increased costs and reduced performance.

The most interesting case of synchronous languages for embedded systems is Esterel, a language for control-dominated applications. Synchronous languages such as Lustre or Signal can also represent dataflow-dominated applications, but in this case the advantages over the approaches we describe in a later section are less clear. In particular, Lustre and Signal require the designer to statically determine buffering rather than its being computed as part of the scheduling task.

Esterel lets us describe reactive tasks as concurrent modules communicating via signals. The module notion serves the sole purpose of a structuring mechanism for readability because the Esterel compiler merges all modules into one extended finite-state machine.^{5,6}

As mentioned earlier, one of the basic notions common to all synchronous languages is that time is divided into a discrete sequence of instants in which something interesting happens and idle intervals. Time in an Esterel program can elapse only when executing a Halt statement. Otherwise, as

soon as something happens, the Esterel program is synchronously executed until the next Halt.

An Esterel program communicates with the environment via a set of signals, which may or may not carry a value. A signal with a value can be a temperature sample or the fact that a user has hit a key on a keyboard. Valueless, or pure, signals can indicate when a period of time (a millisecond, for example) has elapsed or when the pressure in a chamber falls below a threshold.

At any instant, a mechanism that is outside the scope of Esterel determines the presence of a subset of the input signals to the program. Note, however, that synchronous programming is akin to static scheduling and closely related to the polling of external inputs at the beginning of each instant.

Esterel includes several constructs to represent useful notions in reactive real-time programming, such as preemption, watchdogs, exceptions, and so on. For example, the statement “do sub-statement Watching signal” instantaneously preempts “sub-statement,” without allowing it to do anything for the current instant. Assume that “msec” is an input signal present once every millisecond and that the following code describes concisely and explicitly a time-out mechanism:

```

...
do halt watching msec; % synchronize
  with msec
do
    some-long-task % execute unless
    preempted
watching msec;
present msec then      % check if
    preempted
    recover-from-timeout
end
...

```

The mechanism may interrupt a task if it does not complete within one millisecond. (The Present statement is exactly like an If statement in classical programming languages but is used to detect a signal’s presence in an instant.) Note that in synchronous programming we don’t need to worry about an occurrence of “msec” after “some-long-task” has finished but before “present msec” is checked. The two events occur exactly in the same instant, that is atomically. The first use of “do ... watching,” to kill a Halt statement is so common that it has the shorthand version “await msec.”

There is one possible problem. A synchronous module is equivalent to a Mealy type of extended finite-state machine in which outputs depend on the inputs in the same instant. This obviously can cause problems due to undelayed feedback loops, which are at the core of the Esterel compilation problem.

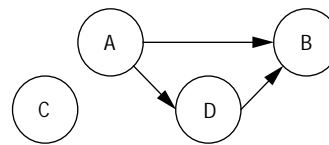


Figure 2. APG for graph in Figure 1.

There is an intrinsic difficulty of statically detecting the absence of loops that do not permit the generation of an extended finite-state machine and thus the solution of the scheduling problem. That is one of the reasons why synchronous scheduling is not generally applicable to all scheduling problems.

Dataflow systems

For scheduling purposes, designers often use dataflow graphs to describe DSP applications. Each actor in a dataflow graph corresponds to a computation, and each edge corresponds to a communication channel between the actors it connects. An edge such as (A, B) imposes a precedence constraint: Actor B may not execute until actor A has finished executing. Each time an actor fires (executes), it produces one token on each of its output edges and consumes one token from each of its input edges. Edges have buffers that hold data tokens, and the buffers are implemented as FIFO queues. Each graph edge is also annotated with a number that represents the number of initial tokens on that edge.

Formally, a dataflow graph G is a directed graph $G = (V, E, d)$, where V is the set of actors, E is the set of edges, and d is the function that assigns to each edge the initial number of tokens called delays. We refer to this dataflow model as homogenous synchronous dataflow (HSDF). The actors and edges may have other parameters as well; for example, communication time, number of data items sent, and so on. A common parameter needed is the execution time of each actor; this is modeled by a function t that assigns each actor an execution time.

In a slightly more general dataflow model, each actor consumes and produces a fixed number of tokens (where these numbers can be greater than one). This model is called synchronous dataflow (SDF). A key SDF strength is its amenability to static scheduling; this means that we can construct a schedule at compile time and not have to make decisions (about which actor to fire) at runtime. Dataflow models in which this property does not hold are sometimes referred to as dynamic dataflow or control dataflow. These models are technically more powerful than SDF since it is possible to simulate Turing machines with them. Static scheduling is not usually possible, and some runtime decision making is required. We can model most DSP algorithms using SDF graphs since DSP

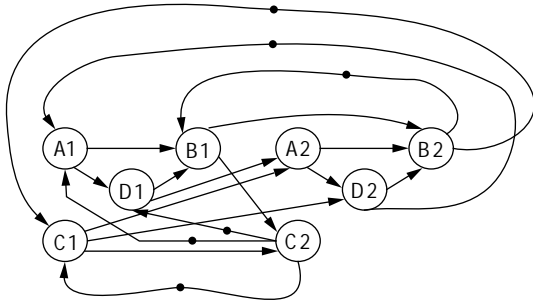


Figure 3. The 2-unfolded graph.

algorithms typically do not require much decision making. A valid schedule for an HSDF graph is a firing order for the actors, possibly on different processors, so that all precedence constraints are met and the graph returns to the original state: the number of tokens residing on each edge. Each actor in the graph appears once in the schedule. This schedule is repeated in an infinite loop since, as mentioned earlier, DSP programs do not generally terminate and run indefinitely, at least until the power fails! Hence, the k th schedule repetition is called the k th iteration.

As an example, consider the dataflow graph shown earlier in Figure 1. A delay on an edge such as (B, C) imposes an interiteration precedence constraint because the data produced by B is not needed by C until the next iteration. In contrast, the absence of a delay on edge (A, B) means that B will consume the data token produced by A in the same iteration. This idea is represented by the graph in Figure 2 called the acyclic precedence graph (APG). This graph shows only the intra-iteration precedences, and we see that in any iteration, actor C can be executed independently of actors A , D , and B . We obtain the APG by removing all edges that have one or more delays.

We can unfold the graph in Figure 1 to see the precedence constraints over multiple iterations. Figure 3 shows the unfolded graph of blocking factor 2. This 2-unfolded graph shows the precedence constraints over two iterations. We use the notation $A1, A2$, etc. to mean the first execution of actor A , the second execution of A , and so on. The executions are now divided conceptually into even- and odd-numbered iterations. The third iteration of actor A is the second iteration of actor $A1$, and the fourth iteration is the second iteration of $A2$ in the 2-unfolded graph, and so on. Since the second iteration of C feeds data to the third iteration of A , we model it by a feedback edge with a delay from $C2$ to $A1$ in the 2-unfolded graph. Similarly, we obtain the APG for the 2-unfolded graph, as shown in Figure 4a.

Throughput. If the dataflow graph does not have cycles, and we have an infinite number of processing elements, there is no theoretical limitation on how fast the graph can

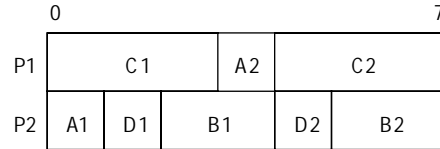
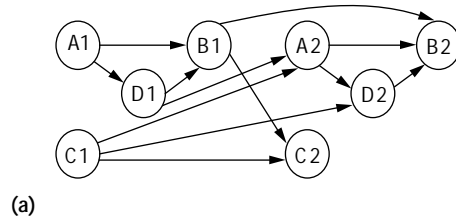


Figure 4. The APG for the 2-unfolded graph (a) and a two-processor schedule (b).

be executed. After all, we could simply execute every iteration of the graph at the same time, thereby achieving a throughput of infinity. However, the presence of resource constraints, and of cycles, limits how fast a dataflow graph can be executed.

Suppose that we have an acyclic graph but a limited number of processors. Then determining the maximum throughput we can achieve is an NP-hard (intractable) problem. Conversely, assume that we have an unlimited number of processors, but there are cycles in the graph. Then there is a fundamental upper bound on the throughput achievable that is given by the inverse of the iteration period bound (IPB):⁷

$$\lambda = \text{MAX}_{l \in \Lambda} [T(l)/D(l)] \quad (1)$$

Here, Λ is the set of all cycles in the graph, $T(l)$ is the total computation time of cycle l , and $D(l)$ is the delay count of cycle l . A cycle that achieves this maximum is called a critical cycle. We call a schedule for a dataflow graph rate optimal if the iteration period for the schedule equals the IPB. We can compute quantity λ in polynomial time. For the example in Figure 1, λ is equal to 3.5, and the critical cycles are $ABCD A$ and $ADBC A$.

In general, we speak of two types of scheduling problems:

- Fixed throughput, where the required iteration period is given and the objective is to minimize the number of processors required.
- Fixed resources, where the number of processors is specified and the objective is to devise a schedule with maximum throughput.

Most forms of these constrained multiprocessor scheduling

problems are NP-hard; so we must resort to heuristics in general.

Scheduling techniques. There are two main ways of tackling these scheduling problems. The first method is to design heuristics derived by some insight into the particular scheduling problem. Well-known heuristics include list scheduling, force-directed scheduling, cyclo-static scheduling, and various forms of software pipelining. A second approach is to cast the scheduling problem as an integer linear programming (ILP) problem and use widely available ILP solvers to obtain a solution. A benefit of this approach is that it is quite general, gives exact results, and can incorporate many different types of constraints easily. On the other hand, ILP is itself an intractable problem: solvers can require inordinate amounts of time. Also, the actors may be large-grained—that is, have execution times of more than one or two, which is typical for fine-grained actors. Then, the size of the ILP formulation itself is exponentially bigger, multiplying the disadvantage of this approach.

Nonoverlapped scheduling techniques. Let's first consider the problem of scheduling a dataflow graph, possibly with cycles, onto an unlimited number of processors. Assume that interprocessor communication costs are zero. As we have seen, the throughput bound limits the fastest rate at which we can execute the graph.

An obvious first strategy would be to use classical multi-processor scheduling heuristics such as list scheduling (reviewed later) on the APG. The schedule that we derive for the APG can be executed in an infinite loop, giving us a schedule for the dataflow graph.

The minimum amount of time required to execute all actors in the APG is clearly given by the critical path: the path in the graph that has the largest execution time. For example, the APG in Figure 2 has a critical path of time $4(A \rightarrow D \rightarrow B)$, and a schedule on two processors with an execution rate of $1/4$ is depicted in Figure 5. This rate is smaller than the $1/3.5$ bound we calculated in Equation 1 and is not optimal. Clearly, we won't get a better schedule if we restrict ourselves to the APG because we are ignoring all of the interiteration parallelism that is present.

One way of exploiting interiteration parallelism is to unfold the graph. Consider the APG for the 2-unfolded graph in Figure 4b; its critical path has a time of 7 units ($A1 \rightarrow D1 \rightarrow B1 \rightarrow C2$). The two-processor schedule shown in Figure 4a achieves a $1/3.5$ execution rate since there are two invocations of each actor every 7 time units. We could do this because we were able to use up the idle time from 3 to 4 in the schedule of Figure 5. This allowed more efficient resource usage. The total schedule length for an APG is called the makespan of the schedule, and the iteration period of the in-

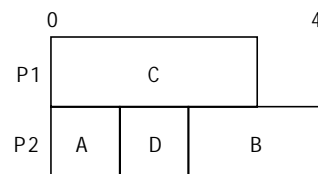


Figure 5. A 2-processor, nonoverlapped schedule.

finitely repeated schedule equals the makespan in this case.

Unfortunately, it isn't possible to obtain a rate-optimal schedule by unfolding the graph more and more. We know of graphs for which the length of the critical path divided by the unfolding factor approaches the iteration period bound asymptotically but never equals it.⁸

The nonoverlapped scheduling approach is so named because the interiteration parallelism is exploited only within a finite number of iterations, not all of them simultaneously.

List scheduling. This is one of the earliest scheduling techniques developed in the operations research community for nonoverlapped scheduling when there are resource constraints. The algorithm maintains a list of nodes that can be scheduled at each step on a particular type of processing element. A priority function breaks ties and chooses a task out of the schedulable tasks at each step. The simplest priority function is the weight of the longest path from the node to the sink node; the larger this weight is, the earlier the task must be scheduled to avoid holding up the succeeding tasks. If there is only one type of processing element, the graph is a tree and each task has unit execution time. Then this algorithm yields optimal schedules.

To minimize resource usage under a makespan constraint, the list scheduling algorithm uses each actor's slack time: this is the difference between the ALAP time (the latest possible time a node can be invoked) and the current scheduling time step. If the slack time is zero, the actor must be scheduled immediately or the makespan bound will be violated. At each time step, the algorithm proceeds by scheduling all actors that have a slack time of zero and increasing the resources as needed to accommodate these actors. Remaining schedulable actors at that time step are scheduled only if they do not need additional resources.

List-scheduling algorithms have a low computational complexity, and solutions usually do not differ much from optimum ones. List scheduling can also be modified to take interprocessor communication cost into account; Liao et al.⁹ has done an extensive comparison of various list-scheduling heuristics that take interprocessor communication into account.

Overlapped scheduling techniques. In contrast to

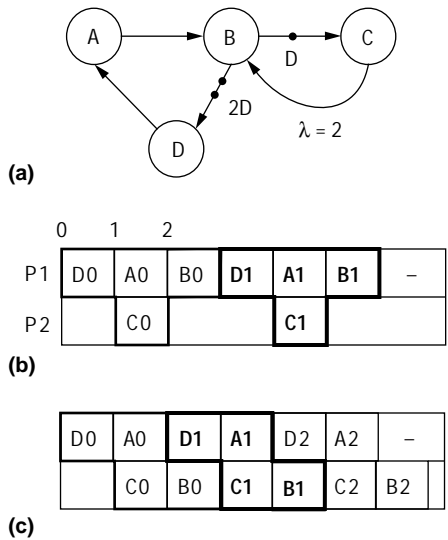


Figure 6. A dataflow graph (a); nonoverlapped, nonrate-optimal schedule (b); possible rate-optimal, overlapped schedule (c).

nonoverlapped scheduling, overlapped scheduling techniques exploit the interiteration parallelism completely and produce rate-optimal schedules when resource constraints and communication costs are ignored. Figure 6a shows a dataflow graph in which each actor has unit execution time. Figure 6b and 6c show two schedules. The first is derived by scheduling the APG and is not rate optimal since the critical path is 3. The other schedule is rate optimal and is overlapped because the i th iteration begins before the $i - 1$ th has finished (Figure 6c). We need to compute the exact starting times of each node such that the next iteration of that node can start λ cycles later. We can do this by using a shortest-paths algorithm in polynomial time. If λ is not an integer, the graph must be unfolded to achieve a rate-optimal schedule. Or, we can simply take the iteration period to be the ceiling of λ and get a schedule that has an iteration period at most one more than the IPB.

Once we add resource constraints to the overlapped scheduling problem, it too becomes intractable, and we have to resort to heuristics. One approach is a pseudopolynomial time, a bin-packing-like approach explored in Heemstra de Groot et al.¹⁰ Even though overlapped techniques are clearly superior, a major reason why nonoverlapped scheduling techniques are still used widely is that resource-constrained, nonoverlapped scheduling heuristics have been around in greater numbers.

Synchronous dataflow models

Recall that in SDF, an actor can produce or consume mul-

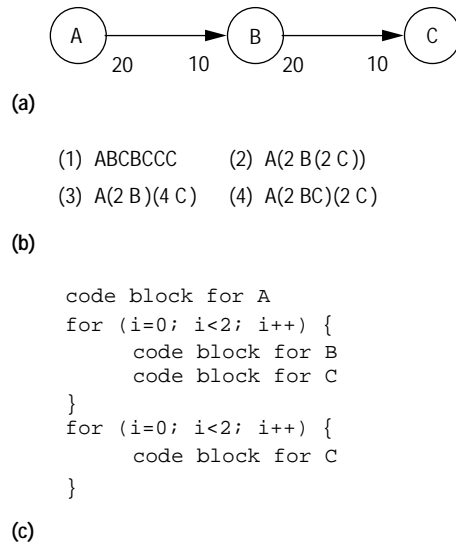


Figure 7. An example illustrating the interaction between scheduling synchronous dataflow graphs and the memory requirements of the generated code. Graph (a), four possible periodic schedules (b), and code (c).

multiple tokens per firing. Designers have used SDF in many block diagram-based rapid-prototyping environments,^{11,12} and code generation for programmable DSPs is a feature in many of these environments. Here we concentrate on certain problems that arise for uniprocessor scheduling.

The code generation strategy used in block diagram environments is called threading; in this method, the underlying model is scheduled to generate a sequence of actor invocations. A code generator then steps through this schedule and inserts the machine instructions necessary for the computation specified by each actor it encounters. These instructions are obtained from a predefined library of actor code blocks. The code generator generates in-line code because the alternative of using subroutine calls can have unacceptable overhead, especially if there are many small tasks.

A key problem in this strategy is the explosion of code size. If an actor appears 20 times in the schedule, there will be 20 code blocks in the generated code. Generally, the only mechanism to combat code size explosion while maintaining in-line code is the use of loops in the target code. If an actor's code block is encapsulated by a loop, multiple invocations of that actor can be carried out without duplicating code.

Loop scheduling. In an SDF graph, iteration of actors in a periodic schedule arises whenever the production and consumption parameters along an edge in the graph differ. For example, the 2-to-1 mismatch on the left edge of Figure 7a implies that within a periodic schedule, B must be invoked twice for every invocation of A .

The parenthesized terms in schedules 2, 3, and 4 in Figure 7b are called schedule loops and allow the code generator to organize loops in the target program, as shown in Figure 7c. The term $(2BC)$ in schedule 4 represents a loop whose iteration count is 2 and whose body is the invocation sequence BC . Thus, $(2BC)$ represents the firing sequence $BCBC$.

If each schedule loop is converted to a loop in the target code, each appearance of an actor in the schedule corresponds to a code block in the target program. Schedules in which each actor appears once (schedules 2 and 3 in Figure 7b) are called single-appearance schedules. If we neglect the code size overhead associated with the loop control, any single-appearance schedule yields an optimally compact in-line implementation of an SDF graph with regard to code size. Typically the loop overhead is small, particularly in many programmable DSPs, which usually have provisions to manage loop indices and perform the loop test in hardware, without explicit software control.

Scheduling can also have a significant impact on the amount of memory required to implement the buffers on the edges in an SDF graph. In Figure 7b, the buffering requirements for the four schedules are 50, 40, 60, and 50, assuming that one separate buffer is implemented for each edge. Note that maintaining a separate memory buffer for each edge is convenient and natural for code generation. It is the model used in SDF-based code generation described in the literature.^{12,13} Bhattacharyya et al. elaborated on more technical advantages of this buffering model.¹²

There are two ways of approaching the problem of jointly minimizing code and buffer size requirements. One gives preference to code size minimization, only considering schedules with minimal code size. From this class, we choose the one that minimizes buffer size. Conversely, we can give preference to buffer size, and choose code size minimal schedules from the class of buffer size minimal schedules. Or we can trade the two parameters systematically.

The loop-scheduling framework in Bhattacharyya et al. focuses on the first angle of attack, assigning first priority to code size minimization and second priority to minimizing the buffer memory requirement. This approach is preferable because, for practical SDF graphs, giving first priority to code size minimization typically yields a significantly more favorable code size/buffer memory trade-off than giving first priority to buffer memory minimization. In the future, researchers should explore alternatives in between these two extremes.


Minimum activation schedules. Heinrich Meyr's group at Aachen has studied the problem of constructing single-appearance schedules that attempt to minimize context-switch overhead. These minimum activation schedules have been

used in the COSSAP environment, now marketed by Synopsys. Each time a new actor is invoked in a schedule, there is a context switch; this overhead includes saving the contents of registers and loading state variables and buffer pointers. In the code generation environment described in Ritz et al.,¹² this overhead includes all the usual ones associated with function calls since the code generated by their system is not in-line.

The objective in minimum activation scheduling is to minimize the number of actor invocations that occur in the schedule. For example, the schedule $[2(2B)(5A)](5C)$ has five invocations per schedule period, while the "flat" version of the schedule $(4B)(10A)(5C)$ has three invocations each schedule period. We define the average activation rate for a periodic schedule as the number of activations divided by the blocking factor of the schedule. If we increase the blocking factor to 2, we can easily verify that the average activation rates for the two schedules discussed earlier become 4.5 and 1.5.

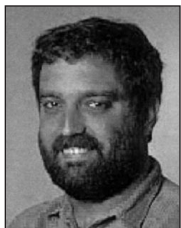
In general, for any consistent acyclic graph, we can make the average activation rate arbitrarily close to zero by using a flat single-appearance schedule (one without nested loops) of an arbitrarily high blocking factor. Thus, the problem becomes more interesting when the synchronous dataflow graph has cycles. Ritz et al.¹² provides algorithms (that in general are not polynomial time) in an attempt to find minimum activation schedules for arbitrary SDF graphs. However, it turns out that multiple-appearance schedules can have a lower average activation. Because the code size is also a primary constraint in the COSSAP code-generation system, Ritz et al. considered only single-appearance schedules.

SCHEDULING FOR EMBEDDED real-time systems is very challenging, and current research (which is unfortunately still far from current practice) has provided efficient solutions only for relatively small classes of problems. The situation is more satisfactory for dataflow systems. There, the regularity of input data streams and control lends itself well to static scheduling, and a variety of techniques to schedule on one or several processors exist, trading off execution time, code, and data memory size.

On the more control-dominated side, on the other hand, we are still lacking a good model to capture intertask dependencies and verify schedulability with realistic constraints. We also need better analysis techniques to determine the worst-case execution time of a task on a complex processor (with pipelining, caches, and so on) without being overly pessimistic. Finally, we are also missing a modeling framework that allows the designer to specify tasks and constraints, and decides which scheduling strategy best satisfies the problem domain, without requiring extensive a priori knowledge and decisions that may be difficult to reverse later on. 

References

1. C. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *ACM J.*, Jan. 1973, Vol. 20, No. 1, pp. 44-61.
2. N.C. Audsley, K.W. Tindell, and A. Burns, "The End of the Line for Static Cyclic Scheduling?" *Proc. Fifth Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 36-41.
3. N.C. Audsley et al., "Applying New Scheduling Theory to Static Priority Preemptive Scheduling," *Software Engineering J.*, Vol. 8, No. 5, Sept. 1993, pp. 284-292.
4. F. Balarin and A. Sangiovanni-Vincentelli, "Schedule Validation for Embedded Reactive Real-Time Systems," *Proc. 34th ACM/IEEE Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., June 1997, pp. 568-571.
5. N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, Boston, 1993.
6. G. Berry, P. Couronné, and G. Gonthier, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE*, Vol. 79, No. 9, Sept. 1991, pp. 1270-1282.
7. R. Reiter, "Scheduling Parallel Computations," *ACM J.*, Vol. 15, No. 4, Oct. 1968.
8. P.K. Murthy and E.A. Lee, "On the Optimal Blocking Factor for Blocked, Non-Overlapped Schedules," *Proc. 28th Asilomar Conf. Signals, Systems, and Computers*, IEEE CS Press, Vol. 2, Nov. 1994, pp. 1052-1057.
9. G. Liao et al., "A Comparative Study of Multiprocessor List Scheduling Heuristics," *Proc. 27th Hawaii Int'l Conf. System Sciences*, Vol. 1, Jan. 1994, pp. 68-77.
10. S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrman, "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," *IEEE Trans. Circuits and Systems-I Fundamental Theory and Applications*, May 1992, Vol. 39, No. 5, pp. 351-64.
11. J.T. Buck et al., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation*, Vol. 4, Apr. 1994.
12. S. Ritz, M. Pankert, and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," *Proc. Int'l Conf. Application-Specific Array Processors*, IEEE CS Press, 1993.
13. S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.



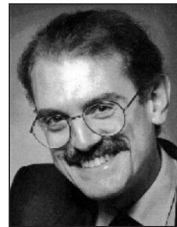
Felice Balarin is a research scientist at the Cadence Berkeley Laboratories. His research is focused on development and application of formal methods to design, verification, control, and timing analysis of systems consisting of both hardware and software. Balarin received his PhD in electrical engineering and computer science from the University of California at Berkeley. He is a member of the IEEE.



Luciano Lavagno is an assistant professor at the Politecnico di Torino, Italy, and a research scientist at Cadence Berkeley Laboratories. His research interests include the synthesis of asynchronous and low-power circuits, the concurrent design of mixed hardware and software systems, and the formal verification of digital systems. Lavagno received his PhD in computer science from the University of California at Berkeley. He is the author of a book on asynchronous circuit design and has published over 60 journal and conference papers. He has also been a consultant for various EDA companies including Synopsys and Cadence. He is a member of the IEEE.



Praveen Murthy is currently with the Alta Group of Cadence Design Systems in Sunnyvale, California. His research interests include techniques for producing optimized software and hardware implementations from dataflow graphs, multiprocessor scheduling techniques, semantics of dataflow networks, multidimensional dataflow, and software tools for rapid prototyping. Murthy received his BSEE from the Georgia Institute of Technology, and his MS and PhD degrees in electrical engineering and computer science from the University of California at Berkeley. He is a coauthor of *Software Synthesis from Dataflow Graphs* and a member of the IEEE Signal Processing Society.



Alberto Sangiovanni-Vincentelli is a professor of electrical engineering and computer science at the University of California at Berkeley. His interests include the design and optimization of digital circuits, the parallelization of complex algorithms in computer-aided design, and design methodologies and tools for mixed-signal integrated circuits including high-frequency and low-power circuits, and for embedded controllers. Sangiovanni-Vincentelli received his Dr.Ing. degree from the Politecnico di Milano in Italy. He is a cofounder of Cadence and Synopsys, and the founder of Cadence Berkeley and Cadence European Laboratories. He is a fellow of the IEEE.

Address questions or comments about this article to Felice Balarin, Cadence Berkeley Laboratories, 2001 Addison Street, Third Floor, Berkeley, CA 94704; felice@cadence.com.