Fine-Grain Adaptive Scheduling using Feedback

Henry Massalin and Calton Pu
Columbia University

ABSTRACT: We describe an implementation of a fine-grain adaptive scheduling mechanism, based on *software feedback*. Conventional scheduling makes job assignment exclusively a function of time. We broaden the meaning of the term "scheduling" to include job assignment as a function of a sequence of events, such as timer interrupts, I/O operations, queue overflow/underflow, and system call traps.

Our implementation of software feedback in the Synthesis operating system is analogous to the hardware phase locked loop. Very low overhead context switches and scheduling cost (a few microseconds on a 68020-based machine) makes this implementation useful to practical applications such as digital signal processing. Since scheduling actions and policy adjustments occur at very fine granularity (sub-millisecond intervals), Synthesis adaptive scheduling is very sensitive. Interesting applications of fine-grain adaptive scheduling include I/O device management, real-time scheduling, and distributed adaptive scheduling.

1. Introduction

Traditional scheduling policies use some global property, such as job priority, to reorder the jobs in the ready queue. A scheduling algorithm is adaptive if the global property changes dynamically; an example of dynamic global properties is the rate a job consumes CPU. Typical global scheduling policies assume that all jobs are independent of each other. Often this assumption does not hold. For example, in a pipeline of UNIX processes, where successive stages are coupled through their input and output. In fact, a global adaptive scheduling algorithm may lower the priority of a CPU-intensive stage, making it the bottleneck and slowing down the whole pipeline.

To take into account the connection between the stages of the pipeline when scheduling them, we use a *software feedback* mechanism to do adaptive scheduling. For example, if a job in the pipeline is "too slow," say its input queue is getting full, we schedule it more often and let it run longer. In general, the software feedback compares the progress of a job to some measurable sequence of events. In this example we monitor the queue length and the job's CPU quantum changes accordingly. This is the first distinguishing character of our work: the ability to monitor a sequence of events. We describe the software feedback for adaptive scheduling in Section 2.

The interaction between jobs may happen very often. In our example, processes may read and write many characters to a *pipe* during their CPU quanta. For our software feedback mechanism to monitor the variations of queue length and take appropriate scheduling action, we need very low-overhead mechanisms. We call a scheduling mechanism *fine-grain* if it is capable of taking many scheduling actions for typical jobs, for example, a few hundred context switches while completing a job of 10 milliseconds. This is the second novel aspect of our work: the extremely fine granularity of adaptation.

We have implemented fine-grain adaptive scheduling in the Synthesis operating system. This includes fast interrupt processing, fast context switching, and fast dispatching, typically costing a few dozen machine instructions. Also, the scheduling algorithm should be simple, since we want to avoid a lengthy search or calculations for each decision. This is the third contribution of our work: a practical demonstration of feasibility. In Section 3 we describe the Synthesis implementation with the necessary performance.

In addition to CPU scheduling, software feedback can be applied to many situations when we have a source of events (called a reference frame) and we want to impose a desired behavior based on the reference frame. In fine-grain scheduling, we divide the job into sufficiently short chunks and schedule them according to the timer interrupts. Other examples of reference frames include I/O device interrupts or output from another program. In Section 4 we outline various applications of software feedback in Synthesis, such as disk sector finding, real-time scheduling, and distributed adaptive scheduling.

2. *Principles of Software Feedback*

2.1 *Software Feedback*

The goal of a feedback system is to adjust an output sequence according to the observation of input, essentially estimating the immediate future based on observations of the recent past. The difference between input and output is called an *error*. A feedback system has two properties: stability and tracking accuracy. A stable system has bounded error. Intuitively it produces a reasonable output sequence, in response to reasonable input. A system tracks well when the error is small.

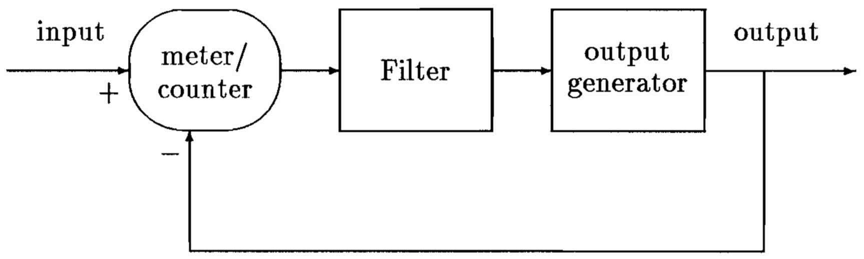

Figure 1: Software Feedback System

Software feedback systems are analogous to hardware feedback systems – one kind of control systems.[1] In Figure 1 we show a generic software feedback system. Its input is a sequence of events and output some other events. The input and output are linked by a feedback loop, which allows the adjustment of output according to the changes in the input. We call the software feedback system that measures and adjusts the event frequency of the input (events/second) an *event frequency feedback system* (EFF). Alternatively, we can measure and adjust the time interval between inputs (seconds/event), called a *time interval feedback system* (TIF). Other kinds of software feedback systems, say measuring time and adjusting frequency, are beyond the scope of this paper.

There are three main components in a software feedback, the meter/counter, the filter, and output generator. The meter captures the observation of the input and compares it to the output. Concretely, EFF measures event frequency by counting them, so the simplest behavior is to maintain the number of output events (and thus the frequency) equal to the input. Since TIF measures intervals, the simplest behavior is to maintain the time interval between consecutive output events equal to the input.

This simple behavior can be modified with *filters*.[2] The overall response of a software feedback system is determined by the kind of filter it uses to transform measurements into adjustments. We have studied filters that have well-understood hardware analogs. For example, a low-pass filter accumulates the

1. For readers unfamiliar with control systems, we include a summary in Appendix A.
2. Filter and error are names chosen because of their hardware analogs (summarized in Appendix A.2).

recent past to eliminate rapid transient changes in the input. It makes the EFF output frequency and TIF output intervals more uniform, but it also delays the feedback response, worsening the tracking accuracy. To decrease the error accumulated due to linear increases or decreases in either frequency or interval, we use an integrator filter, which improves the tracking accuracy of both EFF and TIF. A derivative filter improves response time when the input frequency or interval changes suddenly but stays with the new value. Like their hardware analogs, these filters can be combined to improve both the tracking accuracy and stability of both EFF and TIF.

All the program examples in this paper include filters that show good stability and tracking accuracy. A formal analysis of software filter stability and tracking accuracy is beyond the scope of this paper.

2.2 Application Domains

At first, measuring and adjusting frequency and intervals seem equivalent, since one is the reciprocal of the other, and both kinds of feedback will work. We choose the appropriate feedback mechanism depending on the desired accuracy and application. Accuracy is an important consideration because we can only measure integer quantities: either the number of events (frequency), or the clock ticks between events (interval). We would like to measure the larger quantity of the two since it carries higher precision.

Let us consider a scenario that favors TIF. Suppose we have a microsecond-resolution interval timer and the input event occurs about once per second. To make the output interval match the input interval, the TIF counter measures second-long intervals with a microsecond resolution timer, achieving 6-figure accuracy with only two events. Consequently, TIF stabilizes very quickly. In contrast, by measuring frequency (counting events), EFF needs more events to detect and adjust the error signal. In simple experiment, it takes about 50 input events (in about 50 seconds) for the output to stabilize to within 10% of the desired value.

A second scenario favors EFF. Suppose we have an interval timer with the resolution of one-sixtieth of a second. The input event occurs 30 times a second (once every 33 milliseconds). Since EFF is independent of timer resolution, the output will still stabilize to within 10% after seeing about 50 events (in about 1.7 seconds). However, since the event interval is comparable to the resolution of the timer, TIF will suffer loss of accuracy. In this example, the measured interval will be either 1, 2 or 3 ticks, depending on the relative timing between the clock and input. Thus TIF output can have an error of as much as 50%.

Generally, slow input rates and high resolution timers favor TIF, while high input rates and low resolution timers favor EFF. Sometimes the problem at hand forces a particular choice. For example, in queue handling procedures, the number of get-queue operations must equal the number of put-queue operations. This forces the use of EFF, since the actual number of events control the actions. In another example, subdivision of a time interval (as in the disk sector finder, described in Section 3.3). TIF is best.

3. Synthesis Implementation

3.1 Synthesis Operating System

Synthesis is a distributed operating system being developed by the authors at Columbia University, Department of Computer Science. The combination of a high-level model of computation with high performance distinguishes Synthesis from other operating systems. The Synthesis kernel interface is at a level comparable to that of UNIX, supporting threads (light-weight processes), memory management, and I/O devices. To achieve high performance with a high-level interface, we use kernel code synthesis, which is described in another paper [Pu et al. 1988]. The main idea of kernel code synthesis is to generate specialized (thus short and small) code at run-time for frequently executed kernel calls. A relevant example of kernel-generated code is the context switch. Each thread has its own context switch routines. Two procedures, switch-out and the switch-in, are specifically customized for that thread. During execution, the timer interrupt vector points to the current thread's switch-out procedure. When the CPU quantum is exhausted, timer interrupt branches to the switch-out procedure in the thread's control table, which saves the registers in the thread's control table. The final instruction of the switch-out branches to the next ready thread's switch-in, which restores the registers, changes the timer interrupt vector to its own switch-out procedure, and resumes execution of the thread.

The current implementation of Synthesis runs on an experimental machine (called the *Quamachine*), which is similar to a Sun-3: a Motorola 68020 CPU, 2.5MB no-wait state main memory, 390MB hard disk, 3½ inch floppy drive. In addition, it has some unusual I/O devices: a two-channel 16-bit analog output port, two-channel 16-bit analog input port, a compact disc (CD) player interface, and a 2Kx2Kx8-bit framebuffer with graphics coprocessor. The Quamachine is designed and instrumented to aid systems research. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and a microsecond-resolution interval timer. The CPU can operate at any clock speed from 1 MHz up to 50 MHz. Normally we run the Quamachine at 50 MHz. By setting the CPU speed to 16.7 MHz and introducing 1 wait-state into the memory access, the Quamachine can emulate the performance of a Sun-3/160, allowing a fair comparison. We have validated this emulation with some CPU- and memory-intensive programs, which report the same wall-clock time for the Sun-3 and Quamachine.

Since the low overhead of Synthesis kernel is crucial to the fine granularity of our adaptive scheduling, we include some performance figures here. The kernel call synthesized to read one character from */dev/mem* takes about 15 microseconds on the Quamachine. This and other important aspects of the Synthesis kernel implementation are described in a companion paper [Massalin & Pu 1989]. For this paper, the most important feature to the Synthesis implementation of fine-grain adaptive scheduling is the extremely fast interrupt handling and context switches. Figure 2 contains measurements of Synthesis thread primitives taken from the Quamachine in the Sun-3 emulation mode. For comparison, context switches take a few hundreds of microseconds in a high performance real-time operating system [Karsten et al. 1987].

operation	time (μsec)
create thread	142
start/stop	8
full context switch	11*
EFF/TIF update step	2
Block/Unblock thread	4

*If the thread does not use the Floating Point co-processor

Figure 2: Synthesis Fine-Grain Scheduling

3.2 *EFF Examples*

```
int residue=0, freq=0;

/* Master (reference frame) */
i1()
{
    residue += 4;
    freq += residue;
          .
     <do work>
          .
    return;
}

/* Slave (derived interrupt) */
i2()
{
    freq += residue;
    residue--;
          .
     <do work>
          .
    next_time = NOW + 1/freq;
    schedintr(i2, next_time);
    return;
}
```

Figure 3: Sample EFF – No Filter

```
int residue=0, freq=0, lopass=0;

i1()
{
    residue += 4;
    lopass = (7*lopass + residue)/8;
    freq += lopass;
        .
     <do work>
        .
    return;
}

i2()
{
    residue--;
    lopass = (7*lopass + residue)/8;
    freq += lopass;
        .
     <do work>
        .
    next_time = NOW + 1/freq;
    schedintr(i2, next_time);
    return;
}
```
Figure 4: Sample EFF – Low-pass Filter

```
int residue=0, freq=0, lopass=0, old_r=0;

i1()
{
1.1 residue += 4;
1.2 lopass = (7*lopass + residue)/8;
1.3 freq += lopass + (residue - old_r)
1.4 old_r = residue;
        .
     <do work>
        .
```

```
    return;
}

i2()
{
2.1 residue--;
2.2 lopass = (7*lopass + residue)/8;
2.3 freq += lopass;
          .
     <do work>
          .
2.4 next_time = NOW + 1/freq;
2.5 schedintr(i2, next_time);
    return;
}
```

Figure 5: Sample EFF – Derivative and Low-pass Filter

```
int residue=0, freq=0, integral=0;

i1()
{
    residue += 4;
    integral += residue;
    freq += integral;
          .
     <do work>
          .
    return;
}

i2()
{
    residue--;
    integral += residue;
    freq += integral;
          .
     <do work>
          .
    next_time = NOW + 1/freq;
```

```
        schedintr(i2, next_time);
        return;
    }
```
Figure 6: Sample EFF – Integral Filter

Applying the feedback idea to scheduling, we use EFF to keep two threads or interrupt sources running at some algebraic function of each other. Figure 3 shows the general abstract algorithm (without filters) when one source of interrupts happens at 4 times the rate of the other. The algorithms described in Figures 3, 4, 5, and 6 describe filters to improve the responsiveness and stability of EFF. Each one of them has a control system analog outlined in Appendix A.2. All the sample EFF algorithms shown in this section are meant to illustrate the EFF mechanism; they are not actual Synthesis code. Appendix B holds examples of working code for IBM PC/AT class of machines. Figure 5 contains line numbers (1.1, 2.1, etc.) that are referenced in Appendix B.

The algorithm in Figure 3 is a simple example of EFF. The phase variable, `residue`, keeps track of relative rates of `i1` and `i2`. The variable `freq` holds the frequency of `i2` interrupts, and `1/freq` the time between successive `i2` interrupts. `freq` has `residue` added to it each time `i1` or `i2` executes. Adding `residue` to `freq` is equivalent to an integrator filter, which improves long-term tracking accuracy. The thread `i1` is the reference; it runs at its own rate; and each time it executes it adds 4 to the `residue` counter. Thread `i2` runs at 4 times the rate of `i1` and each time `i2` executes it decrements the `residue` counter. If `i2` and `i1` were running at the perfect relative rate of 4 to 1, `residue` would tend to zero and no correction would result. In contrast, if `i2` is slower than 4 times `i1`, residue will become positive, increasing the frequency of `i2` interrupts and causing `i2` to speed up. Similarly, if `i2` is faster than 4 times `i1`, `i2` will be slowed down. As the difference in relative speeds increases, the correction gets correspondingly larger. As `i1` and `i2` approach the exact rate of 1:4, the difference decreases and we reach the minimum correction with `residue` being decremented by one and incremented by four, therefore cycling between [-2,+2*]*. The cycling `residue` will cause the `i2` execution frequency to oscillate around the ideal execution rate, with error bounded by 2.

A low-pass filter in the program helps eliminate this oscillation at the expense of convergence time. Figure 4 shows an EFF with low-pass filter. The variable `lopass` keeps a "history" of what the most recent value of `residue`. Each update adds 1/8 of the new `residue` to 7/8 of the old `lopass`. This calculation has the effect of taking a weighted average of recent `residues`. When residue is positive for many iterations, as is the case when `i2` is too slow, `lopass` will eventually be equal to `residue`. But if residue oscillates, as in the situation described in the previous paragraph, `lopass` will go to zero.

The problem now is increased convergence time. The low-pass filter has a lag effect on the EFF response. If `i1` speeds up quickly, `i2` will lag behind `i1` while `lopass` "charges up." Convergence time can be decreased by adding a differentiator filter in the `i1` loop (Figure 5). The expression `(residue - old_r)` approximates the first derivative of residue. Since it appears only in the `i1` loop, it does not magnify the high frequency `i2` oscillation. The correction due to derivative is higher when `i1` execution rate varies quickly, pushing `i2` towards the correct rate faster.

Finally, we show an integrator filter in Figure 6. This kind of filter is useful for accurate tracking of interrupt sources with linearly increasing (or decreasing) rates. Since integrals filter out high frequencies naturally, there is less need for a low-pass filter to deal with oscillations.

3.3 Synthesis Examples

We have used the fine-grain adaptive scheduling to handle a wide variety of jobs in Synthesis. These are:

- a special effects sound processing program that uses TIF to find rhythms in music;
- an interrupt generator using TIF to generate an interrupt a few microseconds before each sector passes under the disk head;
- a digital oversampling interpolator for a CD player; the interpolator uses EFF to adjust its I/O rate to match the CD player output.

The special effects program takes as input a stereo sound source sampled at 44,100 Hertz. The program processes the input in real time and produces output which is sent to the digital to analog converters and eventually to the speakers. The program is a pipeline of delay elements, echo and reverberation filters, adjustable low-pass, band-pass and high-pass filters. A correlator and feature extraction unit extracts rhythm pulses from the music. The next stage uses TIF to generate a stream of interrupts synchronized to the beat of the music. These interrupts drive a drum synthesizer, which adds more drum beats to the music. We can also get pretty pictures synchronized to the music when we plot the TIF input versus output on a graphics display. We rely on the system scheduler EFF (described in Section 4.3) to maintain the high data rate flowing through the pipeline in real-time. The fine-grain adaptive scheduler keeps the data flowing smoothly at the 44.1 KHz sampling rate, regardless of how many CPU-intensive jobs might be executing in the background.

In the second application, TIF helps the disk driver minimize rotational delay. Many simple disk controllers generate a hardware interrupt once every disk revolution. This sequence of interrupts (once per revolution) is used by TIF to generate a faster sequence of timer interrupts, which happens once per sector (about 17 times each revolution). With the new sequence of sector interrupts, the disk driver knows what sectors are closest to the disk heads and can minimize rotational delay in addition to normal seek optimization algorithms. Good buffering may increase data throughput for some applications, but important applications such as database logging may benefit from rotational delay optimization. Our example works for simple disk controllers using protocols such as ESDI, but not for other controllers, e.g. those using the SCSI protocol. More intelligent controllers supply the current sector number under the head for OS-level optimization, for example, based on heuristics [Stevens 1989].

In the third application, EFF is used in the digital oversampling interpolator for the CD player. The interpolator takes as input a stream of sampled data and creates additional samples between the original ones by interpolation. This oversampling increases the accuracy of analog reconstruction of digital signals. We generate 4 samples using interpolation from each CD sample

(4:1 oversampling). The CD player produces 44,100 new data samples per second, or one every 22.68 microseconds. The interpolated data output generated by EFF is four times this rate, or one every 5.67 microseconds.[3] This example shows the feasibility of scheduling at this fine granularity.

3.4 Discussion

The implementation and the previous examples support three important points that we make explicit here. First, a conventional scheduler cannot accomplish the kind of adaptive scheduling based on fine-grain feedback. This is illustrated by the rhythm tracker. Without the feedback mechanism it is impossible to capture the rhythm modulation given the amount of processing power available. Second, the processing requirements are so demanding that only an extremely short critical path can keep the programs running in real-time. This is shown by the CD player at the 5.67 microsecond output cycle. Third, the combination of fine-grain and adaptation is uniquely powerful. The disk driver sector interrupt generator is an example. Even if we could speed up the processor (or slow down the disk rotation) to try to do it using conventional scheduling, the output sequence will soon go out of synchronization. Section 4.1 discusses this point in more detail.

A formal analysis of the properties of fine-grain adaptive scheduling is beyond the scope of this paper. However, we would like to give the readers an intuitive feeling about two situations: saturation and cheating. As the CPU becomes saturated (no idle times), the fine-grain adaptive scheduler degrades gradually. Since the hardware interrupts cause scheduling actions, the threads closest to device drivers will get their share of CPU. Consequently, the threads further away from I/O interrupts become slower due to competition. This gradually fills the queues starting from the I/O devices. If the system recovers before the queues overflow, we have a graceful degradation and recovery. This is the desired behavior, especially in a real-time system with high I/O rates.

3. This program runs on the Quamachine at 50 MHz clock rate.

Another potential problem is cheating (consuming resources unnecessarily to increase priority), since feedback-based scheduling tends to give more CPU to threads that consume more. However, cheating cannot be done easily from within a thread or by cooperation of several threads. First, unnecessary loops within a program does not help the cheater, since they do not speed up data flow in the pipeline of threads. Second, the schedule is arranged so that I/O within a group of threads only shifts CPU quanta within the group. A thread that reads from itself gains quantum for input, but loses the exact amount in the self-generated output. To increase the priority of a thread, it must read from a real input device, such as the CD player. In this case, it is virtually impossible for the OS kernel to distinguish the real I/O from cheating I/O. This kind of cheating would succeed under existing schedulers. In Section 4.3 we discuss cheating further.

4. Applications

We apply fine-grain scheduling policies to three kinds of situations:

- interrupt source coupled to interrupt source, described in Section 4.1,
- interrupt source coupled to program progress, described in Section 4.2, and
- program progress coupled to program progress, described in Sections 4.3 and 4.4.

4.1 Clock Accuracy and Synchronization

An interval timer shows differential stability, i.e., the intervals between consecutive clock ticks are very much the same. However, any small inaccuracy in the clock tick will accumulate in the long term into a noticeable difference between the clock and the absolute time. A timing device exhibits integral stability when it is able to limit this difference even for the long run. EFF provides integral stability, i.e., in the long-term the accumulated difference between the reference frame and generated interrupts is bounded by a constant, even though any individual interval may differ from the reference. This property of EFF avoids the need for very accurate interval timers for some important applications.

A fundamental application is clock synchronization. We consider asymmetric clock synchronization first, where slave clocks try to follow accurately a master clock, usually ticking at a coarse granularity. (For example, the world-wide broadcast time reference has the resolution of one second.) EFF solves this problem well, taking the master clock tick as input and producing the slave clock ticks at some rational (p/q) rate of the input, at a higher precision. The feedback adjusts an interval timer to generate the interrupts "missing from the input." The precision of a slave clock is only limited by its interval timer. The integral stability of EFF bounds the difference between the master and slave. Note the similarity of asymmetric clock synchronization with the CD oversampling interpolator (Section 3.3).

A more sophisticated problem is symmetric clock synchronization, where we advance the participating clocks in some kind of lock-step, i.e., the relative differences among the participating clocks should be bounded. Traditional distributed clock synchronization algorithms use agreement protocols to find the average time, and the bound on the difference is tighter if we run the agreement protocols more often. Software feedback can be useful in reducing the error bound or the frequency of agreement protocol, or both. The intuitive idea is that using EFF each clock learns whether it is advancing or lagging with respect to the average and compensates for it.

A third problem is making accurate timing measurements with a coarse interval timer. A recent report [Danzig & Melvin 1990] describes two ways to make accurate measurements on Sun workstations, which have a 20-millisecond resolution interval timer. The first method is to take many measurements and average them. The drawback is that we need a large number (thousands to hundreds of thousands) of data points to achieve one or two orders of magnitude accuracy improvement. The second method is to build a high-resolution hardware clock, but this is expensive. In contrast, software feedback (EFF) can achieve high-resolution timing with only low-resolution clocks. The use of EFF does not significantly reduce the number of measurements that must be made to achieve a specified accuracy, but it can provide a continuous stream of "best guesses" while converging to the ultimate answer. This feature proves useful in some highly dynamic environments when a good guess is better than no information at all. Finally, the fine-grain (cheap) mechanism ensures that the timing is as unobtrusive as possible.

4.2 Real-Time Scheduling

Real-time scheduling is the problem of scheduling jobs with deadlines. A soft-deadline job loses its value gradually as the deadline passes. A hard-deadline job may cause catastrophic system failure if not completed by its deadline. For example, a computer-controlled valve better open before the increasing pressure causes an explosion. Simple versions of real-time scheduling problem have been shown to be NP-hard and the hard-deadline real-time scheduling is in general a very difficult problem [Stankovic & Ramamritham 1988].

In our discussion, we divide hard-deadline jobs into two categories: short ones and long ones. A short job is one that must be completed in a time frame within two orders of magnitude of interrupt and context switch overhead. For example, a job taking up to 50 microseconds would be a short job in Synthesis. Short jobs are scheduled as they arrive and run to completion without preemption.

Long jobs take longer than 100 times the overhead of an interrupt and context switch. In Synthesis this includes all the jobs that take more than 1 millisecond, which includes most of the practical applications. The main problem with long jobs is the variance they introduce into scheduling. Since most traditional hard-deadline scheduling approaches try to guarantee the schedule, they must be prepared for the worst scenario. If the variance is large, the CPU requirements for the worst case is much higher and more expensive than the average case. Therefore, the hardware remains unused most of the time.

To use fine-grain adaptive scheduling for long jobs, we break a long job into small *strips*. If the real-time program is written in assembler language (frequently this is done to minimize overhead and guarantee response-time), the programmer knows enough details of the code to choose the strips. If the program is in a high-level language, say Ada, we can combine compiler support with heuristics. For example, procedure boundaries in modular programs would be good candidates for delimiting strips. Another piece of useful information is the estimated run-time of the entire job. This could be done by measuring the program CPU consumption to find the execution profile statistics of the expected and maximum completion time. In the rest of this section we assume suitable strips can be found, since too complex jobs are not good building blocks of real-time systems.

For simplicity of analysis we assume each strip to have the same execution time ET. We define the estimated CPU time to finish job J as:

$$Estimate(J) = \frac{(strips\ in\ J) * ET}{Deadline(J) - Now}$$

For a long job, it is not necessary to know ET exactly since the software feedback "measures" it and continually adjusts the schedule in lock step with the actual execution time. In particular, if $Estimate(J) > 1$ then we know from the current estimate that J will not make the deadline. If we have two jobs, A and B, with $Estimate(A) + Estimate(B) > 1$ then we may want to consider aborting the less important one and calling an short emergency routine to recover.

Unlike traditional hard-deadline scheduling algorithms, which either guarantee completion or nothing, fine-grain scheduling provides the ability to predict the deadline miss. We think this is an important practical concern to real-time application programmers, especially in recovery from faults. (For a good discussion of issues in real-time computing, see Stankovic [1988].)

4.3 Adaptive Scheduling

Current operating systems, e.g. UNIX, use a simplistic adaptive strategy to improve system throughput. Usually, they have a priority-based scheduling mechanism. CPU-intensive jobs have their priority lowered and I/O-intensive jobs priority increased. In UNIX, if a job has exhausted its CPU quantum when de-scheduled then it is likely to be CPU-intensive. Also, if a job accumulates enough CPU minutes it is automatically demoted to a lower priority. This scheme works very well with jobs of long duration, since the scheduling decision is based on the average behavior of the job.

However, as we mentioned in the introduction, this algorithm assumes the jobs are independent of each other. Serious problems arise even in simple cases, for example, the pipeline of threads each with one input and output. Each stage of the pipeline is a consumer of data from the previous stage and producer of data for the next stage. If one particular stage in the pipeline needs a relatively large amount of CPU (compared to the other stages), the above simple adaptive scheduling would lower its priority, causing congestion to form at the stage. In the Synthesis world of many threads connected in a graph, avoiding this kind of congestion is crucial for good performance.

First we note that a smooth flow of data through the pipeline would best use the resources of the system, since each stage will be running at just the right speed, without idling or congestion. In the above scenario, data flow slows down at the CPU-intensive stage, so the entire pipeline runs at the speed of the lowest priority stage. To solve this problem with fine-grain adaptive scheduling, we adjust the thread priority according to the length of its input queue. The frequency and length of the CPU quantum of a stage is directly proportional to the length of its input queue and inversely proportional to the length of its output queue. If the input queue is full, then this stage is a bottleneck and should be scheduled more often and with a larger quantum of CPU, in the hope that it will start consuming its input faster. Similarly, if a thread has filled its output queue, then it is "too fast" and should have a smaller quantum and be scheduled less often.

In the Synthesis fine-grain adaptive scheduling, the kernel detects when a queue approaches empty or full and makes the adjustments to the thread's quantum. This way, a CPU-intensive thread in a pipeline will be scheduled more often and "run faster" to keep up with the rest of the pipeline. We should note that Synthesis does not currently do any significant global CPU accounting. With only local adjustments based on queue length, fine-grain scheduling policies exhibits global stability, i.e., the pipeline of threads run smoothly.

Since the Synthesis adaptive algorithm varies thread priority dynamically, we need to put a limit on the CPU allocated to each thread to avoid monopoly. For example, a thread that reads from a high data rate I/O source (say a sound digitizer at 50,000 samples per second) may be able to capture a lot of CPU because the digitized sound data come in at a very high rate. We impose an upper limit on the thread CPU quantum and scheduling frequency to prevent any thread from monopolizing the CPU. This restriction may be relaxed for dedicated real-time systems.

4.4 Multiprocessor and Distributed Scheduling

We think the adaptiveness of EFF promises good results in multiprocessor and distributed systems. At the risk of oversimplification, we describe an example with fixed buffer size and execution time. We recognize that given a load we can always find the optimal scheduling statically by calculating the best buffer size and CPU quantum. We emphasize the main advantage of software feedback: the ability to dynamically adjust towards the best buffer size and CPU quantum. This is important when we have a system with variable load, jobs with variable CPU demands, a reconfigurable system with a variable number of CPUs, or an application program designed for machines with different speed parameters.

Figure 7 shows static scheduling for a two-processor shared-memory system with a common disk (transfer rate of 2 MByte/second). We assume that both processes access the disk drive at the full transfer rate, e.g. reading and writing entire tracks. Process 1 runs on processor 1 (P1) and process 2 runs on

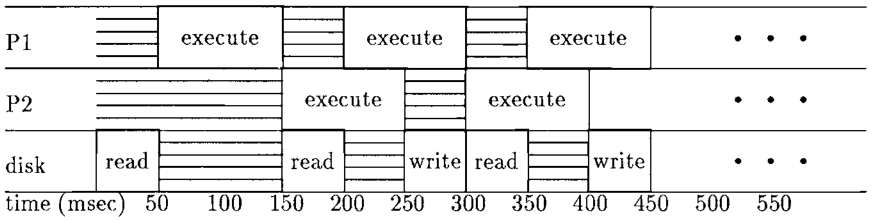

Figure 7: Two Processors, Static Scheduling

processor 2 (P2). Process 1 reads 100 KByte from the disk into a buffer, takes 100 msec to process the data, and writes 100 KByte through a pipe into process 2. Process 2 reads 100 KByte from the pipe, takes another 100 msec to process them, and writes 100 KByte out to disk. In the figure, process 1 starts to read at time 0. All disk activities appear in the bottom row, P1 and P2 show the processor usage, and shaded quadrangles show idle time. Figure 8 shows the fine-grain scheduling mechanism (using EFF) for the same system. We assume that process 1 starts by filling its 100 Kbyte buffer, but soon after it starts to write to the output pipe, process 2 starts. Both processes run to exhaust the buffer, when process 1 will read from the disk again. After some convergence time, depending on the filter used in the software feedback, the stable situation is for the disk to remain continuously active, alternatively reading into process 1 and writing from process 2. Both processes will also run continuously, with the smallest buffer that maintains the nominal transfer rate.

The above example illustrates the benefits of fine-grain scheduling policies in parallel processing. In a distributed environment, the analysis is more complicated due to network message overhead and variance. In those situations, calculating statically the optimal scheduling becomes increasingly difficult. We expect the fine-grain scheduling to show increasing usefulness as it adapts to an increasingly complicated environment.

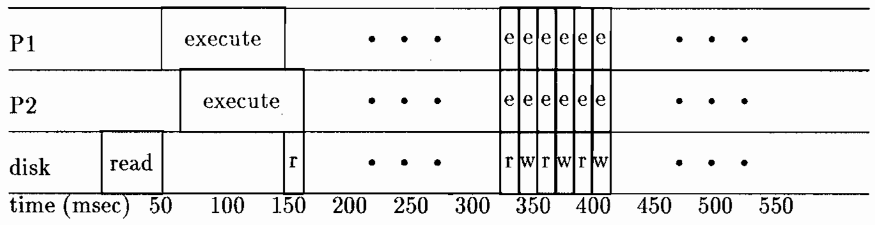

Figure 8: Two Processors, Fine-Grain Scheduling

5. *Conclusion*

We have described a fine-grain adaptive scheduling mechanism based on software feedback. The fine granularity uses frequent state checks and dispatching actions to adapt quickly to system changes. Quick adjustments make better use of system resources, since we avoid queue/buffer overflow and other mismatches between the old scheduling policy and the new situation. The fine granularity also allows the software feedback to take into account local state changes, such as the queue length connecting the consecutive stages of a pipeline. Finally, the fine granularity enables early warning of deadline misses, giving the real-time application programs more time to attempt an emergency recovery before the disaster strikes.

We have implemented the fine-grain adaptive scheduling in the Synthesis distributed operating system. Synthesis kernel makes adjustments every few hundreds of microseconds based on local information, such as the number of characters waiting in an input queue. Fundamental to our implementation are the low-overhead kernel facilities. These include scheduling actions (a few tens of microseconds), context switch for dispatching (less then ten microseconds), checking local state (a few machine instructions), and interrupt processing (also a few machine instructions).

The software feedback mechanism can be used in situations other than CPU scheduling. Given a sequence of events we can generate another related sequence. In the current version of Synthesis, we have used it in I/O device management and real-time scheduling. Looking into the future, distributed applications stand to benefit from the software feedback, since they will be able to track the input events despite variances introduced by message delays. Concrete applications we are studying include load balancing, distributed clock synchronization, smart caching in memory management and real-time scheduling.

Acknowledgements

We would like to thank the reviewers for the Special Issue. They made valuable comments that improved the presentation of this paper considerably.

This work is partially funded by the New York State Center for Advanced Technology on Computer and Information Systems under the grant NYSSTF CU-0112580, the National Science Foundation under the grant CDA-88-20754, the AT&T Foundation under a Special Purpose Grant, the Digital Equipment Corporation under the External Research Program, IBM Fellowships, and Sun Microsystems. Also, we would like to thank AMD, Hitachi, Intel, Motorola, National Semiconductor and Mupac for their contributions of hardware samples for the experimental machine.

Appendix A: Properties of Hardware Feedback Systems

A.1 Phase Locked Loops

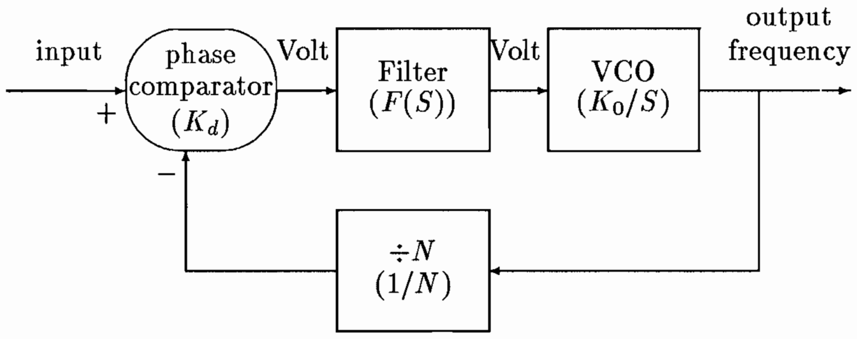

Figure 9: PLL Picture

To explain the software feedback systems, we start with a hardware analog, the phase locked loop (PLL). Figure 9 shows the PLL as a block diagram. The PLL synchronizes an output frequency with an input frequency. If the rate divider (N) is set to unity, then the PLL generates an output that is frequency and phase synchronized to the input (frequency is the time derivative of phase). The phase comparator outputs a signal proportional to the difference in phase (frequency) between its two inputs. The filter is used to tailor the time-domain response of the loop. An example is a low-pass filter that attenuates the quickly varying phase differences and passes the slowly varying phase differences. The oscillator (in hardware implemented as a voltage-controlled oscillator – VCO) generates an output frequency proportional to its input, which comes from the output of the filter. The overall loop operates to compensate for variations on input, so that if the output rate is lower (higher) than the input rate, the phase comparator, filter, and oscillator work together to increase (decrease) the output rate until it matches the input. When the two rates match, the output rate tracks the input rate, and the loop is said to be locked to the input rate.

A software feedback system has the same three elements of the PLL. First, we track the difference between the running rate of a job and the reference frame; this is analogous to a phase comparator. Second, we use a filter to dampen the oscillations in the difference, like the PLL filter. Third, we re-schedule the running job to minimize its error compared to the reference, in the same way the VCO is adjusted.

A.2 Filters in PLL

Like all feedback control systems, a PLL may be stable or unstable. The informal arguments for the stability presented in this section follow established control theory, in particular PLL frequency synthesis [Rohde 1983]. A PLL can be modeled as a three-stage cascaded feedback system (Figure 10). In control theory, we use Laplace Transforms to analyze system behavior and determine stability. In Figure 10 we have a generic feedback system with the open-loop function G and the feedback function H. The Laplace Transform (function of S) of the closed-loop function is:

$$\frac{out(S)}{in(S)} = \frac{G}{1 + G * H}$$

For the PLL in Figure 10, we have

$$G = K_d * F(S) * K_0/S = K * F(S)/S$$

where K_d is the sensitivity of phase comparator (also called gain factor) in volts/radian, K_0 is the responsiveness of VCO in

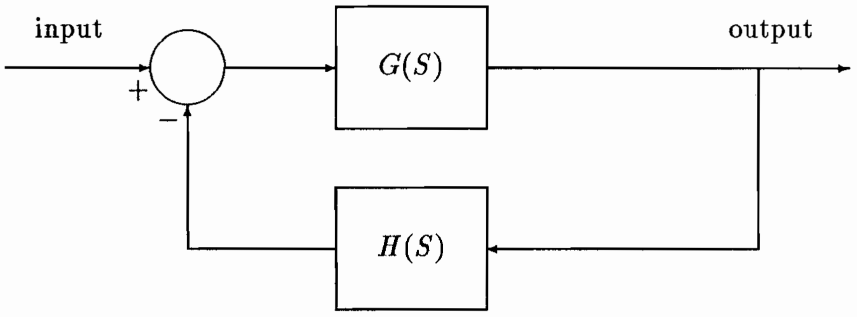

Figure 10: Feedback System

(radian/second)/volt. The loop gain K is the constant product of K_d by K_0. The feedback function is $H = 1/N$ because the output is divided by N before it is fed into the phase comparator. Substituting the terms we have:

$$\frac{out(S)}{in(S)} = \frac{K * F(S)/S}{1 + K * F(S)/S/N} = \frac{K * F(S)}{S + K * F(S)/N}$$

We say that a feedback system is stable if for all bounded input it produces bounded output. A particular feedback system is stable if all the singularities of its closed-loop function have the real part less than zero.

To study the stability of PLLs, we need to fill in the unspecified filter function $F(S)$. Some useful filter functions and their Laplace Transforms are given below. These filters may be used alone, cascaded, or superimposed. We can calculate the composite filter functions using two properties of Laplace Transforms: cascaded elements (serial connection) multiply, and combined signals (superimposed) add.

- Null filter: $F(S)=1$.
- Gain (proportional) filter: $F(S)=C$ (some constant).
- Simple low-pass filter: $F(S) = \dfrac{1}{1+(\frac{1}{\omega}) * S}$ where ω is the cut-off frequency in radian/sec.
- Integrator filter: $F(S)=1/S$.
- Differentiator filter: $F(S)=S$.

Each of these filters has certain general tendencies (not iron-clad rules) summarized in Table 1. Table 1 tells the influence of each filter on the feedback system: tracking accuracy, stability, and convergence time. Tracking accuracy refers to the classes of input functions a feedback system can follow without steady-state error. Improving stability moves the real part of the closed-loop function singularity towards minus infinite. Convergence time is the time it takes for the output to reach its final value. Noise immunity tells how sensitive the system is to input noise and oscillations.

The simplest PLLs have no filter at all, or perhaps just a gain filter. This type of PLL will always have some error. The steady

	tracking accuracy	Stability	Convergence Time	Noise Immunity
null	–	–	–	–
low-pass	–	improve	lengthen	improve
integrator	improve	lessen	lengthen	improve
differentiator	lessen	improve	shorten	lessen

Table 1: Informal Filter Characteristics

state error in response to step changes in input frequency is inversely proportional to the open loop gain (K). High gain is required to reduce the error, but increased gain makes the PLL response noisy and eventually unstable.

To reduce the noise in the output and make PLL more stable, we can add a low-pass filter, which removes the high frequency transients. Low-pass filtering results in a weighted average of recent input. Occasional input transients will contribute little to the weighted average and affect the output less.

To remove the error completely, we use integrator filters. A feedback system with one integrator filter will track constant input with no error, linearly increasing input with constant (non-increasing) error, and quadratic input with ever-increasing error. A feedback system with two integrator filters will track linearly increasing input with no error, quadratic input with constant error, and so on. In general, feedback system tracking accuracy improves with the number of integrator filters.

In a PLL, the VCO contains an implicit integrator, since it generates constant frequency (linearly increasing phase angles) for constant input. Consequently, a PLL with a null filter behaves as a feedback system with one integrator, which appears as the K_0/S factor in PLL's open-loop function.

Integrator filters lessens PLL stability and lengthens convergence time. We can use differentiator filters to counteract these tendencies. Differentiator filters are never connected in cascade, since they will cancel the effect of integrator filters. They are superimposed with gain filters or more integrator filters. Differentiator filters increase loop stability, allowing the loop gain to be increased, therefore decreasing convergence time. In general, one can use integrator and differentiator filters to tailor system response according to the expected input.

A.3 Analyzing PLL and Software Feedback

In the previous section, we have summarized the mathematical analysis of filters for hardware PLL. Since the software implementation of feedback is analogous to the hardware one, we would like to analyze the software feedback similarly. However, two kinds of significant differences make the formal analysis difficult: the ones dealing with the discrete-time nature of the software feedback, and the ones dealing with frequency and interval measurements and adjustments. We will consider each in turn.

In the first place, the hardware PLL is a continuous-time system, while the software feedback is a discrete-time system. In particular, the hardware PLL has a continuous frequency range, while the EFF has a discrete frequency range. The resolution of the hardware interval timer limits the choice of frequencies in the EFF. When a desired frequency falls between two available frequencies, the EFF will alternate between the two available ones. This behavior can be modeled by inserting a floor function filter in the hardware PLL to simulate the quantized nature of the EFF. The floor function introduces non-linearity into the model, making it difficult to analyze.

If the discreteness were the only difference, we could use Z-Transforms instead of Laplace Transforms to analyze the software feedback systems. One example of additional complications is the non-constant sampling rate in EFF: samples occur whenever `i1` or `i2` runs. We can change the variables appropriately to place the sample points on a fixed grid, but these changes of variables will introduce other non-linearities, complicating the model further.

Yet another problem is peculiar to TIF, which measures time intervals intead of frequency. Since the period is the inverse of frequency, we could insert an $1/x$ filter right before the VCO in the PLL in Figure 9 to model TIF, converting a frequency adjustment into a period adjustment. The $1/x$ function would introduce another non-linearity into the model. Similarly, TIF adjustment of time intervals requires another $1/x$ function between the comparator and the filter.

Our work on mathematically analyzing the software feedback is in progress. In the mean time, we have written a few test programs to empirically observe the stability of different algorithms and filters. One such program is found in Appendix B.

Appendix B: Fun Demo for Your PC

Readers who want to play with software feedback can run the following programs (Figures 11 and 12), written for the IBM PC and compatibles. Type it in and compile! It has been tested using the Microsoft C compiler. You may have to change the `define getkey()` on other compilers. The function *getkey*() is a non-blocking read of keyboard that returns the key pressed or −1 if no key has been pressed. Run it. Push and hold down the "1" key, letting auto-repeat generate a steady stream of "events" and watch the error oscillate a few times to stabilize towards zero. Try playing with the parameters, change the filters, the gain constant, ... Enjoy!

The EFF demo program is patterned after the template program in Figure 5 (EFF – Derivative and Low-Pass Filter). All the line references point to that figure, including the line numbers and specific statements.

- The variable *time* keeps track of simulated time, which is incremented by one each loop.
- EFF synchronizes to a multiple of an external event, in this case the pushing of a key. The multiple is determined by the numeric key being pushed (1 = ½, 2 = 1, ..., 9 = 4½) [Key = multiple].
- The output is two rotating bars, the first rotating with each key push, the second rotating at the EFF output frequency. The other numbers displayed show the current error and instantaneous frequency.
- The statement `err += 3 * (c-'0');` corresponds to line 1.1.
- The statement `if(time > next)` corresponds to the interval timer expiring and causing an interrupt.
- The statement `err -= 6` corresponds to line 2.1.
- The statement `x = filter(err)` corresponds to line 2.2.
- The statement `freq += x` corresponds to line 2.3.

- The statement `next += 200000L/freq` corresponds to the line `schedintr(...)` in Figure 5.

```
/* an example of EFF */
#include <stdio.h>
#define ESC     27
#define getkey()    (kbhit() ? getch() : -1)

filter(x)
int     x;
{
static  int     lopass, old_x;
    int     r;

    lopass = (3*lopass + x) >> 2;
    r = lopass + 15*(x - old_x);
    old_x = x;
    return r;
}

main()
{
static  char    bar[] = "|/-\\";
    int     i1, i2, event, freq, err, x, c;
    long    time, next, last, tmp;

    i1 = i2 = err = event = 0;
    time = next = last = 0;
    freq = 200;
    while((c = getkey()) != ESC)
    {
        time++;

        /* this is i1 */
        if(c >= '1' && c <= '9') {
            err += 3 * (c-'0');
            x = filter(err);
            freq += x; if(freq <= 0) freq = 1;
            next = last + 200000L/freq;
            i1 = (i1+1) & 3;
```

```
                event = 1;
            }

            /* this is i2 */
            if(time > next) {
                last = next;
                err -= 6;
                x = filter(err);
                freq += x; if(freq <= 0) freq = 1;
                next = last + 200000L/freq;
                i2 = (i2+1) & 3;
                event = 1;
            }

            if(event) {
                event = 0;
                printf("%c %c %+4.3d %+4.3d %5d\r",
                    bar[i1], bar[i2], err, x, freq);
            }
        }
}
```

Figure 11: EFF Demo Program

```
/* a example of TIF */
#include <stdio.h>
#define ESC     27
#define getkey()    (kbhit() ? getch() : -1)

long    filter(x)
long    x;
{
static  int     lopass;
    x <<= 8;
    lopass = (63*lopass + x) >> 6;
    return (lopass+128)>>8;
}
```

```
main()
{
static  char    bar[] = "|/-\\";
    int  i1, i2, event, c, mul;
    long time, next, last1, last2,
         intv1, intv2, err, x;

    time = next = last1 = last2 = intv1 = intv2 = 0;
    i1 = i2 = event = 0;
    err = 0;
    mul = 2;
    while((c = getkey()) != ESC)
    {
        time++;

        /* this is i1 */
        if(c >= '1' && c <= '9') {
            mul = (c-'0');
            intv1 = time - last1;
            last1 = time;
            err = ((intv1<<1)/mul) - intv2;
            x = filter(err);
            intv2 += x; if(intv2 <= 0) intv2 = 0;
            next = last2 + intv2;
            i1 = (i1+1) & 3;
            event = 1;
        }

        /* this is i2 */
        if(time > next) {
            if (time - last1 > intv1)
                intv1 = time - last1;
            err = ((intv1<<1)/mul) - intv2;
            x = filter(err);
            intv2 += x; if(intv2 <= 0) intv2 = 0;
            last2 = next;
            next = last2 + intv2;
            i2 = (i2+1) & 3;
            event = 1;
        }
        if(event) {
            event = 0;
            printf("%c %c %9ld %9ld\r",
                bar[i1], bar[i2], intv1, intv2);
        }
    }
}
```
Figure 12: TIF Demo Program

References

P. Danzig and S. Melvin, High resolution timing with low resolution clocks and a microsecond resolution timer for Sun workstations, *ACM SIGOPS Operating Systems Review*, 24(1):23-26, January 1990.

S. Karsten, T. Bihari, B. W. Weide, and G. Taulbee, High-performance operating system primitives for robotics and real-time control systems, *ACM Transactions on Computer Systems*, 5(3):189-231, August 1987.

H. Massalin and C. Pu, Threads and input/output in the Synthesis kernel, In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Arizona, December 1989.

C. Pu, H. Massalin, and J. Ioannidis, The Synthesis kernel, *Computing Systems*, 1(1):11-32, Winter 1988.

U. L. Rohde, *Digital PLL Frequency Synthesizers: Theory and Design*, Prentice-Hall, Inc., first edition, 1983.

J. A. Stankovic, Misconceptions about real-time computing: A serious problem for next-generation systems, *IEEE Computer*, 21(10):10-19, October 1988.

J. A. Stankovic and K. Ramamritham, editors, *Tutorial: Hard Real-Time Systems*, Computer Society Press of IEEE, 1988.

W. R. Stevens, Heuristics for disk drive positioning in 4.3BSD, *Computing Systems*, 2(3):251-273, Summer 1989.