Thrifty Execution of Task Pipelines*

Fred B. Schneider
Richard Conway
Dale Skeen

TR 84-615
June 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work is supported in part by NSF Grant MCS-8003340 and MCS-8103605. Schneider is also supported by a Faculty Development Grant from IBM Corp.

Thrifty Execution of Task Pipelines*

Fred B. Schneider
Richard Conway
Dale Skeen

Department of Computer Science
Cornell University
Ithaca, New York 14853

June 11, 1984

ABSTRACT

A sequence of tasks that must be performed on a sequential database can be scheduled in various ways. Schedules will differ with respect to the number of accesses made to peripheral storage devices and the amount of memory space consumed by buffers. Buffer requirements are discussed for task schedules that avoid accesses to peripherals storing the sequential database. The relationship between certain thrifty scheduling policies and loop jamming, a standard code optimization technique, is also identified. Application to UNIX pipelines and to file processing is discussed.

*This work is supported in part by NSF Grant MCS-8003340 and MCS-8103605. Schneider is also supported by a Faculty Development Grant from IBM Corp.

1. Introduction

Consider a *task* to be a computation that obtains input from one or more sequential files and produces one or more sequential files as output. A *task pipeline* is a sequence of tasks where one of the output files of each task is an input file for the next task in the sequence, and no other files are shared by tasks. The *input* to the task pipeline are those files read by tasks in the pipeline but not written by other tasks; the *output* of the task pipeline are those files written by tasks in the pipeline but not read by other tasks.

Task pipelines arise in a variety of contexts. For example, in UNIX[1] [RT78] the pipe operator ("|") allows commands to be combined into a task pipeline. Another common situation in which a task pipeline is useful arises when a sequence of independent tasks must be performed on a database, where each task involves reading and possibly writing all of the records in the database, in the same order.

There are alternative strategies for executing of a task pipeline. One of these is *serial execution*, where each task is run to completion before the next is started. Users are entitled to expect the results of executing a task pipeline to be those that would be obtained by serial execution. Consequently, in this paper we explore task pipeline execution strategies that produce results equivalent to serial execution, but with lower cost.

We seek strategies that minimize the number of operations involving peripheral storage devices, since these operations are relatively time-consuming and can dominate the execution time of a task pipeline. Operations involving input and output files are inescapable, but there are thrifty ways of handling files used only for inter-task communication. To the extent that these *communication files* can be buffered in main memory, costly operations on peripheral devices can be avoided. While it is impractical

[1]UNIX is a trademark of ATT Bell Laboratories.

to retain an entire communication file in main memory at one time, this is unnecessary if execution of tasks can be interleaved. By beginning execution of a task before its predecessor has completed, the entire communication file linking these two tasks need never exist at any one point in time.

Interleaving tasks involves process switching, so this strategy is only attractive if process switching does not involve access to peripheral storage devices. We assume that performing a process switch is inexpensive relative to accessing a peripheral storage device. This assumption is valid when main memory is large enough to store the code for all the processes comprising the task pipeline.

We proceed as follows. In Section 2, we prove that every execution interleaving that arises from concurrent execution of the tasks in a pipeline will produce results equivalent to those of serial execution. Not every interleaving is equally effective in reducing the number of peripheral storage accesses. In Section 3, we address this issue, and give bounds on the amount of memory required to store communication files when interleaved execution is used. In Section 4, we define Record Oriented Scheduling, a scheduling policy that allows interleaved execution with a minimal amount of main memory. Section 5 explores the relationship among Record Oriented Scheduling, loop jamming, and concurrent processing.

2. Interleaved Execution of Task Pipelines

Consider a task pipeline TP consisting of the sequence of tasks T_1, T_2, ..., T_n, which we denote:

$$TP:\ T_1 \mid T_2 \mid \cdots \mid T_n .$$

Each task can be viewed as specifying a sequence of *actions*. A *schedule* of TP is a sequence of the actions of the tasks comprising TP that corresponds to some execution of the concurrent program:

$$C(TP)\text{: } \textbf{cobegin}\ \ T_1\ //\ T_2\ //\ \cdots\ //\ T_n\ \ \textbf{coend}.$$

A prefix of a schedule of TP is called a ***partial schedule*** of TP.

A ***serial schedule*** of TP results from executing the sequential program:

$$S(TP)\text{: } T_1;\ T_2;\ \cdots;\ T_n.$$

For a given task pipeline, two schedules are ***equivalent*** if identical output is produced by each, for the same input.

We show below that every schedule of a task pipeline is equivalent to the serial schedule. This equivalence is due to the synchronization imposed on task execution in $C(TP)$ by communication files. A task attempting to read the r^{th} record from a file containing fewer than r records is delayed.

Lemma 2.1: For any partial schedule s, it is possible to construct a partial schedule s' such that:

(i) s' and s are equivalent and

(ii) the actions in s from T_n, the last task in the pipeline, appear as the last elements of s'.

Proof: The proof is by induction on the number of actions in s.

Base Case: Suppose s consists of a single action. Then $s' = s$ and the lemma follows trivially.

Induction Case: Suppose the lemma is true for any partial schedule consisting of k or fewer actions, and given is a partial schedule s_{k+1} with $k+1$ actions. Let α be the last action appearing in s_{k+1} from T_n, the last task in the pipeline. Interchange α with each action to its right in s_{k+1}, forming partial schedule $s*_{k+1} = s_k\ \alpha$. To see that $s*_{k+1}$ is equivalent to s_{k+1} note the following:

First, delaying α cannot cause α to execute differently. Execution of α is completely determined by the records and program variables it reads. No action after α in s_{k+1} reads any records from the files that α reads because each file is read by only one task and, by construction, α was the last action in that task. No action after α in s_{k+1} modifies any program variables that α reads because tasks do not share storage and α was the last action in its task. Thus, α will not execute differently in $s*_{k+1}$ than it does in s_{k+1}. Therefore, α will write the same record(s) to output files and will leave program variables of T_n in the same state in $s*_{k+1}$ as it did in s_{k+1}.

Second, delaying α will have no effect on the execution of other actions. This is because the output of α is to files that are read by no other task in the pipeline, since α is from the last task in the pipeline.

By the induction hypothesis, it is possible to construct partial schedule s_k' from s_k so that s_k' is equivalent to s_k and all actions in s_k from T_n appear at the end of s_k'. Thus, $s_{k+1}' = s_k' \alpha$ is equivalent to s_{k+1} and satisfies (i) and (ii).

Q.E.D.

Theorem 2.2: Every schedule of a task pipeline is equivalent to the serial schedule of that pipeline.

Proof: The proof is by induction on the number of tasks in the pipeline.

Base Case: Suppose there is only 1 task. Then, the theorem follows because a concurrent program with only one process is a sequential program.

Induction Case: Suppose the theorem is true for all pipelines consisting of t or fewer tasks. Consider any schedule s that results from execution of a given task pipeline with $t+1$ tasks. By Lemma 2.1, it is possible to construct a schedule s' from s, where (i) s' and s are equivalent, (ii) all the actions from the $t+1^{st}$ task appear as the last elements of s'. Thus, we can write $s' = s_a; s_b$ where s_a is a schedule of a task pipeline involving the first t tasks and s_b contains only actions from the $t+1^{st}$ task.

By the induction hypothesis, s_a, which is a schedule of a task pipeline involving the first t tasks, is equivalent to the serial schedule for those tasks. By definition, executing s_b corresponds to the serial execution of a task pipeline consisting of the $t+1^{st}$ task. Thus, by the definition of serial execution, s' is equivalent to the serial execution of the given task pipeline.

Q.E.D.

3. Thrifty Execution

Suppose there is sufficient memory available to accommodate the code for all tasks in a task pipeline and to provide B record-size buffers. If B is large enough then it should be possible to store communication files in memory rather than on peripheral devices. We shall refer to this way of executing a pipeline as *B-thrifty* since it avoids time-consuming accesses to peripheral devices at a cost of memory space for B buffers.

3.1. Implementing Thrifty Execution

In thrifty execution, each communication file is implemented by a queue of buffers named by the filename. Buffers not in use are stored on a special queue named *FREE*. For each queue Q, let $cap(Q)$ be the maximum number of buffers that can be accommodated in Q. We define $cap(FREE) = B$, but assume only that $cap(Q) > 0$ for all other queues Q.[2]

The following procedures allow queues of buffers to be manipulated.

acquire(Q, b)	Delay until queue Q contains at least one buffer. Remove the buffer at the head of Q and assign its address to b.
release(Q, b)	Delay until queue Q contains fewer than $cap(Q)$ buffers. Add the buffer pointed to by b to the tail of Q. Assign Φ to b.

Read and write operations on communication files are implemented using these operations. The read operation

read *rcd* **from** *INP*,

where *INP* is a communication file, is implemented by:[3]

```
acquire(INP, temp);
rcd := DeRef(temp);
release(FREE, temp)
```

The write operation

write *rcd* **to** *OUT*,

where *OUT* is a communication file, is implemented by:

```
acquire(FREE, temp);
DeRef(temp) := rcd;
release(OUT, temp)
```

Read and write operations to other files remain unchanged. However, we do assume

[2]The relationship between $cap(Q)$ and B is discussed below.

[3]$DeRef(temp)$ evaluates to the contents of the buffer pointed to by $temp$.

that after the last record has been read, an *eof* record is returned by the next read operation to that file.

Finally, we assume that tasks are well behaved in the following sense:

B1: A task that reads an *eof* record from an input file does not subsequently attempt to read from that file.

B2: No task writes an *eof* record until it has read one from each of its input files.

B3: A task that writes an *eof* record to any output file, eventually terminates.

B4: No task terminates until it has written an *eof* record to all its output files.

B-thrifty execution of task pipeline TP is defined by:

```
$Th_B(TP)$:  $FREE$ := Queue of $B$ buffers;
             cobegin  $T_1$ // $T_2$ //  ···  // $T_n$  coend
```

The equivalence of this mode of execution and concurrent execution when files are stored on peripheral devices follows from our implementation of read and write operations on communication files in terms of **acquire** and **release**. But for the added possibility of deadlock, our implementations of read and write are equivalent to read and write operations when files are stored on peripheral devices.

3.2. Sufficiently Large B

If B-thrifty execution of a pipeline is attempted and B is not large enough, it is possible that one or more tasks will be delayed forever—the task pipeline will become deadlocked. In this section, we present bounds on B that ensure freedom from deadlock during B-thrifty execution.

Let F_i be the communication file used by adjacent tasks T_i and T_{i+1} in $Th_B(TP)$. The following result will be useful.

Lemma 3.1: During execution of $Th_B(TP)$, if task T_n, the last task in the pipeline, terminates then so do all other tasks.

Proof: The proof is by induction on the number of tasks in the pipeline.

Base Case: If there is one task in the pipeline then the lemma follows trivially.

Induction Case: Suppose the lemma is true for every task pipeline involving t or fewer tasks. Consider a pipeline with $n = t+1$ tasks. If T_n terminates, then by B4 it must have written an *eof* record and by B2 it must have read an *eof* record from F_{n-1}. Thus, T_{n-1} must have written an *eof* record to F_{n-1}, and according to B3 it will terminate. By the induction hypothesis, tasks T_1 through T_{n-2} will terminate if T_{n-1} terminates.

Q.E.D.

The following theorem defines the number of buffers sufficient to avoid deadlock in thrifty execution.

Theorem 3.2: Execution of $Th_B(TP)$ does not deadlock if $B \geq \sum_{1 \leq i < n} cap(F_i)$

Proof: The proof is by contradiction. Suppose there is a deadlock.

Since there is a deadlock, at least one task is blocked and all others are blocked or have terminated.

By Lemma 3.1 we conclude that T_n has not terminated. Therefore, it must be blocked. Tasks block only at **release** and **acquire** operations. Since T_n is the last task in the pipeline, the only operations that can block it are **acquire** on F_{n-1} and **release** on *FREE*. A **release** on *FREE* never causes blocking, so we conclude that T_n is blocked at the **acquire**. This means that F_{n-1} is empty, and, due to B1, that T_{n-1} has not written an *eof* record to F_{n-1}. Since F_{n-1} is empty, *FREE* is not empty, due to our choice of B and the fact that buffers are conserved by our implementations of read and write.

Since T_{n-1} has not written an *eof* record to F_{n-1}, it has not terminated, according to B4. Therefore it must be blocked. It can't be blocked at a write to F_{n-1}, because *FREE* contains at least one buffer and F_{n-1} is empty. Thus, T_{n-1} must be blocked at a read to a communication file.

If $n-1 = 1$ then we have a contradiction since T_1 never reads from a communication file.

If $n-1 \neq 1$ then T_{n-1} is blocked at an **acquire** from F_{n-2}, because a task is never blocked trying to **release** to *FREE*. This means that F_{n-2} is empty and, due to B1, that T_{n-2} has not written an *eof* record to F_{n-1}. This argument is repeated, until we conclude the contradiction that T_1 is blocked at a read on a communication file.

Q.E.D.

B buffers, as defined in Theorem 3.2, are necessary as well as sufficient to avoid deadlock. Necessity is demonstrated by an example:

T_1: *COPY* | T_2: *PREFIX* | T_3: *COPY* where *COPY* simply copies its input file to its output, one record at a time; *PREFIX* writes a header record to its output, and then copies its input file to its output, one record at a time. The output of T_1 is the input of T_2; and the output of T_2 is the input of T_3. Let $cap(F_1) = k$, $cap(F_2) = 1$, and choose $B = k$. Thus, $B < cap(F_1) + cap(F_2)$. The following schedule DS, which is possible under B-thrifty execution, results in deadlock:

DS: Run T_1 long enough to read k records and write k records.

After running DS, T_1 will be blocked because F_1 is full; T_2 will be blocked because *FREE* is empty and the first operation done by T_2 is to write to F_2; and T_3 is blocked because its first operation is to read from F_2, which is empty. Thus, execution of this task pipeline is deadlocked.

4. Record Oriented Scheduling

By careful scheduling, it is possible to reduce the number of buffers required to execute a task pipeline using thrifty execution, without risking deadlock. We show below that only one buffer is required for thrifty execution according to:

Record Oriented Scheduling: Always execute the runnable task T_i with greatest index i.

This means that with Record Oriented Scheduling, a task pipeline can be run without storing communication files on peripheral devices, using negligible storage beyond that needed to keep the tasks themselves resident in memory. Note also that Record Oriented Scheduling can be simply implemented on an operating system that supports preemptive priority scheduling.

Theorem 4.1: Execution of $Th_B(TP)$ using Record Oriented Scheduling does not deadlock if $B = 1$ and $cap(F_i) = 1$ for $1 \leq i < n$.

Proof: The proof is by induction on the number of task switches performed during execution.

Base Case: Suppose no task has been scheduled. Even if no task T_i where $i>1$ is runnable, T_1 will be runnable because it performs **acquire** operations only on *FREE* and **release** operations only on F_1. Initially, *FREE* contains 1 buffer, so the **acquire** won't be delayed, and F_1 is empty, so a **release** won't be delayed either.

Induction Case: Suppose the pipeline has not deadlocked after h task switches have occurred. We now show there is guaranteed to be a runnable task for the $h+1^{st}$ task switch. Two cases are considered:

Case 1: Suppose the buffer is in *FREE*. Let T_i be the task with smallest index that has not terminated. If $i=1$ then by the argument given for the Base Case, T_i is runnable. If $i>1$ then by our choice of T_i, we conclude that T_{i-1} has terminated. Therefore, according to B4, T_{i-1} must have written an *eof* record to F_{i-1}. By hypothesis, the buffer is in *FREE*, so every communication file is empty. Thus, F_{i-1} must be empty. Therefore T_i must have read the *eof* record written by T_{i-1}. Due to B1, T_i cannot subsequently attempt to read from F_{i-1}. If $i=n$ then T_i is runnable because it accesses no other communication file. If $i<n$ then T_i is runnable because it will not be delayed if it executes a write to a communication file: the **acquire** to *FREE* will not block—*FREE* contains 1 buffer; the **release** to F_i will not block—F_i contains no buffers.

Case 2: Suppose the buffer is not in *FREE*. Let the buffer be in F_i. Thus, T_i is now blocked and must have been the last task to run. Also, the last action completed by T_i involving a communication file must have been to write a record to F_i. Prior to that write, (1) the buffer was in *FREE*, since write first does an **acquire** from *FREE*, and (2) T_i was the runnable task with highest index, according to the Record Oriented Scheduling rule. From (2) we deduce that T_{i+1} was not runnable at the time T_i was last scheduled, and from (1), that T_{i+1} was not waiting to perform a write to a communication file at that time. Therefore, T_{i+1} must have been waiting to read from communication file F_i. F_i now contains a record, however, so T_{i+1} is runnable.

Q.E.D.

5. Compiler Implemented Record Oriented Scheduling

In data processing applications that access a single sequential master file, a well-known strategy is to combine several separate tasks into a single *job* and make one pass through the master file. The result is a task pipeline where the updated master file record written by each task (but the last) is internally treated as the input of the next task in the job. The outputs of the pipeline are the reports produced by each task and the updated master file written by the last task. For example, in typical accounts-receivable systems, a single pass through the file posts transactions and produces invoices and summary reports.

The logical extension of this concept is to separate the record-accessing function from the individual tasks. This results in a generalized record-accessing program into which the user can insert multiple independent tasks. In effect, this provides automatic batching and can yield substantial reductions in the number of accesses made to peripheral devices. This was, in fact, done in some of the early file management systems (circa 1970). Since these were commercial systems that were not discussed or cited in the literature, it is difficult to establish precedence,[4] but one of the first to emphasize such a facility was ASAP [CMM72]. This system permitted both update and retrieval, using an English-like source language. Some examples of ASAP tasks are shown below.

Example 1:

```
FOR ALL ACCOUNTS WITH AREA = 'NYC'
   AND SALESMAN = 'SMITH, J.' AND BALANCE > 0,
   PRINT A LIST OF NAME, BALANCE, OVERDUE_AMOUNT, YTD_SALES,
      ORDERED BY DISTRICT, ORDERED BY YTD_SALES DECREASING.
```

Example 2:

```
FOR ALL OF THE ACCOUNT RECORDS
   SELECTED BY KEY IN THE FEB_TRANSACTIONS,
      FORMATTED BY TRANSACTION_LAYOUT,
   UPDATE THE RECORD.
```

Example 3:

```
FOR THE FIRST ACCOUNT WITH NAME = 'BATH IRON WORKS',
   DELETE THE RECORD.
```

[4]The earliest mention we are aware of is [K68].

Example 4:

```
FOR ALL THE ACTIVE ACCOUNTS:
   SET OLD_BALANCE = BALANCE;
   SET BALANCE = 0;
   IF OLD_BALANCE > 0 AND PAYMENTS = 0,
      SET OVERDUE_AMOUNT = OLD_BALANCE,
      INCREASE BALANCE BY OVERDUE_AMOUNT, END;
   FOR ALL ITEMS WITH AMOUNT NOT = 0,
      INCREASE BALANCE BY AMOUNT, END;
   IF BALANCE > 0, PRINT A REPORT STATEMENT,
      ORDERED BY ZIP_CODE, ON FORM BILL_STOCK, END;
   INCREASE SUM_OF_BALANCES BY BALANCE.
```

In ASAP, independent tasks to be performed on the same file that are submitted together are batched automatically. Each ASAP task A is really the specification for a task:

```
asap(A):  read rcd from $INP_i$;
          while $rcd \neq eof$ do
                 $r(A)$;
                 read rcd from $INP_i$;
          write rcd to $INP_{i+1}$   /* writes eof */
```

where $r(A)$ is the result of the ASAP compiler

(1) translating DELETE THE RECORD clauses (see example 3) to code that sets an additional *deleted* field in the record to *true*,[5]

(2) translating FOR clauses (see examples 1-4) into **if** statements that check whether a record is one that should be processed by A and does not already have $deleted = true$, and

(3) appending a statement

write *rcd* **to** INP_{i+1}

to the end of the translated code. If in the last task in the batch, the **write** is guarded by $deleted \neq true$.

Execution of a batch of ASAP tasks $A_1, A_2, ..., A_n$ is equivalent to executing the task pipeline

$$TP_{ASAP}: \quad asap(A_1) \;\mid\; asap(A_2) \;\mid\; \cdots \;\mid\; asap(A_n).$$

[5]Actually, the *deleted* field is unnecessary. A DELETE THE RECORD clause can be translated to a (forward) jump to the end of the loop body. This was not realized by the designers of ASAP.

However, using 1-thrifty execution with Record Oriented Scheduling, then from the definition of Record Oriented Scheduling, we see that execution of TP_{ASAP} turns out to be the same as:

```
ROS: read rcd from INP;
     while $rcd \neq eof$ do;
          $LoopJam(\tau(A_1), \tau(A_2), \ldots, \tau(A_n))$;
          read rcd from INP;
     write rcd to OUT   /* writes eof */
```

where *LoopJam* denotes the code that results from performing "loop jamming" [AU77], a standard code optimization technique where the body of two or more loops are combined if the i^{th} iteration of one loop does not depend on results computed by the j^{th} iteration, $j > i$, of a preceding loop.

The ASAP compiler automatically performs this optimization when it batches tasks. ASAP programs therefore enjoy the benefits of 1-thrifty execution with Record Oriented Scheduling, without incurring the complexity of executing tasks concurrently. A batch of tasks in ASAP are executed by making a single pass through the input file with a single buffer being passed from one task to the next. Although it was not well-understood when ASAP was introduced[6], it is now clear that the ASAP compiler could batch tasks in this way only because the ASAP source language offered the user no way to specify a task that was not subject to loop jamming.

Several other early file management systems also provided task batching, but only for retrieval tasks (see SRS360 [C73]). Such a facility—often called multiple report generation—is also present in some current database management systems. For example, NATURAL/ADABAS [Nxx] is very similar to ASAP in concept and source language, but curiously, allows only retrieval tasks to be batched. No contemporary database

[6]One of the designers of ASAP is an author of this paper.

system [KW81] seems to offer generalized automatic task batching.

6. Discussion

We have analyized how to execute task pipelines so as to avoid storing on peripheral devices files used strictly for communication between tasks. We have derived bounds for the number of buffers required when the tasks of a pipeline are simply executed concurrently. And, we presented Record Oriented Scheduling, a scheduling policy that allows tasks to be scheduled in an environment with space for only 1 buffer.

The basis for thrifty execution of pipelines is the way pipes are implemented in UNIX. There, piped commands are executed as asynchronous processes that share a (communication) file. However, UNIX files are streams of characters and are automatically buffered in memory to the extent possible. The combination of a small record size and automatic buffering makes concern about the size of communication files unnecessary. However, generalization of pipes to accommodate larger records—an idea being explored by a number of researchers—requires the analysis of buffer requirements done in this paper.

The basis for Record Oriented Scheduling is the way ASAP compiles batches of tasks. Our discovery of how to relate loop jamming as used by ASAP to Record Oriented Scheduling is satisfying. Thus, while we have discovered no new ways of executing task pipelines, we now understand why some important old ways work and how they are related.

A number of interesting questions arise when tasks in a pipeline can read and write more than one communication file, and we are presently pursuing them as part of the design of a new UNIX shell. Two cases that appear to merit further study are (1) when the task communication graph is acyclic and (2) when loops are permitted in the task communication graph. The bounds derived for thrifty execution in this paper do not apply to such extended pipelines—we have counterexamples to prove this for B-thrifty execution both with and without Record Oriented Scheduling. Nevertheless, these more general task pipelines present some intriguing possibilities, especially in an operating system command language.

Acknowledgments

We would like to thank two referees whose questions lead to a generalization of some of our earlier results.

References

[AU77] Aho, A., J. Ullman. *Principles of Compiler Design.* Addison Wesley, Mass. 1977.

[CMM72] Conway, R., W. Maxwell and H. Morgan. Selective Security Capabilities in ASAP — A File Management System. *AFIPS Conf. Proc. Vol. 40* (SJCC 1972), 1181-1185.

[C73] Simultaneous Reporting System. *SRS360 Reference Manual,* Chilton Computer Company, 1973.

[K68] Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Mass. 1968, pp. 194-196.

[KW81] Krass, P., and H. Wiener. The DBMS Market is Booming. *Datamation, Vol. 27,* No. 10 (September 1981), 153-170.

[Nxx] *NATURAL Reference Manual.* ADABAS Systems, Software AG of North America.

[RT78] Ritchie, D., and K. Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal Vol. 57,* No. 6, July-August 1978, 1905-1929.