Effective Control for Pipelined Computers[†]

by

Edward S. Davidson
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Leonard E. Shar
Hewlett-Packard Company
Cupertino, California

A. Thampy Thomas
Intersil, Inc.
Cupertino, California

Janak H. Patel
Coordinated Science Laboratory
University of Illinois
Urbana, Illnois 61801

I. PIPELINES

A pipelined system, or subsystem, is broadly defined as a system with several resources, called segments, controlled in such a manner that several segments may be busy simultaneously. As such, it does not differ from a parallel system. However, in most common usage, each segment is specialized to be capable of performing only a particular class of subtasks so that a single simple task must flow from segment to segment for its execution. This flow is normally synchronous. Segment specialization results in the inherent economy of pipelined computers over parallel computers of equivalent performance with general purpose resources.

Pipelines are becoming increasingly common, [1, 2,3,4,5], even in small and medium sized computers with some proposals involving more than simple instruction prefetch [6,7]. Although some simplified system analysis is available, [8,9](also [10] which has some inaccuracies), and some detailed logic design principles developed, [11,12], there has been no comprehensive theory for their effective control. We here propose such a theory which encompasses a broad class of pipelines (some of which has been described in [13,14,15,16]).

In addition to accepting the connotations of pipelining above, it is further assumed that the precise time-flow pattern of a task through the segments is known at the time of initiating the task in the pipeline. It is also assumed in Sections II and III that the only control decision which can be made is when to initiate each task in the pipeline. Once initiated, a task must observe its time-flow pattern precisely without encountering any delays between segments. Buffering between segments is discussed in Section IV.

A task flow pattern may be displayed in a reservation table (see Fig. 1a). Rows correspond to segments and columns to units of time following initiation. An x is placed in cell (i,j) whenever the task requires segment i at time unit j. Any pattern of x's is legal. Multiple x's in a column indicate parallel reservations. Multiple x's in a row indicate slow segments or flow feedback.

The control problem is to schedule queued tasks awaiting initiation in the pipeline so as to achieve high throughput and avoid collisions; i.e., two or more tasks attempting to use the same segment at the same time. The (propagation) delay of a pipeline is the number of time units between initiation and termination of a task; the latency is the number of time units between successive initiations. Note that throughput, in tasks per time unit, is the reciprocal of average latency for a large number of tasks and is independent of delay.

For single function units, all tasks use the same reservation table. For a p-function unit, p reservation tables pertain. The reservation table for function K uses K's in place of x's. An overlaid reservation table (see Fig. 2a) formed from simply overlaying the p reservation tables is useful here. Each task requesting initiation must have an associated function tag which selects its reservation table.

II. SINGLE FUNCTION UNITS

A lower bound, ℓ, on achievable minimum average latency (MAL) is the maximum number of x's occurring in any single row of the reservation table (or better, occurring in any row of any reservation table which yields the same collision vector -- see below). If MAL were less than ℓ, the segment with ℓ x's would be busy more than 100% of the time, which is impossible. Achieving the lower bound means that at least one segment is busy all the time. For some pipelines, the lower bound cannot be achieved. What then is achievable and how can the function unit be controlled?

Collisions occur when two tasks are initiated with an initiation interval equal to the column distance between two x's in some row of the reservation table. Thus the set of distances between each pair of x's in the same row of the reservation table, over all rows, is the forbidden initiation interval set. The collision vector, $c_n c_{n-1} \cdots c_1$, where n is the largest set element, has $c_i = 1$ if i is in the set and $c_i = 0$ otherwise.

Collisions may be avoided by using an n-bit shift-right register which shifts once each time unit introducing 0's on the left. An initiation is allowed next time if and only if the rightmost bit of the shift register is 0. The collision vector is ORed into the shift register immediately after the shift accompanying an initiation. The shifted state prevents future task collisions with previously initiated tasks; the collision vector

[†]This research was supported by National Science Foundation Grants GJ-35584X and GJ-40584.

prevents collisions with the task just initiated.

A modified state diagram (see Fig. 1b) has states which correspond to all the shift register states that can immediately follow an initiation. Arcs correspond to initiations and are labeled with the latency of the initiation (initiation interval between it and the immediately preceding task). Every state has an outbound arc to the initial state, which is the collision vector itself, labeled with all integers greater than n. For convenience these are represented by a single arc simply inbound to the initial state labeled $n+1^+$. The modified state diagram contains all possible collision-free latency sequences of the function unit.

Cycles indicate steady-state sustainable sequences provided that the input queue is never empty at initiation time (heavy load). A greedy cycle always uses the outbound arc with the lowest label from each of its states. Greedy control follows greedy cycles and always corresponds to initiating waiting tasks at the first opportunity. It is simple to implement with the shift register controller, but is often nonoptimum under heavy load. It can be shown that an upper bound on the average latency of any greedy cycle is the number of 1's in the collision vector plus 1. The upper bound equals the lower bound if and only if the collision vector is periodic; i.e., $c_i = 1$ if and only if $i = j, 2j, \ldots, mj = n$, for some j.

Optimum control follows MAL cycles which are either simple cycles or complex cycles made up from simple MAL cycles which share states. A straightforward algorithm has been found which involves a branch-and-bound search of the modified state diagram and produces all simple MAL cycles. A simple optimum controller uses an end-around shift register with 0's spaced to correspond to the desired MAL cycle latency sequence. It is always true that lower bound $\leq$ MAL $\leq$ all greedy cycle average latencies $\leq$ upper bound (see Fig. 1).

If constant latency is required, a latency of m is achievable if and only if $c_m, c_{2m}, c_{3m}, \ldots$ are all 0 ($c_k \triangleq 0$ whenever $k > n$). The minimum constant latency, however, may even exceed the upper bound on greedy latency (see Fig. 1). Any constant latency $\geq$ the reservation table lower bound can be achieved by using noncompute delays (see Section IV).

III. MULTIFUNCTION UNITS

For a p-function unit, a computation with function tag Q may now collide with a computation with a computation with function tag R initiated t time units previously if and only if some row of the overlaid reservation table has a Q in column c (for some c) and an R in column c+t. A pXn collision matrix, M_R (see Fig. 2a), is formed with row Q containing a 1 in column t for each collision-causing t. If desired $t = 0$ may also be considered, in which case M_R becomes p x (n+1), M_R indicates forbidden initiation intervals for all functions initiated henceforth resulting from initiating a computation with function tag R. Thus p collision matrices are formed and n is the largest overall collision-causing value of t. An analogous shift register controller contains p n-bit shift registers. Initiation of a computation with function tag Q is allowed next time if and only if the least significant bit of shift register Q is 0. M_Q is OR'ed into the shift register bank immediately after the shift accompanying an initiation of a computation with function tag Q.

A unified state diagram (see Fig. 2b) is analogously constructed with shift register bank states immediately following initiations. Arcs are labeled with the latency and function tag of the initiation. Greedy control initiates tasks, in the order in which they arrive at the input queue, at the first possible opportunity. The lower and upper bounds may be generalized for multifunction units only for some given function mix; i.e., the fraction of the tasks to be performed which is associated with each function tag. Greedy control is useful under light load or under an unstable function mix. Performance under greedy control does not necessarily reach optimum, but is generally superior to the common practice of never allowing functions with distinct function tags to be simultaneously in process.

MAL control is defined only for some given function mix with a task arrival pattern which allows "sufficient" freedom in scheduling; i.e., when the controller desires a particular function there is always a computation with the appropriate tag waiting in the input queue. The MAL cycles are those with the lowest average latency among cycles which match the given function mix. However, it is possible that no MAL cycle for some mix is a simple cycle: the mix may be attainable only by some linear combination of mixes of simple cycles. While the number of simple cycles is finite, the number of complex cycles is infinite. The branch-and-bound MAL cycle-finding algorithm is thus not applicable.

An easily solvable linear program has been formulated which, given a mix and a set of simple cycles, each characterized by its latency and mix, will find a linear combination of the cycles which achieves the given mix with MAL. One procedure would be to give the linear program all simple cycles of the unified state diagram.

However, only certain simple cycles are ever combined to form a complex MAL cycle. These "good cycles" are the simple cycles which attain MAL for their own function mix. The branch-and-bound algorithm does generalize to find all good cycles. Furthermore, some good cycles are redundant in the sense that their mix can be realized with equal latency by a linear combination of other good cycles. The modified branch-and-bound algorithm actually finds an irredundant good cycle set. This set always contains a MAL cycle for each "pure" mix (100% of one function tag). The good cycle set is thus complete and spans all possible mixes by linear combination. Only such a good cycle set need be given to the linear program. The branch-and-bound algorithm is run only once for a multifunction unit. The linear program is run once per desired function mix (see Fig. 2).

It has been implicitly assumed here that any MAL linear combination of good cycles forms a complex cycle; i.e., the good cycles have sufficient states in common. Such is unfortunately not the case: some may even have no states in common with the rest. However, any cycle transition may be achieved by at worst increasing a single latency to n+1. In such cases, it is desirable to make such transitions as seldom as possible. How often this must be done depends on the arrival pattern at the input queue of tasks with appropriate function tags.

IV. PIPELINES WITH INTERNAL BUFFERING

Many of the previously described pipelines have a MAL which does not meet the reservation table lower bound. Noncompute delays; i.e., segments inserted in the flow for delay only, can always be used to modify the reservation table so that it does achieve its lower bound. However, such delays have an associated cost. Furthermore, for multifunction units, there may be no insertion pattern which satisfies all mixes.

An alternative modification allows substantial freedom in adapting to changes in the function mix and arrival pattern while producing high performance. Pipeline delay may, however, increase. Temporary storage buffers are employed between segments (sometimes actually between steps or "segment usages"). These buffers also have an associated cost and require distributed control points for the function unit. Many buffer priority schemes can be used, including 1. FIFO-global; 2. LIFO-global; 3. FIFO-step, LPF; 4. LIFO-step, LPF; 5. FIFO-step, MPF; 6. LIFO-step, MPF. In FIFO-global a task has priority over all tasks initiated later than itself; in LIFO-global a task has priority over all tasks initiated earlier than itself. In the rest of the schemes FIFO-step and LIFO-step indicate respectively, a First-In-First-Out queue and a Last-In-First-Out stack at each computation step; LPF and MPF stand for Least-Processed-First and Most-Processed-First to select between several steps which share a physical segment.

Priority schemes 2,3, and 4 have been shown to meet lower bound latency for a p-function unit with any initiation cycle which has that latency.

For scheme 2, buffers are associated with segments rather than steps. The segment i buffer never exceeds $\left(\sum_{1\le j\le p} m_{ij}n_j\right) - \max_{1\le j\le p} n_j$ tasks, where n_j is the number of tasks in the cycle with function tag j and m_{ij} is the number of times a task with function tag j uses segment i. The wait time at any step never exceeds $\left(\sum_{1\le j\le p} m_{ij}n_j\right) - 1$. Note that $\sum_{1\le j\le p} m_{ij}n_j \le t$ where t is the number of time units in a cycle. Schemes 2, 3, and 4 are equivalent for constant latency cycles, while schemes 1 and 5 are equivalent in all cases. Schemes 1, 5, and 6 seem to produce lower bound latency, but this is still a conjecture.

V. CONCLUSION

This purpose of this paper is to demonstrate that even quite complicated pipelines of the classes modeled here can be effectively and simply controlled. One should not fear to employ pipelined multifunction units with complex reservation tables. Internal buffering can be used to advantage in many situations and provides high performance and freedom of control at a price.

It can be argued that pipelines without feedback, which duplicate segments to achieve only one x per reservation table row, achieve MAL = 1 and never involve a percentage cost increase greater than their performance increase. However, the cost increase may be significant and it may be that the work load is not heavy enough to utilize the increased performance capability. Then similar subtasks should share segments. Selective segment duplication and use of noncompute delays can often, however, dramatically increase performance. Note also that nonfeedback design precludes using multifunction units which share segments.

It is implicit that dependencies between tasks are resolved before a task is placed in the input queue. Effective use of pipelining is possible only when such dependencies do not often empty the input queue. The cures for dependency degradation include changing to a multiple task stream environment, task stream restructuring, and controlling pipeline delay.

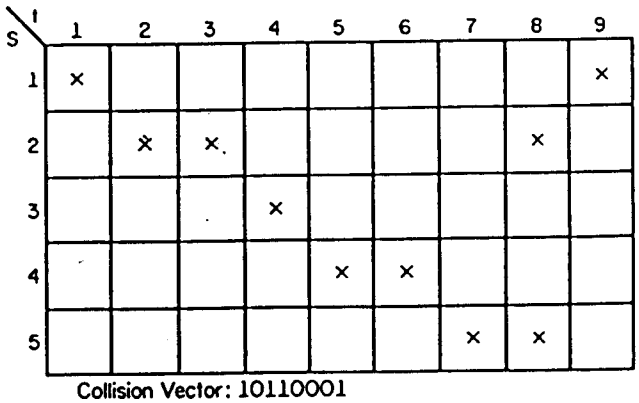
Intersegment switching is a common problem which can be economically solved by setting switches directly from a modified shift register controller which stores an initiation history. Initiation can then be controlled by using p OR gates to generate the least significant bits of the previous shift register controller.

Finally, the pipelines modeled here are generally applicable beyond computational systems as can be seen by thinking of tasks not as "computations," but "processes." The techniques employed here compare favorably to classical job shop-flow shop scheduling which usually requires unbounded "wait in process"; i.e., internal buffering. Some work has been done on "no wait in process" shops [17]. The complexity of our scheduling techniques depend only on the number of functions, p, not on the actual (finite) number of tasks. Greedy control is suggested for light load and optimal (MAL) control for heavy load for an unbounded number of jobs. Our techniques do, however, depend on and capitalize upon knowing task flow patterns precisely and knowing something about task arrival patterns and function mixes.

REFERENCES

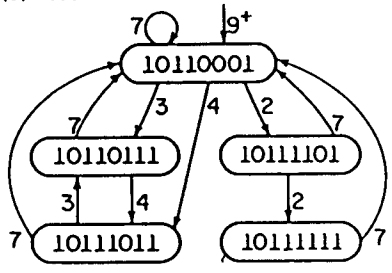
1. Watson, W. J. "The TI ASC - A highly modular and flexible super computer architecture," Proc. FJCC 1972, pp. 221-228.
2. Stephenson, C. "Control of a variable configuration pipelined arithmetic unit," Proc. 11th Annual Allerton Conf. on Circuit and System Theory, Oct. 1973, pp. 558-567.
3. Several papers on IBM 360/91, IBM Journal of R and D, Jan. 1967, pp. 8-53.

4. Session 1: STAR-100, Proc. Compcon '72, pp. 1-16.
5. Ibbett, R. N. "The MU5 instruction pipeline," The Computer Journal, Vol. 15, No. 1, pp. 42-50.
6. Shar, L. E. and Davidson, E. S. "A multiminiprocessor system implemented through pipelining," Computer, Feb. 1974, Vol. 7, No. 2, pp. 42-50.
7. Parasuraman, B. "Pipelined architectures for microprocessors," Proc. Compcon Fall '74, pp. 225-228.
8. Chen, T. C. "Parallelism, pipelining and computer efficiency," Computer Design, Jan. 1971, pp. 69-74.
9. Chen, T. C. "Unconventional superspeed computer systems," Proc. SJCC, 1971, pp. 365-372.
10. Flynn, M. J. "Some computer organizations and their effectiveness," IEEE Trans. Comput., Sept. 1972, Vol. C-21, pp. 948-960.
11. Larson, A. G. and Davidson, E. S. "Cost-Effective design of special-purpose processors: a fast Fourier transform case study," Proc. 11th Annual Allerton Conf. on Circuit and System Theory, Oct. 1973, pp. 547-557.
12. Hallin, T. G. and Flynn, M. J. "Pipelining of arithmetic functions," IEEE Trans. Comput. Aug. 1972, Vol. C-21, pp. 880-886.
13. Davidson, E. S. "The design and control of pipelined function generators," Proc. 1971 Int. IEEE Conf. on Systems, Networks and Computers, Oaxtepec, Mexico, Jan. 1971.
14. Shar, L. E. "Design and scheduling of statically configured pipelines," Tech. Report No. 42, Digital Systems Lab., Stanford University, Sept. 1972.
15. Thomas, A. T. and Davidson, E. S. "Scheduling of multiconfigurable pipelines," Proc. 12th Annual Allerton Conf. on Circuit and System Theory, Oct. 1974, pp. 658-670.
16. Davidson, E. S. "Scheduling for pipelined processors," Proc. 7th Annual Hawaii Intl. Conf. on Systems Sciences, Jan 1974, pp. 58-60
17. Reddi, S. S. and Ramamoorthy, C. V. "A Scheduling Problem," Operational Res. Quarterly, Sept. 1973, Vol. 24, No. 3, pp. 441-446.



Collision Vector: 10110001

(a) Reservation Table



(b) Modified State Diagram

Lower Bound = 3

Minimum Average Latency = $3\frac{1}{2}$

Cycle: (3,4)

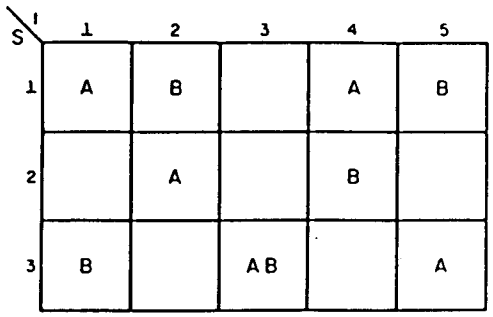
Greedy Latencies = $3\frac{2}{3}$, $3\frac{1}{2}$

Cycles: (2,2,7), (3,4)

Upper Bound = 5

Minimum Constant Latency = 7

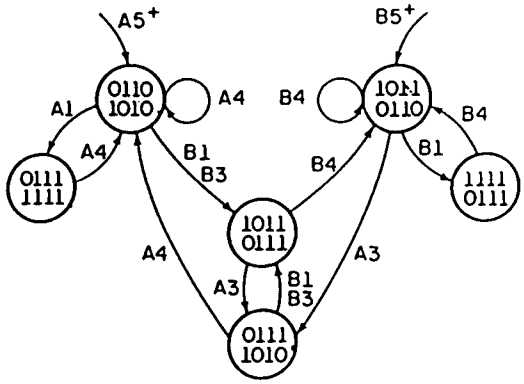
Fig. 1. Single Function Unit



Collision Matrices:

$$M_A = \begin{bmatrix} 0110 \\ 1010 \end{bmatrix} \quad M_B = \begin{bmatrix} 1011 \\ 0110 \end{bmatrix}$$

(a) Overlaid Reservation Table



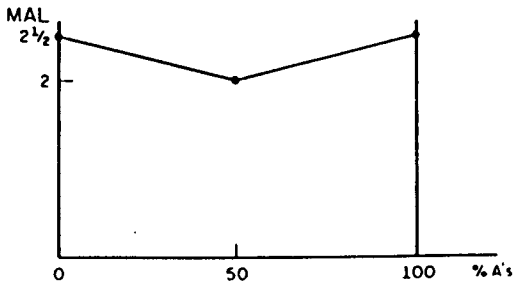
(b) Unified State Diagram

Irredundant Good Cycle Set:

(A1, A4) Av. Latency $2\frac{1}{2}$

(B1, B4) Av. Latency $2\frac{1}{2}$

(B1, A3) Av. Latency 2



(c) Convex Hull for Linear Program

Fig. 2. Multifunction Unit