vol. 33, pp. 493-506, June 1984.
[6] N.K. Jha, "A totally self-checking checker for Borden's code," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 731-736, July 1989.
[7] D. Nikolos, A. Paschalis, T. Haniotakis, and G. Laskaris "Totally self-checking checkers for optimal t-unidirectional error detecting codes," *Proc. 13th Int'l Conf. Fault-Tolerant Systems and Diagnostics*, pp. 326-331, 1990.
[8] C.V. Friedman, "Optimal error detecting codes for completely asymmetric binary channel," *Information and Control*, vol. 5, pp. 64-71, Mar. 1962.
[9] A.M. Paschalis, D. Nikolos, and C. Halatsis, "Efficient modular design of TSC checkers for m-out-of-2m codes," *IEEE Trans. Computers*, vol. 37, pp. 301-309, Mar. 1988.
[10] M.A. Marouf and A.D. Friedman, "Design of self-checking checkers for Berger codes," *Dig. of Papers, Eighth Int'l Symp. Fault-Tolerant Computing*, pp. 179-184, 1978.
[11] T. Haniotakis, A. Paschalis, and D. Nikolos, "Efficient totally self-checking checkers for a class of Borden codes," DEMO Report 93/25, NCSR Demokritos.
[12] G. Laskaris, T. Haniotakis, A. Paschalis, and D. Nikolos, "New design method for low-cost TSC checkers for 1-out-of-n and (n – 1)-out-of-n codes in MOS implementation," *Int'l J. Electronics*, vol. 69, no. 6, pp. 805-817, 1990.
[13] N. Gaitanis and C. Halatsis, "A new design method for m-out-of-n code checkers.," *IEEE Trans. Computers*, vol. 32, pp. 273-283, Mar. 1983.
[14] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, Mass.: Addison-Wesley Publishing Company, 1985.

Efficient Partitioning of Sequences

Bjørn Olstad and Fredrik Manne

***Abstract*—We consider the problem of partitioning a sequence of n real numbers into p intervals such that the cost of the most expensive interval, measured with a cost function f is minimized. This problem is of importance for the scheduling of jobs both in parallel and pipelined environments. We develop a straightforward and practical dynamic programming algorithm that solves this problem in time $O(p(n - p))$, which is an improvement of a factor of log p compared to the previous best algorithm. A number of variants of the problem are also considered.**

***Index Terms*—Partitioning, dynamic programming, the MinMax problem, parallel processing, multiple regression.**

I. Introduction

The scheduling of jobs to processors so as to minimize some cost function is an important problem in many areas of computer science. In many cases, these problems are known to be NP-hard [6]. Thus if scheduling problems are to be solved optimally in polynomial time, they must contain enough restrictions to make them tractable. In this paper we will consider one such problem. To motivate why this particular problem is of interest consider the following example from Bokhari [3].

In communication systems it is often the case that a continuous stream of data packages have to be received and processed in real time. The processing can among other things include demodulation, error correction and possibly decryption of each incoming data package before the contents of the package can be accessed [7]. Assume that n computational operations are to be performed in a pipelined fashion on a continuous sequence of incoming data packages. If we have n processors we can assign one operation to each processor and connect the processors in parallel. The time to process the data will now be dominated by the processor that has to perform the most time consuming operation. With this mapping of the operations to the processors, a processor will be idle once it is done with its operation and have to wait until the processor that has the most time consuming operation is done, before it can get a new data package. This is inefficient if the time to perform each task varies greatly. Thus to be able to utilize the processors more efficiently we get the following problem: Given n consecutively ordered tasks, each taking $f(i)$ time, and p processors. Partition the tasks into p consecutive intervals such that the maximum time needed to execute the tasks in each interval is minimized. In [3] it is also described how a solution to this problem can be used in parallel processing as compared to pipelined.

The outline of this paper is as follows: In Section II, we describe the main partitioning problem and give an overview of recent work. In Section III, we develop a new efficient algorithm for solving this problem. A generalization of the partitioning problem is described and solved in Section IV. In the final section we summarize and point to areas of future work.

II. A Partitioning Problem

We will in this section give a formal definition of the main partitioning problem and also recapitulate previous work. The problem as stated in [11] is as follows.

Manuscript received Nov. 30, 1993; revised Sept. 28, 1994.

B. Olstad is with the Department of Computer Systems and Telematics, The Norwegian Institute of Technology, N-7034 Trondheim-NTH, Norway; e-mail: Bjoern.Olstad@idt.unit.no.

F. Manne is with Norsk Hydro a.s., N-5020 Bergen, Norway; e-mail: fmanne@bg.nho.hydro.com.

To order reprints of this article, e-mail: transactions@computer.org, and reference IEEECS Log Number C95122.

Let the two integers $p \le n$ be given and let $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ be a finite ordered set of bounded real numbers. Let $R = \{r_0, r_1, ..., r_p\}$ be a set of integers such that $r_0 = 0 \le r_1 \le ... \le r_{p-1} \le r_p = n$. Then R defines a partition of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ into p intervals: $\{\sigma_{r_0}, \sigma_1, ..., \sigma_{r_1-1}\}, \{\sigma_{r_1}, ..., \sigma_{r_2-1}\}, ..., \{\sigma_{r_{p-1}}, ..., \sigma_{r_p-1}\}$. If $r_i = r_{i+1}$ then the interval $\{\sigma_{r_i}, ..., \sigma_{r_{i+1}-1}\}$ is empty. We will use the shorthand notation $[i, j] = \{\sigma_i, \sigma_{i+1}, ..., \sigma_j\}$ for intervals in the partition R.

Let f be a function, defined on intervals taken from the sequence $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$, that satisfies the following two conditions of nonnegativity and monotonicity:

$$f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) = f([i, j]) = f(i, j) \ge 0 \tag{1}$$

for $0 \le i, j \le n-1$, with equality if and only if $j < i$.

$$f(i+1, j) < f(i, j) < f(i, j+1) \tag{2}$$

for $0 \le i \le j < n-1$. In addition we will only assume the following mild conditions on the computability of the set-function f to be satisfied:

1) The function $f(\sigma_i)$ can be computed in $O(1)$ time and
2) Given $f(i, j)$ we can calculate f in $O(1)$ time, when either i or j has been increased or decreased by one.

The problem is then

MinMax Find a partition R such that the associated cost

$$\max_{i=0}^{p-1}\{f(r_i, r_{i+1}-1)\}$$

is minimum over all partitions of $\{\sigma_0, ..., \sigma_{n-1}\}$.

Bokhari [3] presents the MinMax problem and gives an $O(n^3p)$ algorithm using a bottleneck-path algorithm. Anily and Federgruen [2] and Hansen and Lih [8] independently presented the same dynamic programming algorithm with time complexity $O(n^2p)$. Manne and Sørevik [11] then presented an $O(p(n-p)\log p)$ algorithm based on iteratively improving a given partition. They also described a bisection method based on a simple $O(n)$ feasibility test for finding an approximate solution which runs in time $O(n \log (f(0, n-1)/\epsilon))$, where ϵ is the desired precision.

Frederickson [4], [5] has studied partitioning of trees with parametric search techniques and gives a good account on related work. Frederickson gives an $O(n)$ solution to the MinMax problem in the special case where $f(i, j)$ is computable in $O(1)$ time for all i and j after an $O(n)$ preprocessing algorithm.

In the next section we show how the dynamic programming algorithm first presented by Anily and Federgruen can be improved to run in $O(p(n-p))$ time.

III. A New Algorithm

In this section we describe a new efficient dynamic programming algorithm for solving the MinMax problem. We start by describing the algorithm by Anily and Federgruen.

Let $g(i, k)$ be the cost of the most expensive interval in an optimal partition of $[i, n-1]$ into k intervals where $1 \le k \le p$ and $0 \le i \le n$. The cost of the optimal solution is then given by $g(0, p)$. Once $g(0, p)$ is known, the positions of the delimiters can be obtained by a straightforward $O(n)$ calculation that grows intervals from left to right. A delimiter is inserted each time $f(i, j)$ has exceeded $g(0, p)$. The following boundary conditions apply to g:

$$g(i, 1) = f(i, n-1) \tag{3}$$

$$g(i, n-i) = \max_{i \le j < n} f(j, j) \tag{4}$$

Note that for $n - i < k \le p$ we have $g(i, k) = g(i, n-i)$.

The following recursion, first presented by Anily and Federgruen [2], shows how $g(i, k)$ can be computed for $2 \le k < n-i$:

$$g(i, k) = \min_{i \le j \le n-k} \max\{f(i, j), g(j+1, k-1)\} \tag{5}$$

This formula suggests that if one has access to each value of $g(j+1, k-1)$, $i \le j \le n-k$, then $g(i, k)$ can be computed by looking up $n-k-i+1$ values of g and by calculating $n-k-i+1$ values of f. This gives a total time complexity of $O(n^2p)$ and a space complexity of $O(n)$ to compute $g(0, p)$.

In the rest of this section we show how $g(0, p)$ can be computed in $O(p(n-p))$ time. This result is due to the fact that $g(i, k)$ is decreasing in i. As a consequence the optimal value of j in (5) increases as i increases. We use this monotonicity to obtain the given speed improvements. The approach is similar to the one found in [12]. The reader is referred to [1], [13] and the references therein for a recent account on algorithmic improvements on dynamic programming algorithms including array searching techniques.

If $R = \{r_0 = i, r_1, ..., r_k = n\}$ defines a partitioning of $[i, n-1]$ of cost $g(i, k)$ we will say that R is *implied* by $g(i, k)$.

It can easily be shown that $g(i, k)$ increases monotonically as i decreases.

Lemma 1. *Let i, i' be integers such that $0 \le i \le i' \le n$. Then $g(i, k) \ge g(i', k)$ for $1 \le k \le p$.*

From Lemma 1 it follows that for fixed k the function g increases monotonically in the size of the interval to be partitioned. The following lemma shows how in certain cases we can compute $g(i, k)$ using the first interval $[i+1, j]$ in a partitioning implied by $g(i+1, k)$ and the value of $g(j+1, k-1)$.

Lemma 2. *Let $[i+1, j]$ be the first interval in a partitioning implied by $g(i+1, k)$, $k > 1$. If $f(i, j) \le g(j+1, k-1)$ then $g(i, k) = g(j+1, k-1)$.*

Proof. If $f(i, j) \le g(j+1, k-1)$ then $f(i+1, j) < g(j+1, k-1)$. Since $g(i+1, k) = \max\{f(i+1, j), g(j+1, k-1)\}$ we have $g(i+1, k) = g(j+1, k-1)$. Together with $g(i, k) \le \max\{f(i, j), g(j+1, k-1)\} = g(j+1, k-1)$ this shows that $g(i, k) \le g(i+1, k)$. From Lemma 1 we know that $g(i, k) \ge g(i+1, k)$ and it follows that $g(i, k) = g(i+1, k)$ and thus $g(i, k) = g(j+1, k-1)$. □

Note that Lemma 2 could also have been stated: If $f(i, j) \le g(i+1, k)$ then $g(i, k) = g(i+1, k)$. The present formulation was chosen in order to emphasize the relationship to (5).

We now discuss how $g(i, k)$ can be computed efficiently if Lemma 2 does not apply. For this purpose we need the following definition:

Definition 3. *Let i and k be integers such that $0 \le i < n$ and $2 \le k \le p$. Further let $s_{i,k}$ be an integer, $i \le s_{i,k} < n$, such that $f(i, s_{i,k}-1) < g(s_{i,k}, k-1)$ and $f(i, s_{i,k}) \ge g(s_{i,k}+1, k-1)$. Then $s_{i,k}$ is a balance point.*

It follows from Definition 3 that $\max\{f(i, s_{i,k}), g(s_{i,k}+1, k-1)\} = f(i, s_{i,k})$ and that $\max\{f(i, s_{i,k}-1), g(s_{i,k}, k-1)\} = g(s_{i,k}, k-1)$. We now show that $s_{i,k}$ is well defined.

Lemma 4. *The balance point $s_{i,k}$ exists and is unique.*

Proof. The result follows directly from the following two facts:

1) $f(i, j)$ is a *strictly monotonically increasing* function of j on $i-1 \le j \le n$ for which $f(i, i-1) = 0$, and
2) $g(j, k-1)$ is a *monotonically decreasing* function of j on $i-1 \le j \le n$ for which $g(n, k-1) = 0$. □

We now state our main theorem. It tells us how the balance point $s_{i,k}$ can be used to compute $g(i, k)$.

THEOREM 5. *For* $k \geq 2$, $g(i, k) = \min\{f(i, s_{i,k}), g(s_{i,k}, k-1)\}$.

PROOF. We consider two cases: Suppose first that $f(i, s_{i,k}) \leq g(s_{i,k}, k-1)$. Since $f(i, s_{i,k}) \geq g(s_{i,k}+1, k-1)$, a partitioning of $[i, n-1]$ into k intervals that costs less than $f(i, s_{i,k})$ must have a first interval $[i, j]$ where $j < s_{i,k}$. The cost of such a partitioning is $\gamma = \max\{f(i, j), g(j+1, k-1)\}$. From Lemma 1 it follows that $g(s_{i,k}, k-1) \leq g(j+1, k-1)$ and since $g(j+1, k-1) \leq \gamma$ we have $g(s_{i,k}, k-1) \leq \gamma$. By the assumption $f(i, s_{i,k}) \leq g(s_{i,k}, k-1)$ we have that $f(i, s_{i,k}) \leq \gamma$ in contradiction to the definition of j. Thus it follows that $g(i, k) = f(i, s_{i,k})$.

Assume now that $f(i, s_{i,k}) > g(s_{i,k}, k-1)$. Since $f(i, s_{i,k}-1) < g(s_{i,k}, k-1)$ a partitioning of $[i, n-1]$ into k intervals that costs less than $g(s_{i,k}, k-1)$ must have the first interval $[i, j]$ such that $j \geq s_{i,k}$. The cost of such a partitioning is $\gamma = \max\{f(i, j), g(j+1, k-1)\}$. Since $f(i, s_{i,k}) \leq f(i, j)$ and $f(i, j) \leq \gamma$ it follows that $f(i, s_{i,k}) \leq \gamma$. Together with the assumption that $f(i, s_{i,k}) > g(s_{i,k}, k-1)$ this shows that $g(s_{i,k}, k-1) < \gamma$ in contradiction to the definition of j and therefore $g(i, k) = g(s_{i,k}, k-1)$. □

We can now show how $g(i, k)$ and the first interval implied by $g(i, k)$ can be computed efficiently from the following information: The first interval $[i+1, j]$ implied by $g(i+1, k)$ and $g(l, k-1)$ for $i < l \leq j+1$.

If $f(i, j) \leq g(j+1, k-1)$ then as noted in Lemma 2 we have $g(i, k) = g(j+1, k-1)$ and the size of the first interval implied by $g(i, k)$ is $[i, j]$.

If $f(i, j) > g(j+1, k-1)$ then we locate $s_{i,k}$ and apply Theorem 5. From the definition of $s_{i,k}$ and Lemma 4 we see that $f(i, j) > g(j+1, k-1)$ implies that $i \leq s_{i,k} \leq j$. We first test if $f(i, i) \geq g(i+1, k-1)$. If this is true then $s_{i,k} = i$ (since $f(i, i-1) = 0$) and from Theorem 5 it follows that $g(i, k) = f(i, i)$ and the first interval contains only σ_i. If $f(i, i) < g(i+1, k-1)$ then we know that $i < s_{i,k}$.

Assume now that $f(i, j) > g(j+1, k-1)$ and $f(i, i) < g(i+1, k-1)$. To locate $s_{i,k}$, we compute $f(i, j-1)$ and compare with $g(j, k-1)$. If $f(i, j-1) < g(j, k-1)$, then $j = s_{i,k}$. If $g(j, k-1) \leq f(i, j-1)$ we reduce j by one and repeat the process. This way we will eventually get j such that $f(i, j-1) < g(j, k-1)$ and $f(i, j) \geq g(j+1, k-1)$. From Definition 3, it follows that $j = s_{i,k}$. We can now compute $g(i, k)$ by applying Theorem 5. The size of the first interval is $[i, j]$ if $f(i, j) < g(j, k-1)$ and $[i, j-1]$ if $f(i, j) \geq g(j, k-1)$.

From the above it is clear that to compute $g(i, k)$ we only make use of $g(l, k-1)$ where $i < l \leq j+1$. This implies that for a fixed value of k we only need to compute $g(i, k)$ for $p-k \leq i \leq n-k$ to be able to compute $g(0, p)$.

Before giving the complete algorithm we note that (4) can be transformed into the following recursive formula:

$$g(n-i, i) = \max\{f(n-i, n-i), g(n-i+1, i-1)\}.$$

We use (3) to compute $g(i, 1)$ for $p-1 \leq i < n$. The complete algorithm is shown in Fig. 1.

Now we show that the time complexity of this algorithm is $O(p(n-p))$. First we argue that under the assumptions on f made in Section II, we can calculate each needed value of f in $O(1)$ time. It is clear that this is true when evaluating $g(i, 1)$ in (6). In (7) and (9), we evaluate f on only one element which can be done in $O(1)$ time. If we ignore (9) then each calculation (except one) of $f(i, j)$ in (8), (10), and (11) is directly preceded by one of the following calculations: $f(i+1, j), f(i, j+1), f(i, j-1)$. The only exception occurs when calculating $f(i, j)$ in (8) for $i < n-k-1$ and (12) was true for $i+1$. Then $f(i+1, j+1)$ was calculated in (11) prior to (8). Since the argument of f is shifted by only two elements from (11) to (8) we can still calculate $f(i, j)$ in (8) using only $O(1)$ time. (Note also that in this case $f(i+1, j)$ was calculated in (10)). From this we see that we can calculate each needed value of f in $O(1)$ time.

```
for i := n - 1 down to p - 1 do                                   (6)
    g(i, 1) := f(i, n - 1);

for k := 2 to p do
    g(n - k, k) := max{f(n - k, n - k), g(n - k + 1, k - 1)};     (7)
    j := n - k;

    for i := n - k - 1 down to p - k do
        if f(i, j) ≤ g(j + 1, k - 1) then                         (8)
            g(i, k) := g(j + 1, k - 1);
        else
            if f(i, i) ≥ g(i + 1, k - 1) then                     (9)
                g(i, k) := f(i, i);
                j := i;
            else
                while f(i, j - 1) ≥ g(j, k - 1) do                (10)
                    j := j - 1;
                g(i, k) := min{f(i, j), g(j, k - 1)};             (11)
                if g(i, k) = g(j, k - 1) then                     (12)
                    j := j - 1;
            end-else
        end-else
    end-do
end-do
```

Fig. 1. The new algorithm for solving the MinMax problem.

Now we turn our attention to the overall time complexity of the algorithm. The initialization in (6) can be done in $O(n-p)$ time. In the innermost for-loop the only statement that cannot be executed in $O(1)$ time is the while-loop (10). We argue that for a fixed value of k, (10) is not true more than $n-p-1$ times. The value of j is initially set to $n-k$. Whenever (10) is reached the value of j is reduced in steps of one until $j = s_{i,k}$. Since $s_{i,k} > i$ it follows that j will never be reduced below $i+1$ in (10). For fixed k the lowest value i can have is $p-k$. Thus for each value of k, (10) can at most be true $n-p-1$ times.

If we ignore (10) then the time complexity of the innermost for-loop is $O(n-p)$. Thus by amortizing the time spent on (10) over the time spent on the innermost for-loop we see that (10) can be regarded as taking constant time. The for-loop involving k is executed $p-1$ times giving a total time complexity of $O(p(n-p))$ for the algorithm. We observe that the algorithm degenerates to an $O(n)$ algorithm for $p = 1$ and $p = n$.

As stated earlier, this is an improvement by a factor of $\log p$ compared to the Manne and Sørevik algorithm [11]. It should be noted that the algorithm presented in this paper has time complexity $\Omega(p(n-p))$ on *every* input. The highest known lower bound for the Manne and Sørevik algorithm is also $\Omega(p(n-p))$, but for an actual set of values it might take less time.

In order to compute $g(i, k)$ for fixed values of i and k, we need only $g(j, k-1)$, $i < j \leq n-k+1$, and the length of the first interval implied by $g(i+1, k)$. Thus our algorithm like that of Anily and Federgruen can be implemented using only $O(n)$ space.

IV. THE GENERALIZED MINMAX PROBLEM

When scheduling jobs to processors each processor might have limited storage for its job queue. This constraint can be included in the following **Bounded MinMax** problem: Given p positive real numbers $U_0, U_1, \ldots, U_{p-1}$, find an optimal partition for the MinMax problem with the constraint that $s(r_j, r_{j+1}-1) \leq U_j$ for $0 \leq j < p$.

$s(i)$ denotes the *size* of element σ_i where s is a function defined on consecutive intervals of $\{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$ such that s satisfies (1) and (2). We also assume that the time complexity of computing s is similar to that of f.

We will first demonstrate how the bounded MinMax problem can be viewed as a special case of the generalized MinMax problem.

Generalized MinMax. Find a partition R such that

$$\max_{i=0}^{p-1}\left\{f_i\left(\sigma_{r_i,\ldots,}\,\sigma_{r_{i+1}-1}\right)\right\}$$

is minimum over all partitions of $\{\sigma_0, \ldots, \sigma_{n-1}\}$.

Note that the generalized MinMax problem is defined with different cost functions for each of the intervals in the partition. This could be convenient if we were to distribute data on a sequence of processors where the processors operate at different speeds. We will require that each of the f_i functions satisfy the properties required by f in the original MinMax problem. We can now think of the bounded MinMax problem as a generalized MinMax problem where the cost functions are defined as follows:

$$f_k(i, j) = f(i, j) + T_k(s(i, j)) \text{ for } 0 \le k < p \tag{13}$$

T_k is a threshold operator that encodes the constraints in the bounded MinMax problem by $T_k(x) = \infty$ if $x > U_k$ and otherwise $T_k(x) = 0$.

The algorithm for solving the generalized MinMax problem is given in Fig. 2. Two modifications have been made to the algorithm in Fig. 1. First of all, every occurrence of $f(*, *)$ have been replaced by the appropriate $f_k(*, *)$ function. Secondly, we have to handle the fact that different cost functions can give solutions with empty intervals. The i-loop must therefore be repeated n times in order to investigate all possible start positions for the kth interval. Similarly, j is initialized to $n - 1$ in order investigate all possible stop positions for the kth interval. These modifications increase the time complexity, but not the validity of the main algorithm. Finally, we have removed the if-test in (9) in order to include the possibility of an empty interval. In terms of correctness, the if-test assured that the *while*-loop (10) would terminate with $j > i$. The corresponding *while*-loop in Fig. 2 terminates with $j \ge i$ because $f_{p-k}(i, i - 1) = 0$. The interval is left empty if the *while*-loop terminates with $j = i$ and $g(i, k - 1) \le f_{p-k}(i, i)$.

```
for i := n down to 0 do
    g(i, 1) := f_{p-1}(i, n - 1);
for k := 2 to p do
    g(n, k) := 0;
    j := n - 1;
    for i := n - 1 down to 0 do
        if f_{p-k}(i, j) ≤ g(j + 1, k - 1) then
            g(i, k) := g(j + 1, k - 1);
        else
            while f_{p-k}(i, j - 1) ≥ g(j, k - 1) do
                j := j - 1;
            g(i, k) := min{f_{p-k}(i, j), g(j, k - 1)};
            if g(i, k) = g(j, k - 1) then
                j := j - 1;
        end-else
    end-do
end-do
```

Fig. 2. The algorithm for solving the generalized MinMax problem. We assume that $f_k(i, j) = 0$ if $j < i$.

The asymptotic time complexity of the algorithm is $O(np)$. This is still an improvement of a factor of $\log p$ compared to the algorithm presented by Manne and Sørevik for the bounded MinMax problem. The parametric search procedure by Fredrickson [4], [5] will also give an $O(np)$ solution when it is applied to the generalized MinMax problem, but our simple and straightforward algorithm must be expected to have a much smaller constant term. The number of candidate values have increased to n^2p and the parametric search must explore p unrelated $n \times n$ sorted matrices.

V. Conclusion

We have in this paper shown how the complexity of the dynamic programming method used by Anily and Federgruen to solve the MinMax problem can be reduced from $O(n^2p)$ to $O(p(n - p))$. We obtain the improvement by taking advantage of the monotone properties of the cost functions. Where applicable, this technique seems to be a useful way of reducing the complexity of dynamic programming algorithms.

Our algorithm has the advantage of being straightforward and practical compared to the parametric search approach developed by Frederickson [4], [5]. In addition, our mild conditions on the computability of f include nonassociative operators that for example occur in signal regression problems [9]. The asymptotic running time for our algorithm and that of Frederickson both equal $O(np)$ for the generalized MinMax problem given in Section IV. Both algorithms also have $O(np)$ running time for the problem of solving the MinMax problem for p' segments with $p' = 1, \ldots, p$. The last problem is relevant in signal analysis where p is not known a priori but rather a crucial parameter for optimization.

In [11] a number of problems related to the MinMax problem were described. Each of these problems can also be solved by slight modifications of our main algorithm. We note that the circular version of the MinMax problem is solved in $O(n(n - p))$ time and that the bounded MinMax problem can be solved by the algorithm for the generalized MinMax problem even if we had used an independent size function s_i as long as each s_i satisfies (1) and (2).

The MinMax problem can be generalized to higher dimensions than one. In [10] it is described how a solution of the two dimensional partitioning problem can be used to speed up sparse matrix-vector multiplication on a systolic array of processors.

Acknowledgments

The authors thank Bengt Aspvall and Tor Sørevik for constructive comments. We would also like to thank the referees for their many helpful comments and for bringing some interesting references to our attention. Finally, we would like to thank the Norwegian Technical Research Council for supporting this study.

References

[1] A. Aggarwal and J.K. Park, "Improved algorithms for economic lot size problems," *Operations Research*, vol. 41, pp. 549-571, 1993.

[2] S. Anily and A. Federgruen, "Structured partitioning problems," *Operations Research*, vol. 13, pp. 130-149, 1991.

[3] S.H. Bokhari, "Partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. Computers*, vol. 37, pp. 48-57, 1988.

[4] G.N. Frederickson, "Optimal algorithms for partitioning trees and locating p-centers in trees," Technical Report CSD-TR-1029, Purdue Univ., 1990.

[5] G.N. Frederickson, "Optimal algorithms for partitioning trees and locating p-centers in trees," *Proc. Second ACM-SIAM Symp. Discrete Algorithms*, pp. 168-177, San Francisco, 1991.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability*. Freeman, 1979.

[7] F. Halsall, *Data Communications, Computer Networks and OSI*. Reading, Mass.: Addison Wesley, 1988.

[8] P. Hansen and K.-W. Lih, "Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. Computers*, vol. 41, pp. 769-771, 1992.

[9] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in Pascal*. Cambridge Univ. Press, 1989.

[10] F. Manne, "Load balancing in parallel sparse matrix computations," PhD thesis, Univ. of Bergen, Norway, 1993.

[11] F. Manne and T. Sørevik, "Optimal partitioning of sequences," Tech. Report CS-92-62, Univ. of Bergen, Norway, 1992.

[12] B. Olstad and H.E. Tysdahl, "Improving the computational complexity of active contour algorithms," *Proc. Eighth Scandinavian Conf. Image Analysis*, Tromsø, Norway, 1993.

[13] F.F. Yao, "Speed-up in dynamic programming," *SIAM J. Alg. Discrete Meth.*, vol. 3, pp. 532-540, 1982.

Division Using a Logarithmic-Exponential Transform to Form a Short Reciprocal

David M. Mandelbaum

***Abstract*—Two trees are used sequentially to calculate an approximation to 1/A, where $1 \le A < 2$. These trees calculate the logarithm and exponential, and the division (reciprocation) process can be described by $1/A = e^{-\ln A}$. For bit skip accuracy of six to 10, this logarithmic-exponential method uses significantly less hardware with respect to the scheme in [3], and the delays may be greater or less than those of [3], depending on the method used and the minimum bit skip.**

***Index Terms*—Array, division, exponential, logarithm, reciprocation, tree.**

I. Introduction

Much investigation has been done in developing methods to obtain an initial quotient or short reciprocal, to be used in a startup manner with iterative algorithms to obtain a quotient of two numbers. Original work in this area has been done by Svoboda [23], Krishnamurthy [19], [20], Matula [21], Sharma and Matula [22], Ercegovac and Lang [18], Wong and Flynn [7], and Schwarz and Flynn, [4], [6]. Recently, work has been done on division or reciprocation by means of carry save adder trees based on the equations $AQ = C$, or $AQ = 1$, where A is the divisor, [2], [3], [4], [6]. Initially, Stefanelli [1] utilized these equations to define an open-ended tree for division. The aim of this paper is the same as that of [3], to develop an approximation to $1/A$ (short reciprocal) to a given accuracy. However, the proposed method is more economical than [3] in certain cases, as determined by complete simulation. In this paper, reciprocation of A is accomplished by a new multistage method. The natural logarithm of A, $(\ln A)$, is calculated by a tree. Then $e^{-\ln A} = \exp(-\ln A)$ is calculated by another tree (or two trees simultaneously) to give $1/A$. Thus it can be said that $1/A$ can be obtained by the logarithmic-exponential transform (LET): $\exp(-\ln A)$.

Comparision between the LET scheme and dedicated division (short reciprocal) trees of [3], was made for certain minimum bit skip accuracy using a complete range of binary input numbers. This was also the criterion used in [7]. The bit skip, which is a criterion of accuracy, is defined by the number of left bit shifts needed to normalize the remainder $R = 1 - AQ'$ where Q' is the approximation (short reciprocal) of $1/A$ obtained from the LET. The parameters considered were the amount of hardware required to implement the LET and the delay through the circuits being considered. This was compared with the equivalent parameters of [3]. Comparisions are made with other methods in [3], including table look-up, (see also Section IV). Thus comparing the results presented here with [3], also compares them directly with other methods. For minimum bit skips of six through 10, complete simulation shows significantly less hardware is needed for the LET than [3], while the delay time varies from less to more than the trees of [3], depending on the amount and type of tree hardware and minimum bit skip. (Closed analysis may be very difficult if not impossible in determining these parameters.) For the case of a minimum bit skip of seven, one of the proposed schemes also gives less delay than the corresponding dedicated division (reciprocal) tree of [3]. However, if one wishes to consider average bit skip instead of minimum bit skip, then the trees of [3] are superior. The LET trees would be used where a guaranteed minimum bit skip for every division operation is required, as well as a minimum tree size. Certain concepts introduced by Schwarz and Flynn,

Manuscript received Sept. 15, 1993; revised Apr. 28, 1994.

The author is at 168 Hollingston Pl., East Windsor, NJ 08520..

To order reprints of this article, e-mail: transactions@computer.org, and reference IEEECS Log Number C95109.