Parallel Computing 19 (1993) 1303–1311
North-Holland

PARCO 805

Short communication

Systolic arrays for integer Chinese remaindering

Çetin Kaya Koç [a,*,**], Peter Cappello [b,†]

[a] *Department of Electrical & Computer Engineering, Oregon State University, Corvallis, OR 97331, USA*
[b] *Department of Computer Science, University of California, Santa Barbara, CA 93106, USA*

Received 26 September 1990
Revised 6 May 1991, 3 November 1992, 22 March 1993

Abstract

This paper presents time-minimal and processor-time-minimal systolic arrays for computing a process dependence graph corresponding to the mixed-radix conversion algorithm. The arrays are particularly suitable for implementation of algorithms from the applications of residue number systems on programmable systolic/wavefront arrays and in VLSI. Examples of such applications are exact solution of linear equations and computation of pseudo-inverses of matrices.

Keywords. Residue arithmetic; Chinese remainder theorem; mixed-radix conversion; Systolic arrays

1. Introduction

Residue Number Systems (RNS) provides an alternative to the weighted number systems for doing arithmetic on large integers. The advantages of residue representation are that addition, subtraction, and multiplication can be performed in a modular fashion without carry propagation. Most work on residue arithmetic is oriented towards hardware design, since carry-free properties of RNS make it attractive to implement digital signal processing algorithms using table-lookup techniques. Also, redundant moduli RNS provides fault tolerance in specialized signal processing architectures due to its error detection and correction properties [15]. RNS has applications in software as well. For example, it can be used to find solutions of equations over integers and rational numbers [14,2,8]. This allows computation of the exact solution of linear equations when the matrix of the coefficients is ill-conditioned.

A disadvantage of the RNS is that a number represented in residue notation does not have magnitude information. Conversion to a weighted number system is performed to extract this information. Methods for conversion of a residue number to a weighted number system are

* *Corresponding author.* Email: koc@ece.orst.edu
** This work is supported in part by the US Army Research Office Grant No. DAAL03-91-G-0106.
† This material is based on work supported by the National Science Foundation under grant MIP89-20598 and the Lawrence Livermore National Laboratories.

0167-8191/93/$06.00 © 1993 – Elsevier Science Publishers B.V. All rights reserved

based on two different constructive proofs of the *Chinese Remainder Theorem*. In the first case, the number is converted to a *single-radix* weighted number system, whereas in the second case it is converted to a *mixed-radix* weighted number system. The similarity between interpolation of polynomials and Chinese remaindering of integers is well-known [13]. As also noted by Bareiss [2], the Lagrange polynomial interpolation formula is the analogue of the single-radix conversion algorithm for integer Chinese remaindering (see, page 270 in [8]). The mixed-radix conversion algorithm given by Szabo and Tanaka [16] is attributed to Garner [6] (see [8,13]). The mixed-radix conversion algorithm is in fact the analogue of Aitken's algorithm for Newton polynomial interpolation. Henceforth, we refer to this algorithm as *Garner's Algorithm*.

In this paper, we present time-minimal and processor-time-minimal systolic arrays for computing a process dependence graph corresponding to Garner's algorithm. The arrays are especially suitable for software implementations of algorithms for exact solution of linear systems on a programmable systolic/wavefront array [10] and VLSI implementations of algorithms for exact computation of pseudo-inverses of matrices [4,9].

In Section 2, we present Garner's algorithm and its process dependence graph. In Section 3, we embed the process dependence graph of Garner's algorithm in processor-time, obtaining several alternative systolic implementations of Garner's algorithm, some of which are optimal. The processor-time embeddings and the resulting systolic arrays are summarized in Section 4.

2. Garner's algorithm

Given the moduli set $\{m_0, m_1, m_2, \ldots, m_n\}$ consisting of $n+1$ pairwise relatively prime numbers and the residues of a weighted number u with respect to each modulus $u = u_i \pmod{m_i}$ for $0 \leqslant i \leqslant n$, Garner's algorithm computes the mixed-radix representation $(v_0, v_1, v_2, \ldots, v_n)$ of u such that

$$u = v_0 + v_1 m_0 + v_2 m_0 m_1 + \cdots + v_n m_0 m_1 \cdots m_{n-1}.$$

The mixed-radix representation $(v_0, v_1, \ldots, v_n)$ can further be converted to a single-radix representation by using the above formula.

Garner's algorithm constructs an upper triangular table of values termed as the *multiplied differences* [12]. The entries of this table are V_{ij} for $-1 \leqslant i < j \leqslant n$ such that $V_{-1,i} = u_i$ for $0 \leqslant i \leqslant n$. The subsequent column entries are produced from the previous column entries by performing the operations

$$c_{ij} = (m_i)^{-1} \pmod{m_j}, \tag{1}$$

$$V_{ij} = (V_{i-1,j} - V_{i-1,i}) c_{ij} \pmod{m_j}. \tag{2}$$

The constants c_{ij} are the inverses of m_i modulo m_j for all $0 \leqslant i < j \leqslant n$ and can be computed using the extended Euclid algorithm (see e.g. [8,13]). The final diagonal entries of the multiplied difference table are are $V_{i-1,i} = v_i$, the coefficients we seek, for $0 \leqslant i \leqslant n$. For $n = 4$, the table looks as follows:

$$\begin{array}{lllll}
V_{04} = (u_4 - u_0)c_{04} & V_{14} = (V_{04} - V_{01})c_{14} & V_{24} = (V_{14} - V_{12})c_{24} & V_{34} = (V_{24} - V_{23})c_{34} & \pmod{m_4} \\
V_{03} = (u_3 - u_0)c_{03} & V_{13} = (V_{03} - V_{01})c_{13} & V_{23} = (V_{13} - V_{12})c_{23} & & \pmod{m_3} \\
V_{02} = (u_2 - u_0)c_{02} & V_{12} = (V_{02} - V_{01})c_{12} & & & \pmod{m_2} \\
V_{01} = (u_1 - u_0)c_{01} & & & & \pmod{m_1}
\end{array}$$

The data dependences among the entries in the above table lend themselves to systolic implementation. The first step in achieving a systolic implementation is to form the *process*

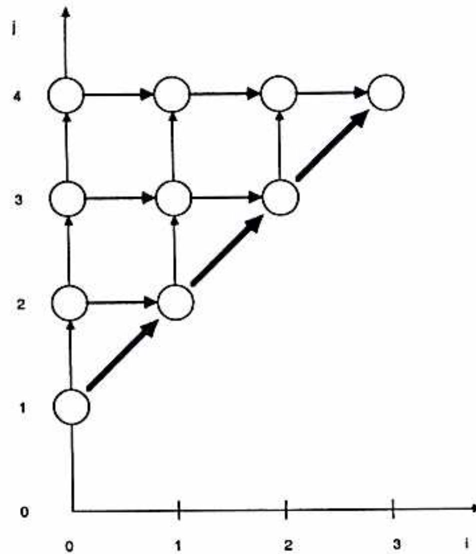

Fig. 1. Process dependence dag for Garner's algorithm ($n = 4$).

dependence graph of Garner's algorithm. Coefficient c_{ij} is in column i and row j. The positions of the numerator terms (i.e. the V_{ij}) are arranged as follows: First, a term of the form $V_{i-1,i}$ is computed on the diagonal, then this term is used in every operation along the ith column. Based on these observations. *Fig. 1* presents the process dependence graph of Garner's algorithm, for $n = 4$.

The graph is drawn on the (i, j) coordinate system. The nodes of this directed acyclic graph (dag) represent the operations; the arcs correspond to dependences between the variables used in the operations. The node at point (i, j) receives the data items from the input buffer, computes V_{ij} by performing the operations (1) and (2), places the data items to the output buffer. These operations are counted as one time step.

3. Systolic arrays

In this section, we embed the process dependence graph of Garner's algorithm in spacetime to produce systolic schedules, some of which are optimal (for processor-time embedding techniques, see [11] and the references therein). Each schedule is described briefly. Its associated measure of optimality also is noted. The synthesis methods described by Kung [11] do not *a priori* guarantee that the resulting multiprocessor schedules are time-minimal, processor-time-minimal, or period-processor-time-minimal. Much of what is interesting about the schedules we present thus is not merely that they are systolic, but that they are systolic and are optimal according to a measure mentioned above. Indeed, we know of no polynomial algorithm for finding such schedules (except for those that are required to be only time-minimal linear schedules).

3.1. A time-minimal systolic array

In this subsection, we briefly present a time-minimal array. We embed the process dependence graph for Garner's algorithm, G, in processor-time. The abscissa is interpreted as time (t); the ordinate as processor (s). The linear embedding, E_1, is as follows:

$$t := i + j;$$
$$s := i.$$

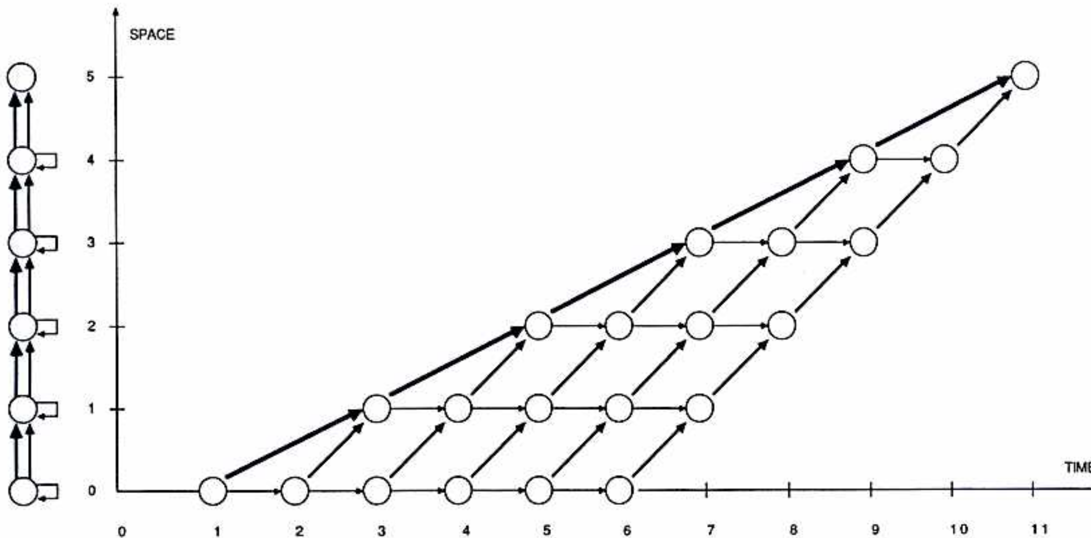

Fig. 2. Embedding E_1 which produces a time-minimal linear array ($n = 6$).

The result, depicted in *Fig. 2* for $n = 6$, is a time-minimal array. Data that flows south → north in *Fig. 1*, flows in the direction of time (perpendicular to processors) in this design: It is in the processors' memory. Data that flows west → east in *Fig. 1*, flows up through the array. Data that flows south → east in *Fig. 1*, also flows up through the array, but at half the speed of the west → east data.

Process (i, j) is executed at time $i + j$ in processor i. By inspection, we see that the array uses n processors, finishing the computation in $2n - 1$ steps. The number of vertices (processes) in a longest directed path in any process dependence graph is a lower bound on the number of steps of any schedule for computing the processes. In our graph, the number of vertices in a longest path is $2n - 1$. This array thus uses a processor-time embedding that is minimal with respect to the number of steps used. Such an embedding is referred to as *time-minimal*. The *period* (i.e. the reciprocal of throughput rate) is n. That is, a conversion completes every n steps.

3.2. Processor-time-minimal systolic arrays

We now make a slight modification to the above array, producing a *processor-time-minimal* array. A graph's embedding is processor-time-minimal when it is processor-minimal among those embeddings that are time-minimal.

There is unused timed on the lower numbered processors. We reschedule the computation done on the upper processors onto these lower processors. More formally, we embed the process dependence graph as follows:

$$t := i + j;$$

$$s := \begin{cases} i & \text{for } i + j \leqslant n \\ n - j & \text{for } i + j > n \end{cases}.$$

This embedding, E_2, is illustrated in *Fig. 3* for $n = 6$. This design has 2 phases of data movement. In the 1st phase, data moves as in the embedding E_1. As the 1st phase ends, and the 2nd begins, there is a transition: When n is even (as depicted in *Fig. 3*), there are 2 time steps in which the south → east data flows in the direction of time: It is in the uppermost processor's memory for these 2 steps. When n is odd, the south → east data always moves through the array.

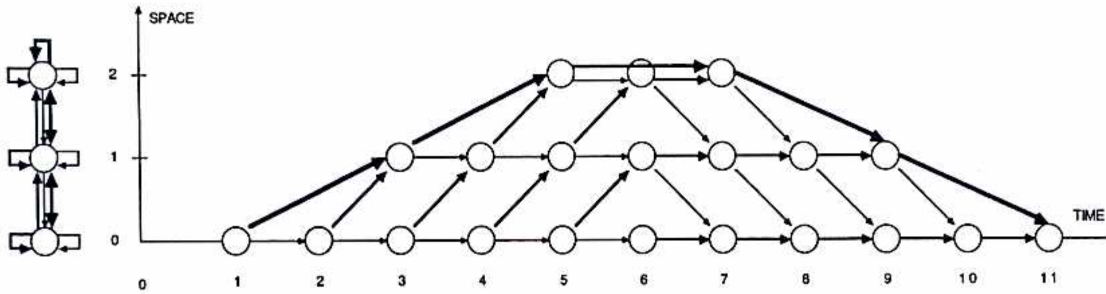

Fig. 3. Embedding E_2 which produces a processor-time-minimal linear array.

In the 2nd phase, data moves as follows. Data that flows south → north in *Fig. 1*, flows down through the array. Data that flows west → east in Figure 1, flows in the direction of time: It is remembered in this phase. Data that flows south → east in *Fig. 1*, also flows down through this array, but at half the speed of the south → north data. Of those processor-time embeddings that are time-minimal, this embedding also is a processor-minimal.

Theorem 1. *Embedding E_2 of G is processor-time-minimal.*

Proof. The embedding E_2 is identical to E_1 with respect to time; it too is time-minimal. We now argue that the embedding is processor-minimal among time-minimal embeddings. Let us focus on the time steps in which all 3 processors are used (which we refer to as the *processor-maximal* time steps). They are time step 5, 6, and 7. In order to reduce the number of processors, during the processor-maximal time steps the nodes scheduled for some processor must be rescheduled onto the other 2 processors. Two processors suffice, for example, if the nodes named (5, 2), (6, 2), and (7, 2) can be rescheduled from processor 2 onto processors 0 and 1. These nodes are in a longest directed path in the process dependence dag. This means that none can be rescheduled for earlier completion without violating a dependence. Neither can they be scheduled for later completion without either violating a dependence, or extending the overall completion time, violating time-minimality. In fact, in this dag, every node is on some longest directed path, and hence can be rescheduled onto neither an earlier nor a later time step. In particular, the nodes scheduled for the processor-maximal time steps cannot be rescheduled. The number of processors therefore cannot be reduced: The design is processor-time-minimal. Any processor-time embedding of this process dependence graph that completes in $2n - 1$ cycles, must use at east $\lceil n/2 \rceil$ processors. □

Moreover, the nonlinearity of our processor-time transformation is necessary: There does not exist a linear embedding of the initial indices that is processor-time-minimal.

We now present 3 other processor-time-minimal embeddings of the process dependence graph of *Fig. 1*. The first is another variation of the array resulting from embedding E_1. We again reschedule the computation done on the upper processors onto the lower processors. To do this, we connect the endpoints of the linear array, making a *ring* of processors. More formally, we nonlinearly embed the process dependence graph as follows:

$$t := i + j;$$

$$s := i \bmod \lfloor n/2 \rfloor.$$

This embedding, E_3, is illustrated in *Fig. 4* for $n = 6$. Its data flow characteristics are identical to those of embedding E_1, except that the upper processor is attached to the lower processor, and data movement wraps around.

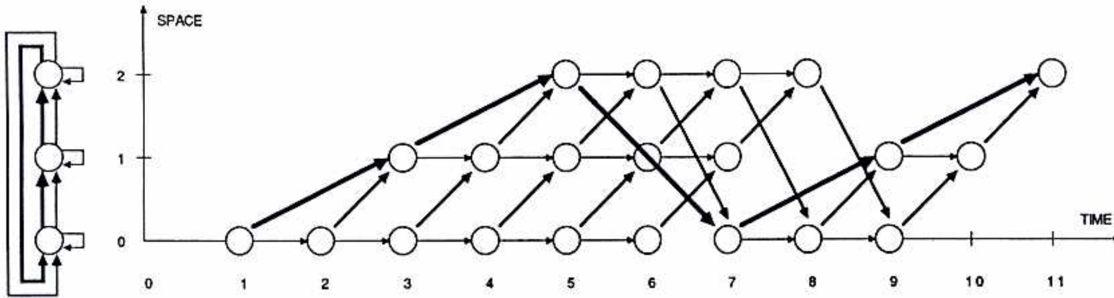

Fig. 4. Embedding E_3 which produces a processor-time-minimal ring array.

Since this embedding results in a computation of the process dependence graph that uses $2n-1$ steps and $\lfloor n/2 \rfloor$ processors, it too is processor-time-minimal.

Finally, we present a bilateral array in which the south → north data of *Fig. 1* moves up through the array, while the west → east data moves down through the array. Such a data movement scheme may be useful, depending on the larger context of which this computation is a part. The processor-time embedding, E_4, is presented in 2 steps:

(1) First, we embed the process dependence graph, illustrated in *Fig. 5*, as follows:

$$t' := i + j - 1;$$

$$s' := -i + j - 1.$$

In this processor-time embedding, when processors are used, they are used every other time step.

(2) We now compress the spatial extent of this embedding with the following nonlinear transformation:

$$t := t';$$

$$s := \lfloor s'/2 \rfloor.$$

Processor efficiency (i.e. the percentage of time that a processor is used) is doubled asymptotically by this nonlinear transformation. *Fig. 6* illustrates the result.

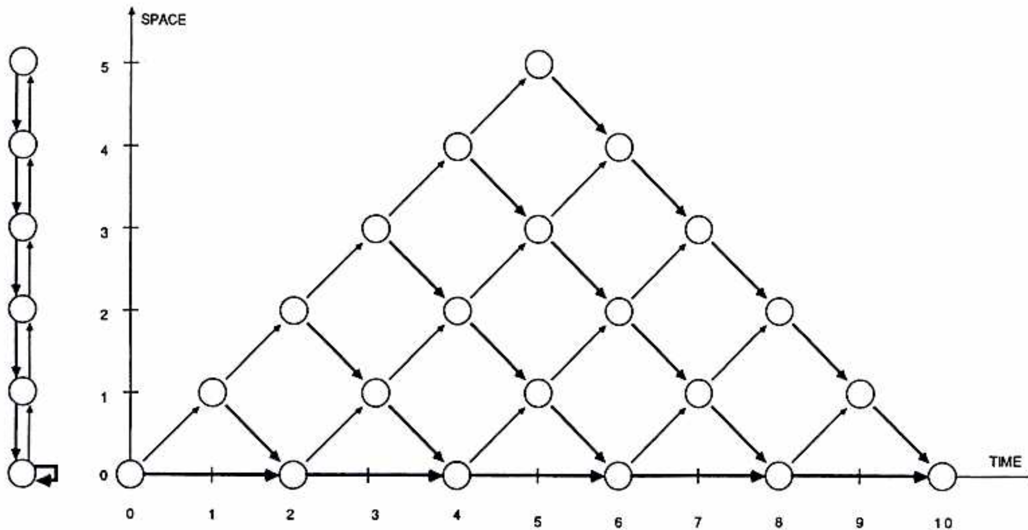

Fig. 5. An embedding which produces a bidirectional array of n processors.

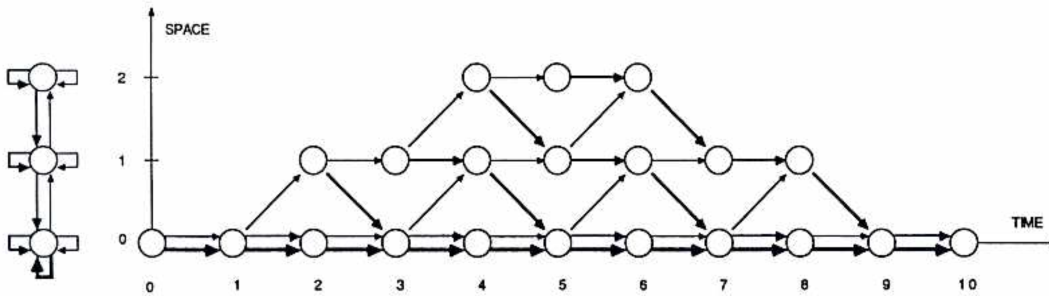

Fig. 6. Embedding E_4 which producing s-time-minimal bidirectional linear array.

This design has 2 phases of data movement which alternate with each time step. Regardless of the phase, data flowing south → east in *Fig. 1*, flows in the direction of time: It is remembered. In phase A, data flowing south → north in *Fig. 1*, flows up through the array; data flowing west → east in *Fig. 1*, flows in the direction of time. In phase B, data flowing south → north in *Fig. 1*, flows in the direction of time; data flowing west → east in *Fig. 1*, flows up through the array. Since this second transformation results in an embedding that uses $2n-1$ steps and $\lceil n/2 \rceil$ processors, it too is processor-time-minimal.

The processor-time-minimal systolic arrays that we have presented (embeddings E_2, E_3, and E_4) have a period of $2n-1$ steps. This is to be contrasted with the period of n steps for the time-minimal (but not processor-time-minimal) systolic array.

3.3. On period-processor-time-minimality

We now sketch another processor-time-minimal systolic array that has a period of $n+1$. It is presented in *Fig. 7*. As can be seen from the figure, the computation proceeds in 2 phases. Data flow in the first phase is identical to that in *Fig. 2*; data flow in the second phase is a compressed version of that depicted in *Fig. 2*.

This systolic array completes in $2n-1$ steps, using $\lceil n/2 \rceil$ processors. It thus is processor-time-minimal. Notice that each of $\lceil n/2 \rceil$ processors is used exactly $n+1$ steps. The computation, therefore, is perfectly balanced with a resulting period of $n+1$. We moreover can argue that this is a *minimal* period, when $\lceil n/2 \rceil$ processors are being used and n is even: The period of a computation on a systolic array is the maximum number of steps taken by any processor for the computation. Given a dependence dag, a lower bound on the maximum number of steps taken by any processor is the total number of nodes in the dag divided by the number of processors. In this case, the total number of nodes is $\sum_{i=1}^{n} i = n(n+1)/2$. Since the number of processors is $\lceil n/2 \rceil$, the minimal period is equal to $n+1$. When n is even, this argument

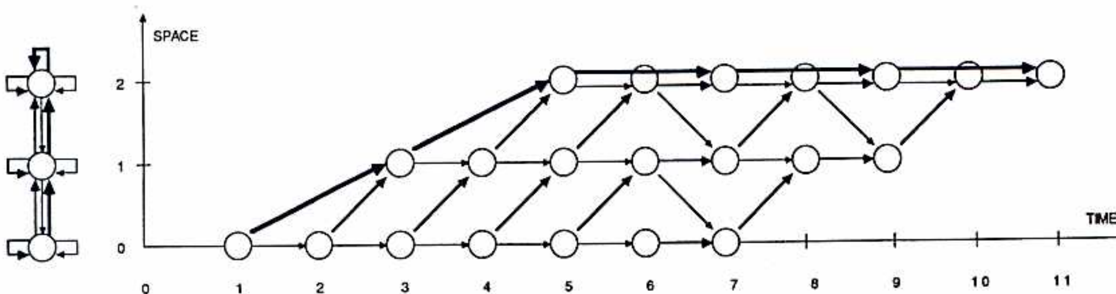

Fig. 7. An embedding which produces an array within 1 step of period-processor-time-minimality.

Table 1
Summary of processor-time embeddings

Figure (embedding)	Processors	Time	Period	Minimality
$2(E_1)$	n	$2n-1$	n	time
$3(E_2)$, $4(E_3)$, $6(E_4)$	$\lceil n/2 \rceil$	$2n-1$	$2n-1$	processor-time
7	$\lceil n/2 \rceil$	$2n-1$	$n+1$	n even: period-processor-time n odd: processor-time

yields a lower bound of $n+1$ steps: This systolic array's period thus is *period-processor-time-minimal* (i.e. a schedule whose period is minimized over all processor-time-minimal schedules for the dag), for this case. When n is odd, the period argument yields a lower bound of n steps. This systolic array's period thus is within 1 step of period-processor-time-minimality, for this case. Like the array in *Fig. 2*, it is time-minimal and has a period that is 1 greater than that of *Fig. 2*, yet uses only *half* as many processors!

4. Conclusions

RNS simplifies large-number computations by resolving a problem into a set of parallel, independent computations performed in modularly formed channels. Consequently, combining RNS with systolic arrays is particularly suitable for a VLSI implementation. Several implementations of Garner's algorithm are given in the literature. These implementations are mostly hardware oriented, using table lookup techniques [15,3]. They thus put restrictions on the size and cardinality of the moduli. The systolic arrays described in this paper have no such restrictions.

We have presented designs that are time-minimal and processor-time-minimal, and also described a design which is within 1 step of period-processor-time-minimality. We summarize the spacetime embeddings, and the resulting systolic arrays, in *Table 1*. Each of the presented schedules clearly have a place, indicating the potential usefulness of a retargetable or programmable systolic/wavefront array. Examples of such software-oriented systolic computing systems include, but are not limited to, an array of Transputers [7], the Matrix-1 [5], and the Warp [1].

References

[1] A.M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu and J. Webb. The WARP computer: Architecture, implementation, and performance, *IEEE Trans. Comput.* 36(12) (Dec. 1987) 1523–1538.

[2] E.H. Bareiss, Computational solution of matrix problems over an integral domain, *J. Inst. Math. Appl.* 10(1) (August 1972) 68–104.

[3] N.B. Chakraborti, J.S. Soundararajan and A.L.N. Reddy, An implementation of mixed-radix conversion for residue number applications, *IEEE Trans. Comput* 35 (8) (Aug. 1986) 762–764.

[4] P.R. Chang and C.S.G. Lee, Residue arithmetic VLSI array architecture for manipulator pseudoinverse Jacobian computation, *IEEE Trans. Robotics and Automation* 5(5) (Oct. 1989) 569–582.

[5] D.E. Foulser and R. Schreiber, The Saxpy Matrix-1: A general-purpose systolic computer, *IEEE Comput. Mag.* 20(7) (July 1987) 35–43.

[6] H.L. Garner, The residue number systems, *IRE Trans. Electronic Comput.* 8(6) (June 1959) 140–147.

[7] INMOS Ltd., Almondsbury, Bristol, UK, IMS T800 Transputer, November 1986.

[8] D.E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms, vol. 2* (Addison-Wesley, Reading, MA: 1981) Second edition.

[9] Ç.K. Koç, Comments on 'Residue arithmetic VLSI array architecture for manipulator pseudoinverse Jacobian computation', *IEEE Trans. Robotic and Automation* 7(5) (Oct. 1991) 715–716.

[10] Ç.K. Koç, A parallel algorithm for exact solution of linear equations via congruence technique, *Comput. Math. Appl.* 23(12) (1992) 13–24.

[11] S.Y. Kung, *VLSI Array Processors* (Prentice-Hall, Englewood Cliffs, NJ 1988).

[12] J.D. Lipson, Chinese remainder and interpolation algorithms, in: *Proc. 2nd Symp. Symbolic and Algebraic Manipulation*, Los Angeles, CA, March 23–25, 1971 (ACM Press, New York) 372–391.

[13] J.D. Lipson, *Elements of Algebra and Algebraic Computing* (Addison-Wesley, Reading, MA 1981).

[14] M. Newman, Solving equations exactly, *J. Res. Nat. Bureau of Standards* 71B(4) (Oct.–Dec. 1967) 171–179.

[15] M.A. Soderstrand, W.K. Jenkins, G.A. Jullien and F.J. Taylor, eds., *Residue Arithmetic Modern Applications in Digital Signal Processing* (IEEE Press, New York, NY 1986).

[16] N.S. Szabo and R.I. Tanaka, *Residue Arithmetic and its Applications to Computer Technology* (MacGraw-Hill, New York, NY 1967).