Mapping a Class of Run-Time Dependencies onto Regular Arrays

G.M. Megson

Abstract

The production of regular computations using algorithmic engineering techniques is beginning to play an important role in the synthesis of massively parallel and VLSI processor arrays. In this paper we widen the class of algorithms that can be formally synthesized by introducing a mapping theorem for a class of algorithms with run-time dependencies. The technique is illustrated by deriving uniform recurrences for the so-called knapsack problem, the resulting systolic array is known to be optimal.
Bibliographical details

MEGSON, Graham Martin
Mapping a Class of Run-Time Dependencies onto Regular Arrays
[By] G.M. Megson
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no.397)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract
The production of regular computations using algorithmic engineering techniques is beginning to play an important role in the synthesis of massively parallel and VLSI processor arrays. In this paper we widen the class of algorithms that can be formally synthesized by introducing a mapping theorem for a class of algorithms with run-time dependencies. The technique is illustrated by deriving uniform recurrences for the so-called knapsack problem, the resulting systolic array is known to be optimal.

About the author
Dr. G.M. Megson is a lecturer in the Department of Computing Science at the University of Newcastle upon Tyne.

Suggested keywords
ALGORITHMIC ENGINEERING
KNAPSACK PROBLEM
RUN-TIME DEPENDENCIES
SYNTHESIS-METHODS

Suggested classmarks (primary classmark underlined)
Dewey -(18th): 511.8
U.D.C. 510.5 519.688
MAPPING A CLASS
OF
RUN-TIME DEPENDENCIES
ONTO
REGULAR ARRAYS

G.M. MEGSON
Department of Computing Science
University of Newcastle-Upon-Tyne,
Claremont Tower, Claremont Rd,
Newcastle-Upon-Tyne,
NE1 7RU,
U.K.

ABSTRACT

The production of regular computations using algorithmic engineering techniques is beginning to play an important role in the synthesis of massively parallel and VLSI processor arrays. In this paper we widen the class of algorithms that can be formally synthesized by introducing a mapping theorem for a class of algorithms with run-time dependencies. The technique is illustrated by deriving uniform recurrences for the so-called knapsack problem, the resulting systolic array is known to be optimal.

Keywords: synthesis methods, run-time dependencies, Knapsack problem, algorithmic engineering.

1. Introduction

In recent years many so-called synthesis techniques for the automatic mapping of algorithms onto regular arrays have been proposed. Most of these methods rely on the representation of nested for-loop programs as systems of recurrence equations. These equations are represented geometrically by forming a Data Dependency Graph (DDG) and embedding the graph into euclidean space (n loops imply n-dimensions). A regular array (or architecture) is then specified by a timing and allocation function pair [1]. The timing function describes a schedule which indicates when equations at the DDG nodes are to be computed. The allocation (in its simplest form) projects n-d space-time into (n - 1)-d space mapping many DDG nodes to a single processor to determine a parallel array for the original algorithm. If in addition the system of recurrences is uniformized prior to scheduling and allocation the resulting architecture will be locally connected (or systolic).

Quite a lot is known about the derivation of linear timing functions for Uniform [1], Affine [2], and Linear [3] recurrences which define a wide class of algorithms. Similarly linear allocation functions correspond to the standard notions of projection in geometric modelling. Some work on non-linear timing and allocation functions is beginning to appear, this work is
related to the idea of space-time optimal arrays which minimise both algorithm computation time and processor counts. In particular the idea of mapping designs to fixed sized arrays [4] using partitioning and tiling techniques [5] together with the generation of arrays for a predefined topology have received some consideration. It is important that such synthesis techniques can be readily automated and software design tools are beginning to emerge. Many of these systems are aimed at VLSI design (e.g. HIFI [6], VACS [7], DIASTOL [8], ADVIS [9]) but one or two most notable the work of Ribas [10] is aimed at programmable arrays.

Past work has concentrated on the synthesis of data flow in arrays but now attention is moving towards the synthesis of control flow [11],[12]. In addition problems such as the Knapsack problem [13-20] have identified deficiencies in current design techniques. Consequently further work is required on the extension of these valuable design methods to cope with larger problem classes together with integration of new techniques into software tools. In this paper we consider the problem of systematically mapping a restricted class of non-uniform recurrences, in which the structure of the DDG is not known until run-time, onto a regular architecture. A pumping lemma is introduced which allows the process to be automated.

The rest of the paper is organised as follows. Section 2 reviews the basic terminology and concepts of recurrence mapping procedures. Section 3 defines what we mean by run-time and compile-time dependencies and identifies the class of recurrences considered. Section 4 introduces the mapping theorem the mapping method is briefly discussed in Section 5. Section 6 shows how it can be used to derive a set of uniform recurrence equations for the Knapsack problem. Section 7 is a conclusion.

2. Basic Terminology

Before proceeding let us briefly review the concepts behind existing mapping (or space-time transformation) techniques. Following Quinton and Van Dongen [3] we will use the following definitions.

A recurrence equation will have the form

\[ U(z) = f(\ldots V(I(z))\ldots) \text{ for } z \text{ in } D \]  \hspace{1cm} (2.1)

where \( z \) is a point of \( \mathbb{Z}^n \) and \( Z \) is the set of non-negative integers. The set of integral points belonging to convex polyhedron \( D \) is called the domain of the problem and normally resides in the positive orthant of \( \mathbb{Z}^n \). We assume that \( D \) is defined by a set of halfspaces derived from the upper and lower bounds of a nested-loop program for computing (2.1) and is finite. The function \( I(z) \) is a mapping from \( \mathbb{Z}^n \rightarrow \mathbb{Z}^l \) where usually the equation (2.1) is fully indexed so that \( l = n \) and all variables have the same index dimension. As we will see the precise definition of \( I(z) \) together with restrictions on variable usage can be used to characterize run-time dependencies. The variables \( U \) and \( V \) are taken from the set \( \mathbb{V} \) of all variables with \( U \) being the result and \( V \) an argument of (2.1). In principle there can be any number of arguments but we will assume that only a small fixed number \( K > 0 \) are permitted (to ensure
bounded fan-in and fan-out of cells in the final design). In such cases the k th argument will be written \( l_k(z) \). Finally the function \( f \) is a strict function and requires unit time to complete its evaluation.

To simplify the discussion we will also make the following assumptions:

- A system of recurrence equations is a finite set of equations with the form (2.1). We will consider only a single equation but the result is generalized in a straightforward manner to systems of equations.

- The system of recurrences is in normal form (this can always be achieved by a series of transformations, see [3]). We will concentrate on the computation equations and ignore input/output equations which do not affect the results.

- All the equations of the system have the same domain. This is not essential but avoids over complication.

A dependence vector of argument \( V \) is defined as \( \Phi_V(z) = z - l(z) \) and indicates that the result at point \( z \) in \( D \) relies on a result from point \( l(z) \). The Data Dependency Graph (DDG) is a directed acyclic graph defined by the pair \( G = (N, E) \) where \( N \) is a finite set of nodes and \( E \) is a finite set of edges. In general there is a 1-1 mapping of nodes in \( N \) and points \( z \) in \( D \). Thus dependence vectors indicate edges of the DDG which are embedded into \( Z^n \). The embedded DDG is a geometric form of the algorithm computed by (2.1) and consequently can be regarded as an architecture. Traditionally one determines a timing and allocation pair \( (t(z), a(z)) \) where \( t(z) : Z^n \rightarrow Z \) schedules the computations at each of the DDG nodes and \( a(z) : Z^n \rightarrow Z^{n-1} \) maps multiple nodes onto a single processor. Clearly \( t(p) \neq t(q) \) if \( a(p) = a(q) \) for two points \( p \) and \( q \) in \( D \) otherwise two simultaneous computations would compete for the same processor at the same time. For finite domains and uniform recurrence equations it is known that the DDG can be evaluated using a linear schedule which is determined mechanically using a linear programming method. More general results on computability are known but are not relevant here (see [21]).

3. Run-Time dependencies

Next let us consider more precisely what we mean by run-time dependencies (or RTD's). It should be clear, that with current techniques, the DDG can only be analyzed for a particular \( (t(z), a(z)) \) pair if the edges are static and known at compile time. In this context, compile-time means before computation begins and run-time means while the schedule \( t(z) \) is being executed. If the edges are not static it is impossible to form the linear programming problem and hence derive the timing function. Conversely if a timing schedule is already known the image of the \( a(z) \) mapping function is not known in advance so that the connection topology of the architecture is not known in advance. We conclude that computations can be classified crudely by their dependence vectors, or more precisely, the function \( l_k(z) \) for argument \( k \).

- Compile-Time Dependencies (CTD's)
For a parametrized linear recurrence we can write

\[ I_k(z) = A_k z + B_k p + C_k \]  

(3.1)

where \( A_k, B_k, \) and \( C_k \) are constant integral matrices with dimensions \( l \times n, \) \( l \times m, \) and \( l \times 1 \) respectively. The term \( p \) is an \( m \)-element vector describing the problem or domain size. When \( B = 0 \) (3.1) produces affine recurrence equations (AREs) and when in addition \( A = I \) uniform recurrence equations (UREs). It should be clear that the dependencies of UREs are simple translations. Now for a given \( p, \) the dependence vectors \( \Phi_k(z) \) of (3.1) can be evaluated at compile time to yield a static DDG.

* Run-Time Dependencies (RTD's)*

In contrast let

\[ I_k(z) = g(\ldots W_{kj}(J_{kj}(z))\ldots) \]  

(3.2)

where \( g \) is a potentially arbitrary function, \( W_{kj} \) is the \( j \)th argument of \( g \) with \( W_k \) from the variable set \( V, \) and \( J_{kj}(z) \) is an index mapping function. Here the dependence vectors are dynamic because they depend on the result of \( W_{kj}, \) evaluated at point \( J_{kj}(z), \) which is not generally known until run-time.

If a systematic method can be found for mapping systems of RTD equations into an equivalent set of CTD equations, existing techniques and results on the computability and scheduling of static DDG's can be used to derive regular arrays. The resolution of this problem in a general context is beyond the scope of this paper. However for a restricted and practically useful form of (3.2) some progress is possible.

The form we consider is the class of RTD's that can be characterized by the following index function

\[ I_k(z) = K_k(z) + \sum_{j=1}^{m} W_{kj}(J_{kj}(z))e_{kr_j} \]  

(3.3)

for \( m > 0. \) Where the index mappings \( J_{kj}(z) \) and \( K_k(z) \) are Affine, and \( e_{kr_j} \) is an \( n \)-component elemental vector which is null apart from the \( r_j \) element which is one. To simplify notation and where the intention is clear we will drop the subscript \( k \) and write

\[ I(z) = K(z) + \Sigma^m_1 \]  

(3.4)

with

\[ \Sigma^m_1 = \sum_{j=1}^{m} W_{j}(J_{j}(z))e_{r_j} \]

Finally, we need to restrict the use of variables from \( V. \) First split the variables into two sets \( V_1 \) and \( V_2 \) such that \( V_1 \cup V_2 = 0 \) and define \( W_j \) as variables from \( V_1 \) and all other variables from \( V_2. \) Further suppose that variables from \( V_2 \) cannot appear as arguments of equations with results from \( V_1. \) The class of programs we consider have the form
do {
    for \( z \in D \) do compute all results from \( V_1 \) \( (I) \)
    for \( z \in D \) do compute all results from \( V_2 \) \( (II) \)
} forever

where the size of \( D \) is the same for all iterations. Notice that the DDG for \( (I) \) is compile-time dependent while the DDG for \( (II) \) is run-time dependent. An important class of problems with this structure arise when sets of input data for a problem instance of a certain size are read into a program and processed one after another. Let \( DDG(r) \) be the data dependency graph for \( (II) \) on the \( r \) th iteration of the do-loop. Clearly once \( (I) \) has been computed \( DDG(r) \) is static. Unfortunately \( DDG(r) \neq DDG(r - 1) \) for \( r = 0, 1, \ldots \). In this paper we show that the graphs \( DDG(r) \) can be replaced by a single DDG which is static and involves only compile-time dependencies.

4. Mapping Theorem

In this section we present a number of lemmas which support the mapping technique. We start with a simple problem and build up to the required form (3.3).

**LEMMA 1:** the \( n \)-d recurrence

\[
U(z) = f(\ldots, V(I(z)), \ldots)
\]

where \( I(z) = z + te_r \) with \( t > 0 \) an integer can be re-written as an equivalent \((n+1)\)-dimensional system of conditional recurrence equations dependent only on \( t \).

**Proof:**

First extend the domain \( D \) into a new domain \( \bar{D} \) with \( n + 1 \) dimensions. Let \( \bar{z} \) be an \((n+1)\)-element vector from \( \bar{D} \) with \( z_{n+1} \) the last element. Further suppose that \( z_{n+1} = 0(1)t_{max} \) and that

\[
\begin{align*}
\bar{e}_r &= \begin{pmatrix} -\bar{e}_r \\ 1 \end{pmatrix}, & \bar{e}_r &= \begin{pmatrix} -\bar{e}_r \\ 0 \end{pmatrix}, & \bar{e}_r &= \begin{pmatrix} -\bar{e}_r \\ -1 \end{pmatrix}
\end{align*}
\]

Also let

\[
\alpha = \lfloor t/2 \rfloor, \quad \beta = \lceil t/2 \rceil - \lfloor t/2 \rfloor, \quad \gamma = \lfloor t_{max}/2 \rfloor
\]

The result \( U(z) \) depends on a result at \( I(z) \) and as \( \Phi_{V}(z) = -te_r \) we define a routing path with the form

\[
-te_r = \alpha \bar{e}_r + \beta \bar{e}_r + \gamma \bar{e}_r
\]

so that the equivalent set of equations

\[
U(\bar{z}) = \begin{cases} f(\ldots, R_1(\bar{z} - \bar{e}_r), \ldots), & \bar{z} = 0; \\
\infty, & \bar{z} > 0. \end{cases}
\]

\[
R_1(\bar{z}) = \begin{cases} R_3(\bar{z} - \bar{e}_r), & \text{if } z_{n+1} = \alpha \text{ and } \beta = 1 \\
R_3(\bar{z} - \bar{e}_r), & \text{if } z_{n+1} = \alpha \text{ and } \beta = 0 \\
R_1(\bar{z} - \bar{e}_r), & \text{if } z_{n+1} \neq \gamma \end{cases}
\]
can be used to shift \( V(I(z)) \) to \( z \), with \( \infty \) a dummy operation. Observe that explicit knowledge of \( \alpha \) and \( \beta \) is required at each point in \( D \). To overcome this problem introduce new variables \( S \) and \( C \), to act as pipelined control variables determining the switching on the routing path. The system (4.1) can be augmented to yield

\[
S(z) = \begin{cases} 
S(z - \varepsilon_r) - 1, & z_{n+1} > 0 \\
\alpha, & z_{n+1} = 0 
\end{cases} \tag{4.2a}
\]

\[
C(z) = \begin{cases} 
C(z - \varepsilon_r), & z_{n+1} > 0 \\
\beta, & z_{n+1} = 0 
\end{cases} \tag{4.2b}
\]

\[
U(z) = \begin{cases} 
f(\ldots, R_1(z - \varepsilon_r), \ldots), & z_{n+1} = 0 \\
\infty, & \text{Otherwise} 
\end{cases} \tag{4.2c}
\]

\[
R_4(z) = \begin{cases} 
V(z), & z_{n+1} = 0 \\
R_3(z - \varepsilon_r), & z_{n+1} > 0 
\end{cases} \tag{4.2d}
\]

\[
R_3(z) = \begin{cases} 
R_3(z), & z_{n+1} > 0 \text{ and } S(z) = 0 \text{ and } C(z) = 1 \\
R_2(z - \varepsilon_r), & \text{Otherwise} 
\end{cases} \tag{4.2e}
\]

\[
R_2(z) = \begin{cases} 
R_3(z), & z_{n+1} > 0 \text{ and } S(z) = 0 \text{ and } C(z) = 0 \\
R_1(z - \varepsilon_r), & z_{n+1} > 0 \text{ and } S(z) = 0 \text{ and } C(z) \neq 0 \\
V(z - \varepsilon_r), & z_{n+1} = 0 \text{ and } R_2(z - \varepsilon_r) \neq 0 \\
R_1(z - \varepsilon_r), & z_{n+1} = 0 \text{ and } R_2(z - \varepsilon_r) = 0 
\end{cases} \tag{4.2f}
\]

which preserves the computation and, apart from \( \alpha \) and \( \beta \) which are required at every point in \( D \) (they can be shifted to the boundary of \( D \) using standard pipelining and routing methods), uses only locally available information. \( \square \)

Effectively Lemma 1 introduces a set of routing hyperplanes which can be used to extend the DDG without overloading the input and output degree of nodes in \( D \). It should also be appreciated that our example routing path is non-unique so that the routing planes can be used to avoid formation of a cyclic DDG and hence subsequent breakdown of the synthesis method.

**LEMMA 2**: the \( n \)-dimensional recurrence

\[ U(z) = f(\ldots, V(I(z)), \ldots) \]

with \( I(z) = z + t(z)e_r \) and \( t(z) > 0 \) can be written as an \((n+1)\)-dimensional system of conditional uniform recurrence equations (CUREs).

**Proof**:

Put \( \alpha(z) = [t(z)/2] \) and \( \beta(z) = [t(z)/2] - [t(z)/2] \). Following Lemma 1 we can use the system
of equations (4.2) with (4.2a, b) replaced by

\[
S(\bar{z}) = \begin{cases} 
S(\bar{z} - \delta_r) - 1, & z_{n+1} > 0 \\
\alpha(\bar{z}), & z_{n+1} = 0 
\end{cases} \\
C(\bar{z}) = \begin{cases} 
C(\bar{z} - \delta_r), & z_{n+1} > 0 \\
\beta(\bar{z}), & z_{n+1} = 0 
\end{cases}
\]

where \( \Phi_\alpha(\bar{z}) = \Phi_\beta(\bar{z}) = \bar{z} - z \) are generally non-uniform. However, \( t(z) \) is constant for the computation (because of our assumptions in section 3) so that an alternative pipelined path from the boundary of \( \bar{D} \) can be found. From the definition of \( \Phi_V(z) \), we know that the variable \( V \) has direction \(-e_r\) and is of length \( t(z) \). Further more,

\[-t(z)e_r = t(z)(\delta_r - e_{n+1})\]

so that \( \alpha(\bar{z}) \) and \( \beta(\bar{z}) \) can be pipelined in the \( e_r - e_{n+1} \) direction to reach \( \bar{z} \) from the boundary of \( \bar{D} \). If \( z_{n+1} = t(z) \) the path has length \( t(z) \). Consequently if \( \bar{D} \) extends beyond \( t(z) \) in the direction \( e_{n+1} \) the pipelining path can be extended backwards along \( e_r \) until \( \alpha(\bar{z}) \) and \( \beta(\bar{z}) \) are input from the boundary of \( \bar{D} \). Thus (4.2) can be extended to yield

\[
S(\bar{z}) = \begin{cases} 
S(\bar{z} - \delta_r) - 1, & z_{n+1} > 0 \\
A(\bar{z}), & z_{n+1} = 0 
\end{cases} \\
C(\bar{z}) = \begin{cases} 
C(\bar{z} - \delta_r), & z_{n+1} > 0 \\
B(\bar{z}), & z_{n+1} = 0 
\end{cases} \\
A(\bar{z}) = \begin{cases} 
A(\bar{z} - e_r + e_{n+1}), & \bar{z} = e_r + e_{n+1} \text{ in } \bar{D} \\
\alpha(\bar{z}), & \text{Otherwise} 
\end{cases} \\
B(\bar{z}) = \begin{cases} 
B(\bar{z} - e_r + e_{n+1}), & \bar{z} = e_r + e_{n+1} \text{ in } \bar{D} \\
\beta(\bar{z}), & \text{Otherwise} 
\end{cases}
\]

As \( \alpha(\bar{z}) \) and \( \beta(\bar{z}) \) are inputs the resulting system is a set of conditional uniform recurrence equations (CUREs). The values of \( \alpha(\bar{z}) \) and \( \beta(\bar{z}) \) can be evaluated for all points \( z \) in \( D \) before computation begins. Furthermore the entry points in \( \bar{D} \) can be computed using the point

\[
\begin{pmatrix} z \\ t(z) \end{pmatrix}
\]

and the direction \( \delta_r \), which define a unique line through the extended domain. □

Remark : the pipelining path \( e_r - e_{n+1} \) overlays part of the routing path in lemma 1. As a result the DDG will be cyclic preventing the automatic derivation of a scheduling function. This is a minor point and a new path \( e_r - e_{n+1} - p \) can be used instead where \( p \) is the direction of a simple offset such that

\[
\begin{pmatrix} z \\ t(z) \end{pmatrix} + t(z)p
\]

we conclude that there is considerable flexibility in the choice of the \( t(z) \) routing paths.

**LEMMA 3** : the \( n \)-dimensional recurrence

\[
U(z) = f(\ldots, V(I(z)), \ldots)
\]
where \( f(z) = z + t(J(z))e_r \) and \( J(z) = Cz + d \) for integral \( n \times n \) and \( n \times 1 \) matrices can be re-written as an \( (n+1) \)-dimensional system of Conditional Affine Recurrence Equations (CAREs) when \( J(z) \) has no pipelining vectors and as a set of Conditional Uniform Recurrence equations (CUREs) otherwise.

**Proof:**

The proof is similar to that of lemma 2. First put

\[
\alpha(z) = \lfloor t(J(z))/2 \rfloor, \beta(z) = \lfloor t(J(z))/2 \rfloor - \lfloor t(J(z))/2 \rfloor
\]

and augment the equations in lemma 1 with additional variables to route \( \alpha(z) \) and \( \beta(z) \) from the boundary. There are two cases to consider :

**Case (a):** Suppose that \( J(z) \) has a pipelining vector \( v \) such that \( Cv = 0 \) and \( v \neq e_{n+1} \) (note : that \( e_{n+1} \) is a a pipelining vector because of the domain extension). Clearly if \( v \) exists \( t(J(z)) \) can be moved in direction \( v \) from \( J(z) \) to \( z \). Further, we know that \( t(J(z)) = t(J(z-v)) \) for all \( z \) in \( D \), by direct application of the usual pipelining theorem [21]. Thus \( t(J(z)) = t(z - t(J(z))v) \). The length of the path from \( z \) to \( z - t(J(z))v \) is \( t(J(z)) \) hence

\[
t(J(z))[v + e_r] = t(J(z))[v + e_{n+1} - e_r]
\]

gives the direction of a path of length \( t(J(z)) \) from \( z-t(J(z))v \) to \( \bar{z} = z + t(J(z))e_r \). Consequently a path

\[
\left( \begin{array}{c} z + t(J(z))e_r \\ 0 \\ \end{array} \right) = \left( \begin{array}{c} z - t(J(z))v \\ t(J(z)) \end{array} \right)
\]

is given by the direction vector \( v - e_r = v - e_{n+1} + e_r \). We can extend along the above path until \( t(J(z)) \) leaves \( D \). Now lemma 1 can be augmented with the equations

\[
S(\bar{z}) = \begin{cases} S(\bar{z} - e_r) - 1, & z_{n+1} > 0 \\ A(\bar{z}), & z_{n+1} = 0 \end{cases}
\]

\[
C(\bar{z}) = \begin{cases} C(\bar{z} - e_r), & z_{n+1} > 0 \\ B(\bar{z}), & z_{n+1} = 0 \end{cases}
\]

\[
A(\bar{z}) = \begin{cases} A(\bar{z} - v + e_{n+1} - e_r), & \text{for } \bar{z} - v + e_{n+1} - e_r \text{ in } D \\ \alpha(J(z)), & \text{for } \bar{z} - v + e_{n+1} - e_r \text{ in } D \end{cases}
\]

\[
B(\bar{z}) = \begin{cases} B(\bar{z} - v + e_{n+1} - e_r), & \text{for } \bar{z} - v + e_{n+1} - e_r \text{ in } D \\ \beta(J(z)), & \text{for } \bar{z} - v + e_{n+1} - e_r \text{ in } D \end{cases}
\]

**Case (b):** When \( J(z) \) has no pipelining vectors except \( e_{n+1} \). Observe that

\[
J(\bar{z}) = C\bar{z} + d = Cz + d + t(J(z))Ce_r = J(z) + t(J(z))Ce_r
\]

Consequently a path \( J(\bar{z}) - J(z) \) has direction \( Ce_r \) and is of length \( t(J(z)) \). It follows immediately that the path

\[
\left( \frac{J(\bar{z})}{t(J(z))} \right)
\]
has length \( t(J(z)) \) and direction \( C e_r - e_{n+1} \) and that the \( t(J(z)) \) can be propagated back along this path to the boundary of \( \tilde{D} \). Thus

\[
S(\tilde{z}) = \begin{cases} 
S(\tilde{z} - \delta_r) - 1, & \tilde{z}_{n+1} > 0 \\
A(J(\tilde{z})), & \tilde{z}_{n+1} = 0
\end{cases} 
(4.5a)
\]

\[
C(\tilde{z}) = \begin{cases} 
C(\tilde{z} - \delta_r), & \tilde{z}_{n+1} > 0 \\
B(J(\tilde{z})), & \tilde{z}_{n+1} = 0
\end{cases} 
(4.5b)
\]

\[
A(\tilde{z}) = \begin{cases} 
A(\tilde{z} - C e_r + e_{n+1}), & \tilde{z} - C e_r + e_{n+1} \text{ in } \tilde{D} \\
\alpha(J(\tilde{z})), & \text{Otherwise}
\end{cases} 
(4.5c)
\]

\[
B(\tilde{z}) = \begin{cases} 
B(\tilde{z} - C e_r + e_{n+1}), & \tilde{z} - C e_r + e_{n+1} \text{ in } \tilde{D} \\
\beta(J(\tilde{z})), & \text{Otherwise}
\end{cases} 
(4.5d)
\]

as (4.5) together with lemma 1 produce a system of CAREs the proof is complete. Notice that a system of CUREs result when \( J(z) \) has a pipelining vector different from the trivial vector \( e_{n+1} \) using case (a). □

**COROLLARY 1**: an \( n \)-dimensional recurrence equation

\[ U(z) = g(\ldots, V(I(z)), \ldots) \]

where \( I(z) = K(z) + t(J(z)) \) and \( J(z) = C z + d, K(z) = E z + f \) has an equivalent system of \((n + 1)\)-dimensional CAREs.

**Proof**:

Consider the following three cases

**case (a)**: when \( E = I \) and \( f = 0 \) the problem reduces to that in lemma 3.

**case (b)**: when \( E = I \) and \( f \neq 0 \) apply lemma 3 but update (4.2d) and (4.2f) by replacing \( \tilde{z} \) with \( \tilde{z} + f \) in the mapping functions for variable \( V \).

**case (c)**: when \( E \neq I \) (but non-null) and \( f \) is arbitrary. Use lemma 3 to produce a system of equations but use the path

\[ \begin{pmatrix} J(\tilde{z}) \\ 0 \end{pmatrix} = \begin{pmatrix} J(K(z)) \\ t(J(z)) \end{pmatrix} \]

to deliver \( t(J(z)) \) to \( J(\tilde{z}) \) where \( \tilde{z} = K(z) + t(J(z)) e_r \). This path has the same direction as the one in lemma 3 but the entry point to \( \tilde{D} \) is different. Also modify (4.2c) by substituting \( K(\tilde{z}) \) for \( \tilde{z} \) on the right hand side of the equation. □

**LEMMA 4**: (pumping lemma) the \( n \)-dimensional recurrence

\[ U(z) = f(\ldots, V(I(z)), \ldots) \]

where \( I(z) = z + \Sigma^m \) can be written as a set of \((n + 1)\)-dimensional CAREs.

**Proof**:

First observe that \( \Phi_V(z) = -\Sigma^m \) hence a path from \( I(z) \) to \( z \) can be regarded as a chain passing through the \( m + 1 \) points \( z + \Sigma_j^i \) for \( j = 0(1)m \). The dependency between any two
adjacent points on this chain is given by \( \Sigma_i^{j-1} - \Sigma_i^j = -W_j(J_j(z))c_{r_j} \). Consequently lemma 3 can be applied repeatedly for each link in the chain. All we need to ensure is that the output of the routing path for link \( j \) is the input for the routing path associated with link \( j-1 \) (links being numbered in ascending order from \( z \) to \( l(z) \)). A technical problem with this scheme is that \( t(J_j(z)) \) has to arrive at \( J_j(z + \Sigma_i^j) \) so that it is easily retrieved for input to the \( j \) th routing path. Putting \( z^{(j)} = z + \Sigma_i^j \) it is easy to show that

\[
\begin{bmatrix}
J_j(z^{(j)})

\end{bmatrix}
- \begin{bmatrix}
J_j(z^{(0)})

\end{bmatrix} = \begin{bmatrix}
\left( J_j(z^{(j)}) \right) - \left( J_j(z^{(0)}) \right)

\end{bmatrix} + \begin{bmatrix}
\left( J_j(z^{(j-1)}) \right) - \left( J_j(z^{(0)}) \right)

\end{bmatrix}
\]

so that \( J_j(z^{(0)}) \) can be shifted to \( J_j(z^{(j)}) \) along the path

\[
C_j \sum_{k=0}^{j} W_k(J_k(z^{(0)}))c_{r_k}
\]

It follows that the method for placing \( W_j(J_j(z^{(0)})) \) on a pipelining path from the boundary can be computed locally for each link. With the entry point on the boundary calculated from the point

\[
\begin{bmatrix}
J_j(z^{(j-1)})

\end{bmatrix}
- \begin{bmatrix}
W_j(J_j(z^{(0)}))

\end{bmatrix}
\]

and the line \(-C_j c_{r_j} + e_{n+1}\). Because \( W_j(J_j(z^{(0)})) \) are assumed constant these points can be evaluated at compile-time. Thus the new system of equations has the form

\[
S_i(z) = \begin{cases} 
  S_i(z - e_{r_i}) - 1, & z_{n+1} > 0 \\
  A_i(p_i(z)), & z_{n+1} = 0
\end{cases} \quad \text{(4.6a)}
\]

\[
C_i(z) = \begin{cases} 
  C_i(z - e_{r_i}), & z_{n+1} > 0 \\
  B_i(p_i(z)), & z_{n+1} = 0
\end{cases} \quad \text{(4.6b)}
\]

\[
U(z) = \begin{cases} 
  f, & z_{n+1} = 0 \\
  \infty, & z_{n+1} > 0
\end{cases} \quad \text{(4.6c)}
\]

\[
R_3^{(i)}(z) = \begin{cases} 
  V^{(i)}(z), & z_{n+1} = 0 \\
  R_3^{(i)}(z - e_{r_i}), & z_{n+1} > 0
\end{cases} \quad \text{(4.6d)}
\]

\[
R_2^{(i)}(z) = \begin{cases} 
  R_2^{(i)}(z), & z_{n+1} > 0 \text{ and } S_i(z) = 0 \text{ and } C_i(z) = 1 \\
  R_2^{(i)}(z - e_{r_i}), & \text{Otherwise}
\end{cases} \quad \text{(4.6e)}
\]

\[
R_1^{(i)}(z) = \begin{cases} 
  R_1^{(i)}(z), & z_{n+1} > 0 \text{ and } S_i(z) = 0 \text{ and } C_i(z) = 0 \\
  R_1^{(i)}(z - e_{r_i}), & z_{n+1} > 0 \text{ and } S_i(z) = 0 \text{ and } C_i(z) \neq 0 \\
  R_1^{(i)}(z - e_{r_i}), & z_{n+1} = 0 \text{ and } R_2^{(i)}(z - e_{r_i}) \neq 0 \\
  V^{(i)}(z - e_{r_i}), & z_{n+1} = 0 \text{ and } R_2^{(i)}(z - e_{r_i}) = 0
\end{cases} \quad \text{(4.6f)}
\]

\[
V^{(i)}(z) = \begin{cases} 
  R_1^{(i)}(z), & i = m - 1 \\
  V(z), & i = m
\end{cases} \quad \text{(4.6g)}
\]

\[
A_i(z) = \begin{cases} 
  A_i(z - p_i + e_{n+1}), & \text{for } z - p_i + e_{n+1} \text{ in } \hat{D} \\
  \alpha_i(J_i(z)), & \text{Otherwise}
\end{cases} \quad \text{(4.6h)}
\]

\[
B_i(z) = \begin{cases} 
  A_i(z - p_i + e_{n+1}), & \text{for } z - p_i + e_{n+1} \text{ in } \hat{D} \\
  \beta_i(J_i(z)), & \text{Otherwise}
\end{cases} \quad \text{(4.6i)}
\]
where \( \alpha_i(J_i(z)) = [W_i(J_i(z))/2], \beta_i(J_i(z)) = [W_i(J_i(z))/2] - [W_i(J_i(z))/2], i = 1(1)m, \)
\[
z_{n+1} = 0(1)W_{max}, \text{ with } W_{max} = \max_{i=1}^{m} W_i(J_i(z))
\]

and,
\[
p_i = \begin{cases} 
    v_i + e_{ri}, & \text{when } C_i v_i = 0 \\
    C_i e_{ri}, & \text{Otherwise}
\end{cases}
\]
\[
P_i(\bar{z}) = \begin{cases} 
    \bar{z}, & \text{when } C_i v_i = 0 \\
    J_i(\bar{z}), & \text{Otherwise}
\end{cases}
\]

The extension to corollary 1 for the general case follows immediately. We can now state our mapping theorem as follows

**THEOREM 1:** the \( n \)-dimensional recurrence
\[
U(z) = f(\ldots, V_k(I_k(z)), \ldots)
\]
can be written as a set of \( (n + K) \)-dimensional CAREs where \( k = 1(1)K \).

**Proof:**

Scan the arguments of the above equation for \( k = 1(1)K \) applying the appropriate lemma for each run time dependent \( I_k(z) \). Each application of a lemma increases the dimension of the domain so that in the worst case the final domain is of dimension \( (n + K) \).

Theorem 1 gives us a systematic method for re-writing a recurrence equation argument by argument until all the RTD's been removed. It is tempting, at this stage, to reduce the significance of the result by emphasizing the fact that the variables in each \( I_k(z) \) are known in advance so that the problem of RTD synthesis is a side issue. In general this statement may be true, but the restricted set of problems we have identified in section 3 are precisely the set of RTDs which exhibit this property. This class is of significance because it includes a number of combinatorial algorithms including the Knapsack problem which cannot be solved using existing synthesis techniques. Section 6 demonstrates how the theorem here can be applied directly to synthesize systolic arrays for this problem.

5. Discussion

However before proceeding it is worthwhile spending time on a few paragraphs about the design technique using the results. First let us deal with some general issues. The first point is that it appears straightforward to pick a pipelining path for the \( W_j \) variables based only on the point \( \bar{z} \) and an arbitrary direction. While this technique is valid it is also rather ad-hoc. The lemmas and theorem provide a systematic method of choosing the pipelining vector based on the original point \( z \). Indeed the method of construction can be easily automated. The second (and most important) issue is the increase in dimension associated with each
application of the lemma's. Increasing the dimension of the domain has a positive effect in 
that it also increases the degrees of freedom in the design process. For example more feasible 
allocations are possible and hence more array designs. A negative effect is that the resulting 
arrays can have high dimensions. If one is considering a VLSI implementation where 2-D 
and 1-D dimensions are preferred the technique can produce transformed designs that are 
no longer practical. In contrast if one is considering a logical mapping to a programmable 
or reconfigurable architecture the issue is a minor one. Alternatively we can use a variation 
of the technique which simply increases the dimension of the domain once and then maps 
all routing and pipelining paths through this extra dimension. Confining the new domain to 
\((n+1)\)-dimensions alleviates the allocation problem but reduces the degrees of freedom in the 
design process. So far we have said very little about the formation of cyclic routing paths 
and scheduling of the timing function. Should the extended DDG become cyclic producing 
a timing schedule will be difficult (if not impossible - certainly the automatic methods begin 
to breakdown). Restricting the dimensionality of the domain compounds the problem.

Next consider the program in section 3 where we restricted the scope of variables in the 
recurrence equations splitting them into two classes. The first class produced results that 
could be used as RTD's but did not contain RTD arguments. The second class admitted 
RTD arguments. The computation of the program maybe envisaged as two DDG's \(G_1\) and 
\(G_2\) embedded into the same space. For convenience assume that \(G_1\) has been translated 
away from \(G_2\) into a non-overlapping part of \(\mathbb{Z}^n\) (this can be achieved by a simple geometric 
transformation). By definition \(G_1\) is static (known at compile-time) while \(G_2\) changes from 
one iteration of the loop to the next becoming \(DDG(r)\) on iteration \(r\). \(G_1\) and \(G_2\) are connected. 
The results of the recurrences associated with \(G_1\) are connected to nodes in \(G_2\) and are 
fixed. It is now possible to see the full effect of our mapping technique. Applying the 
results of the previous section produces a static DDG for all iterations of the loop hence 
\(G_2\) is static. The run-time variations present in each iteration \(r\) are "chased" out to the 
boundary of the domain enclosing \(G_2\) where they become inputs. Consequently the static 
connections between \(G_1\) and \(G_2\) corresponding to RTD variables and become dynamic. In a 
general computational strategy different arrays for \(G_1\) and \(G_2\) are produced and connected 
by a reconfigurable interface. Computation proceeds in three stages 1: in the first stage we 
evaluate the recurrences associated with \(G_1\), in the second stage the \(G_1\) and \(G_2\) connections 
are set, and in the third stage the computation associated with \(G_2\) is performed. Clearly 
there are many details related to this organisation, for example can we map \(G_1\) and \(G_2\) on 
the same final array, can we ensure smooth throughput between the stages, and is a general 
timing function over both graphs always possible. These issues are interesting topics for 
future work, for our part and the remainder of this paper we will demonstrate our technique 
on a problem where there is no need to construct \(G_1\) explicitly. The reconfiguration part 
then reduces to the standard problem of host input-output organisation for the array.

6. UREs for the Knapsack Problem

Consider the so-called Knapsack problem. Briefly the problem can be stated as follows.
For a Knapsack of capacity \( c > 0 \) and \( g \) items each with weight \( w_j \) and value \( v_j \) the problem is

\[
\max \left( \sum v_j q_j \right), \; v_j \geq 0, \; j = 1(1)g
\]  \hspace{1cm} (6.1)

Subject to \( \sum w_j q_j \leq c, \; w_j \geq 0, c > 0, j = 1(1)g \)

where \( q_j \) is the number of items of type \( j \) included in the Knapsack [14]. Solving the problem using the dynamic programming problem produces the following recurrence equation

\[
F(k, y) = \max \left( F(k - 1, y), F(k, y - w_k) + v_k \right)
\]  \hspace{1cm} (6.2)

with \( F(k, y) = -\infty \) when \( y < 0, F(0, y) = 0, F(k, 0) = 0 \) for \( y = 0(1)c \) and \( k = 0(1)g \). It is generally acknowledged that regular array for this problem cannot be automatically synthesized using normal techniques [19]. Using our definitions it is clear that this is because the DDG is run-time dependent. However a program for evaluating the table generated by (6.2) for fixed \( g \) and \( c \) satisfies the restrictions in section 3. For example we can compute a number of Knapsack problems by repeatedly reading the weights (i.e. the \( w_k \) values for a particular instance) and then constructing the table. During the table construction the \( w_k \) are constant.

For our purposes a more appropriate form of (6.2) is the 2-D equation

\[
F(z) = f(F(z - e_1), F(z + w(J(z))e_2), v(J(z))
\]  \hspace{1cm} (6.3)

where \( f \) computes the \( \max \) function required and

\[
J(z) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \; e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \; e_2 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}
\]

Clearly our theorem using the pumping lemma with \( m = 1 \) is applicable. First add an extra dimension to all vectors and matrices (i.e. add zero rows and columns) and put

\[
e_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \; \bar{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \; \bar{e}_2 = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}
\]

\[
e_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \; z = \begin{pmatrix} k \\ y \\ x \end{pmatrix}
\]

Clearly \( v' = (0, 1, 1) \) is a valid (non-unique) pipelining vector and \( z = 0(1)c \). Now substituting into (4.6) and some simple manipulation produces the following system of recurrence equations

\[
S_1(k, y, x) = \begin{cases} S_1(k, y - 1, x - 1), & x > 0 \\ A_1(k, y, x), & z = 0 \end{cases}
\]  \hspace{1cm} (6.4a)

\[
C_1(k, y, x) = \begin{cases} C_1(k, y - 1, x - 1), & x > 0 \\ B_1(k, y, x), & z = 0 \end{cases}
\]  \hspace{1cm} (6.4b)

\[
U(k, y, x) = \begin{cases} f(F(k - 1, y, x), F^{(1)}(k, y - 1, x + 1), V(k, y, x)), & x = 0 \\ \infty, & x > 0 \end{cases}
\]  \hspace{1cm} (6.4c)
\[ R^{(1)}_3(k, y, z) = \begin{cases} F(k, y, z), & z = 0 \\ R^{(1)}_3(k, y - 1, z - 1), & x > 0 \end{cases} \quad (6.4d) \]
\[ R^{(1)}_2(k, y, z) = \begin{cases} R^{(1)}_3(k, y, x), & x > 0 \text{ and } S_1(k, y, x) = 0 \text{ and } C_1(k, y, x) = 1 \\ R^{(1)}_2(k, y - 1, x), & \text{Otherwise} \end{cases} \quad (6.4e) \]
\[ R^{(1)}_1(k, y, z) = \begin{cases} R^{(1)}_2(k, y - 1, x + 1), & x > 0 \text{ and } S_1(k, y, x) = 0 \text{ and } C_1(k, y, x) = 0 \\ R^{(1)}_1(k, y - 1, x + 1), & x = 0 \text{ and } R^{(1)}_1(k, y - 1, x) \neq 0 \\ F(k, y - 1, x), & \text{Otherwise} \end{cases} \quad (6.4f) \]
\[ A_1(k, y, z) = \begin{cases} A_1(k, y - 1, x), & \text{for } (k, y - 1, x) \text{ in } \bar{D} \\ [W(J(k, y, z))]/2, & \text{Otherwise} \end{cases} \quad (6.4g) \]
\[ B_1(k, y, z) = \begin{cases} B_1(k, y - 1, x), & \text{for } (k, y - 1, x) \text{ in } \bar{D} \\ [W(J(k, y, z))]/2 - [W(J(k, y, z))/2], & \text{Otherwise} \end{cases} \quad (6.4h) \]
\[ V(k, y, z) = \begin{cases} V(k, y - 1, x), & z = 0 \\ \infty, & x > 0 \end{cases} \quad (6.4i) \]

which are uniform and three dimensional. Observe that (6.4i) is introduced as a result of pipelining the variable \( v \) and that \( V(k, 0, 0) = v_k \). A valid timing function which can be derived using normal scheduling techniques is given by

\[ t(z) = (1, 1, 0)^T z - 1 \]

many allocation functions exist, but two useful ones are

\[ a(z) = (-1, 0, 0)^T z \text{ and } a(z) = (0, -1, 0)^T z \]

Notice that \((0, 0, -1)^T\) is not permitted. Clearly the time for the computation is given directly from \( t(z) \) as \( g + c - 1 \) while allocations produce arrays with \( O(gc) \) and \( O(c^2) \) cells. The exact description of arrays for the above \((t(z), a(z))\) pairs are discussed fully in [23] where a fixed size array is shown to be optimal and the upper bound on \( x \) is reduced to \([c/2]\).

Finally, a variation of the Knapsack problem is the 0/1 Knapsack problem

\[ F(k, y) = \max(F(k - 1, y), F(k - 1, y - w_k) + v_k) \quad (6.5) \]

where the \( x_j \) in (6.1) are either 0 or 1. Using our technique we can see that

\[ I(z) = K(z) + W(J(z))c_2 \]

where \( J(z) \) is the same as before and \( K(z) = z - c_1 \), which according to corollary 1 amends (4.4d and f) using case (b) so that \( F(k - 1, y, x) \) and \( F(k - 1, y - 1, x) \) are referenced on the right hand side. The change is only a minor one, the timing and allocation functions are still valid allowing the derivation of alternative arrays for the 0/1 Knapsack problem.

7. Summary

We have characterized a restricted class of run-time dependent recurrence equations. Such equations produce peculiarities in the associated Data Dependency Graphs of algorithms
which prevent the application of current synthesis techniques. Included in our class of problems is the so-called Knapsack problem which has received attention in the systolic literature recently. Unfortunately all the designs proposed are based on ad-hoc methods.

In this paper we have established a number of lemmas and a theorem for the systematic re-writing of recurrences with run-time dependencies for our class of problems and demonstrated their use on the Knapsack problem. A set of uniform recurrence equations for the Knapsack problem have been derived and arrays for the problem synthesized in the usual manner. One of the arrays produced is known to be optimal (see [23]).

The same techniques are applicable to a large number of algorithms which employ restricted forms of run-time dependencies, many of which can be found in combinatorics - a ripe area for parallel processing. Thus the technique widens the class of algorithms for which regular and in particular systolic arrays can be derived. Further work is required in extending the class of run-time dependencies that can be considered.

Acknowledgements

This work is supported by SERC grant GR/F 80494. Equations have been checked using the DTAGS transformation kernel.

References