Knapsack on VLSI: from Algorithm to Optimal Circuit*

Rumen Andonov† and Sanjay Rajopadhye‡

Abstract

We present a parallel solution to the unbounded knapsack problem on a linear systolic array. It achieves optimal speedup for this well-known, NP-hard problem on a model of computation that is weaker than the PRAM. Our array is correct by construction, as it is formally derived by transforming a recurrence equation specifying the algorithm.

This recurrence has dynamic dependencies, a property that puts it beyond the scope of previous methods for automatic systolic synthesis. Our derivation thus serves as a case study. We generalize the technique and propose a systematic method for deriving systolic arrays by non-linear transformations of recurrences. We give sufficient conditions that the transformations must satisfy, thus extending systolic synthesis methods.

We address a number of pragmatic considerations: implementing the array on only a fixed number of PEs, simplifying the control to just two counters and a few latches, and loading the coefficients so that successive problems can be pipelined without any loss of throughput.

Using a register level model of VLSI, we formulate a non-linear optimization problem to minimize the expected running time of the array. The analytical solution of this problem allows us to choose the memory size of each PE in an optimal manner.

*Supported by the French Coordinated Research Program C^3, and by the Esprit BRA project NANA 2 (No. 6632). The second author was also partially supported by NSF Grant No. MIP-910852. Preliminary versions of parts of this work have been reported in CONPAR '92 [AQ92], and ASAP '93 [AR93].

†Institute de Sciences et Technologie de Valenciennes, Université de Valenciennes, Le Mont Houy, B. P. 311, 59304 Valenciennes Cedex, France; on leave from Center of Computer Science and Technology, Sofia, Bulgaria. Email: `andonov@univ-valenciennes.fr`

‡IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, and Computer Science Dept, Oregon State University, Corvallis, OR, USA. Email: `rajopadhye@irisa.fr`

1 Introduction

The unbounded knapsack problem is a classic NP-*hard*, combinatorial optimization problem with a wide range of applications [Hu69, MT90]. It can be solved sequentially in $\Theta(mc)$ time, where m is the number of objects and c is the capacity. This is called *pseudo polynomial* complexity [GJ79] since it is polynomial with respect to a parameter, c, which itself, can be an exponentially growing function of input size. Many time consuming instances of these problems exist [CHR88], and it is therefore important to investigate efficient parallel implementations. One of the standard approaches to solving this problem is dynamic programming [Bel57, GN72, Hu69]. In this paper, we address the problem of deriving a systolic VLSI array for the dynamic programming algorithm for the knapsack problem.

Systolic array synthesis has been an active research area for some time, and there is now a large body of work on deriving systolic arrays from *systems of recurrence equations.* Typically, one starts with a recurrence of the following form.

$$\text{for all index points, } z, \quad X[z] = g(\ldots Y[d(z)] \ldots) \tag{1}$$

It specifies that in order to calculate the variable X at any point z, it is necessary to apply the atomic function g to its arguments. All arguments are of the form, $Y(d(z))$; i.e., the variable Y at a point $d(z)$. Y may be defined similarly, by its own recurrence equation, or it may be an input. The function d is called a dependency function. The problem of synthesizing systolic arrays from recurrences is now well developed. The simplest case is when all the dependencies are *uniform*, i.e., $d(z) = z + a$ for some *constant* a. The synthesis techniques for this case are now more than a decade old and are based on a seminal paper by Karp et al. [KMW67] which even predates systolic arrays. The synthesis problem can be reduced to determining appropriate linear scheduling and allocation functions. The problem is a little more complicated when the dependency function has the form $d(z) = Az + a$ for some constant matrix, A, and a constant vector, a (this is called an *affine* dependency). In addition to determining linear scheduling and allocation functions, it is also necessary to *localize* the recurrence (i.e., convert it into an equivalent uniform recurrence). This problem, too, has received a satisfactory solution (see [RF90], [QR89] Ch 13, and references therein), and many researchers have developed software systems for systolic synthesis from systems of affine recurrences.

The dynamic programming problem for the knapsack problem has a very simple specification as a recurrence equation. However, we shall see that the dependency function here is neither uniform nor affine. Indeed, the function $d(z)$ is different for different z and may change from one problem instance to another. Such a class of recurrences are said to have *dynamic* (or *run-time*) dependencies [Meg93], and the well-known synthesis techniques are no longer applicable.

A complete synthesis method for such recurrences is not available at present.

Thus, there are two motivations for studying the systolic implementation of the dynamic programming algorithm for the knapsack problem: first, the problem is important in its own right and would benefit from parallel VLSI implementation on dedicated processor arrays; second, a careful study of this particular recurrence may lead to a synthesis method for recurrences with dynamic dependencies. In this paper we present the following results.

- We describe a linear systolic array for the unbounded knapsack problem. The array is modular (all PEs are identical, independent of *all* problem parameters), and each PE has only a fixed memory (α words). The array can be adapted to run on only q PEs (connected in a ring). Then, its *average* time complexity is $\Theta\left(\frac{mc}{q}\left\lceil\frac{w_{\max}+w_{\min}}{\alpha}\right\rceil\right)$, where $w_{\max}$ and $w_{\min}$ are, respectively, the expected largest and smallest object weights among all problem instances. When the PE memory is large enough (this is the case for *general purpose machines*, and is implicitly assumed by all researchers), this yields optimal speedup. For VLSI implementation we use a more precise notion of optimality as discussed below.

- Our array is *correct by construction:* it is formally derived by applying transformations to an initial system of recurrence equations that specify the dynamic programming algorithm.

- The derivation serves as a case study for systolic synthesis in the presence of dynamic dependencies and highlights the fact that it is not enough to merely determine a conflict free space-time transformation, as in existing synthesis methods. We tackle the additional problems that need to be resolved and give sufficient conditions that a transformation must satisfy, in order that the resulting recurrence can be interpreted as a systolic array.

- We address some pragmatic considerations: in addition to implementing the array on only a fixed number of PEs, we simplify the control to just two counters and a few latches and give a method for loading the coefficients so that successive problems can be pipelined without any loss of throughput.

- In terms of practical implementation, we seek an optimal tradeoff (between memory and processors) given that we have fixed silicon resources. Using a register level model of VLSI, we formulate a non-linear optimization that seeks to minimize the expected running time of the array (for a uniform distribution of problem instances). We solve it analytically, enabling us to choose the memory size and yielding an optimal circuit.

It is our strong conviction that in order to obtain good implementations, one must investigate the problem at many different levels: from algorithms, through architecture, and all the way to

VLSI circuit level optimizations. Our results highlight the cross disciplinary nature of designing application-specific array processors, or "algo-tech-cuit engineering".

The remainder of this paper is organized as follows. In the next section, we provide a review of background and notation, define the problem, and derive a linear array for the knapsack problem which, although not systolic, serves as a starting point of our synthesis. In Sec. 3 we give the systematic derivation of the systolic array and illustrate how the synthesis methods must be modified to adapt to dynamic dependencies. In Sec. 4 we deal with the pragmatic aspects, and in Sec. 5 we show how the PE memory size which minimizes the running time can be determined by solving a non-linear optimization problem. In Sec. 6, we develop a general synthesis method for deriving systolic arrays with non-linear transformations. Finally, in Sec. 7, we conclude with a summary of related work, and an outline of future work.

2 Background, Problem and Naive Solution

We briefly review the techniques of systolic synthesis, formally define the algorithm that we want to map to silicon, and present a naive solution that serves as our starting point.

2.1 Review of systolic array synthesis

We start with a specification in the form of a system of uniform recurrence equations, or UREs [KMW67]. Associated with each such system is its *data flow graph*, which models every computation as a distinct node, and whose edges indicate the precedences for the computation. Note that the data flow graph may be arbitrarily large (even infinite for some problems), but due of uniformity, it can be represented compactly by means of a *finite* number of constant vectors, called *dependence vectors* or *dependencies*. To derive a systolic array from such recurrences, we assume that the body of the equation can be computed in one step (by means of special hardware units), and seek two functions, namely, an *allocation function* and a *timing function* (also called a *schedule*), that map each index point to a processor location and a time instant, respectively. Together, they constitute a *space-time* mapping or transformation of the original recurrence and its data flow graph. The timing function must satisfy *causality*, meaning that no computation can be scheduled until its arguments have been computed. Furthermore, the allocation and the timing functions must not *conflict*, i.e., two distinct points must never be mapped to the same PE at the same time. Note that in general, a space-time transformation may yield an array where all PEs are not active at all time instants (we say that a space-time point is *active* if it is the image of some point in the original index domain). For a number of reasons, it is very useful to restrict these functions to be linear (or affine) functions of the indices:

- Many techniques have been developed to derive linear schedules systematically using integer linear programming. The number of constraints is the (finite, and usually small) number of distinct dependence vectors, independent of the size of the data flow graph.
- A linear schedule is specified by a single vector, λ.
- Such schedules are known to be optimal for UREs.
- A linear *allocation function* is characterized by a single projection vector, u.
- Ensuring that the mapping is conflict free involves a simple test that $\lambda^{\mathrm{T}} u \neq 0$.
- Techniques exist for systematically enumerating linear allocation functions.
- If linear space-time transformations are used, the resulting recurrence (and its data flow graph) remain uniform. Indeed, the new URE can be *automatically* derived by means of simple rewriting rules. We say that UREs are *closed* under linear transformations. As we shall see, this is crucial for the synthsis, because the new dependencies can be directly interpreted as the interconnection links of the processor array and may be used to generate masks automatically for VLSI layouts.

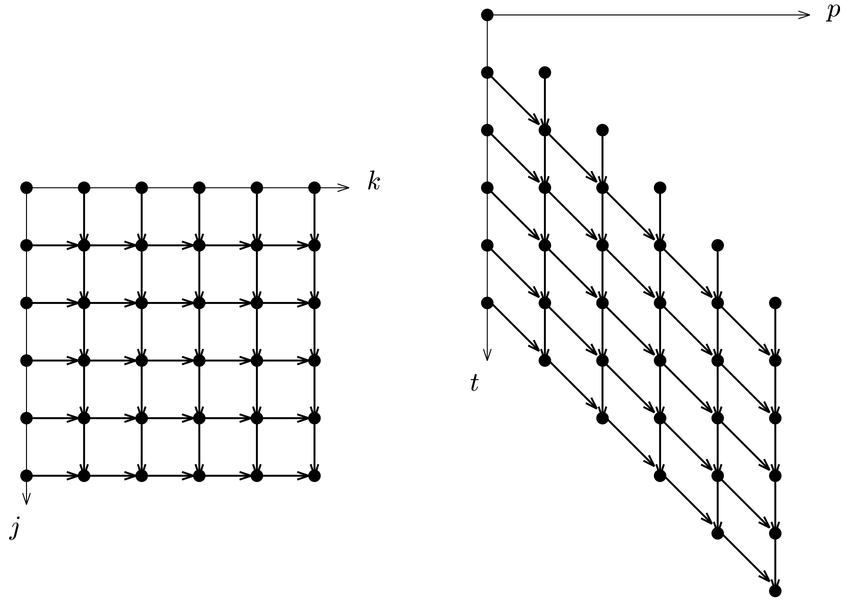
To illustrate this, consider the problem of designing a systolic array for the URE of Eqn. (2), below. It specifies the computation of X over an $N \times N$ *domain*, initialized to 0 at the boundaries. It has two dependencies, $[0,1]$ and $[1,0]$, and its data flow graph is shown in Fig. 1-a.

$$X(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0 \\ X(j,k-1) * X(j-1,k)) & \text{if } 0 < j,k < N \end{cases} \tag{2}$$

Because the URE has two dependencies, the (linear) schedules must satisfy just two constraints, namely that $t(0,1) > 0$, and $t(1,0) > 0$, and thus, $t(j,k) = j + k$ is a valid (indeed, the optimal) schedule. Also, $a(j,k) = k$ is a valid allocation function. The corresponding space-time transformation is shown in Fig.1-b, and when it is applied to Eqn. (2), we obtain the URE below, whose data flow graph is shown in Fig.1-c.

$$X(t,x) = \begin{cases} 0 & \text{if } t = x \text{ or } x = 0 \\ X(t-1,x-1) * X(t-1,x)) & \text{if } 0 < x < t \leq N \end{cases} \tag{3}$$

Its dependencies are $[1,1]$ and $[1,0]$, and it can be interpreted as an array where each PE sends one value to its right neighbor (with one clock cycle delay), and updates an internal register every cycle (see Fig. 1-d).

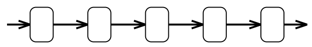


(a) Original data flow graph

(c) Transformed data flow graph

$$\begin{bmatrix} t \\ p \end{bmatrix} = \begin{bmatrix} t(j,k) \\ a(j,k) \end{bmatrix} = \begin{bmatrix} j+k \\ k \end{bmatrix}$$

(b) Space time transformation



(d) Final systolic array

Figure 1: Overview of systolic synthesis

2.2 The Knapsack Problem

Suppose that m types of objects are being considered for inclusion in a knapsack of capacity c. For $i = 1, 2, \ldots, m$, let p_i be the *profit* and w_i the *weight* of the i-th type of object, where w_i, p_i, and c are all positive integers. The knapsack problem is to choose a collection of objects so as to maximize the total profit without exceeding the capacity constraint, i.e.,

$$\max \left\{ \sum_{i=1}^{m} p_i z_i \;:\; \sum_{i=1}^{m} w_i z_i \leq c,\ z_i \geq 0 \text{ integer},\ i = 1, 2, \ldots, m \right\} \tag{4}$$

where z_i is the number of i-th type objects included in the knapsack. The problem, as specified above is often called the *unbounded knapsack problem*, since the only constraint on the solution (other than the capacity constraint) is that it is non-negative. There are many variations of this problem [MT90], such as the *bounded knapsack problem* (here, additional constraints of the form

$z_i \leq b_i$ must be satisfied), the *0/1 knapsack problem* (a particular case of the bounded problem, where $b_i = 1$: there is exactly one copy of each type of object), the *subset sum problem* (a 0/1 problem with $w_i = p_i$), the *change making problem*, etc. They arise in different application domains, and are all NP-hard.

The most common (exact) algorithmic techniques for solving knapsack problems are *dynamic programming* [LSS88, CCJ90, LS91, Ten90] and *branch-and-bound* [LS84, JF88]. In this paper, we concentrate on dynamic programming, which is based on the *principle of optimality* [Bel57] and consists of two phases. In the first (forward) phase, the optimal value of the profit function is computed; this is used in the second (backtracking) phase to find the actual solution. The forward phase is usually formulated in a recursive manner by decomposing the problem into sub-problems. In particular, we define a function $f(j,k)$ which denotes the value of an optimal solution of the sub-problem, where only the first k objects are considered and only a capacity of j is available. It is well-known that the computation of $f(j,k)$ is specified recursively as follows. For $[j,k] \in \mathcal{D}$

$$f(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0 \\ f(j,k-1) & \text{if } k > 0 \text{ and } j < w_k \\ f(j,k-1) \oplus (p_k + f(j-w_k, k-\beta)) & \text{if } k > 0 \text{ and } j \geq w_k \end{cases} \quad (5)$$

where $\mathcal{D} = \{[j,k] : 0 \leq j \leq c; 0 \leq k \leq m\}$, is the domain of our recurrence. The different variations of the knapsack problem merely correspond to different choices of the operator $\oplus$ and the constant β in Eqn (5): in the unbounded knapsack problem, $\oplus$ is max, and $\beta = 0$; in the 0/1 and subset-sum problems, $\oplus$ is max and $\beta = 1$; in the change making problem $\oplus$ is min and $\beta = 0$ (see [MT90] for details).

Traditionally, $f(j,k)$ is viewed as a $m \times c$ table which is "filled up" during the forward phase. The backtracking phase consists of using this table of $f(j,k)$ to determine the actual solution. For the unbounded knapsack problem, Hu has given an elegant, memory efficient implementation of this phase [Hu69]. If certain book-keeping information is maintained during the forward phase, the solution can be obtained in $\Theta(c+m)$ time and c space (only the last column needs to be saved, not the entire table). The book-keeping corresponds to simply keeping track of which of the two arguments $f(j,k-1)$ or $p_k + f(j-w_k,k)$ was chosen in calculating $f(j,k)$. At present, the question of whether a similar technique exists for the other variants of the knapsack problem is still open. This is one of the reasons why we consider only the unbounded knapsack problem in this paper. All our results can be trivially extended to the *forward* phase of the other variants.

2.3 A Naive Array

Our problem is formally stated as follows. For the following system of recurrences, defined over the domain, $\mathcal{D}$, derive a systolic array that computes, for $0 < j \leq c$, the values of $f(j,m)$ and $u(j,m)$.

$$f(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0 \\ f(j,k-1) & \text{if } k > 0 \text{ and } j < \boxed{w_k} \\ f(j,k-1) & \text{if } k > 0 \text{ and } j \geq \boxed{w_k} \text{ and} \\ & \qquad f(j,k-1) > p_k + f(j - \boxed{w_k}, k) \\ p_k + f(j - \boxed{w_k}, k) & \text{if } k > 0 \text{ and } j \geq \boxed{w_k} \text{ and} \\ & \qquad f(j,k-1) \leq p_k + f(j - \boxed{w_k}, k) \end{cases} \tag{6}$$

$$u(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0 \\ u(j,k-1) & \text{if } k > 0 \text{ and } j < \boxed{w_k} \\ u(j,k-1) & \text{if } k > 0 \text{ and } j \geq \boxed{w_k} \text{ and} \\ & \qquad f(j,k-1) > p_k + f(j - \boxed{w_k}, k) \\ k & \text{if } k > 0 \text{ and } j \geq \boxed{w_k} \text{ and} \\ & \qquad f(j,k-1) \leq p_k + f(j - \boxed{w_k}, k) \end{cases} \tag{7}$$

Observe that we have rewritten Eqn. (5) to account for the fact that $\oplus$ is max and $\beta = 0$, and have split the last line into two conditions. By doing so, we see that the two recurrences have *identical* conditional branches. Also note that the body of Eqn. (7) involves no real computation. Hence, any PE that computes $f(j,k)$ can be easily adapted to also compute $u(j,k)$ by adding two registers and a multiplexor. Henceforth, we will not deal explicitly with the calculation of $u(j,k)$, except to account for this additional hardware cost.

Now, we see that in order to compute $f(j,k)$ at any given point (other than those near the boundaries), we need two arguments: one that comes from the point $[j, k-1]$, and the other from $[j - w_k, k]$. The dependencies are thus $[0,1]$, which is uniform, and $[w_k, 0]$, which is different from one column to another, and will doubtless be different for different problem instances (see Fig. 2). It is this that complicates the problem. Indeed, this recurrence is neither uniform nor even affine, since the dependency is not even known at compile time. Such recurrences are said to have *dynamic* or *run-time* dependencies [Meg93]. In spite of this, let us continue to use the notion of timing and allocation functions, and try to obtain a systolic array for it. There are two crucial observations that allow us to synthesize a locally connected (albeit, non-systolic) array, even though we do not completely know the dependency graph statically:

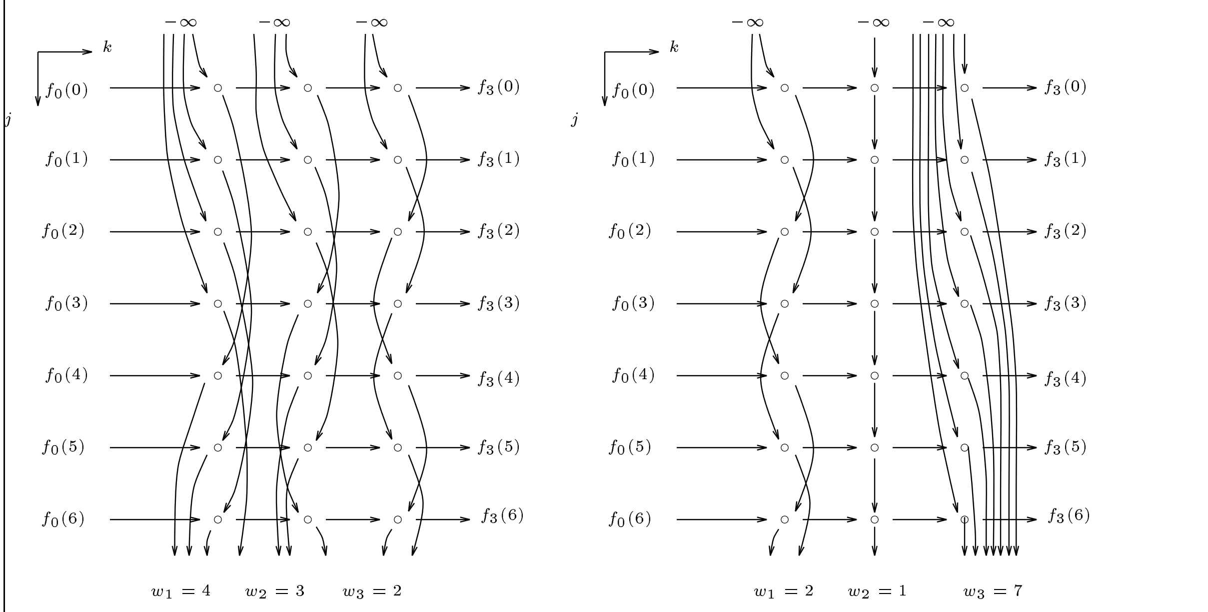


Figure 2: Two knapsack problem instances: note how the graphs vary with the w_i's

- The dynamic component of the dependencies is **vertical**. This implies that if all the points in the k-th column are allocated to the k-th PE then, for any computation performed by this PE, its two arguments must come from the $k-1$ th PE, and the k-th PE itself respectively . In other words, if we choose the allocation function to be $a(j,k) = k$, all the "irregularity is projected within each processor" and the interconnections will be localized.

- The dynamic component of the dependencies is also **downward**. As a result, a valid linear schedule is given by $t(j,k) = j + k$.

Notice that our timing and allocation functions are linear, and can be used to obtain the transformed recurrence shown below. Please try to visualize the transformed data flow graph (the transformation is the same as in Fig. 1) and verify the boundary conditions and dependencies in the following recurrence:

$$f(t,x) = \begin{cases} 0 & \text{if } t = x \text{ or } x = 0 \\ f(t-1,x-1) & \text{if } x > 0 \text{ and } t - x < \boxed{w_x} \\ f(t-1,x-1) & \text{if } x > 0 \text{ and } t - x \geq \boxed{w_x} \text{ and} \\ & \qquad f(t-1,x-1) > p_x + f(t-\boxed{w_x},x) \\ p_x + f(t-\boxed{w_x},x) & \text{if } x > 0 \text{ and } t - x \geq \boxed{w_x} \text{ and} \\ & \qquad f(t-1,x-1) \leq p_x + f(t-\boxed{w_x},x) \end{cases} \tag{8}$$

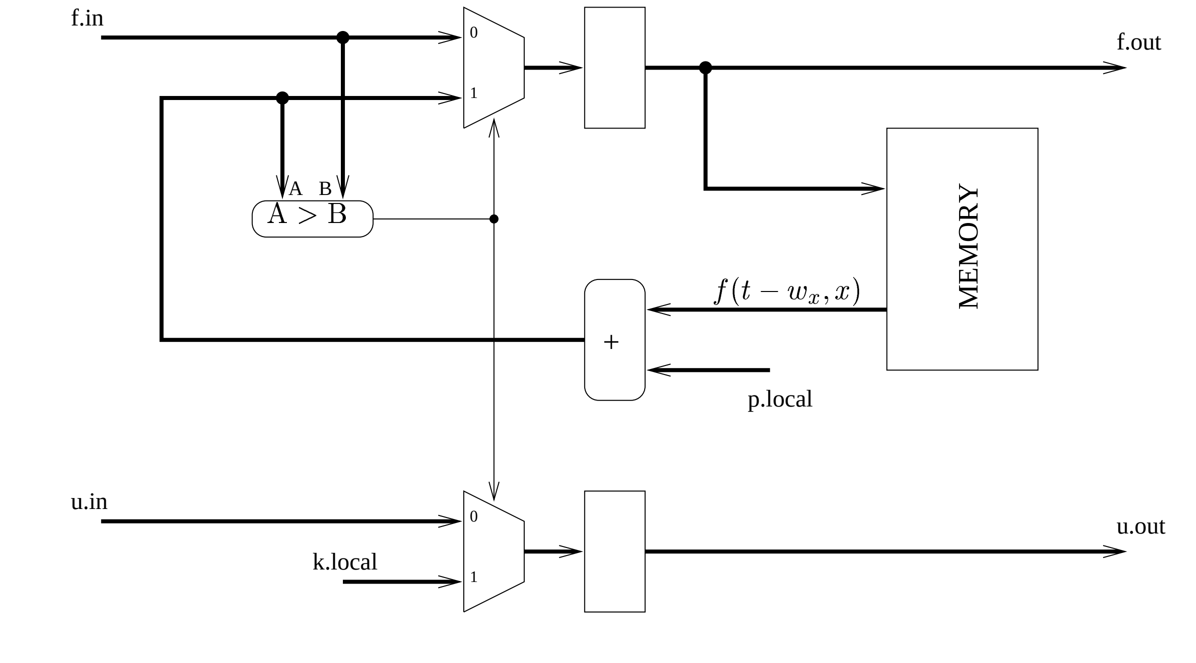


Figure 3: Naive data path (registers p.local and k.local are initialized for each problem)

Let us look closely at this recurrence and interpret x as the PE label, and t as the time. Note that when $x = 0$, the value produced is always 0, so PE 0 is trivial. All other PEs have the same structure (no need to implement the test for $x > 0$ explicitly). The PEs start at $t = x$, and this can be implemented as a control signal propagating from one PE to the next. The value produced by any PE is either $f(t-1, x-1)$, i.e., the value produced by the preceding PE in the previous cycle, or $p_x + f(t - w_x, x)$. Thus, we have a linear array, where each PE has the simple architecture shown in Fig. 3 (we have omitted the additional control for implementing the second line of Eqn. (8), for simplicity). However, notice that at any time instant, the PE needs a value that was calculated at $t - w_x$. Hence any result must be stored for exactly w_x time steps, and this implies a memory of size w_x in each processor, accessed cyclically, modulo w_x (alternatively, it may be viewed as a shift register of length w_x). Since the w's are problem parameters, and could be arbitrarily large, this architecture *does not represent a systolic array!*

3 The Systolic Knapsack Array and its Synthesis

We first present an intuitive explanation of our array and its derivation, then give a formal proof of correctness, and finally address the necessary control.

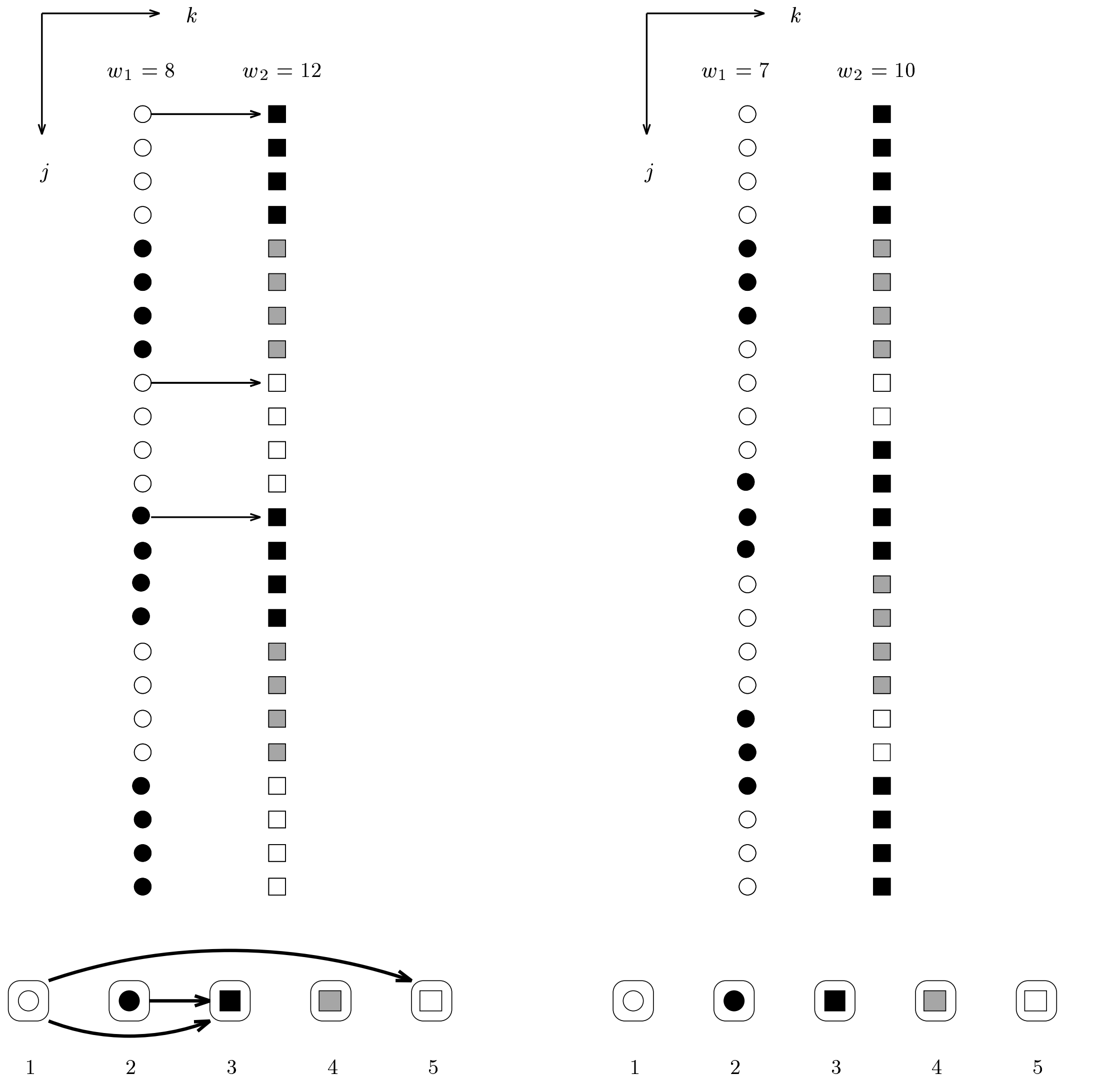
3.1 Informal Explanation

In order to obtain a systolic array, we pose the question from the opposite perspective: how can we implement Eqn. (6) on an array whose PEs have only constant memory (say each PE has only α words of memory)? An obvious answer is to continue with a space-time transformation similar to the one above, but to allocate each column to a *block* of contiguous PEs rather than a single PE. In general, the k-th column will be allocated to $\lceil \frac{w_k}{\alpha} \rceil$ PEs, so that, between them, they have sufficient memory. In other words, Eqn. (8) corresponds to a *virtual* processor array, which is *emulated* in a blocked fashion by a linear array with only α memory per PE. Although this idea seems straightforward, there are a number of complications that arise.

In order to ensure that the dynamic dependency is properly handled, we would still like to allocate the point, $[j, k]$ to the same PE as the point $[j - w_k, k]$. One way to do this is to divide the column into *blocks* of size w_k, allocate the first α points of each block to the first PE, the next α points to the second one, and so on. Subsequent blocks are allocated similarly. This is illustrated in Fig. 4. Please do the exercise given there, and it will be clear that in general, the $[0, 1]$ dependency must cross a number of PEs, and that different instances (of the *same* dependency) must reach different destination PEs. Add to this the fact that the w_k's may change from one problem instance to another, and it is impossible to predict statically which specific PEs will require the outputs of any given PE. Hence, if we naively interpret the image of the dependency vector by the allocation function as an interconnection link of the derived architecture, we will end up with a fully connected network! We conclude that although the idea of blocking appropriately many PEs into a single virtual processor is simple, there are many subtle problems. The problems arise because we are using a non-linear allocation function, and this class of mappings itself was imposed on us because the dependencies are dynamic.

Propagation and propagation conflicts To solve this problem we again use one of the most well-known systolic synthesis techniques: instead of requiring dedicated interconnections for *each* communication, we *pipeline* the data values through the intervening PEs. Although this idea, too, seems simple at first glance, it requires a careful choice of the timing function so that the intermediate processors are idle at the times when they are supposed to be propagating (otherwise there will be a conflict).

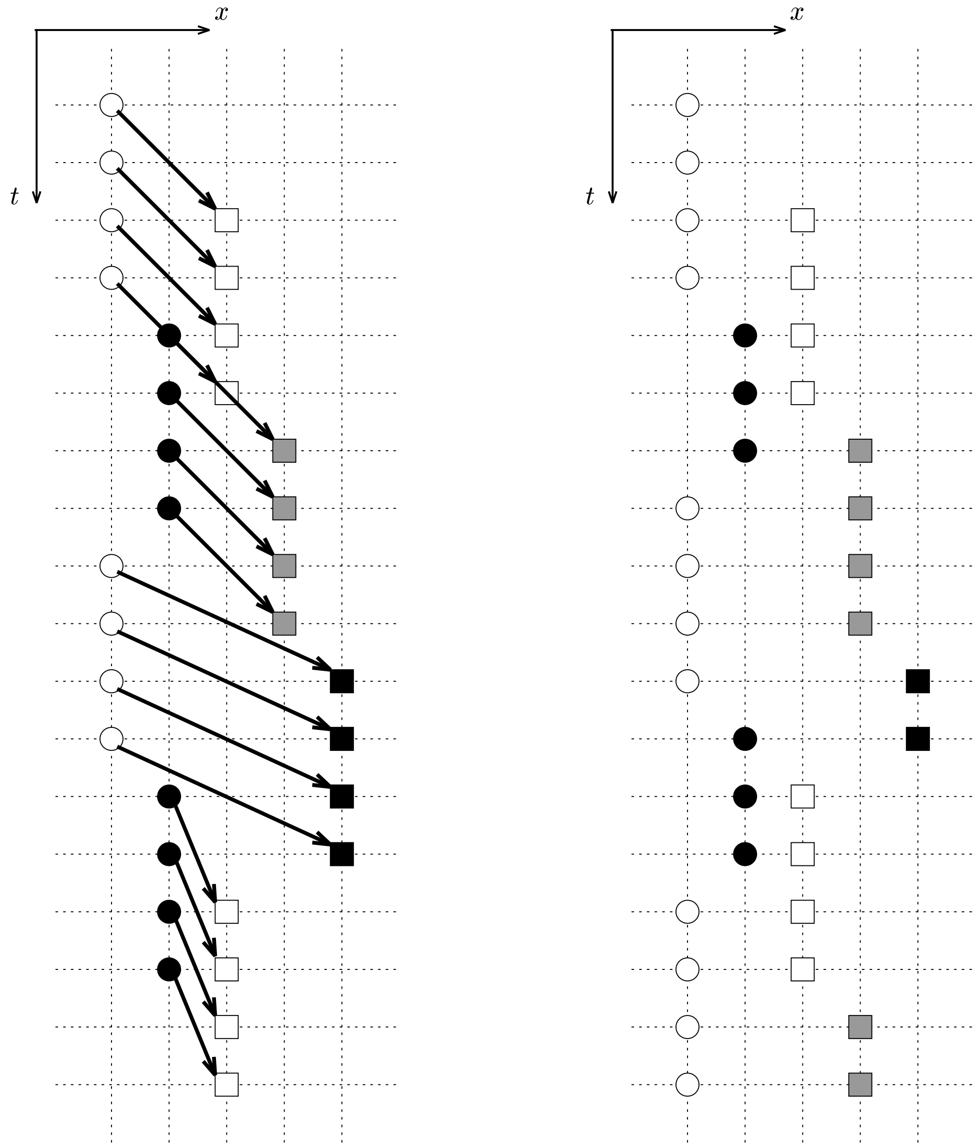
Consider, Fig. 5-a, which illustrates a particular space time mapping for our example ($w_1 = 8, w_2 = 12$ and $\alpha = 4$). Ignore the arrows and observe that each point is assigned a PE and a time instant. The allocation function is as described above; the timing function is explained intuitively as follows. Every PE in the k-th block computes cyclically, with a period of w_k. Each PE in turn, is active for exactly α cycles (the last PE in the block may be active for fewer



The picture on the left shows how two adjacent columns are mapped to PEs with $\alpha = 4$, for a particular example ($w_1 = 8$ and $w_2 = 12$). The first two PEs, (marked with open and solid circles), are responsible for the correspondingly marked nodes of the first column, and the other three (marked with squares) compute the second column. Three instances of the $[0, 1]$ dependency and their images in the processor space are also shown. They highlight the fact that values produced by PE 1 may be required by PEs 3 and 5, and that data required by PE 3 may come from different PEs (1 and 2). This is actually the simple case (the w_k's are multiples of α). The picture on the right, is an exercise. Determine all the inter-PE communications possible and draw the corresponding dependencies.

Figure 4: Illustration of the allocation function for fixed-size PEs.

cycles). Note that the start of the first PE in each block is delayed (with respect to the first PE in the preceding block) in order to allow its inputs to arrive. It is easy to verify (for the example) that the timing function does not conflict with the allocation function.



(a) The simple case (b) The exercise

Figure 5: A particular space-time mapping

Now consider the arrows. They are the images in the space-time domain, of the individual instances of the $[0, 1]$ dependency, and we shall call them *propagation paths*. We see right away that there are a number of problems. The arrows are not parallel, and this means that the propagated values will not flow at constant rates. Some paths must cross 2 PEs in 2 time steps, some must cross 4 PEs in 2 steps, and some must go to the next PE in 2 steps (does the figure show all possibilities?) This also implies that we will not have nearest-neighbor interconnections. Another problem is that there is at least one instance where two propagation paths are superimposed: the value produced by PE 1 at $t = 4$ will pass through PE 2 at $t = 5$. But at this time, PE 2 is active and will be computing its value, and will try to write a different value on its output line, and this leads to a conflict.

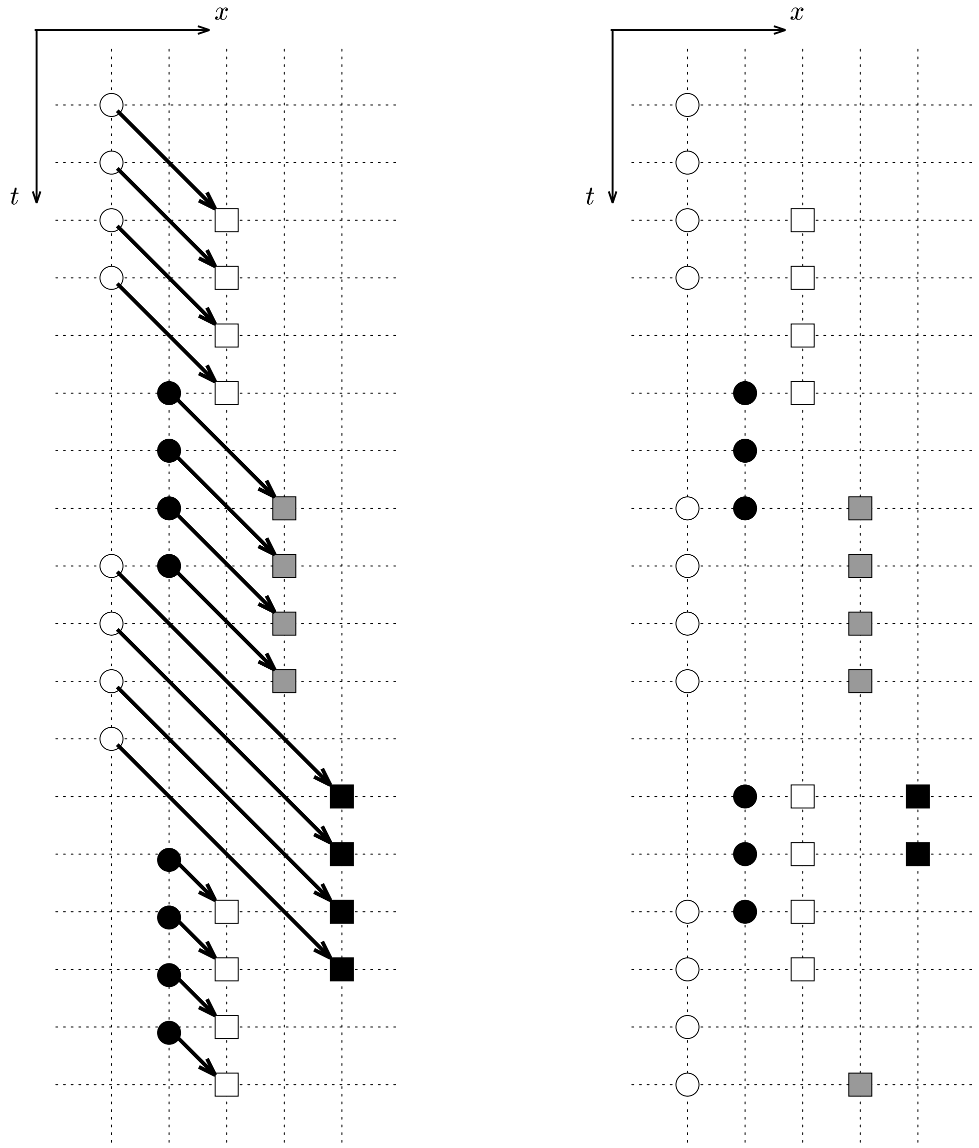
Please study Fig. 5-b and draw the propagation paths for the more complicated case when w_k is not a multiple of α. It highlights a number of interesting questions. How many distinct propagation paths are there (are you sure that all the cases are included in the figure, or is it necessary to consider more points)? How many instances of propagation conflict are there? How many distinct interconnection links are required? What is the output of PE 2 at $t = 12$?

It is this clear that the timing function must be chosen carefully. Before reading further, please try to devise a valid space-time mapping. It may be helpful to draw figures similar to the ones presented here to ensure that all the problems raised above are addressed.

Resolving the propagation conflict Let us first try to address the problem of conflict, without worrying about whether the propagation paths are parallel or not. One of the reasons for the conflict can be seen as follows: consider the last value computed by a PE in α consecutive steps (for example the value computed by PE 1 at $t = 4$). This value needs to *pass* through the next PE at the next time instant, but at this instant, this PE has just started its computation (PE 2 starts its block at $t = 5$, the same time that it is supposed to be transmitting the value computed by PE 1 at $t = 4$).

To resolve the conflict, why not delay the start of PE 2 by one cycle, to allow the propagated value to pass through? In general, we delay the start of *each* PE by one cycle (relative to its predecessor). This is illustrated in Fig. 6-a (once again for the simple case). First ignore the arrows and compare with Fig. 5-a. Note how columns 2, 4 and 5 are shifted down. Surprisingly, this simple change also makes the new propagation paths parallel to each other. As before, we leave it as an exercise to work out the details of the complicated case (this is the simplest exercise).

There still remains one final problem. Note that although the propagation paths are parallel, their *lengths* are different. So we still need a proper control mechanism that will inform each PE when to compute a new result, and when to propagate. Before we address this problem, let us formally prove our results so far. Doing so will also lead to a solution for the control problem.



(a) The simple case (b) The exercise

Figure 6: Illustration of the final space-time mapping

3.2 Proof of Correctness

We first prove that the space-time mapping is conflict free in the conventional sense. Formally, our (non-linear) allocation function is as follows.

$$a(j,k) = \lceil (j \bmod w_k + 1)/\alpha \rceil + \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil \tag{9}$$

Since $\lceil w_i/\alpha \rceil$ PEs are needed to execute the i-th column, the last term represents all the PEs in the first $k-1$ blocks (the summation ends at $k-1$); the first term gives the precise PE in the k-th block. Note that the presence of the (j mod w_k) implies that any point $[j,k]$ is mapped to the *same* PE as $[j-w_k,k]$, and the dynamic dependency is properly localized. The schedule is specified as follows.

$$t(j,k) = j + \lceil (j \bmod w_k + 1)/\alpha \rceil + \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil \tag{10}$$

Lemma 1 The space-time map given by Eqns (9-10) is free of conflict.

Proof: We show that for any two points, $[j,k]$ and $[j',k']$, in $\mathcal{D}$, if

$$\begin{bmatrix} a(j,k) \\ t(j,k) \end{bmatrix} = \begin{bmatrix} a(j',k') \\ t(j',k') \end{bmatrix}$$

then $j = j'$ and $k = k'$. The details are in the Appendix. ∎

However, we have seen that this alone is not enough (otherwise the space-time mapping of Fig. 5 would work just as well). To understand and systematically derive the propagation paths, consider the function defined below.

$$\begin{aligned} \Delta(j,k) &= a(j,k+1) - a(j,k) \\ &= \left\lceil \frac{j \bmod w_{k+1} + 1}{\alpha} \right\rceil - \left\lceil \frac{j \bmod w_k + 1}{\alpha} \right\rceil + \left\lceil \frac{w_k}{\alpha} \right\rceil \end{aligned} \tag{11}$$

Note that, by definition, $\Delta(j,k)$ is a "distance" function, since it is the spatial distance that the value $f(j,k)$, computed in PE $a(j,k)$ must travel to reach the PE $a(j,k+1)$, where it will be consumed. In other words, the dependency $[j,k] \to [j,k+1]$ is mapped to PEs that are separated in space by $\Delta(j,k)$. Observe another important property of this function, that we can verify by simple substitution.

$$\Delta(j,k) = t(j,k+1) - t(j,k)$$

This has a very fortunate practical implication: after space-time mapping, the $[j,k] \to [j,k+1]$ dependency has the *same* spatial and temporal components. Thus, a value produced by any PE x at time t needs to be sent to PE, $x + \Delta(j,k)$ at time $t + \Delta(j,k)$. This is why all the propagation vectors in Fig. 6 had a 45 degree slope. Our propagation strategy thus corresponds to sending the data by repeatedly (exactly $\Delta(j,k)$ times) passing it from one PE to the next over a link of unit delay.

In order to formally prove that this strategy works correctly we must show that when a data value is being propagated in this manner, the PEs that it passes through are neither performing some other computation, nor propagating any other data value (otherwise there will be a conflict).

Theorem 1 The tagging strategy is conflict free.

Proof: We need to show that at any instant a PE is propagating a value, it is not expected to be (a) performing some other computation; nor (b) propagating any other value. To do this, we show that that for two distinct points, $[j,k]$ and $[j',k']$, and for any $x : 0 < x < \Delta(j,k)$ and $y : 0 \leq y < \Delta(j',k')$

$$\begin{bmatrix} a(j,k)+x \\ t(j,k)+x \end{bmatrix} = \begin{bmatrix} a(j',k')+y \\ t(j',k')+y \end{bmatrix} \tag{12}$$

leads to a contradiction. The details are in the Appendix. ∎

3.3 Control

Let us now return to the problem of control. Since $\Delta(j,k)$ is the distance that $f(j,k)$ needs to travel, a naive control mechanism is as follows. For each $f(j,k)$ value that is computed, we attach a tag which denotes the distance that it needs to go. Each PE first tests to see whether the tag on the incoming data is 1. If so, the PE "knows" that the value is intended for itself, so it performs a new computation (of $f(j,k)$ and its new tag) and sends these values on its output lines. Otherwise, the PE simply transmits the input values unchanged, but decrements the tag by 1. The functions $\Delta(j,k)$ thus serves as a *dynamically computed routing tag.*

Note that since $\Delta(j,k)$ is a function of the index point and w_k and w_{k+1}, we can view Eqn. (11) as yet another equation, just like $f(j,k)$, and defined over the same domain, $\mathcal{D}$. It too, is mapped using the *same* space-time transformation, and gives us an array where each PE also has a special unit to calculate $\Delta(j,k)$, in addition to $f(j,k)$ and $u(j,k)$. The result is used to produce the tag and is a major component of the PE controller shown in Fig. 7. Each PE receives a tag.in and produces a tag.out. It has a special comparator that tests whether tag.in $= 1$, and its result determines whether tag.in is propagated (decremented by 1) or whether the output of the Δ.local unit is sent as tag.out. The same signal also controls the multiplexors of Fig. 3, and also (write enable) whether the output register is saved into memory or not.

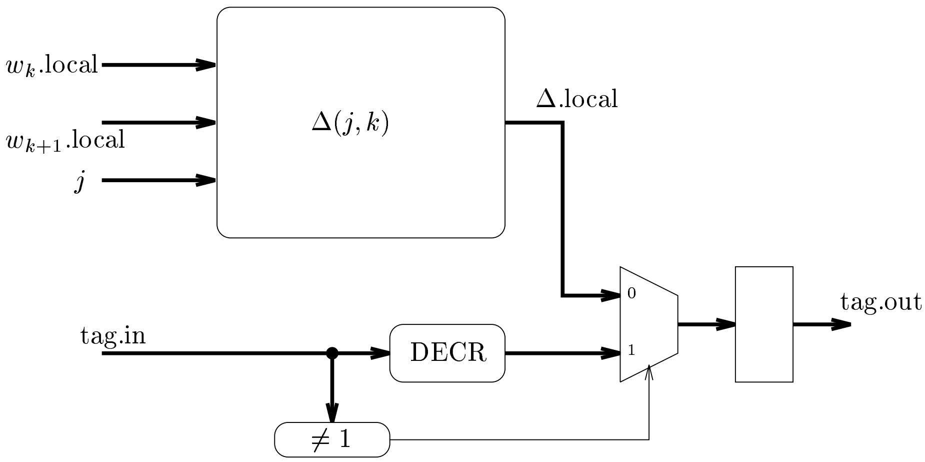


Figure 7: Control Unit of the knapsack array

4 Pragmatic Considerations

We shall now address a few implementation related problems. In particular, we discuss how standard techniques can be used to obtain a fixed size array and derive the performance of this implementation. We will also address two other problems: an efficient control mechanism and coefficient loading and setup.

4.1 Partitioning on a fixed size ring

Since $[c, m]$ is the last point to be computed, by substituting this in Eqn (10), the total computation time of the array is given by

$$T_{c,m} = t(c,m) = c + \lceil (c \bmod w_m + 1)/\alpha \rceil + \sum_{i=1}^{m-1} \lceil w_i/\alpha \rceil \tag{13}$$

The number of PEs needed to achieve this is

$$P_m = \sum_{i=1}^{m} \left\lceil \frac{w_i}{\alpha} \right\rceil \tag{14}$$

These are the two standard metrics of any systolic (indeed, any parallel) implementation, and as expected, they depend on some of the problem parameters. Note that here, they both depend on the w_k's and hence will be different from one problem instance to another. In practice, it is important to have a fixed number of PEs, since we envisage the array as a specialized co-processor board. For this, we use the well-known *locally parallel, globally sequential* (LPGS) partitioning scheme [MF86]. The approach is as follows: assume that only q PEs are available, rather than P_m. Then we "time multiplex" the array so that it solves any instance of the knapsack problem in multiple passes. Of course, the results that are produced by the last PE must be buffered and fed back into the first PE, and this necessitates a ring topology, and some intervention by the host for proper synchronization. In other words, the domain $\mathcal{D}$ is partitioned into subdomains, which are individually executed in a pipelined fashion as explained in the previous section (locally parallel). Viewing this as a single "step", the steps are executed serially on the array (globally sequential).

To determine the running time of the algorithm using this strategy we make the following observations which can be readily verified. First, note from Eqns (9) and (10), that $t(j,k) = a(j,k)+j$. Second, the very first "computation" performed by any PE x is either the calculation of $f(0,k)$ (if x is the first PE in its block), or the propagation of this value. This occurs at time $t = x$. Finally, the very last "computation" (which may either be a real computation or a propagation) performed by any PE occurs c steps later. We say that the PE is *busy* from $t = x$ to $t = x + c$.

When the array is partitioned using the LPGS strategy on q PEs, these observations are still valid for each pass: if the r-th pass starts at t_r, PE x is busy for this pass from $t_r + x$ to $t_r + x + c$. The first result of this pass is available from the last PE at $t_r + q$. Recall that c grows exponentially as compared to m (the knapsack problem is NP-hard), and it is thus reasonable to assume that $c > q$. Thus, the start of any two successive passes differs by exactly c clock cycles. Note that this requires buffering the results produced by the last PE (for exactly $c - q$ cycles), and we assume that the host (or possibly a specialized I/O co-processor, or a smart DMA controller) is responsible for this. Since the entire algorithm needs $\lceil P_m/q \rceil$ passes, the last pass can only start at $t_l = c(\lceil P_m/q \rceil - 1)$, its first result is produced at $t_l + q$, and the final result of the array is produced at $t_l + q + c$. Hence, the running time of the fixed size ring is given by:

$$\begin{aligned} T^q_{c,m} &= c\left(\left\lceil \frac{P_m}{q} \right\rceil - 1\right) + c + q \\ &= c\left\lceil \frac{P_m}{q} \right\rceil + q = c\left\lceil \left(\sum_{i=1}^{m} \lceil w_i/\alpha \rceil\right)/q \right\rceil + q \end{aligned} \tag{15}$$

Note that, in addition to c, m, q and α, the performance of this implementation depends on w_k's, and will thus be different for different instances of the problem (even if they may have the same "size" parameters, m and c). In spite of this, it is important to note that the array will correctly implement *any* instance of the unbounded knapsack problem, even when the different PEs that constitute a block (for one column of $\mathcal{D}$) are partitioned across the ends of the ring, or if any individual w_k is so large as to require multiple passes itself, etc. Indeed, there is no need to even prove this formally, since the original array with P_m PEs has been proven correct by construction, and LPGS partitioning is a proven technique, applicable to *any* systolic array.

4.2 Efficient Control

We have seen that our PE has a "black box" within its control unit (see Fig. 7) which is responsible for computing $\Delta(t, x)$ at all times. If this were implemented naively as specified by Eqn. (11) it would require two divisions by α, two incrementers, one modulo w_k and one modulo w_{k+1} (even if we assume that $\lceil w_k/\alpha \rceil$ can be computed off line and latched into a local register when each new problem instance is loaded). This is clearly unrealistic: this heavy machinery is to be used to control a very simple data path, essentially an adder-comparator. Note that our development here (and also in the following subsection) will be more informal, since it involves circuit level optimizations. At present, we are not aware of a formal transformation scheme to obtain the final hardware systematically.

In order to obtain a more practical circuit, let us recall the main purpose of the controller, namely to indicate to the PE when to propagate and when to compute a new result. The formal development and the distance function was necessary in order to prove that the propagation strategy would not have any collisions. For efficient implementation, we would like to achieve the same objective without calculating $\Delta(t, x)$. This can be done if we can study the pattern of activity of the PEs and use it to deduce this information statically. Once again, please return to Fig. 6 and try to determine these patterns before reading any further.

First of all, the activity of each PE in the k-th block is periodic with a period of exactly w_k. Let us first consider the simple case when w_k is a multiple of α. Then each PE is active for exactly α consecutive steps in each period of w_k. As a result, the control can be easily implemented with a stoppable counter that counts modulo α. When it cycles over, the counter stops until it is restarted by another signal. In other words, one can think of a "token" (indicating that the PE is active) being passed among the PEs. Each PE keeps the token for exactly α cycles, and then passes it on to its neighbor. The first PE in a block is special. It ignores any tokens coming from its left neighbors, but simply generates a new token every w_k cycles, using a modulo w_k counter.

A subtle point to note: recall that in our schedule, we need the final result produced by any PE to be propagated through its neighbor *before* the neighbor itself can start. This is easily achieved in our token passing control mechanism: after being active for α cycles, each PE simply delays the passing of the token by one cycle! Because of this, it is possible that more than one PE is simultaneously active (the word token is possibly a misnomer in our explanation, since it does not imply mutual exclusion). Also note that since any PE could be required to be the first PE in its block for some problem instance, each PE must have a counter that counts modulo w_k. In fact, this counter modulo w_k is essential for a couple of other functions, as we shall see.

Now consider that case when w_k is not a multiple of α. The above arguments are still valid for all PEs except the last one in the block, which is active for fewer than α cycles. However, since we know that the PE activity is periodic with a period of w_k, we simply use a counter modulo w_k to "turn off" the last PE in the block. Hence the operating rule of the PEs now becomes as follows: start computing when you are given a token (or generate one, if you are the first PE in the block); stop computing when *either* the counter modulo α or the counter modulo w_k cycles over. Indeed, the PE doesn't even have to "know" whether it is the last one in its block. For this scheme to work, the modulo w_k counters must cycle in a skewed fashion (i.e., the one in PE, x must be reset at $t = x$). Finally, recall from the second line of Eqn. (8), that when $t - x < w_k$, i.e., for the w_k cycles after a PE becomes active for the very first time, each PE must simply propagate its input, and this is also easily controlled by our modulo-w_k counter.

In the above discussion, we have talked about the fact that a PE must know whether it is the first PE in a block or not. This is perfectly reasonable, since for any given problem instance, this information can be determined off line, and loaded with the problem coefficients. Thus, we conclude that the control of the PE can be reduced to two counters: one fixed (that counts modulo-α) and the other programmable (counting modulo-w_k) and a few latches.

4.3 Coefficient loading

For correct operation, each PE must know w_k, p_k, k.local and FIRST_k (a bit that indicates whether it is the first PE in its block). We now discuss how these values can be efficiently loaded into the array. Since the host knows the w_i's for the particular problem instance, there is a simple preprocessing step to determine these values (how many successive PEs have the same coefficients, etc.) and the appropriate sequence of coefficients can be generated off-line.

A standard systolic loading method is to simply pipeline these values into the array from the left (the coefficients of the last PE are entered first), and to issue a broadcast at the q-th step (there are q PEs in the array). However, for good throughput one would like to do this while another problem (or a previous pass) is being solved in the array. So, until the broadcast is issued (i.e., until it is time to commence the next problem/pass), the PE must save the new values, but continue to use the older values. This necessitates a set of "shadow registers" and a

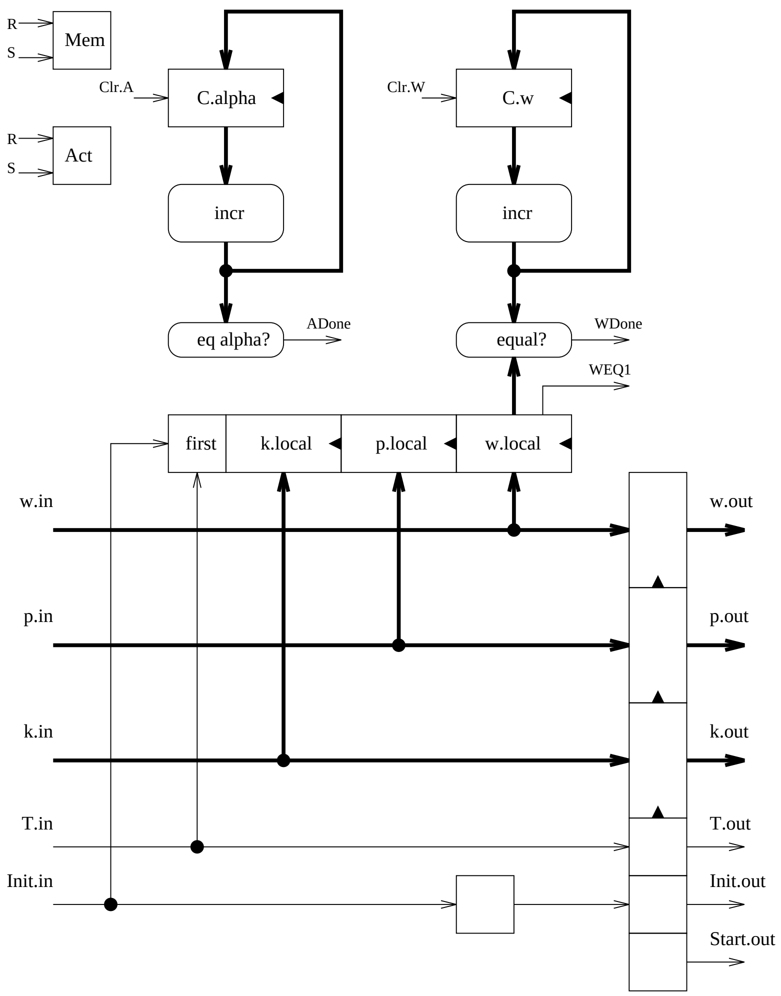


Figure 8: Controller for the final architecture

control signal to indicate the start of a new pass.

Consider another alternative. We load the coefficients in the reverse order (the coefficients of the first PE are the first in the sequence). We have a control signal that is propagated with twice the delay as the coefficients (one extra latch in the control path), which is aligned with the first data value. Hence, this control bit meets the i-th data value at the i-th PE, and can be interpreted as a "LOAD" command. The only problem is that this happens at $t = 2i$. The solution is simple. We run this loading circuitry at twice the clock rate as the main circuitry. This is feasible in terms of speed, because the adder and the comparator in the main data path determine the critical delay. Furthermore, there is no need for shadow registers, and the LOAD signal can also be used to start the computation of the next pass. Fig 8 shows the control for the final design.

5 A VLSI Model and Optimal Memory Size

In our development so far, we have not specified the value of the PE memory, α, other than saying that it is constant, independent of the problem parameters. Of the different parameters that govern the running time of the array, we note that q and α are design parameters, while the others depend on the problem instance. We now address the problem of choosing these two optimally. In particular, we seek to determine "good" values of α and q, i.e., those that minimize the running time of the array. However, the two parameters are related. Suppose that we have only some fixed real estate on silicon. This may be used either to implement a large number of PEs with small memory, or just a few PEs with larger memory. This leads to a classic tradeoff problem. Fortunately for us, our performance metric is easily quantifiable, and we are able to set up a (non-linear) discrete optimization problem, and solve it analytically.

Note that the dominant term in Eqn. (15) is $c\lceil \frac{P_m}{q} \rceil$, and since $\frac{cP_m}{q} \leq c\lceil \frac{P_m}{q} \rceil \leq \frac{cP_m}{q} + c$, we can reasonably approximate it by

$$F(q, \alpha) = \frac{cP_m}{q} = \frac{c}{q} \sum_{i=1}^{m} \lceil w_i/\alpha \rceil \tag{16}$$

It is bounded from below by $\frac{mc}{q}$, the theoretical optimum for executing the knapsack problem on q processors. Even with this assumption, we see that there are too many parameters (the w_k's), and they could vary arbitrarily, even for some fixed c and m. We therefore consider a range of problem instances, whose w_k's are uniformly distributed in an interval, $[w_{\min}, w_{\max}]$, and we seek to minimize the *expected value* of $F(q, \alpha)$ for all problem instances in this range. It is thus, the average case analysis of the performance of our array.

Lemma 2 If the coefficients w_i are uniformly distributed in the interval $[w_{\min}, w_{\max}]$, the expected value of $F(q, \alpha)$ is given by

$$\mathbf{E}[F(q, \alpha)] = \frac{cm}{2q} \left(\frac{w_{\max} + w_{\min} - 1}{\alpha} + 1 \right) \tag{17}$$

Proof: See Appendix ∎

The above result allows us to simplify the cost function for our optimization problem, which can now be formally stated as follows.

For a given resource, R (silicon area), which may be used either as memory or as PEs, what values of α and q minimize the objective function, $\mathbf{E}[F(q, \alpha)]$?

As mentioned above, q and α are related, since the number of PEs that we can implement with a given real estate depends on how much memory they have. Let a_1 be the area of the CPU, control and I/O (i.e., the *fixed* cost of a PE), and let a_2 be the area of the memory per word. The cost of our PE is thus $a_1 + a_2\alpha$. Then the resource restriction is expressed as the non-linear constraint $q(a_1 + a_2\alpha) \leq R$ which is equivalent to

$$q\alpha \leq \overline{R} - q\overline{a}, \quad \text{where } \overline{R} = R/a_2, \quad \text{and } \overline{a} = a_1/a_2$$

If we denote $\overline{w} = w_{\max} + w_{\min} - 1$ our problem is reduced to finding the values of α and q which minimize $\mathbf{E}[F(q,\alpha)]$ and satisfy the resource constraint, i.e.

Pb. 1: minimize $\frac{\overline{w}}{q\alpha} + \frac{1}{q}$

$$\text{subject to} \quad \begin{array}{rll} q\alpha & \leq \overline{R} - q\overline{a} & \\ 1 & \leq \alpha \leq w_{\max}, & \text{integer} \\ 1 & \leq q, & \text{integer} \end{array}$$

In general, solving such a non-linear integer optimization problem is NP-hard. However, as we will see, this particular problem has some nice properties which allows us to show that its solution is close to the solution of the relaxed problem (i.e., the problem when q and α do not have to be integers, also called the continuous version). This latter solution is easily obtained analytically, and hence all we need to do is to look at the nearest integral points to it.

It is clear that the solution of the continuous version of Pb. 1 always satisfies the equality $q\alpha = \overline{R} - q\overline{a}$. Hence, consider the following problem.

Pb. 2: minimize $\frac{\overline{w}}{\overline{R} - q\overline{a}} + \frac{1}{q}$

$$\text{subject to} \quad \begin{array}{rll} q\alpha & \leq \overline{R} - q\overline{a} & \\ 1 & \leq \alpha \leq w_{\max}, & \text{integer} \\ 1 & \leq q, & \text{integer} \end{array} \tag{18}$$

Theorem 2 The solution $(q_{\text{int}}, \alpha_{\text{int}})$ of problem PB. 2 is either $\left(\lfloor q^* \rfloor, \left\lfloor \left(\frac{R}{\lfloor q^* \rfloor} - a_1\right)/a_2 \right\rfloor\right)$, or $\left(\lceil q^* \rceil, \left\lfloor \left(\frac{R}{\lceil q^* \rceil} - a_1\right)/a_2 \right\rfloor\right)$, where (q^*, α^*) is a solution of its relaxation. Moreover, (q^*, α^*) is given by

$$\alpha^* = \sqrt{\frac{a_1(w_{\max} + w_{\min} - 1)}{a_2}} \tag{19}$$

and

$$q^* = \frac{R}{\sqrt{a_1 a_2 (w_{\max} + w_{\min} - 1)} + a_1} \tag{20}$$

provided that $\frac{a_1}{a_2} \leq \frac{w_{\max}^2}{w_{\max} + w_{\min} - 1}$. Otherwise,

$$\alpha^* = w_{\max} \quad \text{and} \quad q^* = \frac{R}{a_2 w_{\max} + a_1}.$$

Proof: See Appendix ∎

Corollary 1 The optimal average time is approximately $\mathbf{E}[F(q^*, \alpha^*)]$ given by

$$\mathbf{E}[F(q^*, \alpha^*)] = \begin{cases} \dfrac{cm}{2R} \left[\sqrt{a_1} + \sqrt{a_2(w_{\max} + w_{\min} - 1)}\right]^2 & \text{if } \frac{a_1}{a_2} \leq \frac{w_{\max}^2}{w_{\max}+w_{\min}-1} \\ \dfrac{cm(2w_{\max} + w_{\min} - 1)}{2R} \left(a_2 + \dfrac{a_1}{w_{\max}}\right) & \text{otherwise} \end{cases} \tag{21}$$

Proof: By substituting α^* and q^* in Eqn (17) and simplifying. ∎

We can see from the above result that α^* does not depend on R, and this has an important practical implication. The design remains optimal, even as more real estate is added. In other words, once an optimal PE is designed, it may be mass produced, and the possibility of adding additional PEs does not sacrifice optimality. Also remember that what we have minimized is the *expected value* of the running time for a range of problems (and this depends on $w_{\min}$ and $w_{\max}$). Nevertheless, the array is capable of solving arbitrary instances of the knapsack problem, and if we occasionally feed it problems whose weights are outside these bounds, all that will happen is a slightly degraded performance for that particular problem instance. Also note that the fixed cost of the PE figures prominently in Eqn. (21) and hence, all the pragmatic improvements that we have developed in Sec. 4 do not simply reduce the area of the PE. They have the added benefit of also reducing the expected running time of the array!

5.1 Performance estimation

We now apply these results to our VLSI design to estimate the improvement that we can obtain. We will compare the expected running time of our optimized architecture to an array that does not choose the memory size optimally. To be fair, we allow the nonoptimized array to use all the hardware improvements presented so far, namely the improved control mechanism and coefficient loading method. Thus we measure only the improvement due to our optimization result.

We assume 1μ CMOS technology, and that 2×2^{10} static registers (16-bit) can be laid out on a 1cm^2 VLSI chip (the actual value is not important since it cancels out later). We also assume that $w_{\max}$ is 1000 (a typical value used in standard texts [MT90]) and $w_{\min} = 1$.

In our design, the main data path for the computation of f is fairly standard, and takes up an area comparable to 13 registers. The control part takes up about 12 registers. Hence, the entire PE can be laid out in 25 registers. We choose to use a DRAM because we know that every memory location (of interest) will be accessed periodically, with a period of exactly w_k, and hence refresh circuitry is not needed (in fact, the minimum speed required to avoid the refresh is only 0.25 MHz). For the 1μ CMOS technology, we assume that each memory cell takes up about half the area of a static register. Hence $a_2 = \frac{1}{2}$, and $a_1 = 25$.

Using this data, we first see that $\frac{a_1}{a_2} \leq \frac{w_{\max}^2}{w_{\max}+w_{\min}-1}$, so the optimal solution is given by Eqn (19) and Eqn (20), and we have $q^* = 14.97$. The nearest integer values which minimize the running time are with 15 PEs, each with 223 words of memory. Recall that this is a solution to an *approximation* of the original optimization problem. To see how far we were off, we made an exhaustive search (indeed, for a practical design, this may well be a valid design strategy, if the best performance is to be squeezed out), and discovered that the true optimum was at $q = 16$, for which $\alpha = 206$. Substituting these into Eqn (17) the expected running time of the optimal array is $0.18275mc$.

On the other hand, the unoptimized array requires 1000 words of memory which take up an area comparable to 500 static registers. This array does not need to use any tagging scheme, so its CPU + control + IO is only 22 registers, enabling 4 PEs to be squeezed on a chip. However, the naive design *always* allocates one PE per column, and the dominant term in the running time is $\dfrac{mc}{q} = 0.25mc$ (obtained from Eqn (15) by letting $\alpha = w_{\max}$); we then have we have the following.

Result 1 The memory optimization reduces expected running time by 28% without any area penalty.

Observe that the choice of $w_{\max} = 1000$ is an arbitrary one, favorable to the naive design. Indeed if $w_{\max} > 4000$, the naive design cannot even fit on a single chip. On the other hand, our modular design can easily adapt to larger values of weights, at the expense of increased running time. This illustrates the importance of designing modular and extensible (i.e., purely systolic) architectures.

6 Non-Linear Mappings to Systolic Arrays

The technique that we used to derive the knapsack array can be easily generalized. In this section, we present a method for systematically deriving systolic arrays from arbitrary recurrences (with or without dynamic dependencies) using non-linear space-time transformations. Recall that the success of the traditional systolic synthesis methods relies very intricately on the coherent interaction of *uniform* dependencies, and *linear* transformations. It is because of this that we can simply interpret the image of the original dependency vectors as the description of the interconnections of the systolic array. Unfortunately, this is no longer true when we use non-linear space-time transformations, as we have already seen.

Consider the general recurrence equation (1), defined over a domain $D \subset Z^n$ with an arbitrary dependency, $d(z)$, which may even be dynamic (repeated below for convenience).

$$\forall z \in D \ : \quad X[z] = g(\ldots Y[d(z)] \ldots)$$

and suppose that we apply a non-linear space-time transformation to all the variables: $T : Z^n \rightarrow Z^n$ (i.e., $T(z) = \begin{bmatrix} a(z) \\ t(z) \end{bmatrix}$, where $t : Z^n \rightarrow N$ is a valid schedule, and $a : Z^n \rightarrow Z^{n-1}$ is an allocation function. The following theorem gives sufficient conditions for the transformed recurrence to be implemented as a systolic array.

Theorem 3 Eqn. (1) can be implemented on a systolic array if there exists a scalar function $H : D \rightarrow N$, and a constant vector, $\rho \in Z^n$ such that:

1. $\forall z \in D, \quad T(z) = T(d(z)) + H(z)\rho$
2. $\forall z, z' \in D, z \neq z'$, and for any two integers, a, a' such that $0 \leq a < H(z)$ and $0 \leq a' < H(z')$,

   $$T(z) + a\rho \neq T(z') + a'\rho$$

Proof: Recall that the space-time transformation, T maps the point z to $T(z)$, and also maps $d(z)$ to $T(d(z))$. Now, if the function $H(z)$ satisfying condition (1) above exists, then it is clear that the vector from $T(z)$ to $T(d(z))$ (i.e., $T(z) - T(d(z))$) can be decomposed into a sequence of displacements by a constant vector, ρ. The number of such displacements is exactly $H(z)$, which is (for all index points, $z \in D$, a strictly positive integer.

Consider condition (2) for the case when $a = a' = 0$. This simply states that the space-time transformation is conflict free: no two distinct points are mapped to the same processor at the same time.

Now consider condition (2) when $a' = 0$, and $a > 0$. The point, $T(z) + a\rho$ represents a space-time index point through which the value $X(z)$, which is computed at $T(z)$, will pass in its a-th step if propagated along ρ. The condition states that this space-time point is not the image of any other point $z' \in D$ in the domain of X.

Finally, by a similar reasoning, when $a, a' > 0$, the condition ensures that if all the values produced in the space-time domain are propagated along the direction ρ, no distinct values will be propagated through the same space-time point. ∎

We call (1) the *feasibility condition* for systolic propagation and (2) the *controllability condition*. When both conditions are satisfied we say that the transformation $T(z)$ is a *realizable* space-time mapping. The function, $H(z)$ is called the *control function*.

The systematic generation of a systolic array following this propagation strategy is also straightforward. As we have done for the knapsack problem, we introduce, in our system of recurrences, a new equation that specifies the computation of $H(z)$ for all $z \in D$. It is transformed using the same space-time mapping, $T(z)$, and its value is used as a dynamic routing tag. The data values are propagated using the space-time vector ρ. Whenever a processor sees that its input tag is 1, it computes a new value (for the data as well as the tag) and propagates it, otherwise it simply propagates the data value unchanged with the tag decremented by 1.

Note that the nature of the control function determines the complexity of the final implementation, which may well be easily amenable for improvement. A systematic method for doing this is at present open. It requires extending the control signal generation methods for systolic arrays [Raj89] to the nonlinear cases. The main idea is as follows. Observe that we need to compute a function $H(z)$ at *all* points in an index domain which may not necessarily be specified as a recurrence relation. It may be easier to implement this function if we are able to decompose it into a recurrence. This is similar to strength reduction techniques used in parallelizing compilers. Furthermore, note that a simple variation is also possible. We may get a systolic array even though the conditions of Theorem 3 are not strictly satisfied. If the cardinality of the number of conflicts is bounded by a constant, we can derive an array with finitely many propagation channels.

7 Related Work and Conclusions

Since the knapsack problem is an important combinatorial optimization problem, a number of researchers have developed parallel algorithms for it. We note that a software version of our algorithm has very good performance, as reported elsewhere [ARQ93] (see references therein for a detailed survey of software implementations).

A parallel dynamic programming implementation for the 0/1 knapsack problem, was presented by Lin and Storer [LS91]. It improves on Lee et al. [LSS88], and its running time is

$\Theta(mc/q)$ on an EREW PRAM of q processors*. This algorithm has optimal speedup and processor efficiency. Kumar et al. [KGGK94, Sec 9.1.2] also give a similar implementation (all these implementations can be viewed as equivalent, in the sense that *rows* of $\mathcal{D}$ are allocated to different processors). Lin and Storer report that on a Connection Machine, the time complexity increases to $\Theta(\frac{mc\log q}{q})$ because of communication costs, and Kumar et al. note that the implementation is cost optimal for $c = \Omega(q \log q)$.

Chen et al. [CCJ90, CJ92] present a linear array of q general purpose processors (each with a sufficiently large RAM). It has a time complexity $\Theta(mc/q)$, which is asymptotically optimal, and is similar to the naive array of Sec. 2.3 (the difference is that because they address the 0/1 problem, they need to retain the entire table of $f(j,k)$ values, and hence their processors require c words of memory).

As noted by Teng [Ten90], the performance of parallel algorithms depends on c, m and also $w_{\max}$ (unlike the sequential case when only m and c are relevant). The number of processors required by many parallel algorithms is exponential in the size of the input and these algorithms have a very low processor efficiency. For example the best time complexity algorithm for Eqn. (4) proposed by Teng [Ten90] requires $M(c)$ processors to solve the problem in $O(\log^2(mc))$ time. The function $M(n)$ above denotes the number of processors needed for multiplying two n by n matrices in $O(\log n)$ parallel time (which is known to be between n^2 and n^3). Hence, at least $O(c^2)$ processors are needed, and $\frac{1}{c}$ is a factor in the efficiency, which approaches zero as c increases.

Note that the systolic model is weaker than other models such as the PRAM in two respects: fixed size PE and restricted connectivity. If we relax the first constraint (i.e., let α be as large as needed), the running time becomes $\Theta(mc/q)$, which is optimal: the sequential running time is reduced by a factor of $\Theta(q)$, which is the best one can expect from any parallel implementation, as illustrated in Sec. 2.3, and by the results of Chen et al. [CCJ90]. This model, although not systolic, is still weaker than a PRAM. We conjecture that the added slowdown factor of our systolic array, $\Theta(w_{\max})$, is a lower bound because of the constraint that each PE has only finite memory. This would be an interesting open problem to resolve.

Independent of the knapsack problem, many researchers have proposed systolic arrays for dynamic programming. This includes arrays for specific instances, such as optimal string parenthesization [GKT79], path planning [BK88], etc., some general arrays [RV84, Myo91] as well as arrays that can solve any instance of dynamic programming, [LW85, LL86]. Usually, the arrays for specific instances have better performance than the general purpose arrays because

*We use the word processors when we discuss software implementations, and PEs when referring to hardware implementations.

they can exploit properties of the particular problem at hand. This is the case for the knapsack problem too—for example, if we use Li and Wah's array [LW85] for our problem, we will need c processors and take mc time steps, which is clearly unacceptable (since we can achieve the same time on a uniprocessor). Even the delta transformation proposed by Lipton and Lopresti [LL86] yields only logarithmic improvement in time and the number of PEs.

Andonov has investigated the problem of implementing the knapsack recurrence on systolic arrays for a number of years. The first solution was proposed by Andonov and Benaini [AB90]. The array is extensible with respect to m, c and $w_{\max}$, but has a running time of $\Theta(mc^2/q\alpha)$ on a ring of q PEs, which is unacceptable (the sequential implementation takes mc steps). The running time was brought down to $\Theta(mcw_{\max}/q)$ by Andonov and Quinton [AQ92]. It was designed in an ad hoc manner without a formal proof of correctness, and has an expensive control strategy.

In this paper, we have presented a number of results: first, we formally derived the Andonov-Quinton array for the knapsack problem. The derivation illustrated the steps to follow in deriving systolic arrays in the presence of dynamic dependencies. We used this derivation as a case study and presented sufficient conditions for the general case, thus advancing the state of the art of systolic synthesis methods. Obviously, many open problems remain. We have only given sufficient conditions. Our results are not constructive, and it needs a lot of designer ingenuity to come up with *realizable* space-time transformations. Automating this is the subject of our ongoing investigation.

Another result of this paper was that we formulated the problem of choosing the memory size as a non-linear optimization problem, which we solved analytically. We reiterate that the design will have optimal *expected running time* for a range of problem instances whose weights are distributed randomly in the interval, $[w_{\min}, w_{\max}]$, but will nevertheless correctly solve *any* instance of the problem. It is also important to note that the optimal memory size does not change with the available resource constraint (silicon real estate), and hence the PE design *remains* optimal when more PEs are cascaded to make a bigger array. The technique of formulating such optimization problems and solving them analytically can be used for many related problems: determining the optimal size for iteration space tiling, a code optimization used by parallelizing compilers, selecting the digit size when deriving digit-based arithmetic circuits from bit-serial dependency graphs, and other related problems.

Finally, we have developed circuit level optimizations to make the final implementation practical for VLSI. These pragmatic improvements allowed us to reduce substantially the control to a couple of counters, and also had the additional benefit of reducing the *time complexity* of the array. Our development of these practical improvements was not formal, and it would be interesting to derive this formally by first localizing the function $\Delta(j,k)$, and then determining an appropriate space-time mapping. We note here that there is still a (minor) problem with the array presented here. The presence of the modulo $w_{\max}$ counter means that the array is not purely systolic: one cannot arbitrarily increase all the problem parameters (specifically, $w_{\max}$) and not have to change the PE design. In practice this is not a serious limitation, since it is just the size of a single register, which can be made large enough (just as one makes arithmetic units large enough to ignore overflow, etc.) Nevertheless, we have recently improved on this to develop a purely systolic array, using the idea of of programmable shift registers [AQRW95]. To boot, this array has almost no control, and thus we obtain further improvement in the expected running time.

We note that there is still scope for improvement. Different memory organizations may be tried, different clocking schemes may be explored, etc. Finally, we could come back full circle, and try to improve the algorithm itself. Indeed, we have recently done precisely that in [AR94], where we use a sparse algorithm, which has better average case running time than the classical dynamic programming, and give an asynchronous VLSI array processor for it. This yields an algorithm that has better average case behavior. However, there is much that needs to be done if that solution is to be implemented in VLSI. It may turn out that the pragmatic complexities may reduce the effectiveness of the improvements. Furthermore, the synthesis methodology used there is completely different and requires much more insight from the designer. A detailed discussion is beyond the scope of this paper.

In conclusion, we have presented a complete derivation of an optimal VLSI design for the unbounded knapsack problem. Since it belongs to the class of NP-*hard* problems its fast parallel implementation is interesting in its own right. Because the algorithm for the dynamic programming solution of this problem has run-time dependencies, it is also interesting from the view of design methodology. We have followed the design trajectory all the way from the level of algorithm through architecture and finally to circuit. At each time we have used appropriate theoretical tools for analysis of the performance. Our results show how it is important to combine many diverse areas—optimal parallel algorithms, VLSI design, discrete optimization and dependence analysis to achieve high performance application specific array processors. In essence, we are really designing "algo-tech-cuits", which span the continuum from algorithms through architectures and all the way to circuits.

Acknowledgments The authors would like to thank J-J. Bellanger, A. Benaini (especially for early insights), P. Quinton and D. Wilde for helpful discussions. We also thank V. Lo and M. Quinn for comments on preliminary drafts of the manuscripts, and the referees for their detailed comments.

References

[AB90] R. Andonov and A. Benaini. A linear systolic array for 0/1 knapsack problem. Technical Report 90-12, LIP-IMAG, Ecole Normale Superieure de Lyon, 1990.

[AQ92] R. Andonov and P. Quinton. Efficient linear systolic array for the knapsack problem. In *CONPAR'92, Lecture Notes in Computer Science 634,* (Lyon, France), pages 247–258, September 1992.

[AQRW95] R. Andonov, P. Quinton, S. Rajopadhye, and D. Wilde. A shift-register based systolic array for the general knapsack problem. *Parallel Processing Letters*, 5(2):251–262, February 1995.

[AR93] R. Andonov and S. V. Rajopadhye. An optimal algo-tech-cuit for the knapsack problem. In *International Conference on Application-Specific Array Processors (ASAP-93)*, pages 548–559, Venice, October 1993. IEEE.

[AR94] R. Andonov and S. V. Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *International Conference on Application-Specific Array Processors (ASAP-94)*, pages 302–313, San Francisco, August 1994. IEEE.

[ARQ93] R. Andonov, F. Raimbault, and P. Quinton. Dynamic Programming Parallel Implementation for the Knapsack Problem. Internal Report 740, IRISA, June 1993.

[AS93] V. Aleksandrov and Fidanova S. On the expected execution time for a class of non-uniform recurrence equations mapped onto 1D regular arrays. *J. of Parallel Algorithms and Applications*, 2(1), 1993.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[BK88] F. Bitz and H. T. Kung. Path planning on the warp computer: using a linear systolic array in dynamic programming. *Int.J. Computer Math.*, 25:173–188, 1988.

[CCJ90] G.H. Chen, M.S. Chern, and J.H. Jang. Pipeline architectures for dynamic programming algorithms. *Parallel Computing*, 13:111–117, 1990.

[CHR88] C-S. Chung, M. S. Hung, and W. O. Rom. A hard knapsack problem. *Naval Research Logistics*, 35:85–98, 1988.

[CJ92] G.H. Chen and J.H. Jang. An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing*, 18:811–821, 1992.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[GKT79] L. Guibas, H. T. Kung, and Clark D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Conference on Very Large Scale Integration: Architecture, Design and Fabrication*, pages 509–525, January 1979.

[GN72] R. Garfinkel and G. Nemhauser. *Integer Programming.* John Wiley and Sons, 1972.

[Hu69] T. C. Hu. *Integer Programming and Network Flows.* Addison-Wesley, 1969.

[JF88] J.M. Jansen and F.M.Sijstermans. Parallel branch-and-bound algorithms. *Future Generation Computer Systems*, 4:271–279, 1988.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel computing.* Benjamin Cummins, 1994.

[KMW67] R. M. Karp, R. E. Miller, and S. V. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[LL86] R. J. Lipton and D. Lopresti. Delta transformation to symplify VLSI processor arrays for serial dynamic programming. In *Proc. International Conference on Parallel Processing*, pages 917–920, 1986.

[LS84] Teng-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *CACM*, 27(6), 1984.

[LS91] J. Lin and J. A. Storer. Processor-efficient hypercube algorithm for the knapsack problem. *J. of Parallel and Distributed Computing*, 13:332–337, 1991.

[LSS88] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problems. *J. of Parallel and Distributed Computing*, 5:438–456, 1988.

[LW85] G. Li and B. W. Wah. Systolic processing for dynamic programming problems. In *Proc. International Conference on Parallel Processing*, pages 434–441, 1985.

[Meg93] G. M. Megson. Mapping a class of run-time dependencies onto regular arrays. In *International Parallel Processing Symposium*, pages 97–107, Newport Beach, CA, April 1993. IEEE.

[MF86] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transaction on Computers*, C-35(1):1–12, January 1986.

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[Myo91] J.F. Myoupo. A fully pipelined solutions constructor for dynamic programming problems. In *Advances in Computing – ICCI'91, Proc. Inter. Conf. Comput. Inform.* Springer-Verlag, 1991. Lecture Notes in Computer Science 497.

[QR89] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques.* Masson, 1989. English translation by Prentice Hall, *Systolic Algorithms and Architectures*, Sept. 1991.

[Raj89] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, May 1989.

[RF90] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, June 1990.

[RV84] I. V. Ramakrishnan and P. J. Varman. Dynamic programming and transitive closure on linear pipelines. In *International Conference on Parallel Processing*, St. Charles, Il, August 1984.

[Ten90] S. Teng. Adaptive parallel algorithm for integral knapsack problems. *J. of Parallel and Distributed Computing*, 8:400–406, 1990.

Appendix

Proof of Lemma 1: First, note (from 9-10) that $t(j,k) = j + a(j,k)$. Let $a(j,k) = a(j',k')$ and $t(j,k) = t(j',k')$. Thus $j + a(j,k) = j' + a(j',k')$, and hence, $j = j'$. Since $a(j,k) = a(j,k')$ (we assume, w.l.o.g., that $k > k'$),

$$\lceil (j \bmod w_k + 1)/\alpha \rceil + \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil = \lceil (j \bmod w_{k'} + 1)/\alpha \rceil + \sum_{i=1}^{k'-1} \lceil w_i/\alpha \rceil$$

or

$$\sum_{i=k'}^{k-1} \lceil w_i/\alpha \rceil = \lceil (j \bmod w_{k'} + 1)/\alpha \rceil - \lceil (j \bmod w_k + 1)/\alpha \rceil$$

so

$$\lceil w_{k'}/\alpha \rceil + \sum_{i=k'+1}^{k-1} \lceil w_i/\alpha \rceil = \lceil (j \bmod w_{k'} + 1)/\alpha \rceil - \lceil (j \bmod w_k + 1)/\alpha \rceil$$

Since the first term on the RHS is no larger than the first term on the LHS,

$$\sum_{i=k'+1}^{k-1} \lceil w_i/\alpha \rceil \leq -\lceil (j \bmod w_k + 1)/\alpha \rceil$$

But each term in the summation on the left is a ceiling, and all the w_i's are strictly positive. Hence, this leads to a contradiction. ∎

Proof of Theorem 1: We first prove the case when $y = 0$, i.e., that the propagation of $f(j,k)$ does not conflict with the *computation* of any other $f(j',k')$. From Eqn. (11), repeated below for convenience, we see that $\Delta(j,k)$ is strictly positive (the three terms are all ceilings, and the last term is an upper bound on the second term).

$$\Delta(j,k) = \lceil (j \bmod w_{k+1} + 1)/\alpha \rceil - \lceil (j \bmod w_k + 1)/\alpha \rceil + \lceil w_k/\alpha \rceil$$

This means that there will never be any need to propagate values to the left. From Eqns (9-10) we have, $t(j,k) = j + a(j,k)$, so $t(j,k) + x = j + a(j,k) + x$. It can also be readily verified that the allocation function is monotonically increasing with k. Let, if possible, for some $x : 0 < x < \Delta(j,k)$,

$$\left[\begin{array}{c} a(j,k)+x \\ t(j,k)+x \end{array}\right] = \left[\begin{array}{c} a(j',k') \\ t(j',k') \end{array}\right]$$

So, $t(j,k) + x = t(j',k') = j' + a(j',k') = j' + a(j,k) + x$. Thus, $j = j'$. Since by our hypothesis, $a(j,k)+x = a(j',k')$, and $x > 0$, we must have by monotonicity, $k' > k$. Since $x < \Delta$, we have $a(j',k') = a(j,k) + x < a(j,k) + \Delta$. We substitute from Eqns (9) and (11), and simplify to obtain

$$\lceil (j \bmod w_{k'} + 1)/\alpha \rceil + \sum_{i=k+1}^{k'-1} \lceil w_i/\alpha \rceil < \lceil (j \bmod w_{k+1} + 1)/\alpha \rceil$$

Now we have two cases. If $k' = k+1$ the summation on the LHS is 0, and the first term on the LHS is identical to the RHS, so we have a contradiction. On the other hand, if $k' > k+1$, then the first term in the summation is an upper the bound on the RHS, and we again have a contradiction.

Now, consider when $y > 0, y = x$. Then subtracting x from both sides of Eqn. (12), we obtain

$$\left[\begin{array}{c} a(j,k) \\ t(j,k) \end{array}\right] = \left[\begin{array}{c} a(j',k') \\ t(j',k') \end{array}\right]$$

which is impossible (by Lemma 1).

Finally, when $x \neq y > 0$, we may assume w.l.o.g.that $x > y$, and subtracting y from both sides of Eqn. (12), we obtain

$$\left[\begin{array}{c} a(j,k)+x' \\ t(j,k)+x' \end{array}\right] = \left[\begin{array}{c} a(j',k') \\ t(j',k') \end{array}\right]$$

for some $0 < x' < \Delta(j,k)$, which we have just shown above is impossible.

Intuitively, if the propagation of $f(j,k)$ conflicts with the propagation of $f(j',k')$, this must imply (by tracing back the propagation paths), that the computation of $f(j,k)$ and $f(j',k')$ conflict, or the propagation of one of them conflicts with the computation of the other. Both of these have been shown to be impossible. ∎

Proof of Lemma 2: Since, the coefficients w_i are uniformly distributed in the interval $[w_{\min}, w_{\max}]$, we have $\lceil w_i/\alpha \rceil = w_i/\alpha + \epsilon_i$, where ϵ_i are uniformly distributed in the interval $[0, \frac{\alpha-1}{\alpha}]$. Using elementary statistical analysis,

$$\begin{aligned}
\mathbf{E}[F(q,\alpha)] &= \frac{c}{q}\mathbf{E}\left[\sum_{i=1}^{m} \lceil w_i/\alpha \rceil\right] &= \frac{c}{q}\sum_{i=1}^{m} \mathbf{E}\left[\lceil w_i/\alpha \rceil\right] \\
&= \frac{c}{q}\sum_{i=1}^{m}(\mathbf{E}[w_i/\alpha] + \mathbf{E}[\epsilon_i]) &= \frac{c}{q}\sum_{i=1}^{m}\left(\frac{w_{\max}+w_{\min}}{2\alpha} + \frac{\alpha-1}{2\alpha}\right) \\
&= \frac{cm}{2q}\left(\frac{w_{\max}+w_{\min}-1}{\alpha} + 1\right)
\end{aligned}$$

We note that a similar result has been obtained by Alexandrov and Fidanova [AS93] using a completely different approach. ∎

Proof of Theorem 2: The objective function in Eqn. (18) is

$$\overline{f}(q,\alpha) = \frac{w_{\max} + w_{\min} - 1}{\overline{R} - q\overline{a}} + \frac{1}{q} \tag{22}$$

Its first and second derivatives with respect to q are respectively

$$\frac{\partial \overline{f}}{\partial q} = \frac{(w_{\max} + w_{\min} - 1)\overline{a}}{(\overline{R} - q\overline{a})^2} - \frac{1}{q^2} \tag{23}$$

and

$$\frac{\partial^2 \overline{f}}{\partial q^2} = \frac{2(w_{\max} + w_{\min} - 1)\overline{a}^2}{(\overline{R} - q\overline{a})^3} + \frac{2}{q^3} \tag{24}$$

Since $\overline{R} - q\overline{a} \geq 1$ then $\frac{\partial^2 \overline{f}}{\partial q^2} > 0$. On the other hand $\overline{f}(q,\alpha)$ does not depend on α and therefore $\overline{f}(q,\alpha)$ is a convex function in the domain of feasible solutions for Eqn. (18). Hence, to find the integer solution of problem PB. 2, it is sufficient to solve its relaxed version and to investigate then the nearest integer points to this solution.

Setting the first derivative Eqn. (23) to zero we get

$$q^* = \frac{R}{\sqrt{a_1 a_2 (w_{\max} + w_{\min} - 1)} + a_1} \quad .$$

From $q\alpha = \overline{R} - q\overline{a}$ we obtain

$$\alpha^* = \sqrt{\frac{a_1 (w_{\max} + w_{\min} - 1)}{a_2}} \quad .$$

∎