An optimal algo-tech-cuit for the knapsack problem

Rumen Andonov, Sanjay Rajopadhye

▶ **To cite this version:**

Rumen Andonov, Sanjay Rajopadhye. An optimal algo-tech-cuit for the knapsack problem. [Research Report] RR-2169, INRIA. 1994. inria-00074503

HAL Id: inria-00074503

https://inria.hal.science/inria-00074503v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

An Optimal Algo-Tech-Cuit for the Knapsack Problem

Rumen Andonov, Sanjay Rajopadhye

N° 2169

Janvier 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

INRIA RENNES

An Optimal Algo-Tech-Cuit for the Knapsack Problem

Rumen Andonov*, Sanjay Rajopadhye**

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués

Projet API

Rapport de recherche n° 2169 — Janvier 1994 — 23 pages

Abstract: We present a formal derivation and proof of correctness of a systolic array for the knapsack problem, a well known, NP-complete problem. The dependency graph of the algorithm is not completely known statically, so the derivation also serves as a case study for systolic synthesis for this class of programs. The array is itself important since it achieves optimal performance on a model much weaker than a PRAM (ring of PE's with a fixed size memory and only nearest neighbor interconnections).

We show how the memory size of each PE can be chosen so that the running time is minimized by formulating and solving a non linear optimization problem. For this, we use the expected running time as the cost function and a register level model of VLSI.

The original array has an intricate tag-based control mechanism which is difficult to implement. We show how this can be reduced to two simple counters and a few flip-flops. Coefficient loading is done with a multi-rate clock which avoids the need for shadow registers.

These results show how it is important to use appropriate tools at different levels, and from different areas for designing application specific array processors.

(Résumé : tsvp)

Supported by the French Coordinated Research Program C^3, and by the Esprit BRA project NANA 2 (No. 6632). A preliminary version was presented at the International Conference on Application-Specific Array Processors, Venice, Oct 1993.

*andonov@irisa.fr (on leave from Center of Computer Science and Technology, Sofia, Bulgaria)

**rajopadhye@irisa.fr (partially supported by NSF Grant No. MIP-910852)

Unité de recherche INRIA Rennes

IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)

Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Un Algo-Tech-Cuit optimal pour le problème du sac à dos

Résumé :

- Nous présentons la dérivation formelle et la preuve d'une solution systolique pour le problème du sac-à-dos, un problème NP complet bien connu. Le graphe des dépendances de cet algorithme n'étant pas statique, la construction d'un réseau systolique pour le sac-à-dos constitue une d'étude de cas pour la synthèse d'architecture. Le réseau proposé atteint des performances optimales tout en restant très simple (un anneau de processeurs élémentaires (PE) possédant une taille mémoire fixe et des connexions locales).
- Nous montrons comment la taille de chaque PE peut être choisie de manière à minimiser le temps d'exécution, en résolvant un problème d'optimisation combinatoire non linéaire. Nous considérons à cet effet le temps moyen d'exécution comme une fonction de coût et un modèle de composant VLSI au niveau registre.
- Le réseau original avait un mécanisme de contrôle difficile à mettre en œuvre. Nous montrons comment il peut être simplifié et remplacé par deux compteurs et des registres. Le chargement des coefficients est réalisé avec une horloge plus rapide qui évite l'utilisation de registres fantômes.

Ces résultats montrent que pour la conception de réseaux spécifiques l'importance, il est important d'utiliser les outils appropriés à chaque niveau de spécifications, ces outils faisant appel à des techniques variés.

1 Introduction

The general knapsack problem is a classic NP-*complete*, combinatorial optimization problem with a wide range of applications [Hu82, MT90]. It can be solved sequentially in $O(mc)$ time, where m is the number of objects and c is the capacity. This is called *pseudo polynomial* complexity [GJ79] since it is polynomial w.r.t. a parameter (which itself, grows exponentially with input size). Many time consuming instances of these problems exist [CHR88, Chv80], and it is therefore important to investigate efficient parallel implementations. One of the standard approaches to solving this problem is dynamic programming [Bel57, GN72, Hu82]. In this paper, we address the problem of deriving a systolic VLSI array for the dynamic programming algorithm for the knapsack problem.

Dynamic programming can usually be specified by a recurrence equation, and there is now a large body of work on synthesizing systolic arrays from such equations. However, we shall see that our recurrence has the form:

$$\text{for all index points, } z \quad X(z) = g(\ldots Y(Az + a + W(z))\ldots) \tag{1}$$

In order to compute X at a point z in the index space, we need the value of Y from the point $Az + a + W(z)$, and this is called a *data dependency* of the equation. The $Az + a$ is a classical affine dependency [RF90], but the presence of a data variable $W(z)$ means that the **dependency itself is not known statically** (see Fig. 1). Such recurrences are said to have "dynamic" (or "run-time") dependencies [Meg93], and the well known dependence projection techniques for systolic array synthesis [QR89] are not 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: the problem is important in its own right, and a careful study may lead to a synthesis method for recurrences with dynamic dependencies. In this paper we present the following results.

- We give a formal derivation (and proof of correctness) of a linear systolic knapsack array. The array is known in the literature [AQ92] and is based on a fixed size ring topology and has optimal speedup. It uses an intricate partitioning scheme, with dynamically computed tags, and our derivation here provides a rigorous proof of correctness.
- We prove the average time complexity of the array is $\Theta\left(\frac{mc}{q}\left\lceil\frac{w_{\max}+w_{\min}}{\alpha}\right\rceil\right)$, where q is the number of PE's, $w_{\max}(w_{\min})$ is the largest(smallest) weight among all problem instances, and α is the size of the memory in each PE, a design parameter. We also formulate and solve a non linear optimization problem which enables us to fix this parameter in an optimal manner.

- The complicated control of the array makes it unrealistic to implement in VLSI. We show how the control can be reduced to two simple counters and a few flip-flops. We also show how the coefficients can be loaded efficiently by means of a multi-rate clocking scheme, which obviates the need of shadow registers. For this final design, we show how the optimal memory size reduces the running time by 26% *without any increase in area.*

This paper is organized as follows. In the next section, we define the problem and the notation, and discuss the state of the art. Then, in Sec. 3, we present a formal derivation of the Andonov-Quinton array [AQ92]. In Sec. 4 we show how the PE memory size which minimizes the running time can be determined by solving a non linear optimization problem. We then address (in Sec. 5) the main drawback of the array, namely the complicated control mechanism. We show how it can can be reduced to two counters and three latches. Finally, we present our conclusions (Sec. 6) and a discussion of future work.

2 Background and Previous Work

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 *value* 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.,

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

where z_i is the number of i-th type objects included in the knapsack. There are many variations of this problem, and since this is an important combinatorial optimization problem, a number of researchers have developed parallel algorithms. Dynamic programming [LSS88, CCJ90, LS91, Ten90], branch-and-bound [LS84, JF88] and approximate algorithms [GRK91, Ger91] are the most popular techniques used. As noted by Teng [Ten90], the performance of parallel algorithms depends on c, m and also $w_{\max}$, the largest object weight, $w_{\max} = \max_{1 \leq i \leq m} w_i$ (unlike the sequential case when only m and c are relevant). The number of processors required by most of the 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 (2) proposed by Teng [Ten90] requires $M(c)$ processors* to solve the problem in $O(\log^2(mc))$ time. The function $M(n)$ above denotes

*We use the word processors when we discuss software implementations, and PE's when refering to hardware implementations.

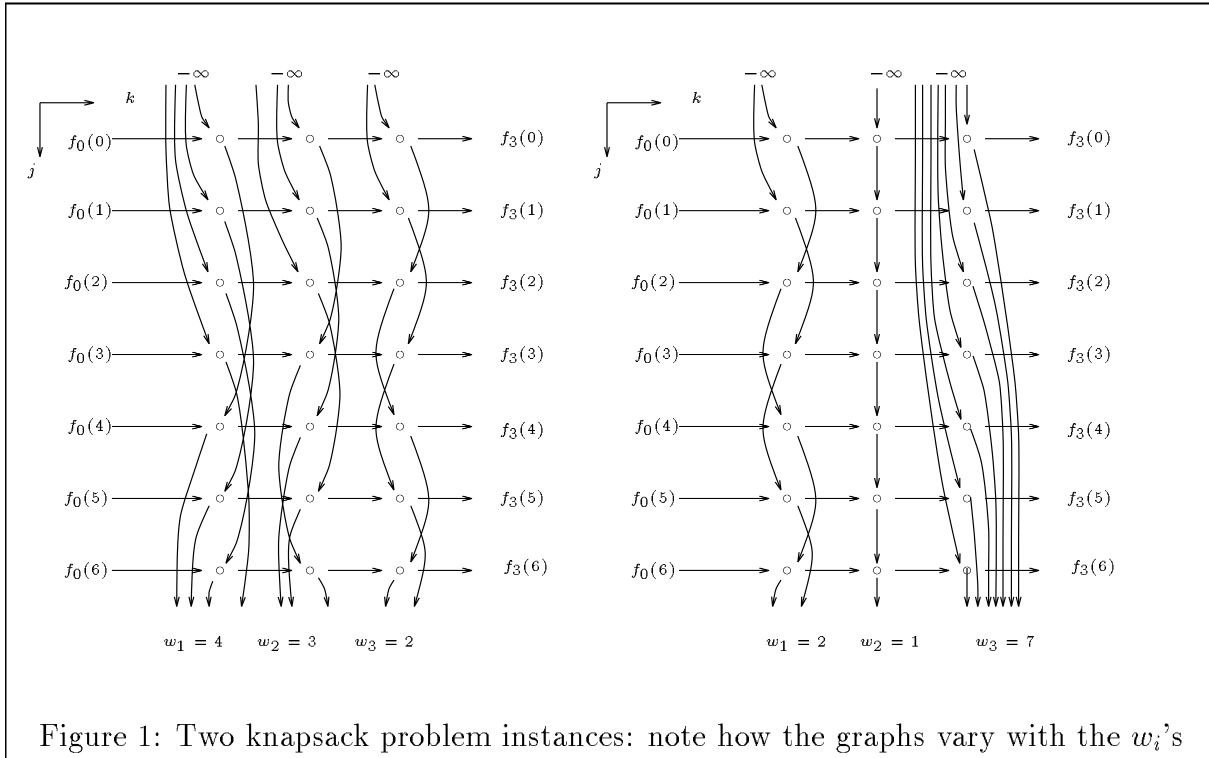


Figure 1: Two knapsack problem instances: note how the graphs vary with the w_i's

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.

Dynamic Programming for Knapsack Problems: One of the most well known approaches for solving knapsack problems is dynamic programming. It is based on the *principle of optimality* [Bel57] and usually contains two phases. In the first (forward) phase the maximum **value** of the objective function is computed, and this is used in the second (backtracking) phase to determine the **actual solution**. It is known that the backtracking phase is inherently sequential [AQ92], and can be performed in $c + m$ steps if certain bookkeeping information is maintained during the forward phase. In our array, this bookkeeping will require one comparator-adder and two registers in each PE. It is thus a constant overhead, and affects only the final PE size, but not the nature of the analysis. We therefore focus on the forward phase, which is specified by the following recurrence:

for all $j, k:\ 0 < j \leq c, 0 < k \leq m$, find $f(c, m)$, where

$$f(j,k) = \oplus\{f(j,k-1), f(j-w_k, k-\beta) + p_k\} \tag{3}$$

with boundary conditions $f(j, 0) = f(0, k) = 0$ and $f(j, k) = -\infty$ when $j < 0$.

It is well known [MT90] that different variations of the knapsack problem merely correspond to different choices of $\oplus$ and the constant β in (3):

- The **general knapsack problem** ($\oplus$ is max, and $\beta = 0$).
- The **0/1 knapsack problem** ($\oplus$ is max and $\beta = 1$).
- The **subset-sum problem** (a 0/1 knapsack problem with $p_i = w_i$).
- The **change making problem** ($\oplus$ is min and $\beta = 0$).

Henceforth, we consider, w.l.o.g., the general knapsack problem—all our results can be extended to the other variants.

Recently, a dynamic programming algorithm 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 and this algorithm has optimal speedup and processor efficiency. The authors report that on a Connection Machine, the time complexity increases to $\Theta(\frac{mc\log q}{q})$ because of communication costs. 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.

Our model of computation is weaker—a systolic ring, where each processor is required to have only a constant size memory; we have optimal performance on this model. Furthermore, a software version of our algorithm has very good performance, as reported in a companion paper [ARQ93] (see references therein for a detailed survey of software implementations).

Systolic Arrays for Dynamic Programming Algorithms: 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 they can exploit properties of the particular problem at hand. This is the case for the knapsack problem too—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 space and time.

Background of this paper: This paper represents a step in an evolutionary development of our results. The main difficulty in obtaining a systolic knapsack array arises from the run-time dependency, which would seem to require unbounded memory in each PE.

The first solution with constant memory was proposed by Andonov, Aleksandrov, and Benaini [AB90, AAB91] (each processor has exactly α words of memory). The crucial idea is a tagging scheme which ensures proper data synchronization. The array is extensible w.r.t. m, c and $w_{\max}$, but has a running time of $\Theta(mc^2/q\alpha)$ on a ring of q PE's, which is unacceptable (the sequential implementation takes mc steps). A two dimensional version was proposed by Andonov and Gruau [AG91] but it also has poor performance. Andonov and Quinton later presented a linear array [AQ92], where the running time was brought down to $\Theta(mc/q)$. They use a complicated control mechanism to implement the tagging scheme which requires each processor to perform a division modulo an arbitrary integer (no larger than $w_{\max}$). This renders it impractical and also sacrifices extensibility w.r.t. $w_{\max}$. Most of the previous results have appeared in conferences or as technical reports. This paper is the first, complete version submitted to an archival journal.

3 Systolic Scheduling of the Knapsack Recurrence

Based on the above considerations, we shall design our array by studying the specific structure of our problem formulation (3). Our final design improves the Andonov-Quinton [AQ92], which was presented in an intuitive manner. Our derivation in this Section is a rigorous proof of correctness, and also serves as a case study for systolic array synthesis for recurrences with run-time dependencies.

There are two crucial observations about the dependencies of (3) that allow us to synthesize an array, even though we do not completely know the dependency graph statically—the dynamic component of the dependencies is **vertical** and **downward** (see Fig 1). The first observation implies that if we choose the allocation function to be $a(j,k) = k$, all the "irregularity" is projected into the processor space. The PE must manage a certain amount of memory (not just a few registers), but the interconnections are purely systolic. The second observation implies that $t(j,k) = j + k$ is a valid linear schedule.

This yields an array with m PE's, with $w_{\max}$ words of memory. The PE's are initialized with w_k, and at each cycle they compute $\oplus$ (i.e., max) using one value that arrives from the left neighbor and one value from the memory; the result is sent to the right neighbor and also stored in the memory (only the first w_k memory locations are used). The program executed by the k-th PE is as follows:

for $t = k$ to $k + c$ do

$$[f_{\text{out}}, \text{mem}(t_{\text{mod} w_k})] = \begin{cases} f_{\text{in}} & \text{if } t < k + w_k \\ \oplus(f_{\text{in}}, \text{mem}(t_{\text{mod} w_k}) + p_k) & \text{otherwise} \end{cases}$$

The problem with this array, is that the memory size of each PE is $w_{\max}$, which is a problem parameter. Therefore the solution is not truly systolic. VLSI implementations of

this array have, nevertheless, been studied and the limitations exposed [MA93]. To solve this, we use a space time map given by the following allocation and timing functions (α, the memory size of each PE, is a constant—a design parameter).

$$a(j,k) = \lceil (j \bmod w_k + 1)/\alpha \rceil + \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil \quad (4)$$

$$t(j,k) = j + \lceil (j \bmod w_k + 1)/\alpha \rceil + \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil \quad (5)$$

We will also need an auxiliary function for our proof of correctness:

$$\Delta(j,k) = \lceil (j \bmod w_{k+1} + 1)/\alpha \rceil - \lceil (j \bmod w_k + 1)/\alpha \rceil + \lceil w_k/\alpha \rceil \quad (6)$$

This space-time map is not a simple linear transformation, as used in the literature [QR89]. Indeed, with such a mapping, there is even no guarantee that the transformed algorithm is systolic, and this needs to be proved formally.

Lemma 1 *The space-time map given by (4-5) is free of conflict.*

Proof: We show that for any two points, $[j,k]$ and $[j',k']$, in the index space, if

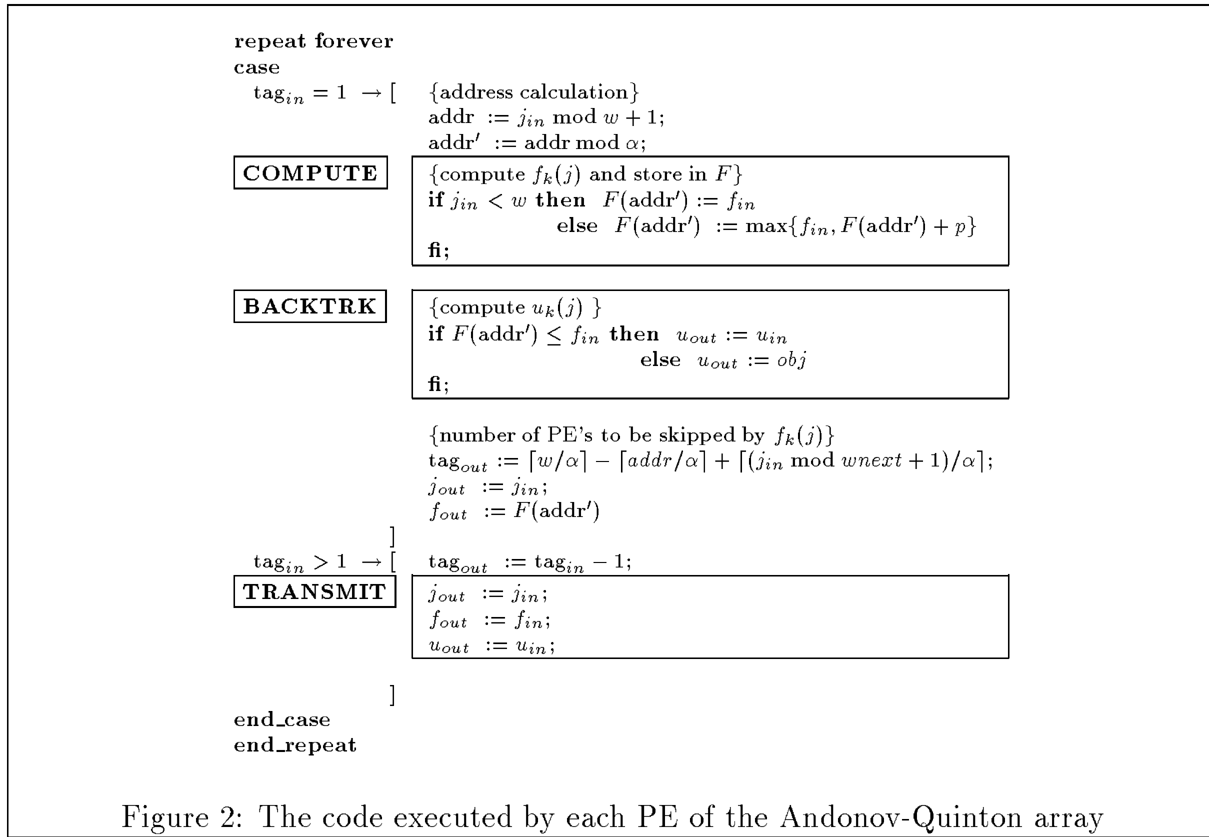
$$\begin{bmatrix} a(j,k) \\ t(j,k) \end{bmatrix} = \begin{bmatrix} a(j',k') \\ t(j',k') \end{bmatrix}$$

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

However, merely proving that the mapping is conflict-free, is not enough. We can readily see from the above definitions, that

$$a(j,k+1) - a(j,k) = \Delta(j,k) \quad (7)$$

Thus, the dependency $[j,k] \to [j,k+1]$ is mapped to processors that are separated by $\Delta(j,k)$. Since this is not a constant vector, but a function of the index point and (more importantly), of w_k which is not known statically, our array does not even have local interconnections! Fortunately, we also see that $\Delta(j,k) = t(j,k+1) - t(j,k)$. Hence, a value produced by any processor, x, at time, t, is required by processor $x + \Delta$ at time $t + \Delta$. As a result, we may design a systolic array that operates as follows. Each value of f computed in the array is *tagged* with a Δ value. Every PE merely transmits it, while decrementing the tag. By the time the value arrives at the destination, the tag has reduced to 1, and the PE "knows" that the value is for it. The code executed by each PE



```
repeat forever
case
  tag_in = 1  → [   {address calculation}
                    addr  := j_in mod w + 1;
                    addr' := addr mod α;
COMPUTE             {compute f_k(j) and store in F}
                    if j_in < w then  F(addr') := f_in
                               else  F(addr')  := max{f_in, F(addr') + p}
                    fi;

BACKTRK             {compute u_k(j) }
                    if F(addr') ≤ f_in then  u_out := u_in
                                      else  u_out := obj
                    fi;

                    {number of PE's to be skipped by f_k(j)}
                    tag_out := ⌈w/α⌉ − ⌈addr/α⌉ + ⌈(j_in mod wnext + 1)/α⌉;
                    j_out  := j_in;
                    f_out  := F(addr')
                ]
  tag_in > 1  → [   tag_out  := tag_in − 1;
TRANSMIT            j_out  := j_in;
                    f_out  := f_in;
                    u_out  := u_in;

                ]
end_case
end_repeat
```

Figure 2: The code executed by each PE of the Andonov-Quinton array

is shown in Fig 2 (it includes the book-keeping required for the backtracking part of the algorithm).

In order to prove that this strategy works correctly we must show that when a data value is being propagated in this manner, the PE's that it passes through are not busy with their own computation (otherwise there will be a conflict for the interconnection wires, and also for the ALU to decrement the tag).

Theorem 1 *The tagging strategy is conflict free.*

Proof: (see Appendix) The main idea is that for two distinct points, $[j,k]$ and $[j',k']$, and for any $x : 0 < x < \Delta(j,k)$

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

leads to a contradiction. ∎

An intuitive explanation of the operation of the array is as follows. To perform the computation associated with the k-th column of the dependency graph, we must store w_k

data values, but we have only α words of memory on each PE. Hence, we allocate $\lceil \frac{w_k}{\alpha} \rceil$ PEs for the job, so that between them, they have sufficient memory. However, this causes two complications: it causes the $[0,1]$ dependency (which is uniform) to be allocated to processors that are no longer adjacent; moreover, the number of PEs required for a column (which is also the distance that a data value has to travel in order to reach the PE that needs it) is not known statically, but depends on the problem instance. The solution is to choose the timing function in such a manner that a tagging scheme can be developed to control the data propagation. For further details, the reader is referred to [AQ92].

4 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. Hence we can treat it as a design parameter. We now show how it can be chosen optimally. Since $[c,m]$ is the last point to be computed, the running time of architecture is obtained by substituting this in (5) as follows:

$$t(c,m) = \frac{c}{q}\sum_{i=1}^{m}\left\lceil\frac{w_i}{\alpha}\right\rceil + c + q + 1 \tag{8}$$

We let $F(q,\alpha) = \frac{c}{q}\sum_{i=1}^{m}\lceil w_i/\alpha\rceil$, which is the dominant term in (8). The problem that we must solve, can be formulated as follows.

For a given resource, R (silicon area), which may be used either as memory or as PE's, what values of α and q minimize the objective function, $F(q,\alpha)$?

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

$$\mathbf{E}(F(q,\alpha)) = \frac{cm}{2q}\left(\frac{w_{\max} + w_{\min} - 1}{\alpha} + 1\right) \tag{9}$$

Proof: See Appendix ∎

Now, q depends on α, since the number of PE's that we can implement on a chip depends crucially on how much memory they have. Let a_1 be the (area) cost of the CPU + control + IO, and let a_2 be the (area) cost of the memory per unit (word). Hence the cost of a PE with α words of memory is $a_1 + a_2\alpha$. Then the ressource restriction is expressed as the non-linear constraint

$$q(a_1 + a_2\alpha) \leq R$$

which is equivalent to

$$q\alpha \leq \overline{R} - q\overline{a},$$

where $\overline{R} = R/a_2$ and $\overline{a} = a_1/a_2$. 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.

$$\begin{array}{lrll} \text{minimize} & \frac{w_{\max}+w_{\min}-1}{q\alpha} + \frac{1}{q} & & \\ \text{subject to} & q\alpha & \leq \overline{R} - q\overline{a} & \\ & 1 & \leq \alpha \leq w_{\max}, & \text{integer} \\ & 1 & \leq q, & \text{integer} \end{array} \tag{10}$$

Instead of problem (10) we will find the exact solution of its approximation, i.e. of the problem given below. Also note that the problems (10) and (11) are equivalent when the requirement for integral q and α is relaxed, since in this case the solution always satisfies the equality $q\alpha = \overline{R} - q\overline{a}$.

$$\begin{array}{lrll} \text{minimize} & \frac{w_{\max}+w_{\min}-1}{\overline{R}-q\overline{a}} + \frac{1}{q} & & \\ \text{subject to} & q\alpha & \leq \overline{R} - q\overline{a} & \\ & 1 & \leq \alpha \leq w_{\max}, & \text{integer} \\ & 1 & \leq q, & \text{integer} \end{array} \tag{11}$$

Theorem 2 *Let (q^*, α^*) be the solution of the relaxed version (i.e., when the requirement of integral solution is relaxed) of (11). It is given by*

$$\alpha^* = \sqrt{\frac{a_1(w_{\max} + w_{\min} - 1)}{a_2}} \tag{12}$$

and

$$q^* = \frac{R}{\sqrt{a_1 a_2(w_{\max} + w_{\min} - 1)} + a_1} \tag{13}$$

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

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

Now, the solution (q_{int}, α_{int}) *of (11) is one of* $(\lfloor q^* \rfloor, \lfloor \alpha^* \rfloor)$, $\left(\lceil q^* \rceil, \left\lfloor \left(\frac{R}{\lceil q^* \rceil} - a_1\right)/a_2 \right\rfloor\right)$, *or* $\left(\left\lfloor \frac{R}{a_1 + a_2 \lceil \alpha^* \rceil} \right\rfloor, \lceil \alpha^* \rceil\right)$

Proof: See Appendix ∎

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

$$\mathbf{E}(F(q^*, \alpha^*)) = \begin{cases} \frac{cm}{2R}\left[\sqrt{a_1} + \sqrt{a_2(w_{\max} + w_{\min} - 1)}\right]^2 & \text{if } \frac{a_1}{a_2} \leq \frac{w_{\max}^2}{w_{\max} + w_{\min} - 1} \\ \frac{cm(2w_{\max} + w_{\min} - 1)}{2R}\left(a_2 + \frac{a_1}{w_{\max}}\right) & \text{otherwise} \end{cases} \tag{14}$$

Proof: By substituting α^* and q^* in (9) and simplifying. ∎

5 An Improved Control Mechanism

We now address the problem of physical implementation. Observe from (14) that one of the most important factors affecting the expected running time of the array is a_1 the "fixed cost" of the PE (all the other terms involve parameters of the problem and/or the hardware technology), and we have much to gain by simplifying the PE design. We see from Fig 2 that the main part of the computation (the lines within the boxes) is fairly simple. It can be implemented with a couple of comparators and an adder, plus a few multiplxers. However, the tag based control is significantly more expensive, since it requires two divisions modulo α, one modulo w_k and one modulo w_{k+1}. This is the principal drawback of the array, which renders it unacceptable for VLSI implementation. We will show how the control can be achieved with two simple counters and a few flip-flops.

As we have seen, the function $\Delta(j, k)$ is crucial for the tagging scheme. Consider the PE, $a(j, k)$ which has to compute $f(j, k)$, and let $p = \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil$, and $x = \lceil \frac{j \bmod w_k + 1}{\alpha} \rceil$. Note that $0 < x \leq \lceil w_k/\alpha \rceil$. Intuitively, the PE's may be viewed as being partitioned into *virtual processors*, one for each column. The first p PE's constitute VP(1) ...VP($k-1$); and VP(k) consists of the PE's $p+1 \ldots p + \lceil w_k/\alpha \rceil$. Hence the points that are mapped to our PE, $p + x$ are all from the k-th column. The smallest j among these points is $(x-1)\alpha = j_0$ (say).

For simplicity, we only consider the case when the w_i's are multiples of α. It is easy (but tedious) to formally prove that the strategy works in the general case. Observe from the following list, that the j values of the set of points allocated to the PE, $p + x$, have the form, $j_0 + \mu w_k + \nu$:

$$\begin{array}{lllllll}
j_0 & j_0+1 & \dots & j_0+\nu & \dots & j_0+\alpha-1 \\
j_0+w_k & j_0+w_k+1 & \dots & j_0+w_k+\nu & \dots & j_0+w_k+\alpha-1 \\
j_0+2w_k & j_0+2w_k+1 & \dots & j_0+2w_k+\nu & \dots & j_0+2w_k+\alpha-1 \\
 & & & \vdots & & \\
j_0+\mu w_k & j_0+\mu w_k+1 & \dots & j_0+\mu w_k+\nu & \dots & j_0+\mu w_k+\alpha-1 \\
 & & & \vdots & &
\end{array}$$

In general, the set of index points computed by the $(p+x)$-th PE is

$$a^{-1}(p+x) = \{[j,k] : p = \sum_{i=1}^{k-1} \lceil w_i/\alpha \rceil,\ j = (x-1)\alpha + \mu w_k + \nu,\ \text{for } 0 \le \nu < \alpha\}$$

Here, μ is bounded from above by c/w_k. The time instants at which this PE is active can be obtained, simply by substituting this in (5) and some simplification, as

$$T(p+x) = \{p + x + (x-1)\alpha + \mu w_k + \nu : 0 \le a,\ 0 \le \nu < \alpha\} \tag{15}$$

Thus we see that each PE starts its activity at $t_0 = p + x + (x-1)\alpha + \mu w_k + \nu$, and after that it stays active for the next α steps, and this pattern repeats with a period of w_k. Based on this, we have the following.

Proposition 1 *The control of each PE may be implemented with two counters, one that counts modulo α and one that counts modulo w_k.*

Fig 3 gives the code executed by each PE with the improved control. **TRANSMIT** corresponds to the propagation of data, **COMPUTE**, performs the main computation (forward phase and also backtracking) and **LOAD-MEMORY** is done when $j < w_k$, when the data in the memory is assumed to be $-\infty$. This part is the same as in the Andonov-Quinton array (except for some reorganization; note that C_α corresponds to addr′, and C_w corresponds to addr), and can be laid out in a fairly standard manner. The control part uses three flip-flops (ACT, MEM, and FIRST) and the two counters, $\mathrm{C_w}$ and C_α. The former is a programmable counter and its upper limit is controlled by w_k, but the latter counts modulo α_{opt}, a fixed constant.

5.1 Coefficient loading

For correct operation, each PE must know w_k, p_k, obj_k and FIRST_k. 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 PE's have the same coefficients, etc.) and the appropriate

```
repeat forever
  {computation}
  if (ACT = 1 and MEM = 0) then          LOAD-MEMORY
  else if (ACT = 1 and MEM = 1) then     COMPUTE
  else                                     TRANSMIT
  fi;
  {control part}
  if (ACT = 1 and C_α = α − 1 and C_w ≠ w − 1) then
                      ACT ← 0;C_α ← 0; MEM ← 1; start_out ← 1
  else if (ACT = 1 and C_w = w − 1) then
                      ACT ← 0; C_α ← 0; MEM ← 1
  else if ((start_in = 1) or (FIRST = 1  and C_w = w − 1)) then
                      ACT ← 1; C_α ← 0; MEM ← 1
  else                C_α ← (C_α + 1) mod α
  fi;
  C_w ← (C_w + 1) mod w;
end_repeat
```

Figure 3: The code executed by the PE in the improved VLSI array

sequence of coefficients can be generated off-line. A standard 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 PE's 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 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. Figs 4 and 5 show the data path and the control for the final design

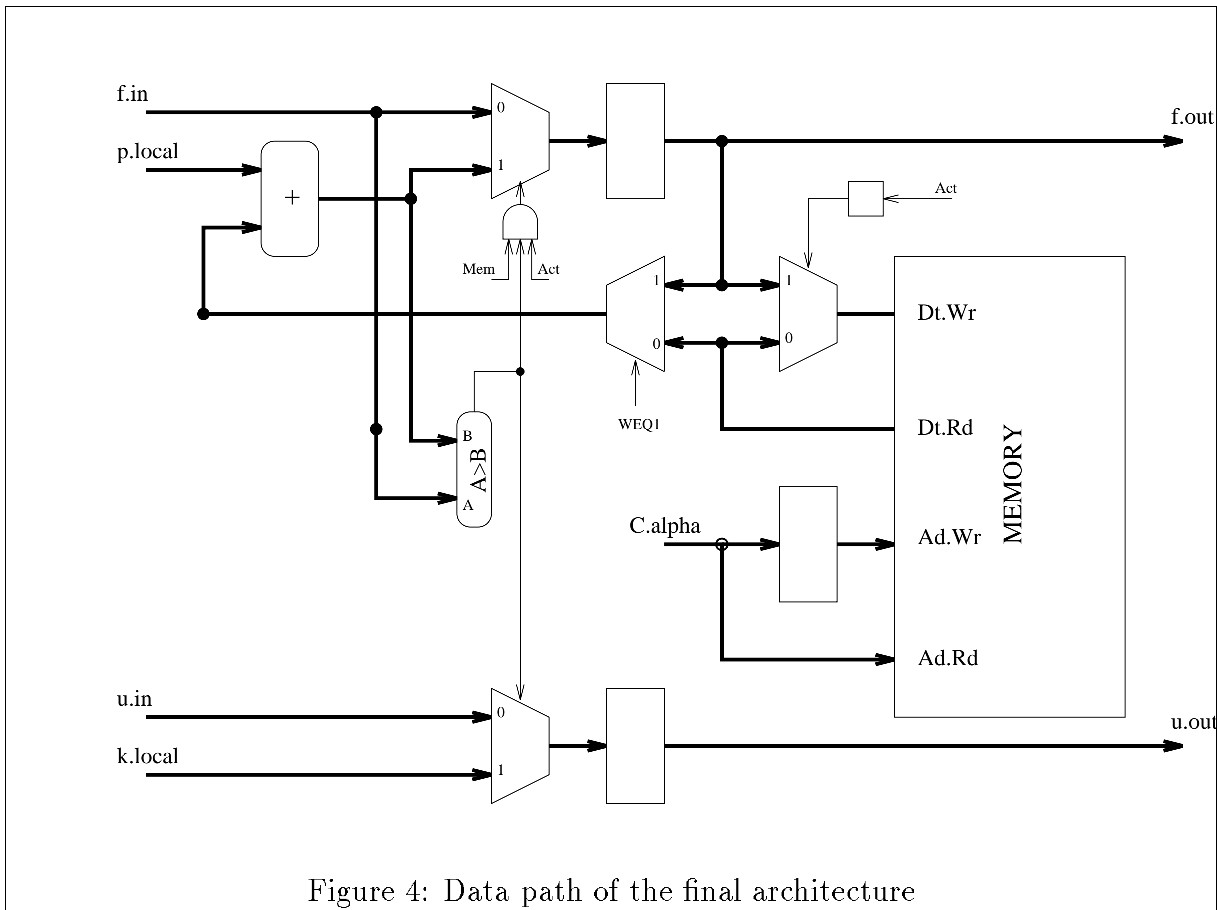


Figure 4: Data path of the final architecture

5.2 Performance Estimation

We now apply the results of Sec. 4 to the VLSI design above, 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 utilize the results of Sec. 4. 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 15 registers. The control part takes up about 12 registers, and thus, the entire PE can be laid out in 27 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

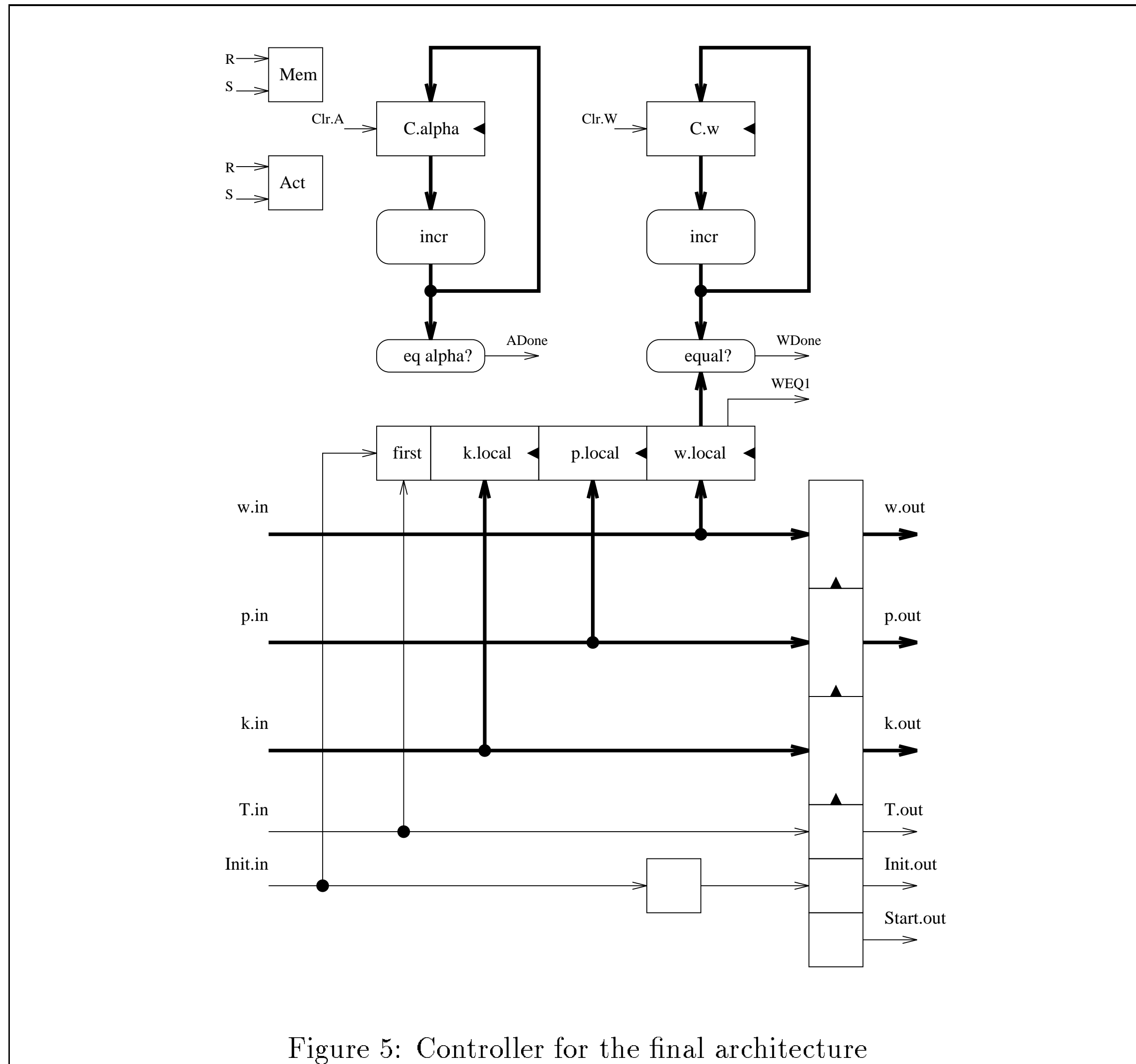


Figure 5: Controller for the final architecture

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 $a_1 = 27$.

Using this data, we first see that $\frac{a_1}{a_2} \leq \frac{w_{\max}^2}{w_{\max}+w_{\min}-1}$ is true, so the optimal solution is given by (12) and (13), and the nearest integer values which minimize the running time are with 15 PE's, each with 219 words of memory. Substituting these into (9) the expected running time of the optimal array is $0.1855mc$.

On the other hand, the unoptimised 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 24 registers, enabling 4 PE's to be squeezed on a chip. However, the naive design *always* allocates one processor per column, and

the dominant term in the running time is $\frac{mc}{q} = 0.25mc$ (obtained from (8) by letting $\alpha = w_{\max}$); we then have we have the following.

Result 1 *The memory optimization reduces expected running time by 25.8% 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 Conclusions and Future Work

We have presented a complete derivation of an optimal VLSI design for the general knapsack problem. Since it belongs to the class of NP-*complete* 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.

In deriving the systolic array, we used the established synthesis strategy as far as possible—determine a timing function and an allocation function, show that they do not cause a scheduling conflict, and use them to determine the interconnections of the array. However, because of the presence of the run-time dependency, we could not use the conventional linear (or even quasi-linear) functions for which the methodology is well established, and we had to prove non conflict by hand. We also saw that we were able to choose an allocation function that localized the dynamic part of the dependency, but in order to derive an array with only finite memory, the uniform dependency was mapped to non adjacent processors. Finally, we saw that by a proper choice of the timing function, this could be controlled with a tagging scheme. We conjecture that such control functions will be an essential feature of systolic arrays for problems with run-time dependencies. A synthesis method could be based on choosing allocation and timing functions to obtain a space-time map of the problem domain, and then using properties of the space-time image of the dependencies to generate control functions (as we did with $\Delta(j,k)$ in this paper). Developing a general methodology for this process is an interesting open problem.

We formulated the problem of choosing the memory size as a non linear optimization problem, which we solved analytically. This approach can be used for many related problems. We have recently used it to determine the optimal size for iteration space tiling (equivalently, the problem of choosing the parameters for an LSGP partitioning when a systolic algorithm is implemented on a fixed sized array). It could be used for selecting

the digit size when deriving digit-based arithmetic circuits from bit-serial dependency graphs, and related problems.

Finally, we have developed circuit level optimizations to make the final implementation practical for VLSI. The improved control function can be viewed as a systolic implementation that computes the function $\Delta(j,k)$ (actually, an equivalent function which computes the predicate $\Delta(j,k)=1$). 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 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. At the final circuit level, there is still scope for improvement. Different memory technologies may be tried out, different clocking schemes may be explored, and so on. Our results show how it is important to combine many diverse areas—optimal parallel algorithms, VLSI design, operations research and dependence analysis to achieve high performance ALGO-TECH-CUIT engineering for application specific array processing*.

One drawback of our final array is that theoretically, the PE design is not purely systolic, since we need a counter modulo $w_{\max}$, which is a problem parameter. However, this is not a practical limitation, since this is only related to the size of a single register. Indeed, by the same token, the array is not extensible w.r.t. c because of the possibility of arithmetic overflow. Nevertheless, we have recently developed another implementation based on programmable shift registers which is truly systolic [AQRW93].

Acknowledgments The authors would like to thank J-J. Bellanger, A. Benaini (especially for early insights) D. Lavenier and D. Wilde for helpful discussions.

References

[AAB91] R. Andonov, V. Aleksandrov, and A. Benaini. *A Linear Systolic Array for the Knapsack Problem.* Technical Report, Center of Computer Science and Technology, Acad. G. Bonchev St., bl. 25a, Sofia 1113, Bulgaria, 1991.

[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.

*It should be clear by now, that ALGO-TECH-CUIT = ALGO-rithm + archi-TECH-ture + cir-CUIT

[AG91] R. Andonov and F. Gruau. A 2D toroidal systolic array for the knapsack problem. In *Algorithms and Parallel VLSI Architectures II,* (Bonas, France), June 1991.

[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), September 1992.

[AQRW93] R. Andonov, P. Quinton, S. Rajopadhye, and D. Wilde. A shift-register implementation of an optimal systolic circuit for the general knapsack problem. Nov 1993. submitted to Parcella 94.

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

[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.

[Chv80] V. Chvatal. Hard knapsack problems. *Operations Research*, 28:1402–1411, 1980.

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

[Ger91] T. E. Gerasch. A parallel approximation algorithm for 0/1 knapsack. In *Proc. International Conference on Parallel Processing*, pages 302–303, 1991.

[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.

[GRK91] P.S. Gopalakrishnan, I.V. Ramakrishnan, and L.N. Kanal. Approximate algorithms for the knapsack problem on parallel computers. *Information and Computation*, 91:155–171, 1991.

[Hu82] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley Publishing Company, 1982.

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

[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.

[MA93] W. P. Marnane and R. Andonov. *Algorithm Engineering and VLSI Design.* Internal Report 748, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, July 1993.

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

[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.

[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 4-5) 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 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: From (6), 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 this fact, and (7), it follows that the allocation function is monotonically increasing with k. From (4-5) we have, $t(j,k) = j + a(j,k)$, so $t(j,k) + x = j + a(j,k) + x$. Let, if possible, for some $x : 0 < x < \Delta(j,k)$ $\begin{bmatrix} a(j,k)+x \\ t(j,k)+x \end{bmatrix} = \begin{bmatrix} a(j',k') \\ t(j',k') \end{bmatrix}$. 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 substitute from (4) and (6), 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 the bound on the LHS, and we again have a contradiction. ∎

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 (11) is

$$\overline{f}(q,\alpha) = \frac{w_{\max}+w_{\min}-1}{\overline{R} - q\overline{a}} + \frac{1}{q} \tag{16}$$

Its first and second derivatives wrt q are respectively

$$\frac{\partial \overline{f}}{\partial q} = \frac{(w_{\max} + w_{\min} - 1)\overline{a}}{(\overline{R} - q\overline{a})^2} - \frac{1}{q^2} \tag{17}$$

and

$$\frac{\partial^2 \overline{f}}{\partial q^2} = \frac{2(w_{\max} + w_{\min} - 1)\overline{a}^2}{(\overline{R} - q\overline{a})^3} + \frac{2}{q^3} \tag{18}$$

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 (11). Hence, to find the integer solution of (11) it is sufficient to solve its relaxed version and to investigate then the nearest integer points to this solution.

Setting the first derivative (17) 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

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

∎

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399