Algorithm engineering and VLSI design

William P. Marnane, Rumen Andonov

▶ To cite this version:

William P. Marnane, Rumen Andonov. Algorithm engineering and VLSI design. [Research Report] RR-2058, INRIA. 1993. inria-00074614

HAL Id: inria-00074614
https://inria.hal.science/inria-00074614

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Algorithm Engineering and VLSI Design

William Marnane and Rumen Andonov

N° 2058

Août 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

1993

ISSN 0249-6399

Algorithm Engineering and VLSI Design

William Marnane * and Rumen Andonov **

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

Rapport de recherche n ° 2058 — Août 1993 — 21 pages

Abstract: The task of producing a VLSI architecture that will solve a given problem contains many design decisions. The effects of these decisions on the final design are often difficult to quantify. We will compare three different implementations of a systolic architecture for solving the knapsack problem using the dynamic programming method. The design decisions involved and the "engineering" of the algorithm (altering the algorithm to improve its implementation), are highlighted and their effects on the resulting implementation are discussed. We will also relate the design area and timing to the parameters of the knapsack problem. We derive $\Theta(\log(w_{max}))$ as estimations of area and circuit timing complexity of the memory control. We will eliminate the influence of the dominant knapsack parameter (the capacity c) and replace it by the maximum weight w_{max}. This is of crucial importance for a realistic and efficient VLSI implementation of this problem since in practice $w_{max} \ll c$.

Key-words: VLSI design, integer programming, dynamic programming, partitioning, recurrences, parallelism, knapsack problem, systolic algorithm

(Résumé : tsvp)

*This work was carried out at IRISA, supported under the EC Human Capital and Mobility Fund, presently at: Department of Electrical Engineering and Microelectronics, University College Cork, Cork, Ireland

**On leave from Center of Computer Science and Technology, Acad. G. Bonchev st., bl. 25-a, 1113 Sofia, Bulgaria

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

Algorithmique et conception VLSI

Résumé : Le passage d'un problème à l'implémentation VLSI d'une architecture qui résoud ce problème comporte de nombreuses choix de mise en œuvre. Les effets de ces choix sur le résultat final sont souvent difficiles à quantifier. Dans cet article, on compare trois implémentations différentes d'une architecture systolique pour résoudre le problème du sac à dos par la méthode de programmation dynamique. Les choix de mise en œuvre ainsi que "l'ingéniérie algorithmique" (l'altération de l'algorithme pour améliorer sa mise en œuvre) sont mis en évidence, et leurs effets sur le résultat sont discutés. Dans la troisième implémentation, on obtient un contrôleur mémoire dont la complexité en surface et en temps est $\Theta(\log(w_{max})$, où w_max est le poids maximum des objets. De la sorte, on élimine dans ce module, l'influence du paramètre dominant de l'algorithme du sac à dos, sa capacité c. Ceci est crucial pour obtenir une implémentation VLSI réaliste et efficace pour ce problème, puisqu'en pratique, $w_{max} \ll c$.

Mots-clé : VLSI design, programmation linéaire, programmation dynamique, récurrences, partitionnement, problème du sac à dos, parallèlisme, algorithme systolique

1 Introduction

Consider the design cycle from the problem domain to a VLSI solution. Generally the designer will chose the most appropriate algorithm to solve the problem. Then follows a long process of adapting the algorithm to the requirements of the considered computer architecture. Very often this process results in significant modifications at algorithmic level. Finally the designer will design a VLSI implementation of the architecture. In this paper we apply this approach to a dynamic programming algorithm solving the knapsack problem on a pipelined parallel architecture [1, 2, 3].

The knapsack problem includes a large class of classical combinatorial optimization problems with a wide range of applications in the the management and efficient use of scarce resources to increase productivity [4, 5]. These types of problems appear frequently in computer science, operations research and cryptology [6, 7, 8]. These are hard problems to solve - i.e. they belong to the class of NP-*complete* problems [9] and polynomial algorithms for their solution are not known. The results in [10, 11] prove that time-consuming instances of these problems are easily generated. Therefore we wish to use the power of current technology in order to get a VLSI implementation for this class of problems.

One of the most well known approaches for solving knapsack problems is dynamic programming. It is based on the *principle of optimality* of Bellman [12]. Efficient dynamic programming algorithm for a class of knapsack problems, was presented recently by Lin and Storer [13]. It improves on Lee *et al.* [14], and its running time is $\Theta(mc/q)$ [1] 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.* [3] present another idea using a linear array of q general purpose processors connected as a ring [2]. It has a time complexity $\Theta(mc/q)$, which is asymptotically optimal. However the authors in [3] do not study the influence of the high communication overhead which is typical for any transputer implementation.

[1] here m and c are the parameters giving the size of the problem (see section 2 for further details)

[2] the authors use transputers for their implementation

Our parallel algorithm is close to the approach of Chen *et al.* [3] and has the same time complexity. But it requires significantly less memory and it is systolic oriented. The considered model meets completely the requirements for communication intensive algorithms. More precisely our work is related to the design of systolic arrays for dynamic programming algorithms. The difficulties in this field of research arise from the necessity for the algorithms to meet the requirements of modularity, ease of layout, simplicity of communication and control and scalability. 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 [15], path planning [16], etc., some general arrays [17, 18] as well as arrays that can solve any instance of dynamic programming, [19, 20]. 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 [19] for our problem, we will get worse results than our approach.

Based on these considerations, our array is designed by studying the specific structure of the corresponding dynamic programming recurrent equation. A starting point of our study are the results in [2, 21] where a processor-efficient systolic algorithm for the knapsack problem has been recently presented. Due to its simplicity, regularity and local interconnections the proposed array is a good candidate for VLSI implementations. The design parameters (the number of processing elements and their memory storage) are problem size independent. Moreover comparisons and additions are the unique operations used in this dynamic programming implementation. However the results in [2] are more theoretical and they raise numerous questions for an efficient implementation of this array. We answer some of them in this paper, raising on the other hand, more questions concerning the VLSI implementation of *pseudo polynomial* algorithms [5].

There are of course many choices in how we implement a given systolic array as a VLSI design. In this study we present three different VLSI implementations of a systolic architecture to compute the knapsack problem. We will compare each implementation in terms of silicon area and clock speed. The design area and timing is also related to the knapsack parameters, thus allowing each design to be evaluated for different problem sizes.

This paper is organized in the following way. Section 2 describes both phases of the dynamic programming approach for the considered problem on a serial machine. In section 3 we discuss the parallel implementation on a linear systolic array. Sections 4 and 5 are devoted to the analysis of the different VLSI implementations and their impact at algorithmic level. Our conclusions are presented in section 6.

2 Dynamic Programming Algorithm

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 unit profit, w_i the unit weight and x_i the number placed in the knapsack of the i-th type of object. The values w_i, p_i, x_i, $i = 1, 2, \ldots, m$, and c are all positive integers. The problem is to find out the maximum total profit without exceeding the total weight constraint or capacity of the knapsack. Thus we must determine how many of each object we may put in the knapsack i.e. the value of x_i for each object. This may be formalized as:

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

In order to use the dynamic programming technique [4, 5] we need to define a new function $f_k(j)$, the maximum profit obtained by using only the first k items subject to weight limit j:

$$f_k(j) = \max \{\sum_{i=1}^{k} p_i x_i : \sum_{i=1}^{k} w_i x_i \leq j, x_i \geq 0 \text{ integer}, i = 1, 2, \ldots, k\}, \tag{2}$$

where $0 \leq j \leq c$ and $1 \leq k \leq m$. This leads us to a recursive relation:

$$f_k(j) = \max \{f_{k-1}(j), f_k(j - w_k) + p_k\}. \tag{3}$$

The meaning of this recurrence is as follows: if $f_{k-1}(j)$ is the maximum then we do not add an object of type k to the knapsack; if, on the other hand, $f_k(j - w_k) + p_k$ is the maximum then we do add another object of type k, ie. we increment x_k. The optimal value of (1) $f_m(c)$, can be found in m stages by generating successively the functions $f_1, f_2, \ldots, f_m$ using equation (3) and the

initial conditions $f_0(j)=0, f_k(0)=0$ and $f_k(i)=-\infty$ for any k,j and i such that $1\le k\le m, 0\le j\le c$ and $i<0$. This approach solves simultaneously a set of knapsack problems with knapsack capacities $1,2,\ldots,c$.

The process of finding the values of $f_k(j), j=0,1,2,\ldots,c,\ k=1,2,\ldots,m$ is called the forward phase. However to determine each value of x_i in problem (1) we need to run a backtracking algorithm [4, 2]. In the course of the forward phase a pointer $u_k(j)$ is associated to any value $f_k(j)$ in such a way that $u_k(j)$ is the index of the last type of object used in $f_k(j)$. In other words if $u_k(j)=r$ this means $x_r\ge 1$, or the r-th object is used in $f_k(j)$ and $x_l=0$ for all $l>r$. The value $u_k(j)$ is used to keep the history of the forward dynamic programming phase. The boundary conditions for $u_k(j)$ are

$$\forall j: 0\le j\le c: u_1(j)=\begin{cases}0 & \text{if } f_1(j)=0\\ 1 & \text{if } f_1(j)\ne 0.\end{cases}$$

In general we set

$$u_k(j)=\begin{cases}k & \text{if } f_k(j-w_k)+p_k\ge f_{k-1}(j)\\ u_{k-1}(j) & \text{otherwise.}\end{cases} \tag{4}$$

for $\forall k: 1<k\le m$ and $\forall j: 0\le j\le c$. The values $u_k(j)$ are computed and propagated through the array simultaneously with the value $f_k(j)$. As shown in [4], definition (4) allows the solution vector in problem (1) $x^*\in\mathbf{N}^m$ (where by $\mathbf{N}$ we denote the set of natural numbers, i.e. $\mathbf{N}=\{0,1,2,\ldots\}$) to be found from the values of the function u_m. The backtracking algorithm is a sequential one and can be executed by the host computer [2].

3 Systolic Architecture

A systolic array consists of several identical Processing Elements (PEs) with identical nearest neighbour inter-connections between PEs throughout the array. These inter-connections can result in a linear (one dimensional) array or a rectangular (two dimensional) mesh. In a systolic system the data flows through the array in a rhythmic fashion in the way that blood circulates in the body. This rhythmic data flow is achieved through the use of pipeline data registers on all PE inter-connections.

The communication required for the execution of equation (3) can be described by means of directed graph, called the *dependence graph (DG)*.

Each node of the DG represents an operation performed by the algorithm and the arcs are used to represent data dependencies or communications. For example figure 1 (a) depicts the DG for a knapsack problem, with 3 objects ($m = 3$), a knapsack capacity of $c = 6$ and the following weights $w_1 = 4, w_2 = 3, w_3 = 2$. Each node represents one calculation, detailed in figure 1 (b). The peculiarity of this graph is that in any column the dependence vectors depend on the weights $w_i, i = 1, 2, \ldots, m$ which are input data for the problem. Such a dependency is not an *affine* dependency. This peculiarity makes the knapsack recurrence equation difficult to transform into a systolic array using the well-known synthesis methods [22, 23, 24].

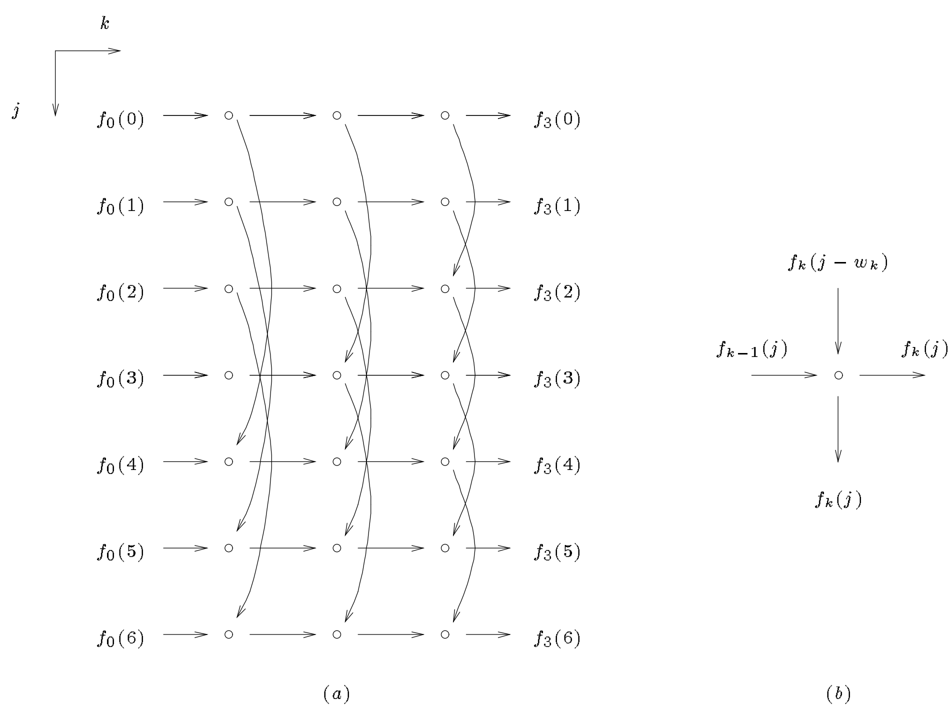


Figure 1: The dependence graph for the given example

However as identified in [3], at each stage the same operations are performed and data dependencies occur between adjacent stages and within the same stage only. Therefore using the dependence projection method [22], we

can map this DG onto a linear array of m identical PEs, $C_k, k = 1, 2, \ldots, m$. Each PE $C_k, 1 \leq k \leq m$, has a memory of size α, where α is a design parameter. The purpose of the memory is to save the values of the function f_k.

The design and operation of the systolic array PE is dependent on the parameter α and the size of the weights $w_k, k = 1, 2, \ldots, m$. We assume $\alpha \geq w_k$ for $\forall k, 1 \leq k \leq m$. Thus a straightforward projection of the dependence graph of figure 1 along the j axis provides a systolic array of m PEs.

Usually the dependence projection method [22] uses two functions, the Timing Function and the Allocation Function. However at this stage we have to take into account the peculiarity of the dependence graph for recurrence (3), where the calculations in each column are dependent on the weights w_k. In [2] a design based on the fact that in any processor only w_k memory locations are used was presented. Thus a new mapping is used called *Address function.*

Address Function: a mapping such that $addr(v)$ is the number of memory locations in processor $a(v)$ where the data v is stored. In [2] it is shown that an address function for the dependence graph of the knapsack problem is $\text{addr}(j, k) = j \bmod w_k$.

The operation of the basic PE during the forward phase is presented in figure 2. Each PE C_k stores the values p_k, w_k, k in registers p, w, k respectively. The data input into the leftmost PE during the first $c + 1$ instants t, $t = 0, 1, \ldots, c$ are $f_{in} = 0, u_{in} = 0$ and $j_{in} = t$. Note the inclusion of a boundary condition multiplexer where $f_{out} = f_{in}$ for $j_{in} < w$. This implements the initial condition where $j - w_k < 0 \Rightarrow f_k(j - w_k) = -\infty$.

If the number of PEs q in the linear array is less than the number of objects m then we need a memory queue to solve the Knapsack problem as reported in [2]. The resulting architecture is called Ring and is well known from systolic computation [25].

4 Systolic Array Design

Let us now consider the implementation of the linear array for solving the knapsack problem as a VLSI design. If we examine figure 2 we can identify four specific instructions carried out every systolic cycle that we need to implement as hardware in our PE design:

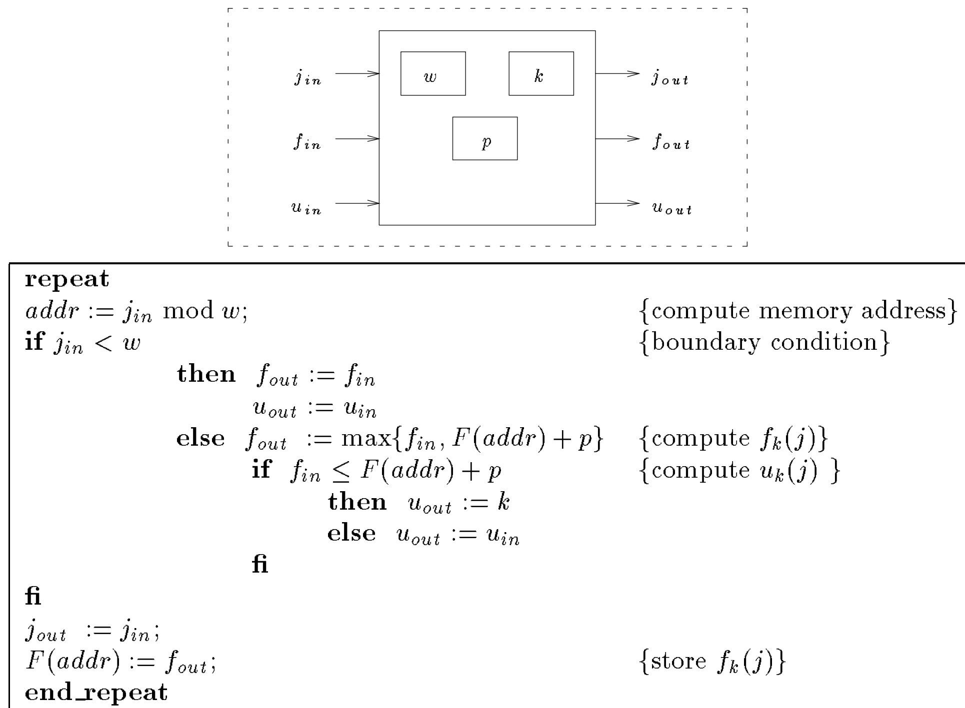


Figure 2: Knapsack Systolic PE and its Operation

calculate memory address:	$addr := j_{in} \bmod w$
memory read:	$F(addr)$
calculate maximum:	$\max\{f_{in}, F(addr) + p\}$
memory write:	$F(addr) := f_{out}$

Using a standard random access memory with single data and address ports we can overlap two of the PE operations, giving us the instruction pipeline illustrated in figure 3.

4.1 ROM PE Design

In order to calculate the memory address we need the function *mod*. This can be implemented as division, but this process slows down the throughput rate of the design. To maintain a high throughput a combinatorial implementation

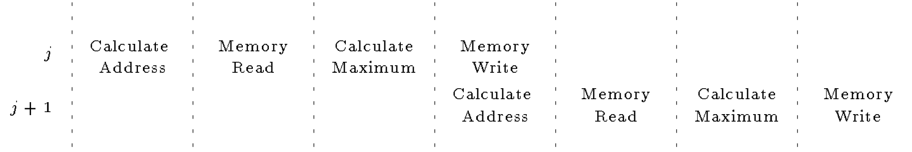


Figure 3: Instruction pipeline for Systolic PE

of *mod* is required. As a starting point, we choose a "naive" design using a look up table (Read Only Memory). In order to latch the ROM address a control signal me_{rom} (memory enable) is required.

Each PE requires a Random Access Memory of size α. A RAM will generally have two control signals $\overline{me}$ and $\overline{we}$. The $\overline{me}$ (Memory Enable) signal triggers the latching of the address and the transfer of data to or from the memory. It is double-edge sensitive: address and data are latched on alternate edges. The $\overline{we}$ (Write Enable) determines whether data is read from or written to the RAM.

The final hardware element in the PE is the ALU to compute $f_k(j)$. This can be achieved using adders for both the addition and the comparison (as a subtractor). A possible VLSI implementation of the basic PE for the Knapsack systolic array is illustrated in figure 4.

Figure 4 includes a systolic clock signal ϕ_1, which controls the data transfer latches between PEs, and three control signals $\overline{me}_{ram}, \overline{we}$ and me_{rom}. Given the timing requirements of the RAM and ROM we note that $\overline{me}_{ram} = \overline{\phi_1} = me_{rom}$. The clock for PE with $k = 0, 2, 4, \ldots$ is ϕ_1 and a non overlapping clock ϕ_2 is used for PEs with $k = 1, 3, 5, \ldots$ as illustrated in figure 5.

The operation of the PE is as follows:

1. On the rising edge of $\overline{\phi_1}$ the ROM address is latched and after a delay of T_{addr} the appropriate RAM address *addr* is produced.

2. On the falling edge of $\overline{\phi_1}$ the memory address *addr* is latched and after a delay T_{ram} the memory contents $F(addr)$ is available.

3. The ALU then performs the calculation and after a further delay of T_{calc} the result f_{out} is clocked to the next PE and written to memory.

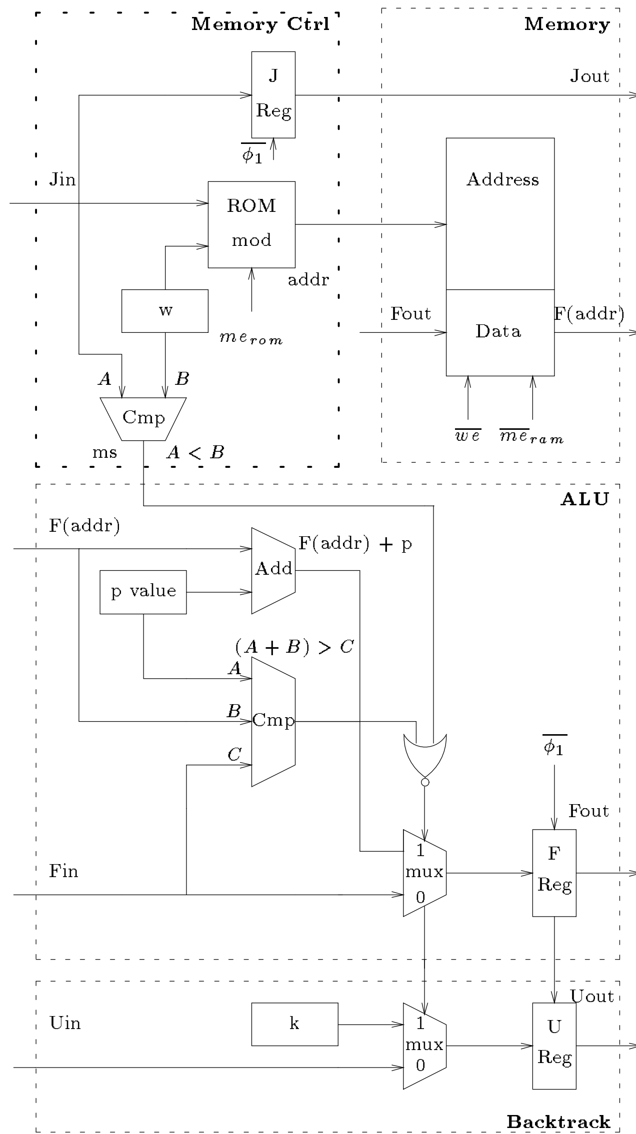


Figure 4: PE Design for Knapsack Systolic Array

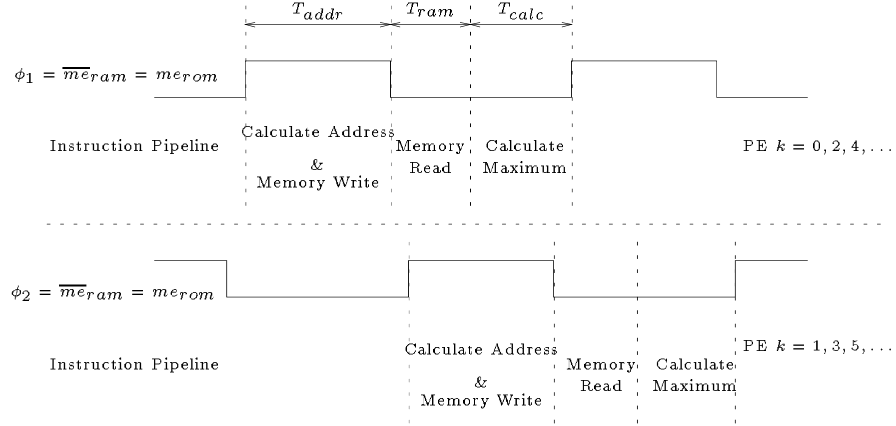


Figure 5: Control and Clock Signals for Knapsack Systolic Array

4.1.1 PE Area

We will now examine the relationship between the knapsack problem parameters and the silicon area of the PE. The parameters are: c the capacity of the knapsack, m the number of objects, $w_{min}(w_{max})$ the minimum(maximum) weight that any of the m objects can have, p_{max} the maximum profit that any of the m objects can have. The purpose of the knapsack array is to calculate the value of $f_m(c)$. An object with weight w_{min} can be placed c/w_{min} times in a knapsack of capacity c. If the object has profit p_{max} then the theoretical maximum value (f_{max}) of $f_m(c)$ in equation 2 is given by: $f_{max} = cp_{max}/w_{min}$.

From figure 4 we can identify four different modules used in the knapsack PE: memory control, memory, ALU and backtracking. Five sub-circuits are used in the design: storage and data registers; adders; multiplexers; ROM and RAM. We can associate with each of these sub-circuits a parameter or constant to represent the area of a one bit instance. For example the area of a single bit adder we call A_{fa}. Likewise a one bit register, a one bit multiplexer, a one bit ROM and a one bit RAM we call respectively A_{latch}, A_{mux}, A_{rom} and A_{ram}. The area of each module in figure 4 can be related to the knapsack parameters as illustrated in table 1.

Module	Area
memory control	$A_{latch}\log(c) + A_{latch}\log(f_{max}) + A_{fa}\log(c)$
	$+A_{rom}w_{max}c\log(w_{max})$
memory	$A_{ram}w_{max}\log(f_{max})$
ALU	$A_{latch}\log(f_{max}) + A_{latch}\log(p_{max}) + 3A_{fa}\log(f_{max})$
	$+A_{mux}\log(f_{max})$
backtracking	$2A_{latch}\log(m) + A_{mux}\log(m)$

Table 1: Circuit Area of Knapsack PE

4.1.2 PE Timing

The clock speed of the PE is determined by the timing through each stage of the pipeline as illustrated in figure 5. As with the area we can relate the timing to the knapsack parameters as illustrated in table 2. Again we base the timing on the bit level where t_{rom} and t_{ram} are the access times of a one bit ROM and RAM and where t_{fa} is the ripple of the carry in a full adder.

Module		Time
memory control	T_{addr}	$t_{rom}\sqrt{w_{max}c\log(w_{max})}$
memory	T_{ram}	$t_{ram}\sqrt{w_{max}\log(f_{max})}$
ALU	T_{calc}	$t_{fa}\log(f_{max})$

Table 2: Circuit Timing of Knapsack PE

4.2 Modified Design

The dominant term in table 1 is the area of ROM which has a complexity of $\Theta(cw_{max}\log(w_{max}))$. Thus to implement ($a$ mod b) is a costly (in terms of area) and unrealistic procedure when using a look-up table. However we note that if b is a power of two ie. $b = 2^x$, then the implementation of the modulo operator is significantly simplified and we further note that α is a power of

two. Thus we present a new address function for use in the projection of the dependence graph into a linear array.

4.2.1 New Address Function

In section 3 we describe the use of the address function in the mapping of the non-affine data dependence graph into a linear systolic array. This resulted in the need to implement digitally the mod operator using a ROM. Therefore we transform the address function $addr(j,k) = j \bmod w_k$ into memory read/write address functions:

- $adread(j,k) = j \bmod w_k = j \bmod \alpha$
  $adwrite(j,k) = (j + w_k) \bmod \alpha$,

  where α is the memory size and $\alpha \geq w_{max}$ is satisfied.

These two address functions implement the recurrence equation (3) as in this case we use two address pointers, one ($adread(j,k)$) to read from the memory the value $f_k(j - w_k)$, the other ($adwrite(j,k)$) to write the value $f_k(j)$. These two memory locations are separated by w_k. This new address function results in the basic PE operation illustrated in figure 6.

4.2.2 VLSI Design

A possible VLSI implementation of this algorithm contains the memory, ALU and backtracking modules of figure 4, but the memory control module is that illustrated in figure 7. The ROM is replace by an adder and multiplexer and we can achieve the (modα) just by choosing the $\log(\alpha)$ least significant bits of the multiplexer output.

This new algorithm requires a memory read and memory write operation to different locations. Two addresses have to be latched into the memory address register by the $\overline{me}$ signal for each systolic cycle. This results in the instruction pipeline and clock signals illustrated in figure 8. The systolic clock is ϕ_1 for PE $k = 0, 2, 4, \ldots$ and ϕ_2 for PE $k = 1, 3, 5, \ldots$.

4.2.3 PE Area

The area for the ROM is removed and replaced by an adder of area $A_{fa} \log(c)$ and a multiplexer of area $A_{mux} \log(w_{max})$. The overall design area for the modified memory control module is given in table 3.

```
repeat
adread := j_in mod α;                              {compute memory address}
adwrite := (adread + w) mod α;
if j_in < w                                        {boundary condition}
          then  f_out := f_in
                u_out := u_in
          else  f_out := max{f_in, F(adread) + p}  {compute f_k(j)}
                if  f_in ≤ F(adread) + p           {compute u_k(j) }
                     then  u_out := k
                     else  u_out := u_in
                fi
fi
j_out := j_in;
F(adwrite) := f_out;                               {store f_k(j)}
end_repeat
```

Figure 6: The modified algorithm

Module	Area
memory control	$A_{latch}\log(c) + A_{latch}\log(w_{max}) + 2A_{fa}\log(c)$
	$+A_{mux}\log(w_{max})$
memory	same as table 1
ALU	same as table 1
backtracking	same as table 1

Table 3: Design Area for Modified Algorithm

4.2.4 PE Timing

The clock period is defined by each stage in the instruction pipeline. The memory access time T_{ram} and the ALU delay T_{calc} are the same as the previous design (see table 2). The relationship between the knapsack parameters and the time taken to calculate the address T_{addr} and the memory write set-up time T_{write} are given in table 4.

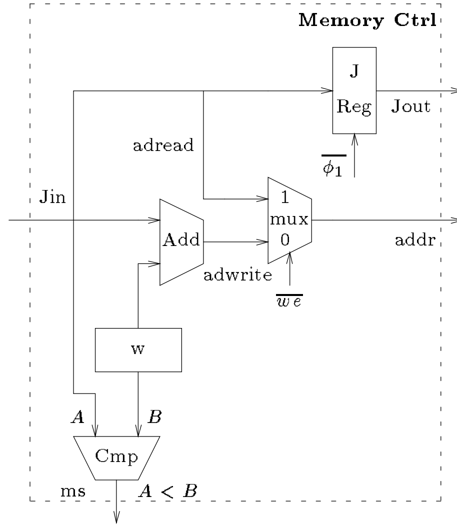


Figure 7: Memory Control for Modified Knapsack Algorithm

module		Time
memory control	T_{addr}	$t_{fa}\log(c)$
memory	T_{ram}	same as table 2
	T_{write}	t_{write} (a constant)
ALU	T_{calc}	same as table 2

Table 4: Circuit Timing of Modified PE

4.3 Counter Design

There is another possible implementation of the address function $addr(j,k) = j \bmod w_k$ without the area penalty of the ROM. We note that at each systolic time step a new value of f_k is entered into a PE. Moreover the data $f_k(j), j = 0, 1, \ldots c$ enters a PE in exactly this order. Therefore we do not need to propagate the argument j and we can generate the memory address by using a re-setable counter. This counter is incremented each time step when a new value of f_k is entered and the counter is reset to 0 after w_k time steps. Thus

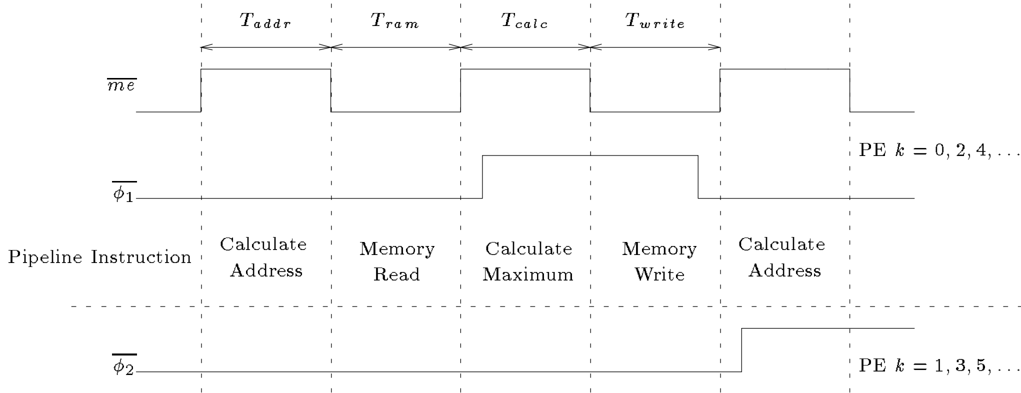


Figure 8: Clocking Scheme and Pipeline for Modified Algorithm

if we write $addr(j, k) = count$, we see that this is another implementation of the address function $addr(j, k) = j \bmod w_k$. The operation of this counter based PE is illustrated in figure 9.

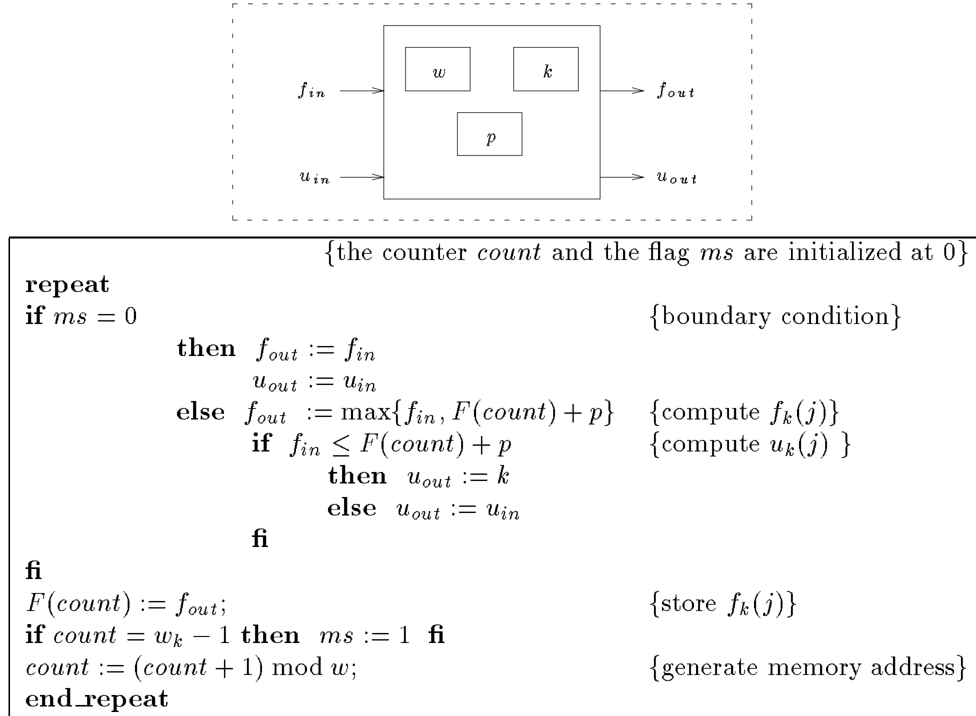
The timing diagram for this design is similar to that of figure 5. A possible VLSI implementation of memory control is illustrated in figure 10. Note the state machine used to determine first w_k values of f_{out}, for the dependence graph boundary condition.

4.3.1 PE Area

In this design we have eliminated the j_{in} data register. However we need two adders of area $A_{fa} \log(w_{max})$, two registers area $A_{latch} \log(w_{max})$ and a set of And gates to reset the counter of area $A_{and} \log(w_{max})$. The design area in terms of the knapsack parameters is given in table 5.

4.3.2 PE Timing

The clock speed is determined by the pipeline delays T_{addr}, T_{ram} and T_{calc} of figure 5. The values of the RAM access time T_{ram} and the ALU calculation time T_{calc} are those given in table 2. The relation between T_{addr} and the knapsack parameters are given in table 6.




```
                       {the counter count and the flag ms are initialized at 0}
repeat
if ms = 0                                             {boundary condition}
        then  f_out := f_in
              u_out := u_in
        else  f_out := max{f_in, F(count) + p}        {compute f_k(j)}
              if  f_in ≤ F(count) + p                 {compute u_k(j) }
                   then  u_out := k
                   else  u_out := u_in
              fi
fi
F(count) := f_out;                                    {store f_k(j)}
if count = w_k − 1 then  ms := 1  fi
count := (count + 1) mod w;                           {generate memory address}
end_repeat
```

Figure 9: Systolic PE for Counter Based Knapsack solution

5 Design Comparison

The first point to note is that the area of the ROM in the naive design depends linearly up to a constant on $cw_{max}\log(w_{max})$. This design is not realizable as the size of ROM necessary cannot be fabricated. This design is included to illustrate the problems encountered if we wish to implement the *mod* function as a combinational circuit in order to maximize throughput. A similar problem will be encountered if the alternative of a PLA is used. However for applications with small values of the product cw_{max}, the ROM/PLA option might be considered due to the ease of design using ROM/PLA generators.

The dominant area term in the modified and counter based designs is the area of the memory as this is linearly dependent on w_{max}. This corroborates our assumption that the size of the memory is sufficiently large ($\alpha \geq w_{max}$).

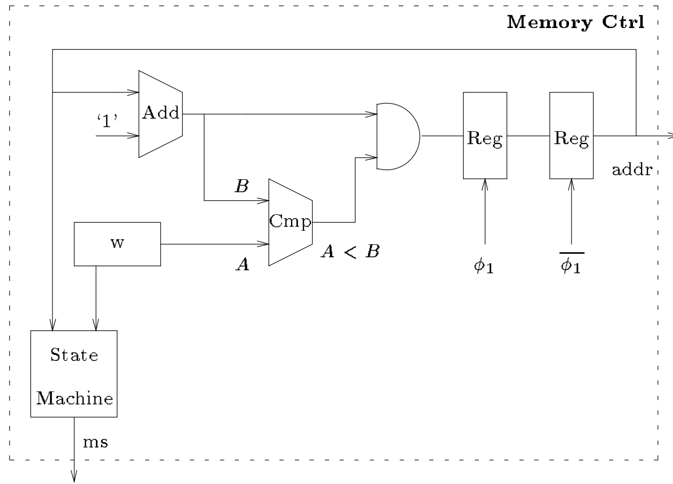


Figure 10: Memory Control for Counter Based Knapsack Algorithm

Module	Area
memory control	$2A_{latch}\log(w_{max}) + 2A_{fa}\log(w_{max}) + A_{and}\log(w_{max})$
memory	same as table 1
ALU	same as table 1
backtracking	same as table 1

Table 5: Design Data for Counter Based Design

We have related the area/time to the knapsack parameters. We wish to emphasize the dominance of the knapsack capacity c in these parameters. In fact it is this term that causes the problem to be NP - *complete* [9]. The relationship between the knapsack parameters and the area/time of the memory, ALU and backtracking modules is fixed as shown in table 7. Here we can see the influence of c. We cannot improve on the design of these three modules, because the dependence of the area/time on c is inherent to the algorithm.

module		Time
memory control	T_{addr}	$2t_{fa}\log(w_{max})$
memory	T_{ram}	same as table 2
ALU	T_{calc}	same as table 2

Table 6: Circuit Timing of Counter based PE

Module	Area	Time
memory	$\Theta(w_{max}\log(c))$	$\Theta(\sqrt{w_{max}\log(c)})$
ALU	$\Theta(\log(c))$	$\Theta(\log(c))$
backtracking	$\Theta(\log(m))$	No influence on timing

Table 7: Dominant Area/Time Variables for Knapsack PE Modules

Let us now consider the remaining module, the memory control. In the ROM design the area dependence is $\Theta(cw_{max}\log(w_{max}))$, while the time dependence is $\Theta(\sqrt{\log(c)+\log(w_{max})})$. Due to the modification proposed to the address function in section 4.2 we get $\Theta(\log(c)+\log(w_{max}))$ and $\Theta(\log(c))$ for area and time respectively. In the final design, counter based, the memory control signal j is not required, eliminating the area of $\log(c)$ routing wires. We also get a complexity of $\Theta(\log(w_{max}))$ for both area and time, thereby eliminating completely the dependence on c in the memory control module.

6 Conclusion

Usually the largest value in the knapsack problem is the capacity c, which provides the main obstacle to an area efficient VLSI implementation. In this paper we highlighted the influence of c and the largest weight coefficient w_{max} on the PE area of three different implementations. We show that the area and time complexity of the memory control module of the first design is $\Theta(cw_{max}\log(w_{max}))$ and $\Theta(\sqrt{\log(c)+\log(w_{max})})$ respectively. We improve

this in the second design where we reduce this dependence to $\Theta(\log(c) + \log(w_{max}))$ and $\Theta(\log(c))$. Finally in the third design we obtain a complexity of $\Theta(\log(w_{max}))$ for both area and time. In this way we have proved that the influence of c can be eliminated completely from the memory control module. Theoretically this is optimal, under the assumption $\alpha \geq w_{max}$, since obviously the other parts of the circuit (the ALU and memory) are inherently dependent of c in order to solve the problem. Thus by 'engineering' the algorithm, we reduce the influence of c on the circuit area to the theoretical minimum and replace it by the influence of w_{max}. This is of crucial importance for an realistic and efficient VLSI implementation of this problem since usually $w_{max} \ll c$.

Acknowledgments

The authors would like to thank Eric Gautrin, Dominique Lavenier, and Doran Wilde for many helpful discussions.

References

[1] S. Teng, "Adaptive parallel algorithm for integral knapsack problems," *Journal of Parallel and Distributed Computing*, vol. 8, pp. 400–406, 1990.

[2] 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.

[3] G. Chen, M. Chern, and J. Jang, "Pipeline architectures for dynamic programming algorithms," *Parallel Computing*, vol. 13, pp. 111–117, 1990.

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

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

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

[7] J.P.Barthélemy, G.Cohen, and A.Lobstein, *Complexité Algorithmique.* Masson, 1992.

[8] D. E. R. Denning, *Cryptography and Data Security.* Addison-Wesley, 1982.

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

[10] C. Chung, M. S. Hung, and W. O. Rom, "A hard knapsack problem," *Naval Research Logistics*, vol. 35, pp. 85–98, 1988.

[11] V. Chvatal, "Hard knapsack problems," *Operations Research*, vol. 28, pp. 1402–1411, 1980.

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

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

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

[15] L.Guibas, H. Kung, and C. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. 1979 Caltech. Conf. on VLSI*, pp. 509–525, 1979.

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

[17] 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.

[18] J. 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.

[19] G. Li and B. W. Wah, "Systolic processing for dynamic programming problems," in *Proc. International Conference on Parallel Processing*, pp. 434–441, 1985.

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

[21] R. Andonov and S. Rajopadhye, "An optimal algo-tech-cuit for the knapsack problem," in *Proc. International Conference on Application-Specific Array Processors - ASAP'93*, 1993. Venice, Italy.

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

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

[24] G. M. Megson, "Mapping a class of run-time dependencies onto regular arrays," in *International Parallel Processing Symposium*, (Newport Beach, CA), pp. 97–107, IEEE, April 1993.

[25] H. T. Kung, "Why systolic architectures," *Computer*, vol. 15, pp. 37–46, 1982.

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