Hardware Solution of a First-Order Diophantine Equation

Ioannis Panagopoulos, Christos Pavlatos, Alexandros Dimopoulos and George Papakonstantinou

***Abstract*-- In this paper we present the theoretical framework for enumerating the solutions of the first order Diophantine equation with unitary coefficients that helps us to define the sets of points of a nested loop that can be executed in parallel. The analytical expression for finding those points is: $i_1 + i_2 + \ldots + i_D = c$ where $i_1, i_2, \ldots, i_D$, $0 \leq i_p \leq L_p$, $\forall p = 1 \ldots D$ are the loop indices, L_p is the loop bound and c defines the time all points satisfying this equation will be executed in parallel. Moreover, we present an innovative "refined" algorithm which speeds up the generation of those solutions compared to the traditional 'brute-force" approach. Finally, we present a modular hardware implementation of this "refined" algorithm on FPGA platforms, an approach which increases even more the algorithm's performance. The presented architecture and theoretical solution is suitable in load balancing applications, consisting of nested for-loops with dependencies, since it allows rapid and dynamic generation of the index points of loop instances that can be executed in parallel. Moreover, this architecture can be easily reconfigured.**

***Index Terms*—FPGA Design, Diophantine equation**

I. Introduction

The platform based design methodology has been proven to be an effective approach for reducing the computational complexity involved in the design process of embedded systems [1]. Reconfigurable platforms consist of several programmable components (microprocessors) interconnected to hardware and reconfigurable components. Reconfigurable components allow the flexibility of selecting specific computationally intensive parts (mainly nested loops) of the initial application to be implemented in hardware, during hardware/software partitioning [2], in the effort of achieving the best possible increase in performance, while obeying specific area, cost and power consumption design constraints.

The most straightforward approach of speeding up the execution of nested loops is realized by the implementation of the nested loop in software using parallel or distributed computer systems [5][6][7][8][9], or in hardware either on an FPGA or as a dedicated ASIC (Application Specific Integrated Circuit) [3][4].

The proposed in this work, framework for enumerating and finding the solutions of the first order Diophantine equation is the core of an innovative dynamic scheduling algorithm implemented on FPGA for fine grain parallelism of nested loops. To the extent of our knowledge this is the first hardware implementation of a dynamic scheduling algorithm which handles data dependencies. We propose the theoretical analysis that enables finding the amount of solutions of the Diophantine equation, a refined algorithm in software, which enumerates them and a modular hardware implementation of this algorithm on FPGAs. The theoretical analysis, in contrast to ref. [10], uses only additions and multiplications which makes it suitable for execution on embedded systems. The refined algorithm avoids the generation of additional points which are not solutions to the equation (a problem encountered on the traditional "Brute Force" approach) reducing the computational steps required by 87%. Finally, the hardware implementation can be easily

Manuscript received February 9, 2007. This work is co - funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS II. Authors are with the Computer Systems Laboratory of the National Technical University of Athens, Dept. of Electrical and Computer Engineering, Zografou 15773, Athens, Greece (e-mail: {ioannis, pavlatos, alexdem, papakon}@cslab.ece.ntua.gr).

reconfigured to handle the enumeration of solutions for Diophantine equation with different number of *c* and *D* parameters (see equation 1).

The rest of the paper is organized as follows. In Section II we present the theoretical analysis for finding the solutions to equation (1). In Section III, we present the refined algorithm that enumerates the solutions of the equation. In Section IV, we present the hardware implementation of the solver. In Section V, we present as a case study, a general "Load Balancing" architecture where this implementation is successfully applied. Finally in Section VI, we summarize the proposed implementation and present future work.

II. Analytical Formula For The Number Of Solutions

This section helps establishing the meaning for the symbols and definitions that will be used throughout this paper. We define loop instances, establish the problem of parallelizing loop instances which are dependent to each other, pinpoint the compromise in parallelization we make for the sake of generality and simplicity and define the notation of Processing Elements that will handle the execution of loop instances.

An example of a nested loop is illustrated in Listing 1.

```
for (I1=0;I1<=4;I1++)
  for (I2=0;I2<=3;I2++)
    S1: a(I1,I2)= a(I1-1,I2)+ a(I1,I2-1)+ a(I1-1,I2-1)
  end
end
```

Listing 1 Genreal abstract representation of a perfectly nested loop.

where I_1 and I_2 are the loop indices, 4 and 3 are the loop bounds and S1 is a set of statements. Statements may be of any kind from conditional branches to I/O operations. All statements within the loop body are executed sequentially.

A loop instance is defined as a single execution of the statements in the loop's body for specific index values. Therefore, every loop instance can be uniquely identified by the values of the indices of the for-loop at the time it is executed. Statements within a loop instance can then be identified by the values of the indices of the loop instance they belong to. Statements assign values to variables. Statements which use values of variables that have been calculated in previous loop instances impose data dependencies which limit the parallelization potential for the for-loop. In the previous example such dependencies are represented in Fig. 1.

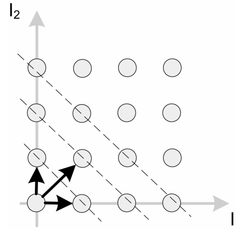


Fig. 1. A 2 dimensional loop with unitary dependencies

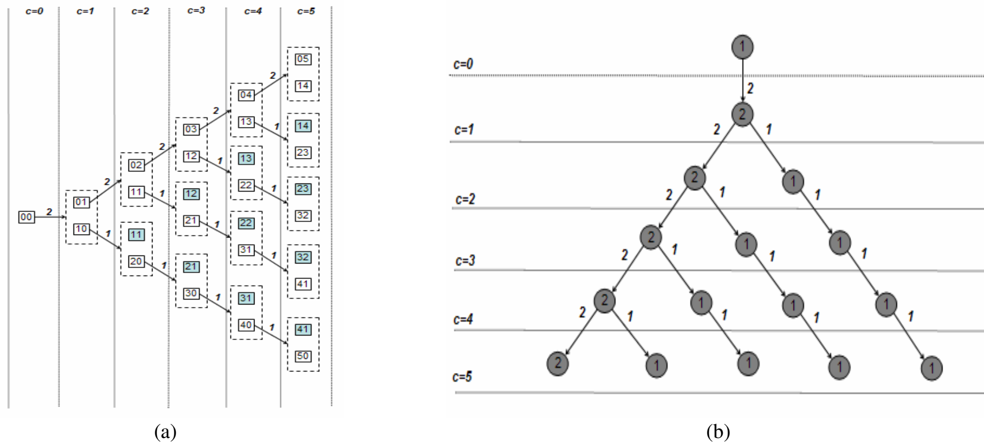
We can represent the loop instances in an n-dimensional space (for an n dimensional loop) where each coordinate is a specific index of the for-loop. Each point in the space represents a specific loop instance. We use arrows between points (loop instances) in the graph to express data dependencies. For the previous example the 2D index space is shown in Fig. 1.

The extracted dependencies among loop instances are used to calculate sets of points that can be executed in parallel. Those sets of points lay on hyperplanes. The hyperplanes formed for the previous example are illustrated in Fig.1 as dashed lines. Obviously all index points on a

hyperplane can be executed in parallel. Moreover, the hyperplanes satisfy the equation (1). Hence, we must find all points on a hyperplane, i.e. the solutions of (1).
We next provide the proof to the following problem. Given a first order Diophantine equation (1) with unitary coefficients, find out the number of solutions that it has for a given c and D.

$$i_1 + i_2 + ... + i_D = c \qquad (1)$$

We define the function $f_c(D)$ as the function that returns the desired result for a given c and D. We first consider the case where $D=1$. It is obvious that $f_c(1) = 1, \forall c$. We then construct the tree of solutions for $D=2$ and recursively from level i we construct level i+1. The resulting solution tree is illustrated in Fig. 2.a.The nodes of the tree represent solutions of equation 1. In each node there are two numbers, the first number represents the value of index i_1 (first dimension), while the second represents the value of index i_2 (second dimension). Furthermore, we label the edges of the tree with the number of solutions they generate as shown in Fig 2.a. Whenever a solution has been generated already this solution appears shaded in the tree, in order not to be counted many times. We also enclose the solutions generated in one step of the algorithm from a single solution from the previous step in dashed rectangles e.g at level c=2 the solutions 02, 11, that both come from solution 01 at level c=1, are enclosed in dashed rectangle. While recursively from level i we construct level i+1, we first increment the second dimension and then the first. From the tree shown in Fig. 2.a, we construct the tree illustrated in Fig. 2.b where nodes correspond to the dashed rectangles and edges to the edges of the tree in Fig. 2.a Nodes are labelled with a number denoting the number of solutions they represent.



(a) (b)

Fig. 2 (a) Solution tree of the equation for D=2 for various c (b) Tree representing the number of solutions in each node.

It can be proved that every node with 2 solutions from the tree in Fig. 2.b, generates two nodes one with two solutions and one with one solution. We also note the recursion in the generated tree which is illustrated in Fig. 3, that is every tree resulting from a node with 2 solutions, leads to a new tree starting from a node with two solutions and a new tree starting from a node with one solution.

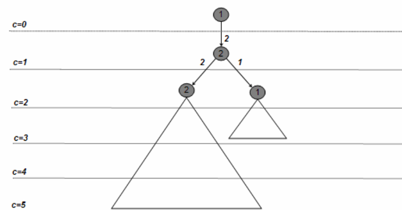


Fig. 3 Tree recursion

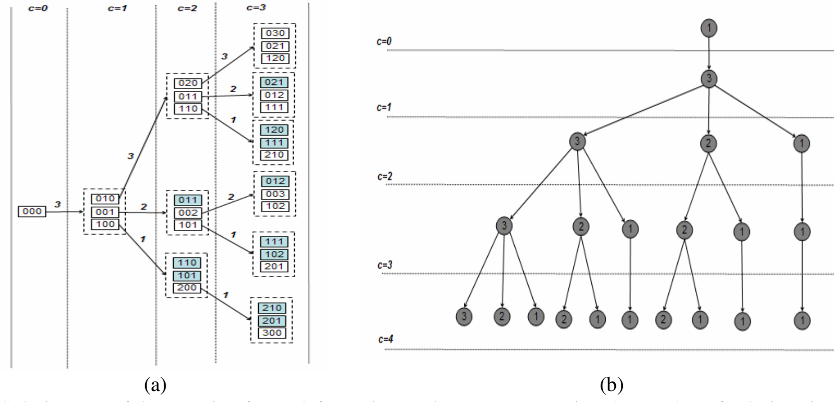
Therefore we can define $f_c(2)$ recursively as follows:

$$\begin{aligned} &f_1(2) = 2 \\ &f_2(2) = f_1(2) + f_1(1) \\ &f_3(2) = f_2(2) + f_2(1) \\ &\ldots \\ &f_c(2) = f_{c-1}(2) + f_{c-1}(1) \end{aligned} \qquad (2)$$

$$\begin{aligned} &f_c(2) = 2 + \sum_{i=1}^{c-1} f_i(1) \Rightarrow \\ &f_c(2) = 2 + (c-1) \end{aligned} \qquad (3)$$

Adding the equations in (2) we get equation (3)

We follow the same approach for *D=3* (Fig. 4a, 4b).



(a) (b)

Fig. 4 (a) Solution tree of the equation for D=3 for various c (b) Tree representing the number of solutions in each node.

The same observations also apply here and therefore we can calculate $f_c(3)$ recursively again as follows:

$$\begin{aligned} &f_1(3) = 3 \\ &f_2(3) = f_1(3) + f_1(2) + f_1(1) \\ &f_3(3) = f_2(3) + f_2(2) + f_2(1) \\ &\ldots \\ &f_c(3) = f_{c-1}(3) + f_{c-1}(2) + f_{c-1}(1) \end{aligned} \qquad (4)$$

$$\begin{aligned} &f_c(3) = 3 + \sum_{i=1}^{c-1} f_i(2) + \sum_{i=1}^{c-1} f_i(1) \Rightarrow \\ &f_c(3) = 3 + \sum_{i=1}^{c-1}\big(2 + (i-1)\big) + (c-1) \Rightarrow \\ &f_c(3) = 3 + \sum_{i=1}^{c-1}\big(1 + i\big) + (c-1) \Rightarrow \\ &f_c(3) = 3 + \frac{(4+c)(c-1)}{2} \end{aligned} \qquad (5)$$

Adding the equations in (4) we get equation (5). For the general case of D we get (6) and adding the equations in (6) we get equation (7). Equation (7) for a given *D, c* yields the total amount of solutions of the Diophantine equation using only multiplications and additions.

$$\begin{aligned} &f_1(D) = D \\ &f_2(D) = f_1(D) + f_1(D-1) + \ldots + f_1(1) \\ &f_3(D) = f_2(D) + f_2(D) + \ldots + f_2(1) \\ &\ldots \\ &f_c(D) = f_{c-1}(D) + f_{c-1}(D-1) + \ldots + f_{c-1}(1) \end{aligned} \qquad (6)$$

$$\begin{aligned} &f_c(D) = D + \sum_{i=1}^{c-1} f_i(D-1) + \sum_{i=1}^{c-1} f_i(D-2) + \ldots + \sum_{i=1}^{c-1} f_i(1) \Rightarrow \\ &f_c(D) = D + \sum_{i=1}^{D-1}\sum_{j=1}^{c-1} f_j(i) \end{aligned} \qquad (7)$$

III. The "Refined Algorithm" For Finding The Solutions

The "I-module" (fig. 5) architecture used for the implementation of the "Point Generator" is based on a refinement of a brute force algorithm for finding and enumerating the solutions to Equation (1) (first-order Diophantine equation with unitary coefficients).

```
for (I[0]=0;I[0]<=L[0];I[0]++)
 for (I[1]=0;I[1]<=L[1];I[1]++)
   …
 for (I[D-1]=0;I[D-1]<=L[D-1];I[D-1]++)
  CheckSolution();

void CheckSolution ()
{
   if (I[0]+I[1]+…+I[D-1])==c
      /* A solution has been found */
}
```

Listing 2 The initial brute force algorithm based on for-loops

```
void FindSolution (int pos)
{
  for (i=0;i<=L[pos];i++) {
    I[pos]=i;
    if (pos==D-1) CheckSolution();
    else FindSolution(pos+1);
  }
}
```

Listing 3 Brute Force algorithm for finding and enumerating the solution of the first order Diophantine equation with unitary coefficients (pos holds the current position in the indices vector)

The trivial approach is to implement a brute force algorithm that iterates through all possible values of the indices and generates the solutions (Listing 2). L[i], i=0...D-1, holds the upper bound for the indices in each dimension. Array I[…] is used for temporal storage of a candidate solution. An alternative Brute Force algorithm is based on a recursive function whose template is shown in Listing 3 and where pos holds the current position in the indices vector. This algorithm is more suitable for its implementation in hardware, since it requires the design of only one hardware module (the `FindSolution()` function) which is replicated identically, depending on the size of the initial vector. The function `CheckSolution()` evaluates the current generated vector and stores the result if the sum of its index values equals to c.

In the effort of improving the performance of the Brute Force algorithm by avoiding the generation of vectors that are not solutions to the equations we observe the following:

1. At a specific index position k, for a solution to exist, the following equation should hold: $i_k + i_{k+1} + ... + i_D = c - i_1 - i_2 - ... - i_{k-1} = r$ .The maximum value that an index can take is its bound. Therefore if $L_k + L_{k+1} + ... + L_D < r$ there is no possibility that a solution will be found.
2. If the value of r (see observation 1) is negative there is no way a solution will be found.
3. At a specific index position k, and for all $i_n = 0, n \in [k+1, D]$ the maximum value that can be stored and can possibly lead to a solution is *min (r, L_k)*
4. If we are located at the last index point, its value for a solution will be r (see observation 1) provided that $L_k + L_{k+1} + ... + L_D < r$ as imposed by Observation 1.

Based on those observations, the proposed, refined algorithm is given in Listing 4. The function `RemSumL(k)` returns the value of: $L_k + L_{k+1} + ... + L_D$ and the initial value of pos is 0 and of r is c.

```
void FindSolution (int pos,int r)
{
  if (pos==(N-1))     //Based on observation 4
  {
    i[pos]=r;    //Based on observation 4
    CheckSolution();
    return;
        }
    For (i=0;i<=Min(L[pos],r);i++)  //Based on observation 3
  {
    I[pos]=i;
    if ((r-i)<=RemSumL(pos+1)) /*Obs 1*/ && ((r-i)>=0) /*Obs 2*/
                      FindSolution(pos+1,r-i);
  }
}
```

Listing 4 Proposed refinement of the Brute Force algorithm for finding and enumerating the solutions of the first order Diophantine equation with unitary coefficients.

This proposed refined algorithm will be used for the generation of the index values of loop

instances to be executed in parallel in the proposed implementation. This algorithm gives an average saving of 87% on the computation steps required compared to the algorithm of Listing 2.

IV. Hardware Implementation Of The Algorithm

We name the proposed implementation as the "Point Generator". To preserve the generality of the approach, the "Point Generator" is implemented in hardware as a line of smaller interconnected modules defined as I-modules. All I-modules are instances of the same I-component that is described in HDL (Hardware Design Language) and specifically in Verilog [11]. The number of I-modules is equal to D and each one corresponds to an index.

In practise, the I-module is described by the use of a Finite State Machine (FSM) which generates an index value participating in a single solution of equation (1) and passes the control either to the next or the previous I-module in line depending on its current state. An I-module, after executing the necessary, in each step computations, may enable, by setting to value 1 the appropriate signal (EnRight, EnLeft signals in Figure 5), the I-module left or right to it. This behaviour imitates the execution of Listing 4 where the function `FindSolution()` either recursively calls a new function or returns (equivalently going right or left in the I-modules). Additionally, when an I-module enables another, it also transmits to the other a bit vector that represents the current value of the variable r.

If control is taken by the last I-module then each I-module passes its current index value to the CheckSolution-Module and respectively the function `CheckSolution()` is executed. CheckSolution-Module adds the current values of points $i_1 ... i_D$ and if the result is equal to c , it stores this correct set of points to a register.

The implementation of the "Point Generator" component, as a series of interconnected modules, guarantees the generation of a new solution in at most D stages (where D is the index vector dimension). Moreover, it can be easily extended for the solution of equations with higher dimensions by adding more I-modules in line (Figure 5).

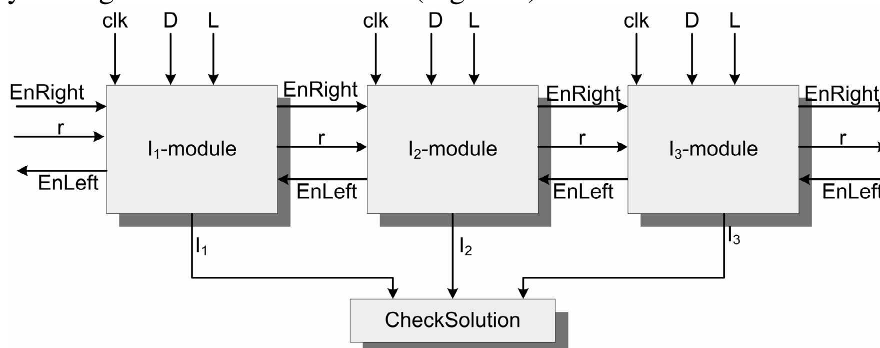


Fig. 5. An interconnected series of I-modules generating the solutions of equation 1 for a 3D vector. Note that additional I-modules can be added at the end of the series to solve the equation for higher dimensional vectors.

Finally, an automated tool (available at http//:www.cslab.ntua.gr/~pavlatos) has been implemented, that taking as parameters *D, c* automatically generates the abovementioned architecture (i.e. the Verilog code which can be downloaded to the FPGA). The generated source has been simulated for validation, synthesized and tested on a Xilinx FPGA (Field Programmable Gate Array) board [12].

The proposed architecture has been tested on a considerable number of applications. Some of the experimental results are presented in Fig.6 for 2D and 3D vector of indices. Obviously, the hardware implementation outperforms all the software solutions (Listings 2, 3 and 4) while the

proposed refinement Brute Force algorithm (Listing 4) surpasses the conventional Brute Force algorithm (Listing 2). The hardware implementation succeeds a speed-up factor of approximately 7.8 compared to the algorithm of Listing 4, independently of the c, D parameters while the proposed the algorithm of Listing 4 succeeds a speed-up factor of 3 orders of magnitude compared to the algorithms of Listing 2 and 3. Taking all the above into consideration, we conclude that the hardware implementation of the proposed refinement Brute Force algorithm achieves a speed up of approximately 4 orders of magnitude.

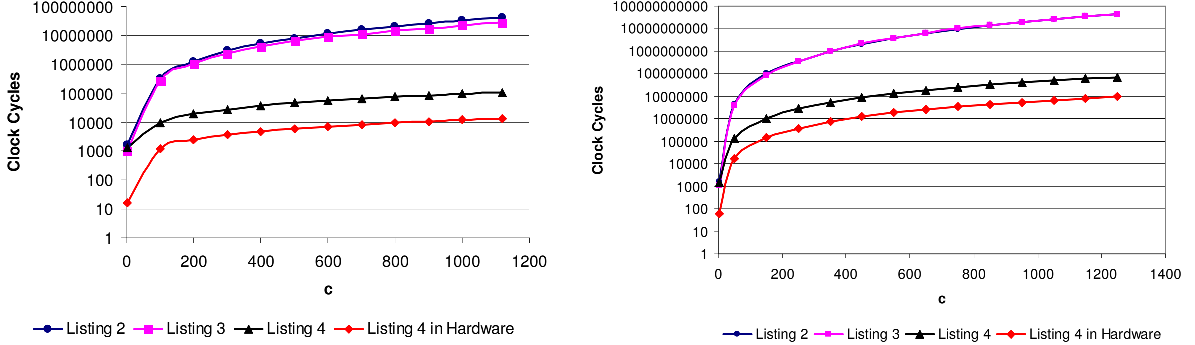


Fig. 6(a) Experimental results for D=2 (b) Experimental results for D=3.

V. LOAD BALANCING PLATFORM

The general architecture [13] of the load balancing design is illustrated in Figure 6a.

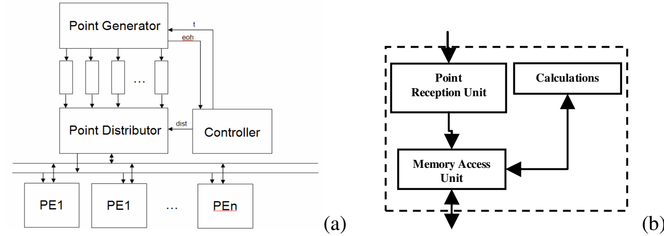


Fig. 7(a) The system's architecture (b) Internal architecture of the PEs.

The pacemaker of the architecture is the “Controller”. The “Controller” component is initiated by the CPU and is responsible for dispatching the current parallel time of execution. The parallel time is defined as the time in which loop instances that belong to a specific hyperplane (Fig.1) can be executed in parallel. The “Controller” component dispatches this time to the “Point Generator Component”. The “Point Generator” is initiated, finds the solutions of equation (1) for the time c, and stores the solution's index values to its interfacing FIFO buffers. The “Point Distributor” Component receives those index values and sends them to the requesting PEs (Processing Elements). PEs receive the index values of the loop instances to be processed, fetch from memory the requested data, perform the loop instance's calculations and store the results to memory. Upon completion of the generation of the index values for a specific parallel set, the “Point Generator” component informs the controller which waits until all PEs have performed their computations and then dispatches the next time unit. This stalling of the “Controller” is crucial to preserve violation of loop dependencies in the case where some PEs are still calculating loop instances of the previous time step. The PEs internal architecture for the execution of the loop instance's statements is presented in Figure 7(b). The “Point Reception Unit” implements the communication protocol with the “Point Distributor Component”. The received index values are used as an offset of a base address by the “Memory Access Unit”

component which fetches the required data from main memory. The "Calculations" component is then initiated that performs the loop instance's calculations. The results are stored to the main memory through the Memory Access Unit. Reconfiguration of the proposed architecture during runtime can occur during the execution of a specific loop for a further speed-up in performance with a cost in area by adding/removing "I-modules", without any further change in the architecture.

VI. CONCLUSION

In this paper we have presented a theoretical framework for finding the amount of solutions to the first order Diophantine equation with unitary coefficients, a refined algorithm for enumerating them and a hardware implementation of this algorithm in FPGAs. We have finally shown how this implementation is successfully integrated in a general load balancing platform for increasing the performance of execution of nested loops in hardware. Our future work is focused on extending the proposed architecture to no unitary coefficients which means general dependence vectors.

REFERENCES

[1] Chang, Henry and Cooke, Larry and Hunt, Merril and Marting, Grant and McNelly, Andrew and Todd, Lee, Surviving the SOC Revolution, Kluwer Academic Publishers",1999
[2] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, Jon Stockwood, Hardware-Software Co-Design of Embedded Reconfigurable Architectures, DAC 2000, pp. 507-512, 2000
[3] G. Economacos, P. Economacos, I. Poulakis, I. Panagopoulos, and G. Papakonstantinou. Behavioural synthesis with Systemc. In Design Automation and Test in Europe Conference and Exhibition (DATE01), Munich, Germany, pages 21–25, 2001.
[4] Michalis D. Galanis, Gregory Dimitroulakos and Costas Goutis, Performance Improvements from Partitioning Applications to FPGA Hardware in Embedded SoCs, The Journal of SuperComputing, Vol 35,2006,pp 185-199
[5] D. Moldovan, Parallel Processing: From Applications to Systems, Morgan Kaufmann, San Mateo, CA, 1993U.
[6] Chao Cheng and Keshab K. Parhi,A Novel Systolic Array Structure for DCT, IEEE Transactions on Cicuits and Systems,Vol 52,No 7,July 2005,pp.366-369
[7] Marcus Bednara and Jurgen Teich, Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms, The Journal of Supercomputing, vol. 26, pp. 149-165, Kluwer Academic Publishers, 2003
[8] C.P. Kruskal and A. Weiss, Allocating independent subtasks on parallel processors, IEEE Trans. On Software Engineering, 11(10): pp.1001-1016, 1985.
[9] F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos, and G. Papakonstantinou, "Dynamic Multi Phase Scheduling for Heterogeneous Clusters", Proc. of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06), Rhodes, Greece
[10] T. Andronikos, N. Koziris, G. Papakonstantinou and P. Tsanakas, "Optimal Scheduling for UET-UCT Generalized n-Dimensional Grid Task Graphs", Proceedings of the 11 th IEEE/ACM International Parallel Processing Symposium (IPPS97), pp 146151, Geneva, Switzerland.
[11] S. Palnitkar, Verilog HDL, Prentice Hall, Second Edition
[12] Xilinx Official Website www.xilinx.com
[13] Ioannis Panagopoulos, Chirstos Pavlatos, George Manis and George Papakonstaninou, A Flexible General-Purpose Parallelizing Architecture for Nested Loops in Reconfigurable Platforms, to be presented in PATMOS 2007, Sweden.