Implementation of Dynamic Loop Scheduling in Reconfigurable Platforms

Ioannis Riakiotakis
Computing Systems Laboratory
National Technical University of Athens
Athens, Greece
Email: iriak@cslab.ece.ntua.gr

George Papakonstantinou
Computing Systems Laboratory
National Technical University of Athens
Athens, Greece
Email: papakon@cslab.ece.ntua.gr

Anthony T. Chronopoulos
Dept. of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
Senior Member IEEE
Email: atc@cs.utsa.edu

***Abstract*— Dynamic scheduling algorithms have been successfully used for parallel computations of nested loops in traditional parallel computers and clusters. In this paper we propose a new architecture, implementing a coarse grain dynamic loop scheduling, suitable for reconfigurable hardware platforms. We use an analytical model and a case study to evaluate the performance of the proposed architecture. This approach makes efficient memory and processing elements use and thus gives better results than previous approaches.**

I. Introduction

Platform-based-design is a widely used and effective approach for the design of embedded systems. In such an approach the source code is partitioned into hardware and software units. Then the hardware part is placed on a special reconfigurable hardware such as FPGAs. Several platform-based-design approaches have been proposed and implemented [1], [2], [3], [4], [17],[5],[16],[18].

Nested loops are some of the most computationally intensive parts of the source programs. Many methodologies have been proposed to speed up further the execution on embedded systems by exploiting the parallelism available in nested loops [6],[7], [8]. All of these methodologies utilize fine-grain parallelism.

On the other hand in distributed memory computer systems, it is known that coarse-grain methods such as the ones presented in [10], [11], [12] offer better data locality and more efficient use of the memory than the fine-grain approaches. The use of coarse-grain methods can lead to performance improvements also in the case of embedded systems. The above coarse-grain methods are static, in the respect that the mapping of loop iterations to executing processors is done prior to the execution. This approach requires a space in memory for storing the execution map, which leads to poor memory utilization. In contrast to static scheduling algorithms, dynamic algorithms create the mapping on the fly during the execution. Another advantage of the dynamic algorithms is that they do not require the prior knowledge of the characteristics of the application and implementation environment in order to achieve a satisfactory output. This makes their use in practical applications very attractive. An important class of dynamic algorithms that has been developed for the parallelization of nested loops and that can provide coarse grain parallelism, is that of *Self-Scheduling* algorithms ([15] and references therein). These algorithms had been devised initially for loops without dependencies but their use was extended in loops with dependencies with the introduction of "Dynamic multi phase scheduling algorithm" (DMPS) [9]. The problem of parallelizing nested loops using dynamic scheduling algorithms in reconfigurable hardware platforms is still an open research issue. Given the success of the self-scheduling algorithms in platforms on parallel computers, it is worthwhile to port these algorithms to reconfigurable hardware platforms, in order to create an autonomous small scale parallel environment and increase the application execution speedup. In this paper the Hardware Dynamic-Multiphase scheduling (H-DMPS) algorithm is presented. H-DMPS is a new dynamic, self-scheduling algorithm based on the DMPS algorithm. The H-DMPS is designed to be used in reconfigurable hardware platforms and it is implemented with the use of the *verilog* hardware description language. With the proposed approach we improve the results existing so far in the literature by using coarse grain parallelism. More specifically the performance of H-DMPS was verified by simulation and it showed satisfactory results when compared to a recent dynamic scheduling method, presented in [6], which uses fine grain parallelism. The main drawback of this method was that the number of processing elements that could be used efficiently was limited by memory congestion and this number has been determined analytically by the authors in [6]. The use of coarse-grain parallelism in our approach, minimizes memory congestion thus allowing more processing elements to be used effectively.

The rest of the paper is organized in the following sections: in section II there is a brief presentation of the nested loops program model and an introduction to the self-scheduling algorithms. In sections III and IV the HDMPS algorithm is presented and analyzed. The experimental results are given in section V. We conclude and give future directions in section VI.

†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.

978-1-4244-1995-1/08/$25.00 ©2008 IEEE.

II. BACKGROUND

A. Algorithmic model

A nested loop of depth n is modeled as a n-dimensional cartesian space J ($J \subset Z^n$) called index or iteration space. We assume that, for each point of this space $(u_1, \ldots, u_n)$, we have that $L_i \leq u_i \leq U_i$, $1 \leq i \leq n$ and L_i and U_i are the lower and upper loop bounds of the $i-th$ dimension of the loop nest respectively. If all the points of this iteration space can be executed simultaneously, then we have a *parallel loop* without dependencies. If on the other hand, iterations depend on each other, then we have *dependent (parallel) loops*. Dependencies impose an iteration execution order and limit the potential parallelism of nested loops, since it is not possible for all iterations to be executed simultaneously. Data dependencies are modeled with the use of dependence vectors, which connect dependent iteration points, and which set is given as $DS = \{\vec{\mathbf{d}}_1, \ldots, \vec{\mathbf{d}}_r\}$, where r is the number of dependence vectors. The body of the nested loop includes general program statements like assignment statements, repetitional statements, conditional branching, etc. A example of a two-dimensional iteration space is given in Figure 1. We also assume, without loss of generality, that a perfectly nested loop is modeled as follows:

for u_1 =L_1 *to* U_1 **do**
  for u_2= L_2 *to* U_2 **do**
    ...
    for u_n= L_n *to* U_n **do**
      Loop Body
    end
    ...
  end
end

Algorithm 1: Programm model of a Perfectly Nested Loop

B. Self-scheduling for loops with dependencies

Self-scheduling algorithms decompose the iteration space of ***parallel loops*** along one loop dimension, which is called scheduling dimension *($u_c = 1, ..., U_c$)* into groups of iterations of V points, called chunks. This creates a pool of independent tasks which, are dynamically assigned to the available processing elements according to the master-worker model. In [9], the dynamic multi-phase scheduling algorithm (DMPS) was proposed to extend the use of self-scheduling algorithms into *loops with dependencies*. In this algorithm a second dimension of the loop was considered along which synchronization points (SPs) were added. The synchronization points were uniformly distributed at every h points. This dimension was called synchronization dimension *($u_s = 1, ..., U_s$)*. This synchronization scheme allowed the self-scheduling algorithms to be applied to loops with dependencies. The partitioning of the index space of a 2D nested loop with the scheduling and synchronization dimensions can been seen in Figure 1. In this figure the terms *chunk* and *subchunk* are illustrated. The lower bound of a chunk is the point along u_c with the smallest index that belong to the particular chunk. The upper bound is the point in u_c with the largest index that belong to this chunk. The same basic notation is used for the description of the H-DMPS in the Section III of this paper.

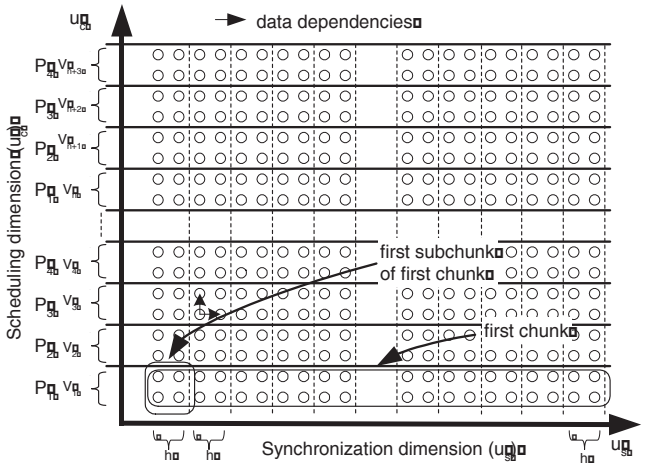


Fig. 1. decomposition of a $2D$ iteration space (J ($J \subset Z^2$)into chunks and the scheduling - synchronization dimensions

In the most basic self-scheduling algorithm (simple self-scheduling) [13], each worker undertakes the execution of a single iteration of the loop as soon as he becomes available, that is when he finishes the execution of the previous iteration that it had undertaken. This algorithm achieves very good load balancing since each worker can finish the execution with maximum difference of just one iteration. However it is obvious that the time required for the assignment of iterations to workers (scheduling cost) is very high. In order to decrease the scheduling cost, the Chunk-scheduling algorithm (CSS) was devised [14]. In CSS each worker undertakes one chunk at each scheduling step instead of one iteration, leading to reduced scheduling overhead. In general the scheduling overhead decreases as the chunk size increases but at the same time the load balancing capability of the scheduling algorithm is reduced, since each worker can finish with maximum difference of one chunk. In this paper we will focus on the CSS dynamic scheduling algorithm. Further information on the self-scheduling algorithms class can be found in [9],[15].

III. THE H-DMPS ALGORITHM, SCHEDULING IN HARDWARE

The DMPS algorithm is based on the Master-worker model which is often encountered in distributed memory architectures. In general the Master undertakes the role of the coordinator which accepts and serves requests from the workers. A request declares that the worker is available to execute a new task and the master replies to this request by assigning to this worker a new chunk of iterations. The parallel execution finishes when no more new tasks exist and all workers have

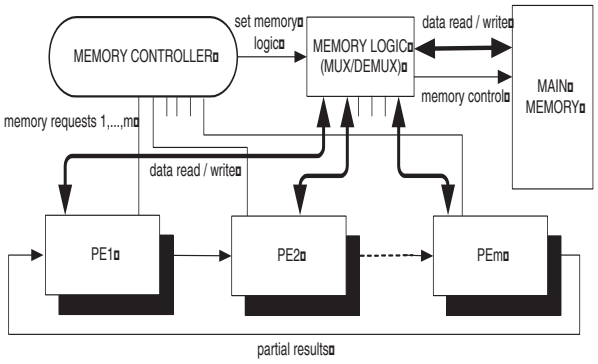


Fig. 2. Block diagram of H-DMPS

completed the execution of the tasks that they have undertaken. The hardware implementation of the DMPS algorithm follows roughly the same approach.In Figure 2 we can see the block diagram of the H-DMPS algorithm. The workers are implemented as a set of special purpose, processing elements (PEs) which have access to a common memory. All the application data are stored in this common memory and only one PE can access the memory at a given time. A memory controller undertakes the role of the master which accepts the requests from the PEs. The memory controller basically controls the access of the PEs to the common memory with the help of the memory logic. The memory logic is a set of multiplexers and de-multiplexers that interface the PEs to the memory read-write and control buses according to instruction signals provided by the memory controller. As we can see from the Figure 2, the PEs are interconnected in a ring topology. In this ring each PE passes data (partial results) to the next, implementing the synchronization mechanism that is required to satisfy the data dependencies. Each PE performs the following basic steps: it reads the data of one subchunk from the memory, it receives partial results from the *previous PE*, it process the subchunk, it sends partial results to the *next PE* and finally it writes the processed data back to the memory. *Previous PE* is the one that is found to the left of the *current PE* in the ring, while *next PE* is the one that is found in the right of the *current PE*.

The H-DMPS algorithm is described briefly in the Algorithm 2, more details can be found in the subsections III-A, III-B, III-C.

A. Memory Architecture

The main memory contains the initial data and assembles the processed results at the end of the parallel execution. Each PE reads part of the data from the main memory, it processes them and writes back the processed data at the same memory locations, updating in this way the contents of the memory. Thus the memory contains processed data up to the point that the execution has advanced, and initial data from this point onward.

The memory architecture of the system is presented in Figure 3. The main memory can be accessed by all PEs through

From the Memory Controller View:

1) Read Memory requests 1 to m from the input port, that correspond to the m PEs,
2) Find the request k with the highest priority
3) Grant memory access to PE_k.
4) **while** Request of PE_k is true **do** configure the Memory Logic to give exclusive access to PE_k
5) Goto step 1

From the Processing Element View:

for *each Processing element* PE_k*, where* $k = 1, \ldots, m$ **do**

1) Calculate current chunk size parameters (start point, stop point)
2) **for** *for all subchunks of current chunk* **do**
   - Request memory access from Memory Controller
   - Wait for memory access:
     - Read Data for the current subchunk from the Main Memory
     - Free memory
   - Read previous partial results, i.e. get data from previous PE, (PE_{k-1})
   - Compute subchunk
   - Write next partial results, i.e., send data to next PE, (PE_{k+1})
   - Request memory access from Memory Controller
   - Wait for memory access:
     - Write Data of the current subchunk to the Memory
     - Free memory

   end
3) **if** There are more chunks left **then** Goto to Step 1 **else** Finish **endif**

end

Algorithm 2: Desrciption of the H-DMPS algorithm

the use of a memory bus (shared memory). In conjunction to the main memory, each PE has its own local memory. This local memory is just large enough to store the data of one subchunk. Prior to the processing of a subchunk each PE transfers the required data from the main-memory to its local memory reducing in this way the amount of the total memory accesses i.e., the local memory plays the role of a cache memory. The advantage of the coarse-grain approach for the partitioning of the index space is that it provides far better data locality in comparison to the fine-grain approach. The data that have to be transferred from the main to the local memories for each subchunk are stored in successive memory locations. The clock cycles required to transfer these data from the main memory is then proportional to the width of the memory bus. If the memory bus is wide enough, a whole subchunk can be transferred in a single clock cycle. Thus, there is a tradeoff between the surface area occupied by the memory bus and the data transfer cost. For example, a subchunk of size 100 words can be transferred in one clock

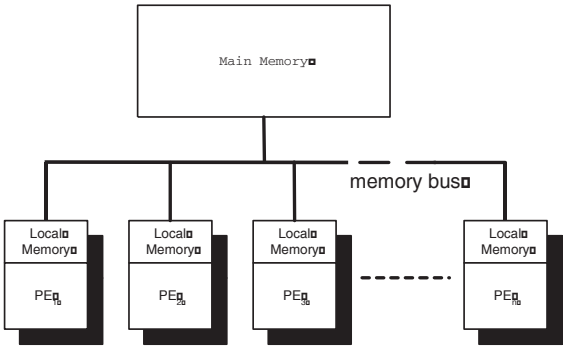


Fig. 3. The memory architecture

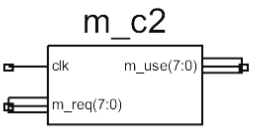


Fig. 4. The memory controller

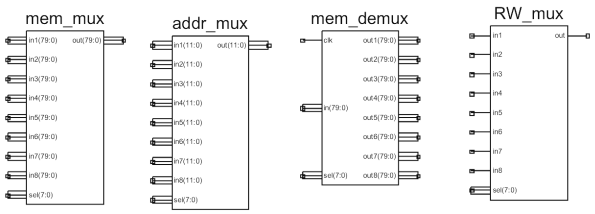


Fig. 5. Memory access sub-units

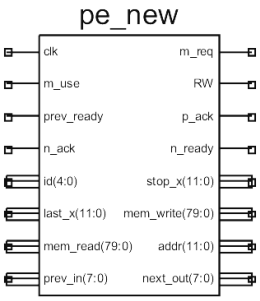


Fig. 6. The processing element component

cycle using a bus that is 100 words wide, in 10 cycles with a bus of 10 words or in 100 cycles with a bus of one word.

B. The Memory Controller

The memory controller coordinates the access to the main memory by the PEs. The memory controller has one input and one output port as can be seen in Figure 4. In the input port (*m_req*) he receives memory access requests from the PEs. The controller grants memory access to one PE according to a simple priority scheme. His decision is then written to the output port (*m_use*). If just one PE is requesting memory access, the controller decides that this PE can access the memory. If there are more than one PEs requesting memory access, the controller chooses the PE with the smallest rank (id). The selected PE can then access the memory exclusively to read or write data. The implementation of the memory controller is pretty straight forward. The input and output ports both have widths of m bits, where m is the number of the available PEs. Each bit corresponds to one PE, with bit one corresponding to PE with rank one (PE_1), bit two to PE_2, etc. The controller perceives a high bit in the input as a memory request, and the position of the high bit as the rank of the requesting PE. He then just has to choose the least significant high bit and set this bit in his output port.

The memory logic is complemented with the series of multiplexers/demultiplexers of Figure 5. The *mem_mux* is used to write data form the processing elements to the memory and the *mem_demux* to read data from memory, the *addr_mux* is used to write the memory address from the processing elements to the memory, and the *RW_mux* to select if the processing elements want to read or write data to the memory. All multiplexers have a number of inputs equal to the number of available processing elements and a single output, and the association of the output with one of the inputs is made using the select input (*sel*) of the multiplexers, which is driven by the *m_use* output of the memory controller. The memory demultiplexer has one input, which is connected to the data read port of the memory (Dout), and a number of outputs equal to the number of processing elements. The association of the input with one of the outputs is made again with the use of the *sel* and *m_use* ports. The lengths of the input/output ports is application/design dependent, and for the current implementation are given in Figure 5.

C. The processing elements

The most important part of the system is the processing elements (PEs), special designed to execute the loop body. A processing element is composed basically of the finite state machine (fsm) of Figure 7, and the local memory described in Subsection III-A. Each PE has a rank ranging from 1 to m, where m is the number of PEs, which in our case is $m = 8$. We consider that the PE with rank 1 (PE_1) starts processing first, then activates PE_2, which activates PE_3 and so on, until the last processing element PE_m is reached. From this point on, all processing elements work concurrently in a form of execution pipeline .

At the beginning of the parallel execution, all PEs are in the state $s0$. This is the scheduling state in which each PE calculates the bounds and the size of their next chunk. This can be seen as a distributed scheduling strategy. Initially PE_1, starts from the point 0 in the scheduling dimension u_c and adds to this the chunk size V_i according to the scheduling algorithm, which is constant in the case of CSS, to find its next chunk upper bound. Then PE_1 writes its upper bound to the output port $stop_x$ and proceeds to $s1$. The next PE,

which is PE_2 reads this value in its input port $last_x$ and this is the lower bound of its next chunk in u_c. just like PE_1 it adds to this lower bound the value V to find the upper bound of its next chunk. PE_2 writes its upper bound to the $stop_x$ output and proceeds to $s1$. This is done until PE_m is reached. States $s1$ to $s3$ are memory related. In $s1$, PEs calculate the memory address in which reside the subchunk data that have to be transferred from the main memory into their local memory, place this address in the address bus and proceed to $s2$. In $s2$ each PE make a memory access request by writing 1 in their m_req ports and wait in this state until the memory access is granted. When the memory access is assured, the PE proceeds to state $s3$. The transfer of subchunk data from the main memory to the local memory, as described in Subsection III-A, is performed in this state ($s3$). When the transfer is completed, the PEs writes 0 in the m_req port to free the memory and proceed to state $s4$. The first phase of synchronization takes place in this state. The current PE waits until the signal *prev_ready* is set, which means that the previous PE has completed the processing of a sub-chunk. The current PE reads the partial results, i.e., the processed data it requires form the previous PE from the port *prev_in* and sets the signal *p_ack* in order to inform the previous PE that it has completed the reception of data. Now, the PE has all the necessary data to compute a subchunk, and the computation is performed in state $s5$. The second phase of synchronization is performed in state $s6$, where the PEs write the partial results in the output port *next_out* and set the signal *n_ready*, so that the next PEs can read them. They wait in this state until the data transfer is acknowledged by checking for the *n_ack* signal. Then, in the states $s7$ and $s8$ the PEs request memory access to write back the processed data, similarly to the states $s2$ and $s3$. At this point, in state $s10$, the computation of a subchunk is completed and the PE has to distinguish among three cases:

- if this is also the end of the current chunk, the next state has to be $s0$, to start the processing of a new chunk
- if this is not the end of the current chunk, the next state has to be $s1$, to start the processing of the next subchunk
- if this is the end of the chunk, but there are no more chunks, this is the end of the parallel execution and the next state has to be $s11$

IV. Analysis

In this section we analyze briefly the H-DMPS algorithm in order to provide an estimate of the parallel execution time and to assess the effects of the use of different memory bus sizes on the overall performance of the scheduling algorithm.

The following additional notations are required for the analysis of the H-DMPS algorithm:

• t_{sr} is the send-receive overhead per word related to the length of dependence vectors (for unitary dependencies $t_{sr} = 1$).

• $T_s = T_r = t_{sr} \times h$, the clock cycles required to send/receive partial result of one subchunk form a neighboring PE.

• W_{bus} is the length of the memory bus is words.

• t_{mr} and t_{mw} are the memory read/write overheads (the number of words that have to be moved from the memory) for each iteration.

• $T_{mr} = t_{mr} \times V \times h/W_{bus}$ and $T_{mw} = t_{mw} \times V \times h/W_{bus}$, clock cycles for reading or writing a subchunk data from/to the main memory.

• T_{sch} is the scheduling cost which is implementation dependent and in the current implementation is 28 cycles.

• $T_p = t_p \times V \times h$ is the subchunk processing time, where t_p is the processing costs per iteration.

The scheduling of 12 chunks on 4 PEs is illustrated in Figure 8. Because of the existence of data dependencies it is not possible for the subchunks to be executed simultaneously, instead we have a pipelined execution. The numbers inside the boxes of Figure 8 designates the time step at which the the corresponding subchunk can be executed. As we can see, there are 3 pipelines, and the execution is completed in 27 time steps. At the end of each time step each PE has to exchange data with its neighboring PEs. The number of time steps is directly proportional to the number of PEs and the chunks sizes. Using this number we can provide an estimate on the parallel execution time. The number of steps is given by $N = (m-1) + k \times U_s/h$, where m is the number of PEs, and $k = U_c/(V \times m)$ is the number of pipelines. In each time step one sub-chunk is processed in time $T_{sbch} = T_{mr} + T_r + T_p + T_s + T_{mw} + T_{sch}$. So the estimated parallel time is $T_{par} = N \times T_{sbch}$. This value is close to the real parallel execution time as long as there are no conflicts in the memory access requests, i.e., when there is no memory congestion. In general, memory congestion is avoided when $T_p \geq T_{mr} + T_{mw}$, i.e., when a PE spends more time computing than using the memory or when the number of PEs is $m \leq \lceil T_p/(T_{mr} + T_{mw}) \rceil$. If we replace the values of T_{mr} and T_{mw}, we end up we the following formula.

$$m \leq \lceil \frac{t_p}{(t_{mr} + t_{mw})} \times W_{bus} \rceil$$

From the above formula we can see that the number of processing elements that can be used effectively without causing memory congestion is proportional to the length of the memory bus.

V. Results

In order to evaluate the performance of the approach described in this article, the H-DMPS algorithm is modeled with the use of the *verilog* hardware description language and simulated using the *modelsim* simulation suite. We give a set of measurements with 2,4,6 and 8 processing elements. Two different memory bus sizes were considered. In the first case the bus is one word wide so that just one word can be transferred from the memory per clock circle and in the second case 10 words can be transferred simultaneously from the main memory per clock cycle. In this way we assess the tradeoff between larger area utilization which is required to build larger data buses, versus the computation speedup. The

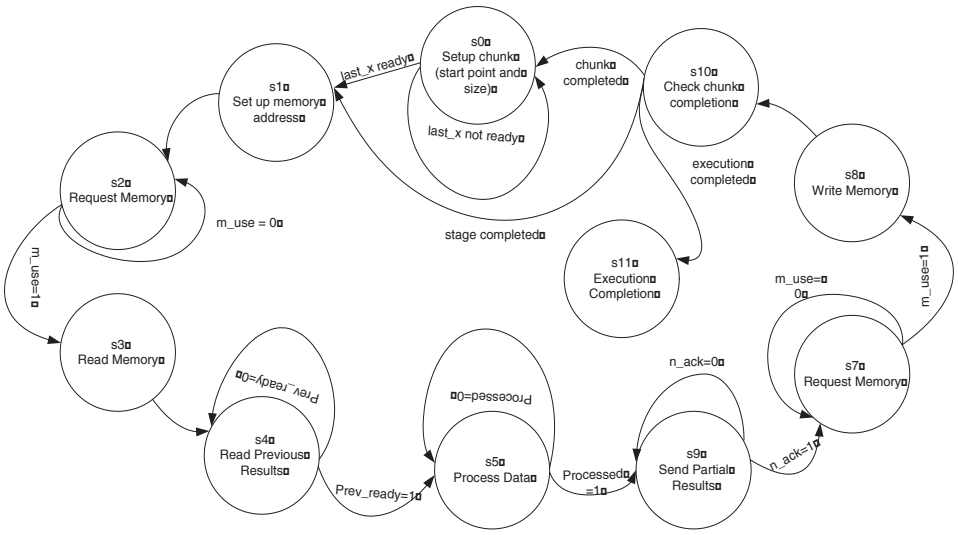
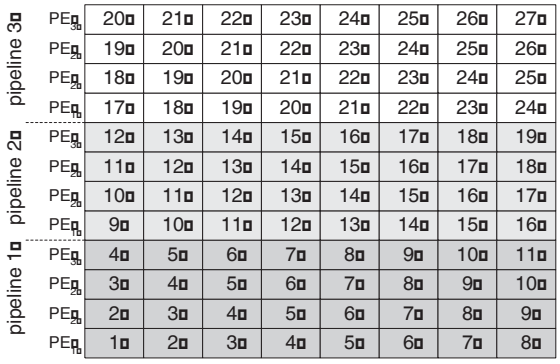


Fig. 7. The state transition diagram of a processing element



PE3	20	21	22	23	24	25	26	27
PE2	19	20	21	22	23	24	25	26
PE1	18	19	20	21	22	23	24	25
PE0	17	18	19	20	21	22	23	24
PE3	12	13	14	15	16	17	18	19
PE2	11	12	13	14	15	16	17	18
PE1	10	11	12	13	14	15	16	17
PE0	9	10	11	12	13	14	15	16
PE3	4	5	6	7	8	9	10	11
PE2	3	4	5	6	7	8	9	10
PE1	2	3	4	5	6	7	8	9
PE0	1	2	3	4	5	6	7	8

Fig. 8. Scheduling of **a dependent loop decomposed in** 12 chunks on 4 PEs

for i = *1 to 160* **do**
  for j= *1 to 63* **do**
    $P = 0.5 * Pm[i-1, j-1] + 0.5 * Pm[i-2, j-2];$
    if $(s[i, j] < zmax)$ **then**
      $s[i, j] += k[i, j] * a[i, j] * P;$
      $Pm[i, j] = (1 - a[i, j] * P);$
    else
      $Pm[i, j] = P;$
    end
  end
end
LastVertexPressure=Pm[159,62];

Algorithm 3: The application pseudocode

synchronization interval in every case is one point ($h = 1$) and the chunk size is 10 points along u_c, i.e., $V = 10$. H-DMPS is the compared to the algorithm described in [6].

The nested loop that is used as a test case is a classic model of a partial differential equation, solved using iterative methods. Its pseudocode is given below and it is part of an application that calculates the pressure exerted in the cells of a uniform plate when specific pressure is applied to one of its corners.

In the pseudocode above we choose the external loop (index i) to be the *chunk dimension* (u_c) and the internal loop (index j) to be the synchronization dimension (u_s). Moreover, the array Pm and the temporary variable P store the values of the pressure on the plate and the k, a, s and $zmax$ are used in the computation.

As we see from the bounds of the nested loop, the loop body will be repeated $160 \times 63 = 10080$ times.

In order to evaluate the proposed algorithm we make the following simplifying assumptions. We assume that all statements are executed in a single clock cycle, apart from multiplications and divisions that require 5 cycles. These assumptions may be not accurate in practice but make some statements more computational expensive that others, so they assist in the evaluation of the proposed architecture. Also, we do not employ any optimizations or code transformations such as loop unrolling for the comparison, since this is outside the scope of this work.

The above loop is executed in 28 clock cycles, hence in the best case the serial implementation it would take $10080 \times 28 = 282240$ clock cycles. We use this value to evaluate the efficiency of our algorithm and to calculate all speedup values in this section.

In Table I we present the simulation results, collected following the above assumptions using the *behavioral simulation* function of the *modelsim* simulator.

The results present the number of clock cycles required for the execution of the parallel algorithm on 2,4,6 or 8 PEs for the cases of simultaneous transfer of one or ten words from the main memory to the processing elements local memory. The speedup over the serial execution, which takes 282240

TABLE I

SIMULATION RESULTS. EXECUTION TIME (IN CLOCK CYCLES) FOR 2,4,6 AND 8 PES

PEs	1 word/cycle	10 words/cycles
2	181730	154514
4	92100	78474
6	61015	54211
8	58271	40812

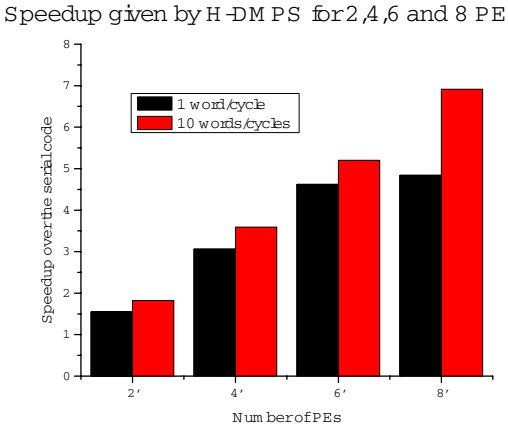


Fig. 9. Speedup of DMPS (2,4,6,8 PEs)

TABLE II

EXECUTION TIME OF THE FINE-GRAIN METHOD

PEs	Execution time (clock cycles)	speedup
1	323598	0.87
2	158697	1.77
3	103796	2.77
4	76395	3.69
5	59994	4.70
6	49093	5.74

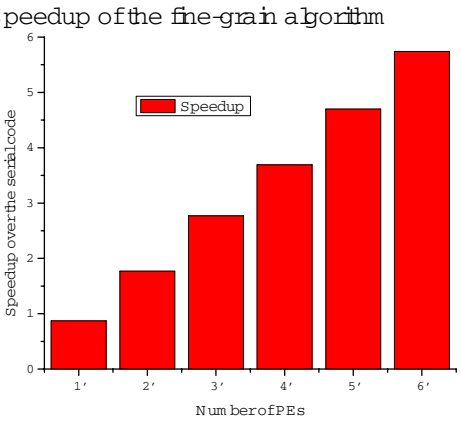


Fig. 10. Speedup of the fine-grain algorithm (1 to 6 PEs)

clock cycles as calculated earlier, is presented in the Graph 9. As we can see, in all cases the proposed algorithm offers significant speedup over the serial execution. Moreover, in the case of simultaneous transfer of 10 words per clock cycle from the memory the performance improvement ranges from 11% to 29% over the case of 1 word per clock cycle.

The next step is to compare the performance of the proposed algorithm to previously proposed dynamic algorithms. This is the first algorithm that combines dynamic scheduling and coarse grain parallelism, so we will compare the obtained performance to that of a recent dynamic and fine grain algorithm. The comparison is valid since both algorithms can be implemented and placed on the same reconfigurable hardware platforms. Table II summarizes the results obtained by the fine-grain algorithm presented in [6] for the same test problem. The optimal number of PEs, as calculated by the formulas provided in [6] is $m_{opt} = 6$, and beyond this number no further improvement can be obtained as stated in [6]. The speedup of this method over the serial code for up to 6 PEs is also illustrated in Figure 10.

As we can see in Table II the fine-grain approach achieves almost ideal speedup, i.e., up to 5.74 on 6 PEs. In the same number of PEs, the H-DMPS algorithms declines more from the ideal speedup, as it is 5.20 times faster than the serial code, in the case of 10 words/cycle. The advantage however of the coarse grain approach is that there is less congestion in the memory bus, so that more PEs can be used. The H-DMPS algorithm when used with two more PEs, i.e., $m = 8$, achieves a significant performance improvement of 17%, as the speedup in this case is 6.91.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we present a novel dynamic algorithm for the scheduling of nested loops with dependencies, in reconfigurable hardware. This algorithm offers a coarse-grain partitioning of the loop nest, which we claim that is more efficient than the fine-grain partitioning, in terms of execution time, data locality and memory management. Our claims were experimentally validated through testing. The results prove that the proposed algorithm achieves performance improvements over both the serial code and over a recently presented fine-grain dynamic scheduling algorithm. In the future, we intend to integrate other dynamic scheduling algorithms into the proposed architecture and evaluate their performance with extensive testing. The ultimate goal is to implement an efficient and easy to use software/hardware platform based on automatic software-hardware code partitioning that will utilize efficiently the proposed hardware architecture.

REFERENCES

[1] A. Sangiovanni-Vincentelli and G. Martin. Platform-Based Design and Software Design Methodology for Embedded Systems, IEEE Transactions on CAD of Integrated Circuits and Systems, 2002, December, Vol. 19, No 12, pp. 1523-1533

[2] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures, In Proc. Design Automation Conf., 2000, pp. 507–512.

[3] X. Wang, S. Ziavras. A configurable multiprocessor and dynamic load balancing for parallel LU factorization, In Proc. 18th International Parallel and Distributed Processing Symposium, 2004.
[4] J. Phillips, M. Areno, C. Rogers, A. Dasu, and B. Eames. A Reconfigurable Load Balancing Architecture for Molecular Dynamics, In Proc. 14th Reconfigurable Architectures Workshop, 2007.
[5] I. Panagopoulos. A. Dimopoulos, G.Manis, G. Papakonstantinou. AMPLE:Automatic mapping of algorithms for embedded systems, PCI2007, Patra, Greece, 2007.
[6] I. Panagopoulos, G. Manis and G. Papakonstantinou. Flexible General-Purpose Parallelizing Architecture for Nested Loops in Reconfigurable Platforms, PATMOS 2007, Sweeden, 2007.
[7] U. Bondhugula, J. Ramanujam, P. Sadayappan, Automatic mapping of nested loops to FPGAs, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07), San Jose California, USA, 2007.
[8] M. Bednara and J. Teich. Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms, The Journal of Supercomputing, vol. 26, pp. 149-165, Kluwer Academic Publishers, 2003
[9] F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos and G. Papakonstantinou. Dynamic Multi Phase Scheduling for Heterogeneous Clusters. 20th IEEE International Parallel & Distributed Processing Symposium, April, Rhodes, Greece, 2006.
[10] J. Xue. On Tiling as a Loop Transformation, Parallel Processing Letters, vol.7, no.4, pp. 409–424, 1997.
[11] G. Goumas, N. Drosinos, M.Athanasaki, N. Koziris. Compiling Tiled Iteration Spaces for Clusters, IEEE International Conference on Cluster Computing, p 360, September 2002
[12] J. Ramanujam, P. Sadayappan. Tiling of Iteration Spaces for Multicomputers, International Conference on Parallel Processing, Vol. II, pp. 179-186, 1990
[13] P. Tang, P.C. Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. Proceedings International Conference on Parallel Processing, pp. 528-535, 1986
[14] C. Kruskal, A. Weiss. Allocating Independent Subtasks on Parallel Processors. IEEE Transactions on Software Engineering, SE-11 (10), pp. 1001-1016, 1985
[15] AR Hurson, JT Lim, KK Kavi, B Lee. Parallelization of DOALL and DOACROSS LoopsAdvances in computers: A Survey. In Advances in Computer, volume 45, 1997.
[16] Y. Dai, Q. Li, Q. Zhang, J. Kuo. SIMD - efficient loop unrolling design for embedded multimedia applications, Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on, pp.1851 - 1854, June 2004.
[17] H. Saito ,A.V. Veidenbaum, X. Tian,M. Girkarmark, A. Nicolau, A. Kejariwal, Challenges in exploitation of loop parallelism in embedded applications Hardware/software codesign and system synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th international conference, pp 173-180, Oct. 2006.
[18] Ying Chen; Shao, Z.; Zhuge, Q.; Xue, C.; Bin Xiao; Sha, Minimizing energy via loop scheduling and DVS for multi-core embedded systems, Proceedings of the 11th International Conference on Parallel and Distributed Systems 2005,V 2, Page(s):2 - 6, July 2005.