Loop Tiling for Reconfigurable Accelerators*

Steven Derrien and Sanjay Rajopadhye

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE
[sderrien|rajopadh]@irisa.fr

Abstract In this paper, we focus on system level-optimizations for automatic parallelization of nested loop on Reconfigurable Accelerators. Specifically, as off-chip bandwidth plays a major role in total performances for such implementations, we propose some partitioning techniques based on loop tiling which can take advantage of the hierarchically structured RA memory systems.

1 Introduction

Although FPGAs are slower and take more area than an equivalent ASIC, Reconfigurable Accelerators (RA's) have demonstrated their ability to significantly outperform software implementations for several applications [11]. Most of these applications are compute-bound and have large number of simple and regular computations which exhibit potentially massive parallelism.

Although many prototype tools (both academic/research & commercial) are now able to perform a large part of this parallelization automatically, we may question whether this performance will scale well with technology. Specifically, it is well known that the gap between available on-chip computational power and off-chip memory bandwidth grows exponentially. Since most target applications usually involve large data-sets, I/O bandwidth is likely to become the limiting factor in the performance of RAs. Automatic parallelization tools must hence take these evolutions into consideration and and provide adequate system level optimizations.

In this paper, we focus on the parallelization of doubly nested loops onto unidirectional linear regular arrays. Such architectures can reach very high performance, but are not very well suited to take advantage of hierarchically structured memory systems. In most cases, data is only seen as a stream read from a source device, processed within the array, and then flushed out to an output device. Besides, data is processed at each clock cycle, and performance is dramatically affected by insufficient bandwidth.

We propose a novel processor array partitioning technique based on loop tiling, that overcomes limitations of traditional methods (LSGP, LPGS, copartitioning) when dealing with size constrained or highly hierarchical memory systems. We also develop an analytical performance model for the execution of nested loops on a generic RA with a two-level memory hierarchy, and obtain closed form solutions for the optimal tiling parameters. We are currently validating the method on a number of applications.

* Supported in part by IFCPAR project 1802-1: CORCoP Compilation and Optimization for Reconfigurable Co-Processors

The paper is organized as follows. Section 2 recapitulates array synthesis techniques and introduces out target architectural model. Section 3 reviews existing partitioning techniques and points out their weakness when it comes to size/bandwidth constrained memory hierarchy. The next few sections describe points enumerated above (partitioning through tiling, the analytic performance model, optimal tiling parameters and preliminary experimental validation). Section 7 presents our conclusions and future work.

2 Background

We consider the parallelization of a restricted class of loop nests: we assume a $W \times H$ perfectly nested rectangular loop with uniform dependences represented by m dependence vectors with positive integer components (this is a prerequisite for tiling), viewed as the columns of a matrix $D = [\overrightarrow{d_1} \ldots \overrightarrow{d_m}]$. Using well established space-time transformation methods [15] we derive an array of processors (PE) executing the loop in parallel (see Fig. 1). In addition, we choose the mapping and scheduling functions such that the interconnections are (i) unidirectional, and (ii) nearest-neighbor (details and justifications are given elsewhere [7], suffice to say that in practice, they do not impose a significant limitation).

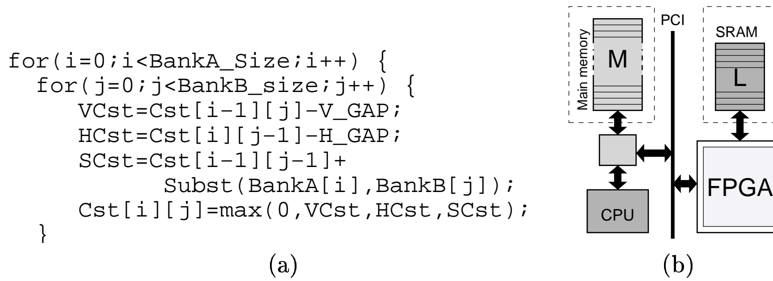

Figure1. (a) A common two dimensional loop nest (sequence comparison); (b) the typical target architecture on which it is to be implemented.

For the example of Fig. 1.a we may obtain a linear array of `BankB_Size` processors operating on a stream of `BankA_Size` data using $\overrightarrow{\tau} = [1\ 1]$ as the schedule vector and $\pi = [0\ 1]$ as the processor allocation.

We seek to implement this array on a target RA (see, eg. Fig. 1) which has a distinct memory hierarchy consisting of a main memory M (assumed to have unlimited capacity), on board memory L, and on chip memory/registers, comprising logic cells (LCs) which may be configured to emulate small grain memories[1]. Input data is present in (M) and results computed by the RA must be written back to (M) to be used by the host CPU application. M is generally accessed through a PCI type bus, offering relatively low bandwidth, often as low as 15MBytes/sec [13].

[1] In this paper we do not try to exploit embedded memory blocks available in most recent FPGA architectures.

When implementing processors arrays on such a target architecture, one difficulty is that the derived array has application dependent characteristics in terms of parallelism level and communication volume with the host. Hence a straightforward implementation, if even possible, is likely to give poor performance. A post-processing step is then required to transform it to a form more suitable for the RA architecture at hand.

3 Array partitioning

The problem of the efficient implementation of processor arrays on resource constrained architectures has been widely studied, and many approaches have been proposed [14,4,7,16,9].

The *serialization* transformation consists of clustering several PEs into a single one, and executing the iterations within this cluster sequentially. For linear arrays, if we *serialize* the array by a factor σ, a *virtual* array containing n PEs will be transformed into a $\lceil \frac{n}{\sigma} \rceil$ PE array. In addition to the array size, *serialization* also affects bandwidth: external I/O accesses (an also the array throughput) are slowed down by a factor σ. Finally *serialization* induces extra area requirements since it duplicates all the *internal registers*[2] by σ and creates a feed-back loop for each input port (see [7] for more details and formal proofs).

An alternative to *serialization* is to use multiple-pass (or LPGS) partitioning. In this approach, the space time domain is split into identical slices, each of size p, and implemented on a p-processor array. The whole loop domain is then computed by a succession of passes, each of them executing iterations from one slice. The number of passes is thus $\lceil \frac{n}{p} \rceil$. Note that this transformation is only valid for arrays with unidirectional connections [14]. Although this approach avoids the area overhead of serialization, it significantly increases (by a factor p) the overall number of I/O accesses, and does not reduce the array throughput requirements. Besides, since each pass reuses results from previous passes, an external FIFO memory (whose size is equal to the entire array output data volume) is required to store all inter-pass temporary data.

Since *serialization* induces an area overhead proportional to σ, it alone is obviously not suited to arrays with a large number of PEs: the larger the array size the larger the will be the σ factor, and hence, each PE will require much more area. Since we are resource constrained by the FPGA area, fewer PEs can be implemented, and the thus the available parallelism decreases.

Eckhardt et al. [9] showed how to combine serialization with a multi-pass approach. The array is then partitioned into p sub-arrays, in which the $\lceil \frac{n}{p} \rceil$ PEs are serialized by a factor σ. Such an approach allows a trade-off between local memory cost and bandwidth adaptation: the *serialization* factor is usually chosen so that the array throughput matches external I/O bandwidth (since the σ needed for this is not too large, the area overhead of *serialization* is not too high). Its most severe limitation is the required FIFO size which is directly dependent on the loop bounds: hence it does not scale well.

This limitation is even more stringent in the scope of an RA architecture: since on-board memory size is fixed, this forces us to use memory M as FIFO memory despite its poor bandwidth. The only cure is then to serialize the array

[2] i.e., registers not connected to a neighboring PE.

to match this low bandwidth, resulting in a severe loss of parallelism due to the local memory area overhead.

4 Loop Tiling

In this section we propose an adaptation of a well known loop transformation called tiling. Contrary to traditional array partitioning where the n dimensional space-time domain is partitioned using $n-1$ hyper-planes, a complete loop tiling will perform the partitioning using n hyper-planes resulting in a sub-domain basic block (or tile) for which all dimensions can be scaled down.

Tiling was initially introduced as a means to improve the performance of nested loop parallelization on (i) hierarchically structured memory systems and (ii) distributed memory machines [12,3,5]. The basic idea is to partition the loop-domain into tiles which are then executed atomically. The tiling problem is then to determine: (i) the tile shape and (ii) the tile dimensions which will give the best performance. Several criteria can be used to influence the choice of the tile, one can choose the tile volume to (i) balance communication and computation volume (ii) adapt the data accesses locality to some hierarchical memory level (from registers to main memory).

4.1 Application to Processor Arrays

Our goal here is to adapt the tiling approach to processor arrays, so that we can tune the data access locality to the the RA memory hierarchy. We will not broach the tiling problem in the general case, instead we will only consider a restrictive (but practical) case : 2D orthogonal tiling [3]. Here, the tile shape is chosen to have its boundaries parallel to the loop domain (see Fig. 2.a). Each of the resulting tiles is then parallelized on a (virtual) linear array of processors. The tile height H_T will correspond to the size of the derived virtual PE array, and the tile width W_T to the size of the data stream processed by the array.

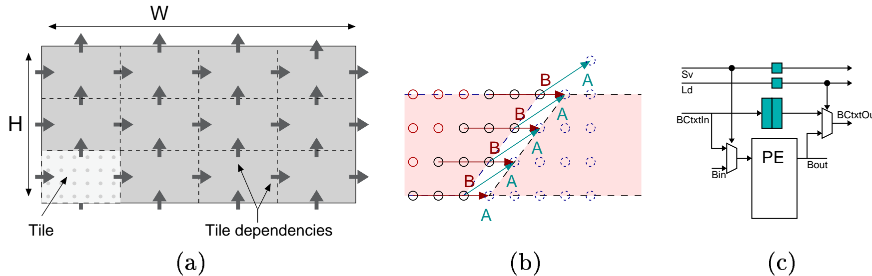

Figure2. (a) Tiled loop domain. The area within the dash-dotted bound is a tile. Dark grey arrows represents the data dependences at the tile level. (b) example of inter-tile horizontal (or temporal) dependences in the space-time domain. The array context data corresponds to data dependences whose source is in previous tile column. (c) context Load/Shift mechanism.

The transformation yields two types of data dependences: dependences with both vertices within the tile correspond to *intra-tile* dependences (leading to local communications in the array), whereas dependences with vertices in different tiles correspond to *inter-tile* dependences and necessitate data to be stored in some external memory and reused when the corresponding tile is executed. Hence, by changing the tile dimensions it is possible to tune the amount of *inter-tile* communication and also the number of external data accesses of a tile.

Contrary to classic array partitioning, tiling requires that between each tile execution, all the array internal variable are *saved* and *restored*. This is intuitively explained as follows. Tiling *breaks* the computation "pipeline" in the temporal dimension, and we need to save (and restore) this pipeline's *temporal context* in order to finish/restart the computation when subsequent tiles are executed. This of course induces an execution overhead since during these steps, the array can not perform any useful computation.

We also need additional control circuitry to save/restore the *array context*. This additional control mechanism (omitted here for brevity, but illustrated in Fig. 2.c, is required for every *variable* of the loop nest, be it local or spatial, and is independent of σ. It is important to note that the *context switching* phases can be overlapped (a tile context can be saved while another context is restored), resulting in an efficient array utilization.

4.2 Tiling strategies

The added degree of freedom provided by tiling gives a larger design space, notably in selecting the "tile-level" schedule, namely the order in which the tiles are executed. Now, row-major scheduling of tiles is nothing but increasing the tile width, but in addition to the column major order, other schedules (e.g. wavefront) are possible. However, the control logic becomes complicated and it is not clear that the yield better performance. Also note that it is possible tile at multiple levels corresponding to the levels of the memory hierarchy [5] and this holds even more so for our target architecture. This opens up different possibilities related to where the data is stored. In the rest of this paper, we consider only (full or partial) column major schedules, and explore two tiling strategies: *vertical tiling* and *horizontal tiling*.

Vertical Tiling is a two level tiling where we call the inner level of tiling as a pass (of size $W_p \times H_p$) and the outer tiling as a band ($W_b \times H_b$). We study the case where $H_b = H$ and $W_p = W_b$ (Fig. 3.a). All data accesses from or to the *band* are made through the PCI bus to the host system main memory M. Internal data access (i.e: between *tiles* of a same *band*) are made through the local on board memory L. Since the tile schedule is column major order, computations associated with a given *band* cannot start before all its preceding *bands* are done. Finally, note that only data associated to vertical dependences are saved in L memory, to be immediately reused in the following pass execution.

Horizontal tiling is also a two-level tiling strategy (we call the outer level *tiles* as ($W_s \times H_s$) *stripes*). We set $W_s = W$, and $H_s = kH_p$, for some positive integer k (see Fig. 3.b). All data accesses inside or outside a *stripe* are made through the PCI bus to memory M, and internal data access (i.e., between *tiles* of a same *stripe*) are made through L memory. As noted above, the *pass* schedule within a *stripe* is column major. As opposed to vertical tiling, here we use L memory to save data associated with both vertical and horizontal tile dependences.

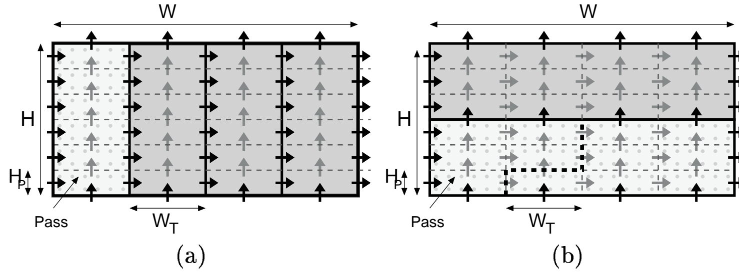

Figure3. (a) Vertical and (b) horizontal tiling. Arrows represents (inter) tile dependences, their color indicate which memory level is used to save intermediate results: black for main memory (L), light grey for local memory (L). The dashed line (in b) represents the maximum data volume that must be saved in L memory.

5 Performance modeling

We now develop an analytical model for our two tiling strategies. This will enable us to predict performance early in the design flow, and also explore the trade-offs involved with the choice of the tile dimension.

Our first step is to model the execution of a *pass* (or inner level tile). We decompose this into two phases (see Fig. 4). Let α and β respectively denote the number of words per-iteration that are used/produced as (i) context data (ii) input data. The pass execution hence consists of: (i) an overlapped *context loading/saving* phase during which βH_p data are loaded from (saved to) the internal registers of the array; (ii) a *computation* phase during which the stream of αW_p data words flows through the array pipeline.

Note that the RA performance is increasingly dictated by I/O bandwidth rather than by effective clock-speed [13]. We have shown that by using *serialization*, the effective array trough-put can be reduced at the price of a certain loss of parallelism [8]. It was also predicted that the best compromise would consist of finding the *serialization* factor which allows the closest match to the one of the external memory bandwidth (either M or L depending on which accesses have the stronger influence on overall execution time). In the scope of this work, we assume large loop domains and seek as much as possible to use the faster L rather than M. We hence make the choice to always *serialize* the array to match L bandwidth (let $\sigma_L$denote this serialization factor). The duration of the computational phase of a tile can then be approximated by the duration of its associated I/O operations (this is a crucial point and allows us to considerably simplify the analytical model).

Knowing this serialization factor, we can now estimate the number of PEs that are likely to fit on our FPGA. Let A_f be the available FPGA real estate (in slices), A_d be the combinational datapath area cost (in slices) of all operators involved in the loop body computation, S_m be the size (in bits) of the original PE local memory, and A_m the average area cost (in slices) of a memory bit for serialization. The number of *physical* PEs n_p is given by (a), and thus the pass height H_p which corresponds to the virtual array size $n_p\sigma_L$ is given by (b):

$$(a) n_p \approx \frac{A_f}{A_d + \sigma_L A_m S_m} \qquad (b) H_p \approx \frac{A_f \sigma_L}{A_d + \sigma_L A_m S_m}$$

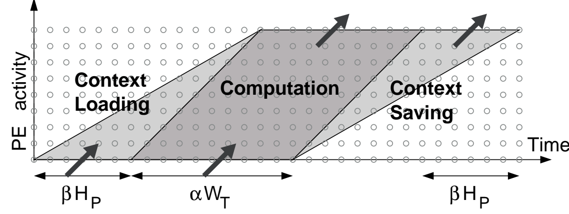

Figure4. Execution time for a $W_p \times H_p$ tile. Context loading and saving each require βH_p I/O accesses, computations requires $2\alpha W_p$ accesses (αW_p read operation and αW_p write operation).

6 Performance Optimization

Our goal of optimal performance, is equivalent to minimizing the time spent per iteration. Let ρ denote this average iteration execution time, T the whole loop execution time, and $T_{i,j}$ the inner tile (i,j) (or pass) execution time. We then have $\rho = \frac{T}{WH}$, $T = \Sigma_{i=1}^{n_h} \Sigma_{j=1}^{n_w} T_{i,j}$ where n_h and n_w represent the number of passes along vertical and horizontal axis. We can hence provide a general closed form for the performance optimization problem (see below), and use it to determine the optimal tile size for our two strategies.

$$\arg_{W_p,H_p} \max \frac{WH}{\Sigma_{i=1}^{n_h} \Sigma_{j=1}^{n_w} T_{i,j}}$$

6.1 Vertical Tiling

We first need to express the pass execution time as a function of its index i,j. Remember that we equate the pass execution time to that of its I/O. We see from Fig. 3.a, that there are three types of pass (in regard of their I/O access scheme). All passes access their context data from M, and only top and bottom passes access their computation stream from M, all other access being performed through L. If t_M and t_L denote the access time respectively for M and L, we obtain

$$T_{i,j} = \begin{cases} (\alpha W_P + 2\beta H_P)\, t_M + \alpha W_P t_L & \text{for a } \textit{top} \text{ or } \textit{bottom} \text{ pass} \\ 2\beta H_P t_M + 2\alpha W_P t_L & \text{otherwise} \end{cases}$$

Using this expression we can formulate our performance metric ρ as shown in (a). We see that H_p is set to a fixed value dependent upon (i) the loop body

area requirements (ii) the available FPGA real estate (iii) L memory bandwidth and (iv) array throughput requirements. Additionally in this tile schedule, W_p is constrained by the size of L since the whole pass computation results must fit within the on-board memory. Let W_L denote L memory capacity, this translate as $\alpha W_p \leq W_L$. Since ρ decreases when W_p grows, it is clear that the optimal inner tile width is $W_p^* = \frac{W_L}{\alpha}$, and thus optimal tile dimension are given by (b).

$$\text{(a) } \rho \approx \frac{2\alpha t_L}{H_P} + \frac{2\alpha(t_M - t_L)}{H} + \frac{2\beta}{W_P} t_M \qquad \text{(b) } \begin{cases} W_P^* = \frac{W_L}{\alpha} \\ H_P^* = \frac{\sigma_L A_f}{A_d + A_m \sigma_L} \end{cases}$$

6.2 Horizontal Tiling

Again, let us express the pass execution time. Here, we can have (see Fig. 3.b) four different type of passes: (i) *corner passes* which access half their context and computation data from M and the rest from L, (ii) *internal passes* access all their data from L, (iii) *horizontal boundary passes* access all their context data from L and half of their computation data from M and (iv) *vertical boundary passes* access half their context data from M, the rest being accessed from L. We can then write $T_{i,j}$ as below:

$$T_{i,j} = \begin{cases} (\alpha W_P + \beta H_P)(t_M + t_L) & \text{for a corner boundary tile} \\ (\alpha W_P + 2\beta H_P) t_L + \alpha W_P t_M & \text{for a horizontal boundary tile} \\ (2\alpha W_P + \beta H_P) t_L + \beta H_P t_M & \text{for a vertical boundary tile} \\ 2\beta H_P t_M + 2\alpha W_P t_L & \text{for an internal tile} \end{cases}$$

The average iteration execution time is then given by the following expression:

$$\rho \approx t_L \left(\frac{2\beta}{W_P} + \frac{2\alpha}{H_P} \right) + \left(\frac{2\alpha}{kH_P} + \frac{2\beta}{W} \right) (t_M - t_L)$$

Contrary to vertical tiling, both W_p and $H_b = kH_p$ (with k positive integer) are variables of the optimization problem, and are coupled together by the constraint on W_L, the size of L memory (where both computation and context data need to be stored). Specifically we have $\alpha W_p + \beta k H_p \leq W_L$. To solve the equivalent continuous constrained optimization problem, we use the well known Lagrange multipliers. Due to space limitations, mathematical details are omitted, and we only provide the closed form solution for the optimal tile dimension:

$$\begin{cases} k^* = \frac{\gamma - \sqrt{\gamma}}{\gamma - 1} \frac{W_L}{\beta H_P} \text{ with } \gamma = \frac{t_M}{t_L} - 1 \\ W_p^* = \frac{1}{\alpha} (W_L - \beta k^* H_p) \end{cases}$$

7 Experimental validation

To validate our model an its predictions, we used a Spyder X2 board [2]. It consists of a Xilinx XCV800-4 and two 256kx32 SRAM memory banks (we currently use only one of the) connected to a PIII-600Mhz through a PCI bus.

Observed PCI bandwidth is approximately 12MBytes/sec while our current SRAM interface allows 110 MBytes/sec. In our implementation, approximately 1300 slices are used to implement the overall system control logic (host interface, memory controller, array control, etc ...), the rest of the resources (8000 slices) being used to implement the PE array.

The target application is a modified version of the loop nest of Fig. 1.a. with a loop domain size of $10^7 \times 10^7$. By integrating a bubble sort mechanism to the PE architecture, we compute a more useful result, namely for each sequence, its K best matches (along with their similarity score) in the other data base (with $K \approx 300$). Due to the lack of space, the architecure is not presented in its details, we will only give an overview of its caracteristics in terms of speed, through-put and resource usage (more details should be available in the extended version of this work [?]):

The PE maximum achievable clock speed of $68MHz$ (estimated after place and route), the array being effectively clocked to $f_c = 66Mhz$. Note that to reach this frequency, we wen through a post-optimization step as described in [7]. The PE datapath is 7 byte wide during the computational phase and 9 bytes wide during context switching phase, leading to the parameters $\alpha = 7$ and $\beta = 9$.

From these figures, we can determine the initial through-put requirements: $\alpha f_c = 912$MBytes/sec during computational phase and $\alpha f_c = 1188$MBytes/sec during context switching phase. Hence, since neither M or L can sustain such a through-put, we are clearly in a case where our implementation performance is I/O bounded.

Let us now exploit the *serialization* transformation to perform bandwidth adaptation. Using the PE array characteristics, it is easy to determine the *serialization* factor σ_M matching M bandwidth: we have $\sigma_M = \lceil \frac{912}{12} \rceil \approx 76$. Using expression 5.b, we can, from the observed PE area cost $A_{p_m} \approx 510$ slices (after mapping) estimate: (i) the number of PEs $n_{p_m} \approx \frac{8000}{510} \approx 15$ and the pass height $H_p = 1140$ (note that this case is a standard co-partitioning scheme, where the pass width w_p is equal to the loop domain width W). Total execution time is thus $T \approx 2t_M \frac{H}{Hp}(\alpha W + \beta H_p)$ leading to an average iteration execution time of $\rho = 2t_M \left(\frac{\alpha}{Hp} + \frac{\beta}{W} \right) \approx 1.0$ ns per iteration.

Our goal is now to quantify the benefit of tiling: as seen previously, although it yields a more efficient utilization of L memory bandwidth, loop tiling induces an execution overhead due to context switching. We must hence understand how severe is the impact of this overhead on overall performance for each of the presented strategies. As seen previously, we make the choice of serializing the array to match L bandwidth, and then use the tiling transformation to minimize the number of M memory I/O accesses. Following previous paragraph, we obtain for $\sigma_L = \lceil \frac{924}{110} \rceil \approx 8$, leading to an observed PE area cost of $A_{p_l} = 301$ slices, an achievable number of processor of $n_{p_l} \approx \frac{8000}{301} \approx 26$, and a pass height $H_p \approx \sigma_L n_{p_l} \approx 208$.

From there, we can determine the (optimal) passes and tile dimensions for both tiling strategies: (i) for vertical tiling, we obtain $W_p^* \approx 146 \times 10^3$and $H_p \approx \sigma_L n_{p_l} \approx 208$. Using our performance model, we then obtain for the average iteration execution time $\rho \approx 0.6$ns per iteration (ii) for horizontal tiling, we have $k^* \approx 166$ and $W_p^* = 98 \times 10^3$, leading to a value of $\rho \approx 0.6$ns practically identical to the one obtained for vertical tiling. In both case, the improvement over the co-

partitioned solution is noticeable (around 40%), even for a target architecture in which the local memory size requirement are small. Hence for applications with higher local memory needs [13], we can expect even better results.

Besides, this improvement corresponds almost exactly to the gain in parallelism due to the smaller *serialization* factor: this means that, in our case, the impact of context switching over global performance is neglectable compared to the area overhead caused by a bandwidth adaptation to M.

8 Conclusion

In this paper we have presented a new partitioning methodology for processors array implementation based on the loop tiling transformation. As opposed to traditional partitioning techniques, this one allow to scale the off-chip memory requirements to suit the target system characteristics. We also provided a performance model which allow (i) to estimate the achievable performance (ii) to obtain a closed from solution for the optimal tile size problem. Since our current model is limited to the 2D orthogonal case, we would like to extend it to higher dimension loop nests and to non orthogonal tiling solutions.

References

1. Spyder Board x2 Manual Rev 1.1. FZI Website and http://www.x2e.de/.
2. R. Andonov, H. Bourzoufi, and S. Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *International Conference on High Performance Computing (HiPC)*, 1996.
3. J. Bu and E.F. Depreterre P. Dewilde. A Design Methodology for Partitioning Systolic Arrays. In *IEEE conference on Application Specific Array Processor*, 1990.
4. L. Carter, J. Ferrante, S. Hummel, B. Alpern, and K. Gatlin. Hierarchical tiling: a methodology for high performance. In *Technical Report CS-96-508 and University of California at San Diego*, 1996.
5. S. Derrien, S. Rajopadhye, and S. Sur-Kolay. Combining Instruction and Loop Level Parrallelism for FPGAs. IRISA Research report N°1376 and February 2001.
6. S. Derrien, S. Rajopadhye, and S. Sur-Kolay. Loop Tiling for Reconfigurable Accelerators. IRISA Research report.
7. S. Derrien, S. Rajopadhye, and S. Sur-Kolay. Optimal partitionning for FPGA based regular array implementations. In *IEEE PARELEC'00*, August 2000.
8. Uwe Eckhardt and Renate Merker. Co-Partitionning - A Method for Hardware/Software design for scalable Systolic Arrays. In *Reconfigurable Architectures and ITPress*, 1997.
9. J. Vuillemin et al. Programmable active memories: Reconfigurable systems comes of age. In *IEEE Transaction on VLSI Systems*, 1991.
10. K. Hogsted, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *ACM Symposium on Parallel Algorithms and Architectures*, 1999.
11. D. Lavenier. FPGA Implementation of the k-means Clustering Algorithm for Hyper-Spectral Images. Los Alamos Unclassified Report 00-3079 and July 2000.
12. D. I. Moldovan and J. A.N. Forbes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. In *IEEE Transactions on Computers*, January 1986.
13. P. Quinton. Automatic Synthesis of Systolic arrays from Recurrent Uniform Equations. In *International Conference on Computer Architecture*, pages 208–214, 1984.
14. L. Thiele J. Teich and L. Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Contrained Resources. In *International Conference on Application Specific Processor Arrays (ASAP)*, 1996.