RaPiD - Reconfigurable Pipelined Datapath[*][†]

Carl Ebeling, Darren C. Cronquist, and Paul Franklin

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract. Configurable computing has captured the imagination of many architects who want the performance of application-specific hardware combined with the reprogrammability of general-purpose computers. Unfortunately, configurable computing has had rather limited success largely because the FPGAs on which they are built are more suited to implementing random logic than computing tasks. This paper presents RaPiD, a new coarse-grained FPGA architecture that is optimized for highly repetitive, computation-intensive tasks. Very deep application-specific computation pipelines can be configured in RaPiD. These pipelines make much more efficient use of silicon than traditional FPGAs and also yield much higher performance for a wide range of applications.

1 Introduction

Configurable computing promises to deliver the high performance required by computationally demanding applications while providing the flexibility and adaptability of programmed processors. As such, configurable computing platforms lie somewhere between ASIC solutions, which provide the highest performance/cost at the expense of flexibility and adaptability, and programmable processors, which provide the greatest flexibility at the expense of performance/cost. Unfortunately the promise of configurable computing has yet to be realized in spite of some very successful examples[1, 5]. There are two main reasons for this. First, configurable computing platforms are currently implemented using commercial FPGAs which are very efficient for implementing random logic functions, but much less so for general arithmetic functions. Building a multiplier using an FPGA incurs a performance/cost penalty of at least 100. Second, current configurable platforms are extremely hard to program[5, 6]. Taking an application from concept to a high-performance implementation is a time-consuming, designer-intensive task. The dream of automatic compilation from high-level specification to a fast and efficient implementation is still unattainable.

[*] This paper appeared in *FPL '96: The 6th International Workshop on Field-Programmable Logic and Applications*, pages 126-135. Springer-Verlag, 1996.

[†] This work was supported in part by the Defense Advanced Research Projects Agency under Contract DAAH04-94-G0272. D. Cronquist was supported in part by an IBM fellowship. P. Franklin was supported by an NSF fellowship.

The RaPiD architecture takes aim at these two problems in the context of computationally demanding tasks such as those found in signal processing applications. RaPiD is a coarse-grained FPGA architecture that allows deeply pipelined computational datapaths to be constructed dynamically from a mix of ALUs, multipliers, registers and local memories. The goal of RaPiD is to compile regular computations like those found in DSP applications into both an application-specific datapath and the program for controlling that datapath. The datapath is controlled using a combination of static and dynamic control signals. The static control determines the underlying structure of the datapath that remains constant for a particular application. The dynamic control signals can change from cycle to cycle and specify the variable operations performed and the data to be used by those operations. The static control signals are generated by static RAM cells that are changed only between applications while the dynamic control is provided by a control program.

The structure of the datapaths constructed in RaPiD is biased strongly towards linear arrays of functional units communicating in mostly a nearest neighbor fashion. Systolic arrays[2], for example, map very well into RaPiD datapaths, which allows the considerable amount of research on compiling to systolic arrays to be applied to compiling computations to RaPiD[4, 3]. RaPiD is not limited to implementing systolic arrays, however. For example, a pipeline can be constructed which comprises different computations at different stages and at different times.

The computational bandwidth provided by a RaPiD array is extremely high and scales with the size of the array. The input and output data bandwidth, however, is limited to the data memory bandwidth which does not scale. Thus the amount of computation performed per I/O operation bounds the amount of parallelism and thus the speedup an application can exhibit when implemented using RaPiD. The RaPiD architecture assumes that at most three memory accesses are made per cycle. Providing even this much bandwidth requires a very high-performance memory architecture.

RaPiD is also not suited for tasks that are unstructured, not highly repetitive, or whose control flow depends strongly on the data. The assumption is that RaPiD will be integrated closely with a RISC engine on the same chip. The RISC would control the overall computational flow, farming out the heavy-duty computation to RaPiD that requires brute force computation.

The concept of RaPiD can in theory be extended to 2-D arrays of functional units. However, dynamically configuring 2-D arrays is much more difficult, and the underlying communication structure is much more costly. Since most 2-D computations can be computed efficiently using a linear array, RaPiD is currently restricted to linear arrays.

The paper begins with a description of the datapath architecture and how computations are configured. This is followed by a description of the way dynamic control signals are generated. Next, a FIR filter example is used to illustrate how computations are mapped to the RaPiD architecture. The paper ends with a discussion of the performance of RaPiD-I and future work.

2 RaPiD Architecture

This section describes the version of the RaPiD architecture, called RaPiD-I, which is currently being implemented at the University of Washington. Variants of this architecture with a different data width and data format, different functional units, different number and configuration of busses and so on, could be defined for different application domains. The RaPiD-I architecture contains all the salient features of RaPiD and will allow us to describe RaPiD computations for a variety of applications.

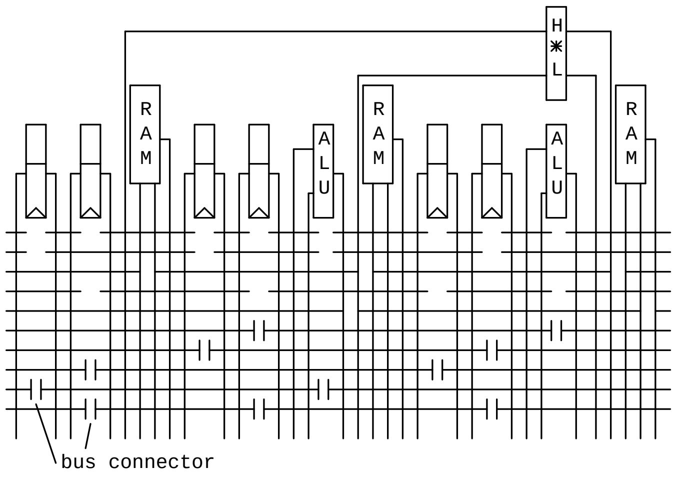


Fig. 1. The basic cell of RaPiD-I. This cell is replicated left to right to form a complete RaPiD array.

RaPiD-I is a linear array of functional units which can be configured to form a (mostly) linear computational pipeline. This array of functional units is divided into identical cells which are replicated to form a complete array. One cell for RaPiD-I is shown in Figure 1. This cell comprises an integer multiplier, two integer ALUs, six general-purpose registers and three small local memories. The complete RaPiD-I array contains 16 of these cells. Although the array is divided into cells, this division is invisible when it comes to mapping an application to the functional units and busses.

The functional units are interconnected using a set of ten segmented busses that run the length of the datapath. Each input of the functional units is attached to a multiplexer which is configured to select one of eight busses. Each output of the functional units is attached to a demultiplexer comprised of tristate drivers, each driving one of eight busses. Each output driver can be configured independently, which allows an output to fan out to several busses, or none at all if the functional unit is not being used. The assignment of operations to functional units must be done so there is a bus segment available to connect units that communicate.

The busses in different tracks are segmented into different lengths so that bus tracks are used efficiently. In several tracks, adjacent bus segments can be connected together using either a buffer or a register. This bus connector is shown in Figure 2 and is represented in Figure 1 as a pair of lines between bus segments. The connection is active and can drive in either direction but not both at once. Many of the registers in a pipelined computation can be implemented using these bus pipeline registers. In theory, all the bus segments in one track could be connected together by bus connectors configured as bypass buffers to provide a broadcast signal the length of the array. In practice, the delay is much too long and all signals are pipelined to some degree.

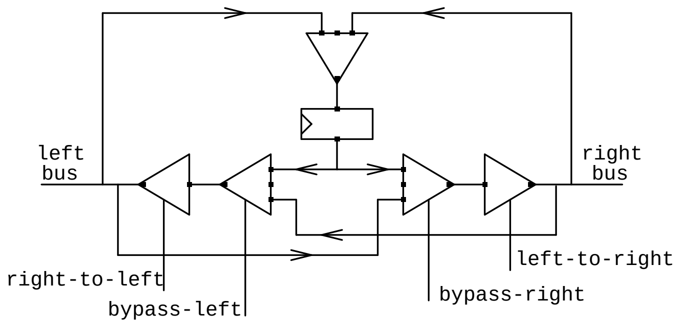


Fig. 2. Bus connectors can be used to connect adjacent bus segments via a buffer or a register.

Functional unit outputs are registered, although this output register can be bypassed via configuration control. Functional units may additionally be pipelined internally depending on their complexity. These pipeline registers can also be bypassed if appropriate.

RaPiD-I operates on 16-bit signed or unsigned fixed-point data which is maintained via shifters in the multipliers. Different fixed point representation can be used in the same application by appropriately configuring the different shifters in the datapath. An extra tag bit is associated with each data value to indicate whether an overflow has occurred. Once set, the overflow value is propagated to all results. The datapath thus generates no exceptions during operation, but incorporates them into the data produced.

The ALUs perform the usual logical and arithmetic operations on 16-bit data. The two ALUs in a cell can be combined to perform a pipelined 32-bit operation, most typically as a 32-bit add for multiply-accumulate computations. The ALU output register can be used as the accumulator for multiply-accumulate operations.

The multiplier multiplies two 16-bit numbers and produces a 32-bit result, shifted by a statically programmed amount to maintain the appropriate fixed-point representation. Both 16-bit halves of the result are available as output via separate bus drivers. Either driver can be turned off to drop the corresponding output if it is not needed. The multiplier uses a modified Booth's algorithm and includes one configurable pipeline register.

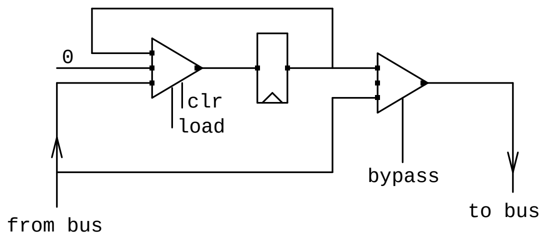


Fig. 3. Datapath register.

The registers in the datapath are used to store constants and temporary values as well as create pipelines of different lengths. These registers are completely general unlike the registers found in the bus connectors and functional units, which are used only for pipelining. Figure 3 shows the design of the datapath registers. The datapath register inputs and outputs are connected to the busses just like other functional units. One configuration signal controls whether the output is driven by the register or the bypass path. This bypass is used to connect a bus segment on one track to a bus segment in a different track. The load and clear signals control the operation of the register. As discussed in Section 3, these control signals must be set dynamically. While datapath registers are very general, they are expensive in terms of both area and bus utilization. While the datapath registers themselves are relatively small, their input multiplexer and output drivers are quite large. Wiring the input and output of a datapath register usually requires bus segments in two different tracks which consumes extra routing resources. Thus the bus pipeline registers and the functional unit registers are used whenever possible.

A limited amount of local memory is provided in the datapath for saving and reusing data over many cycles. In many applications, the input or output data is segmented into blocks that are accessed once, saved locally and reused as needed, and then discarded. Local memory can also be used for constant arrays. RaPiD-I includes three local memories per cell. The input and output data lines are connected to busses as in other functional units. Because of the time needed to read and write memory, configurable registers are included on both the input and output data ports. The memory address is supplied either by a data bus or by a local address generator, shown in Figure 4, that supports simple sequential memory access. If values are read and written to consecutive addresses, which is the most common case, then the memory address generator can supply the addresses without using datapath resources.

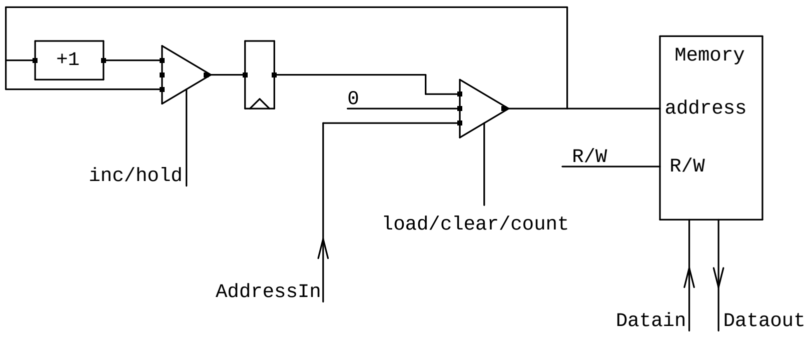


Fig. 4. Local memory.

Input and output data enter and exit the datapath via I/O streams at each end of the datapath. These streams act as the interface to external memory. Each stream contains a FIFO which is filled with data required by the computation or with results produced by the computation. The data for each stream is associated with a predetermined block of memory from which it is read or to which it is written. The datapath reads from an input stream to obtain the next input data value and writes to an output stream to store a result. Address generation and memory reads and writes are handled entirely by the I/O streams themselves. The I/O stream FIFOs operate asynchronously: if the datapath reads a value from an empty FIFO or writes a value to a full FIFO, the datapath is stalled until the FIFO is ready.

3 Datapath Control

For the most part, the signals that control the operation of the functional units and their interconnection can be static over an entire application. However, there are almost always some control signals that must be dynamic. For example, constants are loaded into datapath registers during initialization but then remain unchanged. The load signals of the datapath registers thus take on different values during initialization and computation. More complex examples include double-buffering the local memories and performing data-dependent calculations.

The control signals are thus divided into static control signals provided by configuration memory as in ordinary FPGAs, and dynamic control which must be provided on every cycle. RaPiD is programmed for a particular application by first mapping the computation onto a datapath pipeline. The static programming bits are used to construct this pipeline and the dynamic programming bits are used to schedule the operations of the computation onto the datapath over time. A controller is programmed to generate the dynamic information needed to produce the dynamic programming bits.

Of the 230 control signals in a RaPiD-I cell, 80 are dynamic. Thus there is a total of over 1200 dynamic control signals for the entire datapath. While configuration memory is relatively cheap, producing and communicating the dynamic control signals on every cycle, using a standard microprogram for example, would be very expensive.

The problem of generating static control signals is solved using a control path which parallels the data path. RaPiD applications map into pipelines of similar, if not identical, repeating pipeline stages. The control signals of these stages are thus similar as well, except that their values are skewed in time in the same way the data passing through the pipeline is skewed in time.

The control path is thus a set of segmented busses containing configurable pipeline registers through which control signal values are sent from one end of the datapath to the other. Control values are inserted at one end of the control path and are passed from stage to stage where they are applied to the appropriate control signals. The configurable pipeline registers allow different control signals to travel at different rates through the control path.

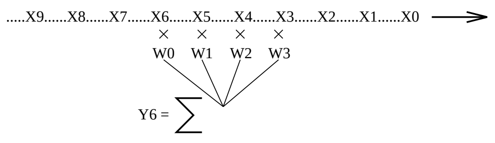
Generating the dynamic control signals is then accomplished by connecting each dynamic control signal to a bus in the control path that carries the appropriate value each cycle. The number of busses required in the control path varies by application, but it is kept manageable because many control signals have identical values. The values inserted into the control path are generated by a simple microprogrammed controller whose microinstructions contain the datapath control information in addition to looping constructs that allow datapath instructions or instruction sequences to be repeated many times.

4 Example Application: FIR Filter

The simple FIR filter provides a good illustration of how RaPiD executes algorithms. Figure 5a gives a specification for a $NumTaps$ filter with $NumX$ inputs. The filter weights are stored in the W array, the input in the X array, and the output in the Y array (starting at array location $NumTaps - 1$). Figure 5b shows the entire computation required for a single output of a 4-tap FIR filter.

```
for i = NumTaps-1 to NumX-1
  Y[i] = 0
  for j = 0 to NumTaps-1
    Y[i] = Y[i] + X[i-j]*W[j]
  end
end
```

(a)


......X9......X8......X7......X6......X5......X4......X3......X2......X1......X0
×
×
×
×
W0
W1
W2
W3
Y6 = ∑

(b)

Fig. 5. (a) Algorithm for FIR filter. (b) Computation for NumTaps=4 and i=6.

As with most applications, there are a variety of ways to map FIR filter to RaPiD. The choice of mapping is driven by the parameters of both the RaPiD array and the application. For example, if the number of taps is less than the number of RaPiD multipliers, then each multiplier is assigned to multiply a specific weight. The weights are first preloaded into datapath registers whose outputs drive the input of a specific multiplier. Pipeline registers are used to stream the X inputs and Y outputs. Since each Y output must see $NumTaps$ inputs, the X and Y busses are pipelined at different rates. Figure 6a shows a schematic diagram for this implementation on a four-tap FIR filter. The X input bus was chosen to be doubly pipelined and the Y input bus singly pipelined. Wires are annotated with the weight, input, and output values from a single point in time during the computation phase.

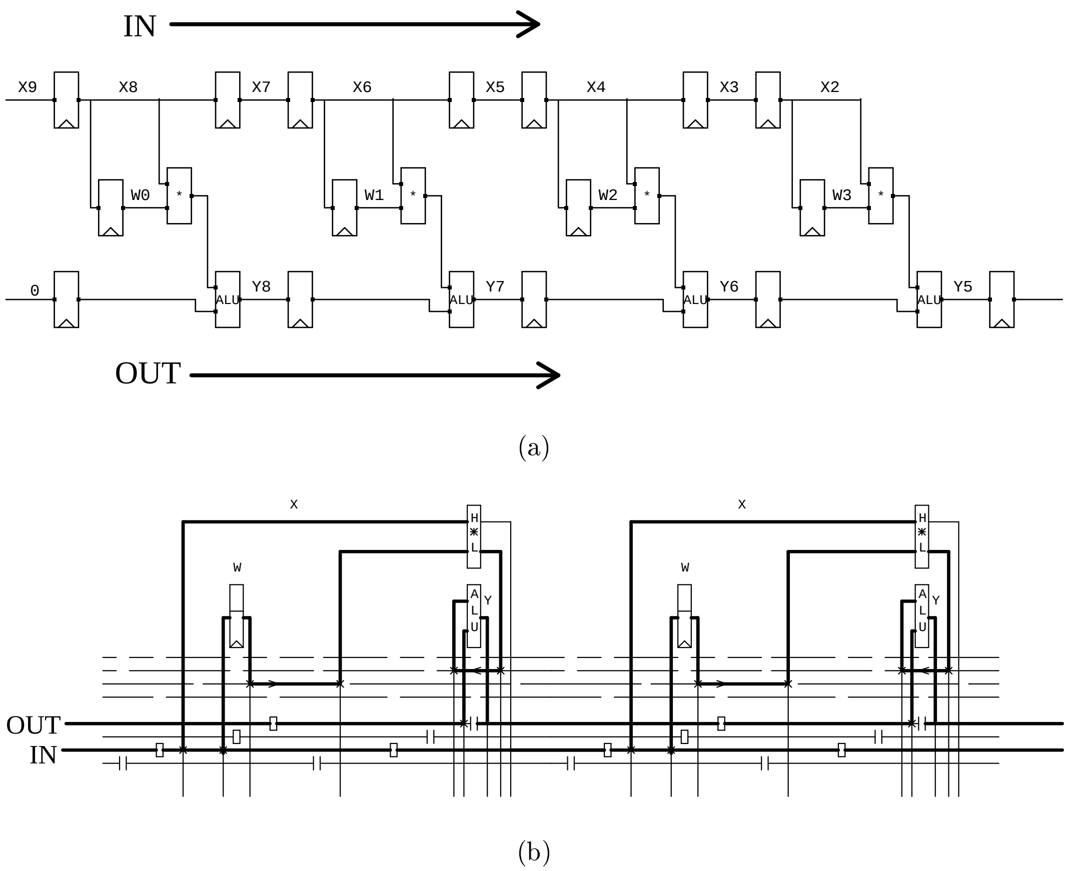


Fig. 6. (a) Schematic diagram for four-tap FIR filter, labeled at a point in time (computing four parallel computations for y5, y6, y7, and y8). (b) Two taps of the FIR filter mapped to the RaPiD array (this is replicated to form more taps).

This implementation maps easily to the RaPiD array, as shown for two taps in Figure 6b. For clarity, all unused functional units are removed, and used busses are highlighted. The bus connectors from Figure 1 are left open to represent no connection and boxed to represent a register. The control for this mapping consists of two phases of execution: loading the weights and computing the output results. In the first phase, the weights are sent down the IN double pipeline along with a singly pipelined control bit (not shown) which sets the state of each datapath register to “LOAD”. When the final weight is inserted, the control bit is switched to “HOLD”. Since the control bit travels twice as fast as the weights, each datapath register will hold a unique weight. No special signals are required to begin the computation; hence, the second phase is started the moment the control bit is set to “HOLD”.

5 Performance

This section evaluates the sustained (ignoring initialization and finalization) computation rate of mapping FIR filter and matrix multiply to the RaPiD array. These results are a function of both the RaPiD array parameters and the algorithmic parameters. The parameters associated with the RaPiD array are the clock rate in MHz (τ), the number of cells (S), and the number of addressable memory locations per cell (M). Because RaPiD by its very nature is heavily pipelined, a conservative estimate on the RaPiD-I clock rate of a mapped application is 100MHz. In addition, conservative estimates of the number of RaPiD-I cells and memory locations per cell are 16 and 96, respectively. Results will be measured in MOPS or GOPS, where an operation is a single multiply-accumulate combination. The maximum rate on RaPiD-I is 1.6 GOPS.

5.1 FIR Filter

The only algorithmic parameter affecting the sustained computation rate of the FIR filter is the number of taps, T. The mapping described in Section 4 produced one output per cycle and thus τT MOPS with the constraint that $T \leq S$. For a more general mapping restricting the taps to $T \leq \frac{1}{3}MS$, the RaPiD array can produce $\min(1, \frac{S}{T})$ outputs per cycle and $\tau \min(T, S)$ MOPS.[5] For example, with $\tau = 100$, $S = 16$, $M = 96$, and $T \geq 16$, RaPiD can perform a sustained rate of 1.6 GOPS on a FIR filter with up to 512 taps (and an unbounded number of input values).

5.2 Matrix Multiply

Matrix multiply takes an $X \times Y$ matrix $\mathbf{A}$ and a $Y \times Z$ matrix $\mathbf{B}$ and computes the $X \times Z$ matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ as $c_{ij} = \sum_{k=0}^{Y-1} a_{ik} b_{kj}$. Many different RaPiD mappings exist, each producing slightly different performance results. In one implementation, the RaPiD array can produce $\min(1, \frac{S}{Y}, \frac{\frac{1}{3}M}{Y})$ operations per cycle and $\tau \min(Y, \frac{1}{3}M, S)$ MOPS. With $\tau = 100$, $S = 16$, $M = 96$, and $Y \geq 16$, RaPiD can perform a sustained rate of 1.6 GOPS (X and Z are unbounded).

[5] This is a simplified version of a more complex formulation which is beyond the scope of this paper.

6 Conclusions and Future Work

The RaPiD architecture potentially provides a very efficient reconfigurable platform for implementing computationally intensive applications. Many applications have been mapped successfully by hand to RaPiD and simulated with very promising results. However, there are several open problems which need to be solved to make RaPiD truly successful.

- The domain of applicability must be explored by mapping more problems from different domains to RaPiD.
- Thus far all RaPiD applications have been designed by hand. The next step will be to apply compiler technology, particularly loop-transformation theory[7] and systolic array compiling methods[4] to build a compiler for RaPiD.
- A memory architecture must be designed which can support the I/O bandwidth required by RaPiD over a wide range of applications.
- Although it is clear that RaPiD should be closely coupled to a generic RISC processor, it is not clear exactly how this should be done. This is a problem being faced by other reconfigurable computers.

Acknowledgments

We would like to thank the rest of the RaPiD team, Chris Fisher, Larry McMurchie and Jeffrey Weener, for their contributions to the project.

References

1. J. M. Arnold, D. A. Buell, D. T. Hoang, D. V. Pryor, N. Shirazi, and M. R. Thistle. The Splash 2 processor and applications. In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 482–5. IEEE Comput. Soc. Press, 1993.
2. H.T. Kung. Let's design algorithms for VLSI systems. Technical Report CMU-CS-79-151, Carnegie-Mellon University, January 1979.
3. P. Lee and Z. M. Kedem. Synthesizing linear array algorithms from nested FOR loop algorithms. *IEEE Transactions on Computers*, 37(12):1578–98, 1988.
4. D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, 1986.
5. J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, 1996.
6. M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II compiler and architecture. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16. IEEE Comput. Soc. Press, 1993.
7. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991.

This article was processed using the LaTeX macro package with LLNCS style