PIPELINING SEQUENCES OF LOOPS: A FIRST EXAMPLE

Rui Rodrigues[1]
and
João M. P. Cardoso[1,2]
[1]*Faculty of Sciences and Technology,*
University of Algarve,
Campus de Gambelas, 8000 – 117 Faro, Portugal
[2]*INESC-ID, Lisbon,*
Email: jmpc@acm.org

ABSTRACT

Sequences of loops or sets of nested loops exist in many applications. This paper shows a scheme to pipeline those sequences of loops in such a way that subsequent loops can start execution before the end of the previous ones. It uses a hardware scheme with decoupled and concurrent datapath and control units that start execution at the same time. The communication of data items between two loops in sequence is conducted by memories. Each element of one of such memories is responsible to flag the availability of the data requested by a subsequence loop. Thus, the control execution of subsequent loops is also orchestrated by data availability and out-of-order produced-consumed pairs are permitted. We apply the concept to a real example: a fast DCT algorithm.

KEYWORDS

Pipelining, Mapping of Algorithms, Hardware Schemes, FPGAs.

1. INTRODUCTION

Due to its impact in performance, loop pipelining has been focus of intense research efforts for many years [1]. It has been considered for both processor and application specific architectures. With respect to compilation for FPGAs, the work in [2] has been one of the first efforts considering pipelining of innermost, well-behaved, loops. Examples of efforts on loop pipelining to FPGA-based systems are presented in [3], [4], [5], [6], [7], and [8]. Beside the efforts on generating the pipelining structures for efficient execution of loops in FPGAs, authors have used some loop transformations (*e.g.*, unrolling, tiling) to make the most promising loops innermost loops since only them are pipelined by most compilers [9]. More complex loop pipelining schemes try to overlap subsequent iterations of an outer loop with an inner loop. Bondalapati [10] illustrates through a single example the pipelining of two such loops. The idea is to overlap the execution of iterations of the outermost loop where each of its iteration executes an inner loop. This scheme needs as pipeline stages memories instead of simple registers since those stages inserted in the body of the inner loop must detach all the iterations of such loop.

Recently, a pipelining scheme at coarse-grained levels has been exploited [11]. The scheme permits to overlap some of the execution steps of sequences of nested loops or functions (*i.e.*, functions or loops waiting for data start computing as soon as the required data items are produced in a previous function or by a certain iteration of a previous loop). They communicate each data item as soon as it is produced by first loops to subsequent loops in the code. Although this pipelining scheme can be efficient for some type of algorithms, it requires, as far as we know, that items produced in the first loops are used in the same order by the subsequent loops. In [12] they present analysis schemes to determine the communication needed by this type of pipelining.

The increasing number of available FPGA resources enables the research of new ideas and hardware schemes. Particularly, schemes without the saving of hardware resources as a first goal may lead to

performance increase otherwise not achieved. The approach presented in this paper decouples the control units of each set of loops and uses a set of memory elements to signal the availability of data items to the subsequent loops. Doing that, the scheme permits pipelining of sequences of loops even when data produced by a set of loops is not consumed in the same order by a subsequence of loops.

This paper is structured as follows. Next section shows the concept and presents the scheme through a real example. Last section concludes the paper and explains the future work.

2. PIPELINING SEQUENCES OF LOOPS

Sequences of loops occur in most applications. Some sequences have loop nests with large iteration space and take long runtimes. Mapping such sequences to hardware structures that maintain the imperative model (i.e., only after a loop or a set of nested loops finishes execution, subsequent loops start their execution) creates a sub-optimal hardware implementation in most cases, as far as performance is concerned. As an example of typical sequences of loops presented in real code, consider a Fast DCT (Discrete Cosine Transform) algorithm, available from Texas Instruments Inc. [13], in Figure 1 a). The algorithm calculates the DCT of an input image using blocks of 8×8 pixels.

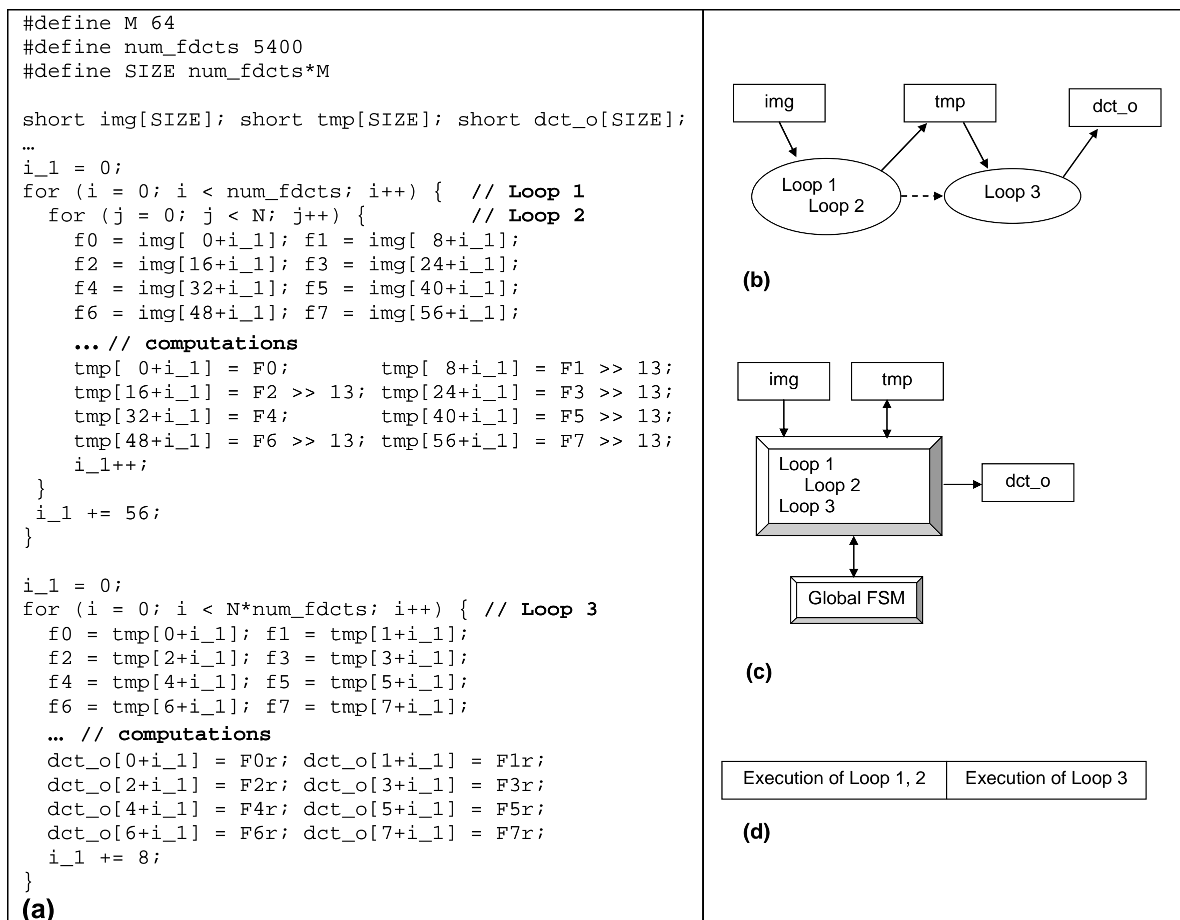

Figure 1. Example: (a) part of the Fast DCT code; (b) Algorithm structure, where bubbles represent sets of loops in the code and rectangles represent the array variables; (c) Typical implementation with a datapath and global FSM to implement the algorithm and the use of one distinct memory for each array variable; (d) Execution of the loops according to the typical implementation.

In this example, we can see that data items of array *tmp* are produced in the first set of nested loops (Loops 1, 2) and used by the third loop (Loop 3). The *tmp* data is generated by Loops 1, 2 in the order 0, 8, 16, 24, 32, 40, 48, 56, 1, 9, 17, … and consumed by Loop 3 in the order 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, …. To simplify the example suppose that each array variable is mapped to a different memory. As a typical hardware implementation we have a global FSM (Finite State Machine) that controls the datapath, memory accesses, and furnishes the loop behaviors, executing them in sequence (see Figure 1 b), c), and d)).

Since data items of array *tmp* produced in the initial iterations of Loops 1, 2 are requested in the initial iterations of Loop 3, there is no constraint to start the execution of Loop 3 before the end of all the iterations of Loops 1, 2. In this case a table of flags indicating if an element has been already produced can be used by the datapath of Loops 1, 2 (see Figure 2 a)), and two concurrent FSMs can be used to control each set of loops. When loading an element of array *tmp*, the second FSM only has a transition to a new state if and only if the correspondent flag in table *tab* has value one, which indicates that the associated data item is ready. Otherwise, the FSM will wait for data availability. The hardware implementation uses two FSMs and two datapaths, all concurrent. In addition, a dual-port memory is used to store the array *tmp* and a dual-port 1-bit table is used to store flag values (similar to ready signals). Figure 2 b) and c) shows the hardware structure and the resulting pipelining execution, respectively.

The results using RTL (Register Transfer Level) simulation of a typical implementation of the Fast DCT and the implementation using the pipelining concept presented here show a speedup of 1.48 with 5,400 image blocks of 8 × 8 pixels (equivalent to NTSC image size: 720 × 480 pixels). This result is very promising and justifies further research efforts.

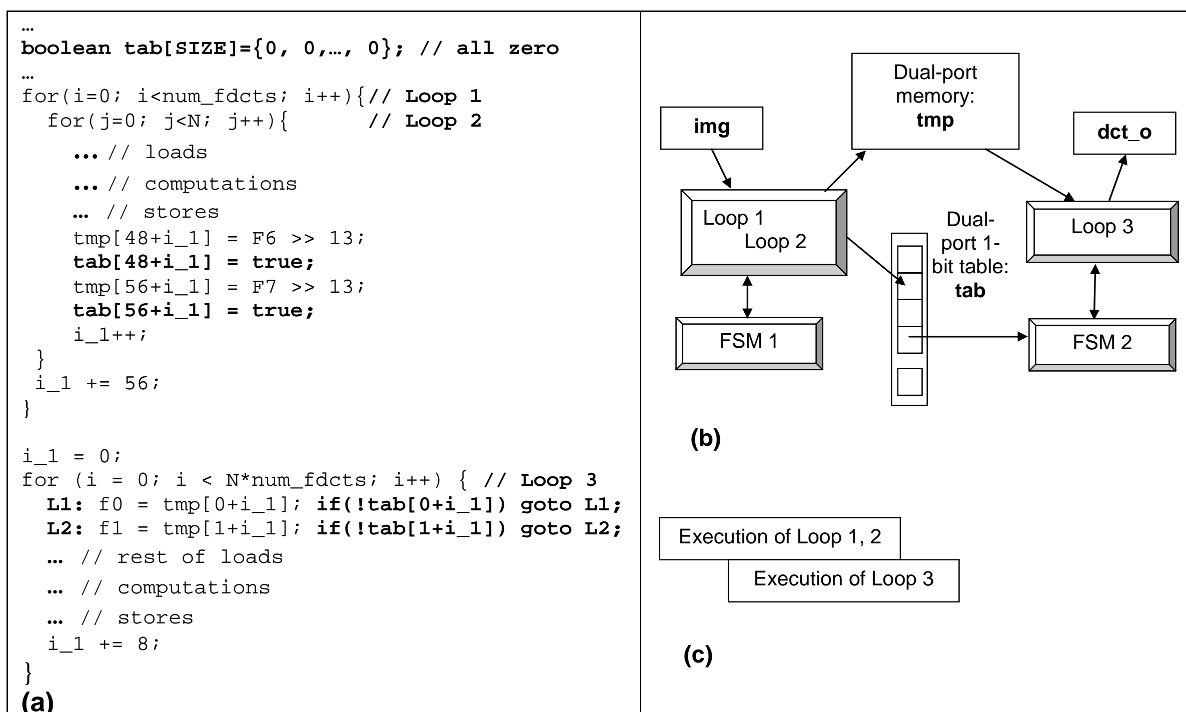

Figure 2. (a) Insertion of the behavior needed in the original code to explain how the concept works. Note that the sequence of loops is implemented as concurrent FSMs and datapaths. (b) Implementation for pipelining the sequence of loops using two decoupled and concurrent FSMs. In this case, the array *tmp* is stored in a dual-port memory and a dual-port 1-bit table is used to store ready signals (*tab* array) for each memory element produced by Loops 1 and 2. This table is accessed in parallel by the FSM2 in order to check the validity of the data item needed by Loop 3 from the memory with *tmp*; (c) Overlap of execution of the two loop sets using pipelining.

When data items are consumed in the same order of the data being produced, FIFOs may be used to communicate data between sequences of loops. This is the scheme used in [12]. However, that approach

requires an analysis to determine the number of FIFO stages needed to maintain high throughput or a handshake protocol in both sides of the FIFOs to stop each of the sets of loops in the sequence. In our case, a table for each array being produced-consumed is used with the same number of elements of the correspondent array. Although further studies are needed, due to the available number of resources in some of the current FPGA devices and the memory elements on- and off-chip, this may not cause a problem.

A problem may occur when there is more than one write to the same array element in a set of loops. This problem requires further research. Further work is also planned to evaluate the use of this form of pipelining in more real applications and to evaluate the increase in hardware resources.

3. CONCLUSIONS

This paper presents a scheme to accelerate applications with sequences of loops by pipelining their execution. With this scheme, before the end of a loop or a set of nested loops a subsequent loop or set of nested loops can start execution based on the data already produced by the first set. The scheme is implemented using dual-port memory elements for accessing the produced-consumed data between sets of loops and to communicate ready signals. The scheme also uses concurrent FSMs to control each loop set with synchronization being achieved by ready signals stored in tables with 1-bit width.

The approach can be seen as an extension of the scheme presented in [12], as it deals with irregular (out-of-order) produced-consumed data pairs in a natural way and without needing advanced compiler analysis techniques.

We expect to realize further experiments in order to evaluate the obtained speedups with more benchmarks.

ACKNOWLEDGEMENTS

This work is partially supported by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002).

REFERENCES

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software Pipelining," in *ACM Computing Surveys*, Vol. 27, Issue 3, Sept. 1995, pp. 367-432.

[2] M. Weinhardt, "Portable Pipeline Synthesis for FCCMs," In *Proc. 6th Int'l Workshop on Field-Programmable Logic and Applications (FPL'96)*, Darmstadt, Germany, Sept. 1996. Springer-Verlag.

[3] T. Maruyama, and T. Hoshino, "A C to HDL compiler for pipeline processing on FPGAs," In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, IEEE CS Press, 2000, pp. 101-110.

[4] P. Diniz, and J. Park, "Automatic Synthesis of Data Storage and Control Structures for FPGA-based Computing Engines," In *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, IEEE CS Press, 2000, pp. 91-100.

[5] T. J. Callahan and J. Wawrzynek, "Adapting Software Pipelining for Reconfigurable Computing," In *Proc. of the Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, San Jose, CA, USA, Nov. 17-19, 2000, ACM Press, New York, pp. 57-64.

[6] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from Matlab®," In *Proc. of IEEE/ACM Int'l Conference on Computer Aided Design (ICCAD'01)*, San Jose, CA, USA, Nov. 4-8, 2001, pp. 314-319.

[7] M. Gokhale, J. M. Stone, and E. Gomersall, "Co-synthesis to a hybrid RISC/FPGA architecture," In *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, Vol. 24, No. 2, March 2000, pp. 165-180.

[8] G. Snider, "Performance-constrained pipelining of software loops onto reconfigurable hardware," In *Proc. of ACM 10th Int'l Symposium on Field-Programmable Gate Arrays (FPGA'02)*, ACM Press, New York, 2002, pp. 177-186.

[9] M. Weinhardt, and W. Luk, "Pipeline Vectorization," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2. Feb. 2001, pp. 234-233.

[10] K. Bondalapati, “Parallelizing of DSP Nested Loops on Reconfigurable Architectures using Data Context Switching,” in *Proc. of IEEE/ACM 38th Design Automation Conference (DAC’01)*, Las Vegas, Nevada, USA, June 18-22, 2001, pp. 273-276.

[11] H. Ziegler, B. So, M. Hall, and P. Diniz, “A Parallelizing Compiler Approach for Coarse-Grain Pipelining on Multiple FPGA Architectures,” in *Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’02)*, Napa, CA, USA, 2002, pp. 77-86.

[12] Heidi E. Ziegler, Mary W. Hall, and Pedro C. Diniz “Compiler-Generated Communication for Pipelined FPGA Applications,” in *Proceedings of the 40th Design Automation Conference (DAC’03)*, pp. 610-615.

[13] Texas Instruments, Inc., “TMS320C6000™ Highest Performance DSP Platform,” 1995-2003, http://www.ti.com/sc/docs/products/dsp/c6000/benchmarks/62x.htm#search