Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach

Ricardo Santos[1,2], Rodolfo Azevedo[1], and Guido Araujo[1]

[1]State University of Campinas
Institute of Computing
Campinas, SP, Brazil
{ricrs, rodolfo, guido}@ic.unicamp.br

[2]Dom Bosco Catholic University
Dept. of Computer Engineering
Campo Grande, MS, Brazil

Abstract

Fast reconfiguration is a mandatory feature for reconfigurable computing architectures. Research in this area has been increasingly focusing on new reconfiguration techniques that can sustain the architecture performance and to allow the simultaneous execution, at the same stage, of configuration and computation tasks. In this context, this paper presents a new dynamic reconfiguration technique, based on a configuration cache, that tackles this challenge by configuring and executing operations on functional units during the execution stage. This approach is implemented in a pipelined reconfigurable multiple-issue architecture called 2D-VLIW. Our dynamic reconfiguration technique takes advantage of the 2D-VLIW pipelined execution by starting reconfiguration concurrently to activities like reading operand registers and executing operations.

1. Introduction

It is well known that the main purpose of reconfigurable computing is to fill in the gap between performance and flexibility. In this sense, it is important to investigate new reconfiguration strategies aiming at mixing reconfiguration and execution time, so as to improve the final architecture performance.

Previous work [4, 5] have shown that it is possible to classify reconfiguration techniques in two classes. First, program patterns are statically recognized and configured into the hardware before program execution. Second, specific instructions are molded into the reconfigurable fabric at run time. Both approaches impact or restrict program performance. The first approach does not allow reconfiguration during execution, thus precluding the use in programs that demand run-time reconfiguration to improve performance. The second approach uses specific instructions to tell hardware what reconfiguration to use. In this case, the overhead comes from the fact that reconfiguration needs to be finished before the next instruction is issued to reconfigurable logic. Several real-world reconfigurable architectures have adopted reconfiguration techniques based on these two classes. The GARP architecture [5] is composed by a main processor and a reconfigurable matrix of logic blocks. Activities such as loading configuration and running data on the matrix are controlled by the main processor through specific instructions. ADRES [7] is a coarse-grained dynamically reconfigurable architecture that includes a tightly coupled VLIW processor and a matrix of reconfigurable functional units running a set of word-level operations selected by a control signal. PipeRench [4] is an architecture based on a dynamic reconfiguration pipeline. The architecture consists of 16 processor elements (PE) per stage and each PE has one ALU and 8 registers. One cycle before a stage is executed, PipeRench configures it by setting functions and interconnections.

Our approach, on the other hand, is strongly based on storing configuration bits into a configuration cache inside the architecture datapath. Basically, we extract at compile-time, configuration bits from single operations and store these bits into a configuration cache. The remainder information from these operations are gathered in large VLIW [3] instruction words. At runtime, the configuration cache is searched for configuration bits. The selected line complements the instruction bits with signals to configure and execute onto functional units. This approach allows one to factorize the instruction operands associated to the Directed

1-4244-0054-6/06/$20.00

Acyclic Graphs (DAGs) from its opcodes. As a result, the size of the instruction fetched from memory is reduced, since it carries only operands.

The remainder of this paper is structured as follows. In Section 2 we describe the approach used in this paper to reach dynamic reconfiguration. In this section we describe the base reconfigurable architecture used to implement this technique and the steps of the dynamic reconfiguration process. Finally, we conclude in Section 3 by listing future work.

2 Dynamic Reconfiguration in the 2D-VLIW Architecture

In this section we present and discuss our dynamic reconfiguration strategy and the base reconfigurable architecture where this strategy is implemented.

2.1 Architecture Background

The 2D-VLIW is a pipelined reconfigurable architecture that exploit instruction level parallelism through a two dimensional VLIW execution model. The execution model deals with large instructions (2D-VLIW instructions) composed of single operations. These operations are executed by a set of functional units (FUs) organized as a matrix. The number of functional units is equivalent to the maximum number of operations in one instruction. In the execution stage, these operations are dispatched to functional units on a per column basis. Figure 1 shows a simplified overview of the 2D-VLIW architecture composed of a 4×4 functional unit matrix and its interconnection network. In Figure 1, the processor dispatches 4 operations to functional units at each execution cycle EX_1, EX_2, EX_3, EX_4. Notice that by using the interconnection network each FU in the column n, $1 \leq n < 4$, can provides operands to two FUs in the column $n + 1$.

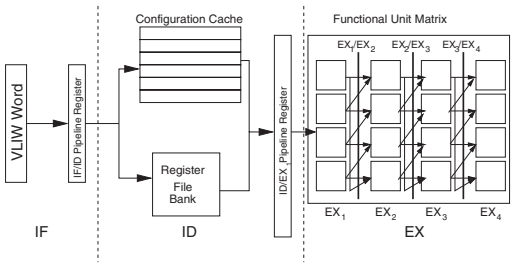


Figure 1. An Overview of the 2D-VLIW arch.

The 2D-VLIW looks like well-known multiple-issue reconfigurable architectures. However, our dynamic reconfiguration strategy takes place at the architecture's pipeline, starting at the decode stage by looking for configuration bits that enable operations to execute. The configuration from each operation is stored in a configuration cache which holds information to configure FUs at the execution stage.

The functional units matrix is composed by generic FUs in order to allow a homogeneous configuration process that is independent of the type of unit being reconfigured. Figure 2 shows all logic blocks and signals to be configured inside a FU. *Operands* input data come from the 2D-VLIW instruction whereas *SelOpnd*1, *SelOpnd*2 and *SelOperation* input signals come from the configuration cache. The *Functional Unit* element is responsible for executing the operations using the given inputs. Currently, results from an operation may be written into a *Temp Register* instead of a global register file. *Temp Register* can be used to minimize the pressure over the global register file, such that temporary results are available to FUs [2] through the interconnection network. Thus, since that the global register file does not need to have one write port per functional unit, the compiler has to avoid conflicts in the same pipeline stage (columns).

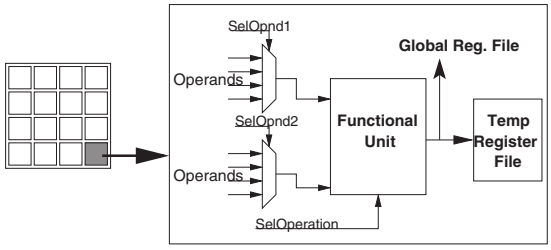
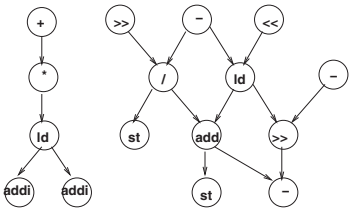


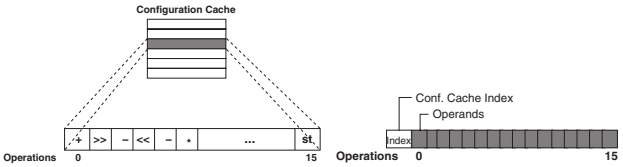
Figure 2. Functional unit.

2.2 Compiler Assistance to Dynamic Reconfiguration

One important step in our dynamic reconfiguration strategy consists in the selection of the configuration data to be stored in the configuration cache. This step takes place during the program compilation, when the compiler collects configuration parameters from the operations in DAGs. The compiler maps DAGs from the program to the architectural resources. Big DAGs can be broken into small ones to optimize the execution. After extracting configuration bits from these operations, the compiler put them on a new configuration cache line. The remainder operation information are put in a 2D-VLIW instruction. Figure 3 shows an example of this step.



(a) Operations' DAGs.



(b) Conf. cache line. (c) 2D-VLIW inst.

Figure 3. Steps to create a 2D-VLIW instruction.

Figure 3(a) shows two DAGs created by the compiler. After extracting configuration data from the operations, the compiler creates a new line in the configuration cache as shown in 3(b). The fields in the cache-line indicates configuration data picked up from each operation. For the sake of simplicity, we have omitted some operations from the cache-line but it must be clear that it has all configuration data from the DAGs in 3(a). The operand information from each operation are encoded in a 2D-VLIW instruction as shown in 3(c). The resulting instruction encodes the operand register numbers and an index field pointing to the cache-line associated to the operations it carries. The configuration information are stored into the configuration cache just before program execution.

A 2D-VLIW instruction can be seen as a function call in which the arguments are encoded into its fields, and the operations are encoded into a reconfiguration cache-line. At run-time the processor fetches an instruction, decodes it, and fetches the cache line associated to it. The processor then provides the operands (from the 2D-VLIW instruction), and the configuration bits (from the fetched cache-line) to the FU matrix.

2.3 Dynamic Reconfiguration Process

Our dynamic reconfiguration process starts on the decode stage when the processor fetches the cache-line associated to the 2D-VLIW instruction in the decode stage and stores it into the ID/EX_1 pipeline register. Simultaneously, it reads input registers from the register file. As said before, the cache-line carries the opcodes associated to the 2D-VLIW operations, as well as the routing information and operand selection bits required to configure the matrix interconnection network. The information collected during the ID stage is enough to prepare the matrix to compute the DAGs associated to the corresponding 2D-VLIW instruction.

Figures 4 and 5 exemplify the 2D-VLIW instruction from Figure 3(c) being executed on the 2D-VLIW datapath. We have left out the instruction fetch stage, because it is the same as in a standard processor.

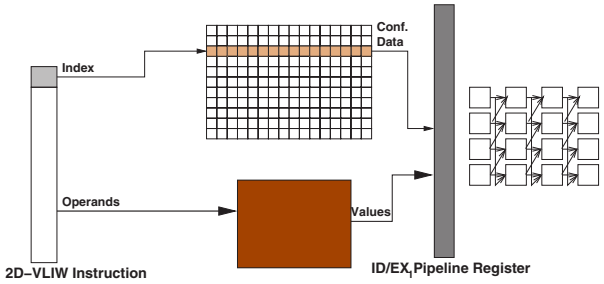


Figure 4. Decode stage of the 2D-VLIW architecture.

Figure 4 shows the decode stage where an instruction is looking for configuration bits whereas its operand registers are read from the register file. After the ID/EX_1 pipeline register has been filled in, configuration and execution activities start over the matrix. Figure 5 describes how these activities are performed at each execution stage.

Figure 5(a) represents the first execution cycle on the matrix. The first column is configured by the data from ID/EX_1 pipeline register. Four functional units from the first column are being configured to execute operations *add*, *shr*, *sub*, and *shl*. The dashed arrows indicate that FU results are being configured as source operands to FUs in the second column. The information to configure and execute the following FUs is been carried to the EX_1/EX_2 pipeline register . At the second execution cycle, 5(b), operations *sub*, *mul*, *div*, and *ld* are configured and executed. Notice that the data to configure and execute on the remainder FUs is been sent to the EX_2/EX_3 pipeline register. The target of their results is also configured. At the third execution stage, 5(c), the EX_2/EX_3 pipeline register configures the operations *shr*, *ld*, *st*, *add* and routing on the functional units. At the fourth execution stage, 5(d), operations *sub*, *addi*, *addi* and *st* are con-

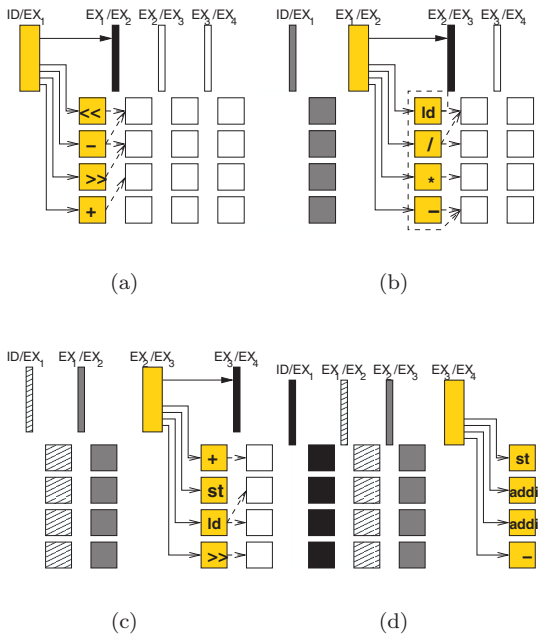


Figure 5. Configuration and execution stages on the 2D-VLIW architecture.

figured into the FUs. Following the pipeline execution, at the fourth execution cycle the matrix is totally filled with operations from four different 2D-VLIW instructions as indicated by the first, second, third, and fourth columns in different colors in 5(d). Also, notice that operations executed in 5(a), 5(b), 5(c), and 5(d) represent exactly all nodes from the DAGs in Figure 3(a).

We have compared the 2D-VLIW performance with an EPIC [8] standard processor called HPL-PD [6]. We designed both architectures through the HMDES language. Moreover, we have used the Trimaran simulator and the optimizations from the Trimaran compiler platform [1]. The results obtained showed that the 2D-VLIW speedup was 1.2-1.6 over the HPL-PD assuming 16 functional units, 32 registers and the same ISA for both architectures. The average speedup was 1.42 for 6 programs from the SPECint00 benchmark.

3 Conclusions and Future Work

A dynamic reconfiguration technique was presented in this paper. Our technique extracts configuration bits during compilation and stores these configuration bits in a configuration cache that is searched for configuration at the decode stage. Configuration data are used to program operations, inputs and routing of the functional units during the execution stage. This reconfiguration mechanism is part of our pipelined reconfigurable multiple-issue architecture named 2D-VLIW.

Currently, we are working on finishing up the implementation of the back-end phases of our own compiler framework. Also, we already have a simulator and assembler tools which implement our dynamic reconfiguration strategy using the 2D-VLIW execution model. An implementation of the architecture, along with a FPGA based reconfiguration strategy is on way.

4 ACKNOWLEDGMENTS

We would like to thank CAPES, CNPq, and Dom Bosco Catholic University (UCDB) for the financial support to this work.

References

[1] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran - an infrastructure for research in instruction-level parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2004.

[2] J. L. Cruz, A. Gonzales, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 316–325, Vancouver, 2000. ACM Press.

[3] J. A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the* 10^{th} *International Symposium on Computer Architecture (ISCA)*, pages 140–150, Los Alamitos, 1983. Computer Society Press.

[4] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–76, 2000.

[5] J. R. Hauser and J. Wawrzynek. Garp: A mips processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM97)*, pages 12–21, 1997.

[6] V. Kathail, M. S. Schlansker, and B. R. Rau. Hpl-pd architecture specification. Technical Report 93–80, Hewlett Packard Laboratories Palo Alto, February 2000.

[7] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Design & Test of Computers*.

[8] M. S. Schlansker and B. R. Rau. Epic: An architecture for instruction-level parallel processors. Technical Report 99–111, Hewlett Packard Laboratories Palo Alto, February 2000.