International Journal of Innovative Research in Information Security (IJIRIS) ISSN:2349-7017(O)
Volume 1 Issue 4 (October 2014) ISSN:2349-7009(P)
www.ijiris.com

Coarse Grain Reconfigurable Floating Point Unit

K.Moorthi * Dr.J.Raja B.Ramya
ECE & Adhiparasakthi Engg. College *ECE & Adhiparasakthi Engg. College* *ECE & Aksheya College of Engg.*

Abstract— Today the most commonly used system architectures in data processing can be divided into three categories, general purpose processors, application specific architectures and reconfigurable architectures. Application specific architectures are efficient and give good performance, but are inflexible. Recently reconfigurable systems have drawn increasing attention due to their combination of flexibility and efficiency. Re-configurable architectures limit their flexibility to a particular algorithm. This paper introduces approaches to mapping point arithmetic. After presenting an optimal formulation using applications onto CGRAs supporting both integer and floating point unit.High-level design entry tools are essential for reconfigurable systems, especially coarse-grained reconfigurable architectures.Coarse-grained reconfigurable architectures have drawn increasing attention due to their performance and flexibility. However, their applications have been restricted to domains based on integer arithmetic since typical CGRAs support only integer arithmetic or logical operations. In this project we introduce an approach to map applications onto CGRAs supporting floating point addition. The increase in requirements for more flexibility and higher performance in embedded systems design, reconfigurable computing is becoming more and more popular.

Keywords— *High-level synthesis, CGRA, FPGA, Floating point unit , Reconfigurable architecture.*

I. INTRODUCTION

Various coarse-grained reconfigurable architectures (CGRAs) have been proposed in recent years [1]–[3], with different target domains of applications and different tradeoffs between flexibility and performance. Typically, they consist is not easy to map an application onto the reconfigurable array host mostly reduced instruction set computing (RISC)] processor. The computation intensive kernels of the applications—typically loop—are mapped to the reconfigurable array while the remaining code is executed by the processor. However, it of a reconfigurable array of processing elements (PEs) and abecause of the high complexity of the problem that requires compiling the application on a dynamically reconfigurable parallel architecture, with additional complexity of dealing with complex routing resources. The problem of mapping an application onto a CGRA to minimize the number of resources giving best performance has been shown to be NP-complete [16].Few automatic mapping/compiling/synthesis tools have been developed to exploit the parallelism found in the applications and extensive computation resources of CGRAs. Some researchers [1], [4] have used structure-based or graphical user interface based design tools to manually generate a mapping, which would have a difficulty in handling big designs. Some researchers [5], [6] have only focused on instruction-level parallelism, failing to fully utilize the resources in CGRAs, which is possible by exploiting loop-level parallelism. Some researchers [7], [8], [17] have introduced a compiler to exploit the parallelism in the CGRA provided by the abundant resources. However, their approaches use shared registers to solve the mapping problem. While these shared registers explicitly during the mapping process. registers can be eliminated if routing resources are considered simplify the mapping process, theycan increase the critical path delay or latency. We show in this paper that the shared More recently, routing-aware mapping algorithms have been introduced [9]–[11]. However, they rely on step-by-step approaches, where scheduling, binding, and routing algorithms are performed sequentially. Thus, it tends to fall into a local integer type application domains, whereas ours extends the routing at the same time, thereby generating better optimized solutions. It also explicitly considers incorporating Steiner points for more efficient routing (some previous incorporate Steiner points although it is not clear whether they approaches use a kind of maze routing algorithm that can are actually incorporated). In [7], they have also presented a unified approach based on the simulated annealing algorithm. However, it takes too much time to get a solution. few minutes. Furthermore, all previous mapping/compiling/synthesis tools have been restricted to coverage to floating pointtype application domains.the approach in [7] takes hundreds of minutes whereas our optimum. Our unified approach considers scheduling and binding.

II. COARSE GRAIN RECONFIGURABLE ARCHITECTURE

CGRAs consist of an array of a large number of function units (FUs) interconnected by a mesh style network. Register files are distributed throughout the CGRAs to hold temporary values and are accessible only by a subset of FUs. The Fig.1. shows the coarse grain architecture, the FUs can execute common word-level operations, including addition, subtraction, and multiplication. In contrast to FPGAs, CGRAs have short reconfiguration times, low delay characteristics, and low power consumption as they are constructed from standard cell implementations. Thus, gate-level re-configurability is sacrificed, but the result is a large increase in hardware efficiency.

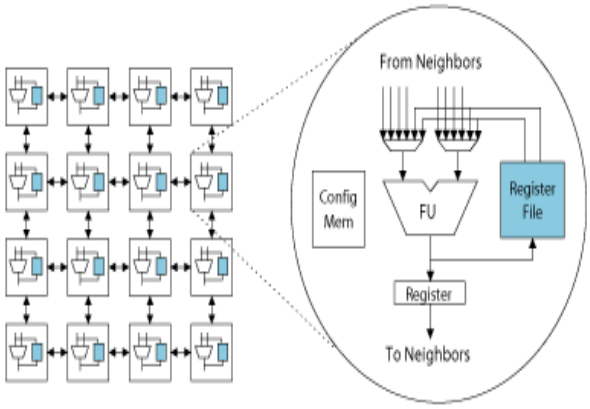

Fig.1. Reconfigurable Hardware

A good compiler is essential for exploiting the abundance of computing resources available on a GRA. However, sparse connectivity and distributed register files present difficult challenges to the scheduling phase of a compiler. The sparse connectivity puts the burden of routing operands from producers to consumers on the compiler. The Fig 2(a) shows the mesh connected processing elements the traditional schedulers that just assign an FU and time to every operation in the program are unsuitable because they do not take routability into consideration. Operand values must be explicitly routed between producing and consuming FUs. Further, dedicated routing resources are not provided. Rather, an FU can serve either as a compute resource or as a routing resource at a given time. A compiler scheduler must thus manage the computation and flow of operands across the array to effectively map applications onto CGRAs. Innermost loop of an application is selected to be executed on CGRA. With the abundance of computation resources, CGRA can efficiently execute compute intensive part of the application. Scheduling difficulties lie in the limited connectivity of CGRA. Without careful scheduling, some values can be stuck in nodes where they cannot be routed any further. Preprocessing is performed on dataflow graph of the target application. It includes identifying recurrence cycles, SSA conversion, and height calculation. Operations are then sorted by their heights and scheduling algorithm is applied to the group of operations with the same height. To guarantee the routing of values over the CGRA interconnect, scheduling space is skewed in a way that slots on the left side of CGRA are utilized first. Since slots on the left are utilized beforehand, operands can be easily routed to the right side of CGRA. An affinity graph which is constructed for each group of operations that have same height in DFG. The idea here is that placing operations with common consumers close to each other. By placing operations with common consumers close to each other, routing cost can be reduced when their consumers are placed later. Affinity values are calculated for every pair of operations in the same height by looking at how far their common consumers are located in DFG. In an affinity graph, nodes represent operations and edges represent affinity values. High affinity values between two operations indicate that they have common consumers in a short distance in DFG. In this step, operations are placed on CGRA in a way that affinity values between operations are minimized. Scheduling of operations is converted into a graph embedding problem where an affinity graph which is used in 3-D scheduling space minimizing total edge length. In the example on the left, nodes are placed in the scheduling space so that total edge length is minimized, giving more weight to edges with high affinity values. When scheduling operations in the next height, routing cost can be effectively reduced since producers of the same consumer are already placed close to each other. After getting an optimal placement of operations in one height, the scheduling space is updated the unused slot can be used in later time slots. Then, scheduler proceeds to next group of operations.

A. Architecture Extension for Floating-Point Operations

A pair of PEs in the PE array is able to compute floating-point operations according to its configuration [21]. While a PE in the pair computes the mantissa part of the floating point operations, the other PE handles the exponent part. Since the data path of a PE is 16 bits wide, the floating-point operations do not support the single precision IEEE-754 standard, but support only reduced mantissa of 15 bits. However, experiments show that the precision is good enough for hand-held embedded systems [21]. Since adjacent PEs are paired for floating-point operations [Fig. 2(c)], the total number of floating-point units that can but PE array as shown in Fig. 2(a) and (b). The PE array has enough interconnections among the PEs so that it makes use of the interconnections for the data exchange between the PEs required in floating-point operations.Mapping a floating-point operation on the PE array with integer operations may take many layers of cache. If a kernel consists of a multitude of floating-point operations, then causing costly fetch of additional context words from the main mapping on the array easily runs out of the cache layers, memory. Instead of using multiple cache layers to

perform such a complex operation, we add some control logic to the PEs so that the operation can be completed in multiple cycles cache logic control.

The layers multiple requiring without be constructed is half the number of integer PEs in the can be implemented with a small finite state machine (FSM) that controls the PE's existing datapath for a fixed number of cycles. B. Architecture Extension for Floating-Point Operations. A pair of PEs in the PE array is able to compute floating-point operations according to its configuration. While a PE in the pair computes the mantissa part of the floating point operations, the other PE handles the exponent part. Since the data path of a PE is 16 bits wide, the floating-point operations do not support the single precision IEEE- To make the RTL implementation flexible, the VHDL generic operator was used in order to control implemented features such as supported operations, number of pipeline registers, word lengths, etc. A cross-bar interconnect provides a connection from every processing elements output to every other processing element input. Partial interconnects, such as nearest neighbour, reduce the interconnect cost. The slice stage is used to split a word operand into sub-word operands and do sign extension to the operational word length. This is used to support wide accesses to memory and split operands to different processing elements which do operations in parallel. In the experimental studies, memories are 32 bits wide and slicing to 16 bits is supported also mapping.

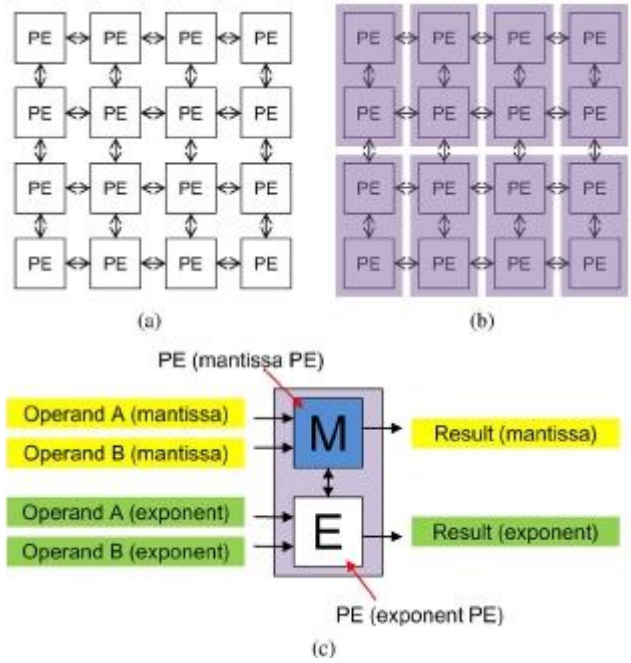

Fig.2 Mesh architecture

B. Processing Elements

To make the RTL implementation flexible, the VHDL generic operator was used in order to control implemented features such as supported operations, number of pipeline registers, word lengths, etc. Hence, instantiated PEs can be made a heterogeneous set by configuring these parameters prior to synthesis shows an example of the processing element. The stages marked at the top of the figure are controlled through a set of software controlled 32-bit wide configuration registers. These registers are written by software in order to route the datapath and configure the ALUoperation. Only PEs used in the computation need to be configured. What follows is a brief summary of the pipeline stages of the developed processing elements. The interconnect stage is used to select source operands from other processing elements, memory system, or FIFO. It is implemented with multiplexers and prior to synthesis, a setup and speed as compared to fine-grained functional blocks. This is because DSP computations in general are characterized by arithmetic operations on multi-bitwords of data. Hence, fine-grained logic blocks, such as 4-input look-up tables, give a large overhead when implementing word-level arithmetic To target reconfigurable architectures for the DSP domain, coarse-grained functional blocks have been proposed and proven more efficient in terms of area. The processing elements developed for Cpac are customized ALUs that support a set of different operations. In addition, post- and preprocessing steps such as data slicing, scaling, and truncation is included as these operations are commonly used in fixed-point data paths.

The operation stage supports a set of fundamental arithmetic and logic operations. In particular, basic operators used in standard codecs from ITU were studied in order to find suitable arithmetic operations. The basic operators are currently used in the standards for G723.1, G729, GSM enhanced full-rate, and adaptive multi-rate speech codec and use 16 and 32 bits numbers. These operators are sufficient to build computational structures such as radix-2 FFT butterfly, complex multiplication, dual and quad multiply and accumulate, accumulated mean squared error, sum of absolute

differences, and basic vector operations. The shift stage is used to scale the result after multiplication, addition, or subtraction to avoid overflow. As many processing elements are envisioned to use shifting by a fixed set of numbers, implementation of the shifter can be selected either as partial shifter, or a barrel shifter that supports all shift operations.The saturation stage is used to saturate the result from the shifter (R4) or from the operation (R3). Saturation values are software controlled through the registers min.

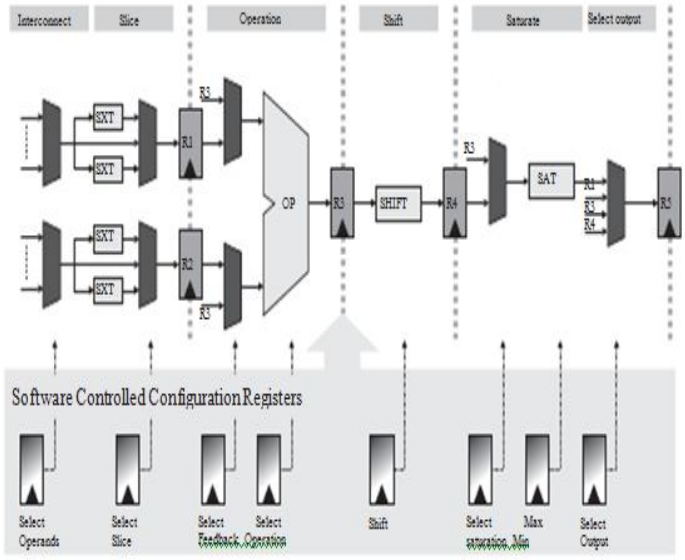

Fig 3. Data path resource.

The floating point input and output was discussed. The floating point conversion using IEEE floating point conversion is used. The Xilinx 9.1 and modelsim 6.1e student edition has been used for the simulation.The floating point conversion is used to convert from hexadecimal number to floating point number. The input is shown in the Fig 3.1 which is a single input floating point number.The floating point number is converted to hexadecimal number for the input to the floating point adder by the use of IEEE floating point conversion software. The 64 bit floating point output which is to be converted to a floating point number by the use of IEEE application software. The floating point conversion unit round off the value, then it will manipulate the hexadecimal value. The hexadecimal value is applied in the coding and the output will be in the hexadecimal. The output is then converted to floating point number by the IEEE floating point conversion unit.

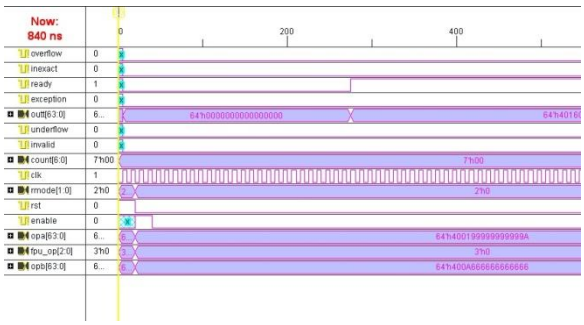

Fig 4. Floating point output

III. CONCLUSIONS

In this paper, we presented a slow but optimal approach and fast heuristic approaches to mapping of applications from multiple domains onto a CGRA supporting floating since it gives better result than spanning tree floating point operations. In particular, we considered Steiner routing. After presenting an ILP formulation for an optimal solution, we presented a fast heuristic approach based on HLS techniques that performs loop unrolling and pipelining which achieves drastic performance improvement. For ran-heuristic approach considering Steiner points finds 97% of domly generated test examples, we showed that the proposed the optimal solutions within a few seconds.

Experiments on only implementation. shown that our approach targeting a CGRA achieves up to119.9 times performance improvement compared to software various benchmark examples and 3-D graphics.

REFERENCES

[1]M. C. Filho, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," IEEE Trans. Comput.,vol. 49, no. 5, pp. 465–481, May 2000.
[2] PACT XPP Technologies [Online]. Available: http://www.pactxpp.com
[3] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in Proc. FPLA, 2003, pp. 61–70.
[4] Chameleon Systems, Inc. [Online]. Available: http://www.chameleon- "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in Proc. ICFPT, Dec. 2002, pp. 166–173.
[5] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for systems.com [1] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E.
[6] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S.reconfigurable computing," in Proc. IWFPL, 1998, pp. 248–257.
[7] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins,P. Amarasinghe, "Space-time scheduling of instruction level parallelismon a RAW machine," in Proc. ASPLOSV, 1998, pp. 46–57.
[8] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, "High-level synthesis challenges and solutions for a dynamicallyreconfigurable processor," in Proc. ICCAD, Nov. 2006, pp. 702–708.
[9] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H. Kim, "Edgecentric modulo scheduling for coarse-grained reconfigurable architec-tures," in Proc. PACT, Oct. 2008, pp. 166–176.
[10] S. Friedman, A. Carroll, B. V. Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An architecture-adaptive CGRA mapping tool," in Proc.FPGA, 2009, pp. 191–200.based spatial mapping algorithm for coarse-grained reconfigurable
[11] J. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek, "A graph draw-architecture," IEEE Trans. Very Large Scale Integr. Syst., vol. 17, no.11, pp. 1565–1578, Jun. 2008.
[12] Y. Ahn, K. Han, G. Lee, H. Song, J. Yoo, and K. Choi, "SoCDAL:System-on-chip design accelerator," ACM Trans. Des. Automat. Electron.Syst., vol. 13, no. 1, pp. 171–176, Jan. 2008.
[13] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing specific optimization," in Proc. DATE, Mar. 2005, pp. 12–17. and pipelining in coarse-grained reconfigurable architecture for domain.
[14] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configurationcache structure and code mapping for coarse-grained reconfigurable architecture," in Proc. ISLPED, Oct. 2006, pp. 310–315.
[15] The SUIF Compiler System [Online]. Available: http://suif.stanford.edu
[16] C. O. Shields, Jr., "Area efficient layouts of binary trees in grids," Ph.D.
[17] G. Lee, S. Lee, and K. Choi, "Automatic mapping of application to dissertation, Dept. Comput. Sci., Univ. Texas, Dallas, 2001. techniques," in Proc. ISOCC, Nov. 2008, pp. 395–398. coarse-grained reconfigurable architecture based on high-level synthesis
[18] K. Han and J. Kim, "Quantum-inspired evolutionary algorithms with a new termination criterion, Hε gate, and two phase scheme," IEEE Trans. Evol. Computat., vol. 8, no. 2, pp. 156–169, Apr. 2004.
[19] GNU Linear Programming Kit [Online]. Available: http://www.gnu.org/
[20] G. Lee, S. Lee, K. Choi, and N. Dutt, "Routing-aware application mapping considering Steiner point for coarse-grained reconfigurable algorithms," in Proc. FPL, 2005, pp. 317–322.
[21] M. Jo, V. K. P. Arava, H. Yang, and K. Choi, "Implementation of floating-point operations for 3-D graphics on a coarse-grained reconfigurable architecture," in Proc. IEEE-SOCC, Sep. 2007, pp. 127–130.
[22] Z. Baidas, A. D. Brown, and A. C. Williams, "Floating-point behavioral synthesis," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.20, no. 7, pp. 828–839, Jul. 2001.

