Pipeline Scheduling with Input Port Constraints for an FPGA-based Biochemical Simulator

Tomoya Ishimori[1], Hideki Yamada[1], Yuichiro Shibata[1], Yasunori Osana[2], Masato Yoshimi[3], Yuri Nishikawa[3], Hideharu Amano[3], Akira Funahashi[4], Noriko Hiroi[5], and Kiyoshi Oguri[1]

[1] Department of Computer and Information Sciences, Nagasaki University
[2] Department of Computer and Information Science, Seikei University
[3] Department of Information and Computer Science, Keio University
[4] Department of Biosciences and Informatics, Keio University
[5] EMBL-EBI
`bio@pca.cis.nagasaki-u.ac.jp`

Abstract. This paper discusses design methodology of high-throughput arithmetic pipeline modules for an FPGA-based biochemical simulator. Since limitation of data-input bandwidth caused by port constraints often has a negative impact on pipeline scheduling results, we propose a priority assignment method of input data which enables efficient arithmetic pipeline scheduling under given input port constraints. Evaluation results with frequently used rate-law functions in biochemical models revealed that the proposed method achieved shorter latency compared to ASAP and ALAP scheduling with random input orders, reducing hardware costs by 17.57 % and by 27.43 % on average, respectively.

1 Introduction

The importance of biochemical simulation is rising in the context of systems biology, which aims at understanding biological processes in a system level. While various biochemical simulators intended for whole-cell simulation have been developed[2, 3], such large scale software simulators require both a fair amount of time and large computing resources. To overcome this problem, we have developed an FPGA-based biochemical simulator called ReCSiP (Reconfigurable Cell Simulation Platform)[1].

ReCSiP executes high-throughput simulation of biochemical model based on ordinary differential equations with custom hardware solvers tailored for given target models. The solvers are essentially deeply pipelined arithmetic function modules to calculate velocity of chemical reactions according to rate-law functions. While high-throughput design of solvers directly improves the overall performance of simulation, compact design allows higher degree of parallel execution with multiple solvers. So far, a wide range of research activities on high-level synthesis techniques that generate high throughput pipeline modules have been carried out [4–9]. However, the following two issues seem to be rarely addressed while they need to be taken into account in practical reconfigurable computing platforms like ReCSiP.

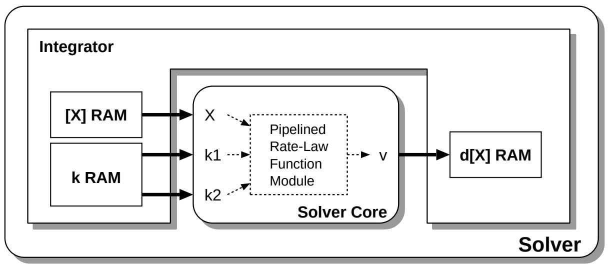

Fig. 1. Structure of a Solver

1. Due to data input bandwidth limitation for pipeline modules, all of input data required to calculate the arithmetic function cannot be fed at the same time.
2. The input data may not necessarily be able to be fed from any input ports. For example, each solver in ReCSiP has three input ports; one for variables conveying chemical concentration and two for variables of reaction rate coefficients. Any concentration variables can not be fed through ports for coefficients. Such architectural constraints effectively reduces switching costs, enabling high operational frequency.

Therefore, scheduling of the data input sequence also plays an important role in terms of the trade-off between performance and hardware costs of solvers.

2 ReCSiP simulator and solver cores

An FPGA on ReCSiP contains ordinary differential equation *Solvers*, switch modules for inter-solver data communication, and PCI interface. Each Solver consists of a *Solver Core* module to calculate the change in concentrations of substances for each time step and an *Integrator* module to perform numerical integration and simulation control. Figure 1 shows the structure of a Solver.

A Solver Core, which is a statically scheduled pipeline module with single precision floating point arithmetic, calculates and outputs a reaction rate v using data from three input ports; one for concentrations of substances and two for reaction coefficients. The structure of Solver Core varies depending on the rate-law function to be calculated. An Integrator has three sets of memories for concentrations (corresponding to [X] RAM), for rate coefficients of each function (k RAM), and for concentration changes occurred by reactions (d[X] RAM). An Integrator performs numerical integration to calculate concentration changes for each time step using Solver Cores.

Since data-input bandwidth constraints to Solver Cores often prevent all the data required by the reaction from being initiated at once, those data should be ordered so as to mitigate arithmetic units' stalls by the inputs. Here, we define pipeline pitch P as the number of clock cycles to complete input. In the case that the value of P is 2 or more, idle clock cycles on arithmetic units might arise. That is, each arithmetic unit operates once every P clock cycles. This control is easily carried out by providing a cyclic state counter with a range of the pipeline pitch. The same arithmetic operators which are scheduled in different states can be shared into the same arithmetic module.

3 Port constrained scheduling

3.1 Data-input scheduling

In our approach, we divide scheduling process into two tasks; we first assign a priority to each input data for a Solver Core based on structural information obtained by the corresponding DFG, then performs pipeline scheduling of minimize the conflicts of arithmetic units. We determine input timing of each data in the descending order of the following three priority metrics. Let us consider the following rate-law function:

$$v = \frac{K_a X_a/(X_a/K_b)}{1 + X_a/K_b + K_c/X_b} \tag{1}$$

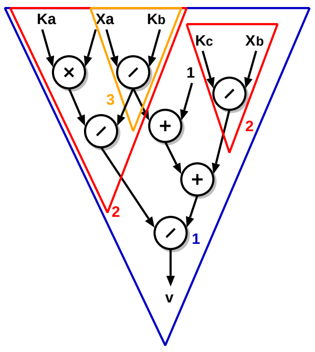

Fig. 2. DFG and subtrees for Function (1)

where X_a, X_b are input through the X port and K_a, K_b, K_c are fed by two K ports.

(1) Usage frequency: Assume that m arithmetic units need the same input data D_m, and n arithmetic units do D_n. When m is larger than n, it might be better to make the priority of D_m higher than that of D_n. To fill in the details, input of D_m allows m arithmetic units to be *half-ready*, in which a node waits for the other input data to initiate the operation, and the same holds true for D_n. Here, D_m could have the higher probability for the beginning of operations than D_n by the assumption, just $m \geq n$. Consequently, giving the higher priority to the input data used on more arithmetic units, might be preferable. Incidentally, usage frequency of input data corresponds directly to out-degree, or the number of edges leaving the node. Taking Function (1) for example, priorities come to be $\{\{X_a\}, \{X_b, K_a, K_b, K_c\}\}$.

(2) The number of successors: Obviously, making the data on the critical path of the DFG input first is better to minimize the latency. The precise critical path, however, can be changed by scheduling of the data input order itself. An easy way to predict the critical path is to use the sum of latencies of arithmetic units along the path, but this idea does not consider the stall time. Since each arithmetic unit may have different stall time, the path whose sum of latencies is small could be critical. Therefore, we propose to use the number of successors from primary input to primary output. The number of successors roughly corresponds to the number of operations performed until the primary output. Taking Function (1) for example, priorities come to be $\{\{X_a, K_b\}, \{X_b, K_a, K_c\}\}$.

(3) Divider subtrees: In general, dividers often have the largest latency among the basic arithmetic units. In fact, the latency of the divider is about 5 times larger than that of the add-subtractor and multiplier in the current ReCSiP design. Therefore, we propose to use the idea of subtrees headed by dividers as shown in Figure 2. To begin with the primary output v, its immediate predecessors are scanned to find dividers. Every time a divider is found, a new subtree is made toward the primary inputs. Finally, the priority of input data is assigned based on how many divider subtrees cover the corresponding the input data. Taking Function (1) for example, priorities come to be $\{\{X_a, K_b\}, \{X_b, K_a, K_c\}\}$.

3.2 Arithmetic pipeline scheduling

As a method for arithmetic scheduling, a wide varieties of algorithms such as the ASAP (As Soon As Possible) /ALAP (As Late As Possible) scheduling [4], heuristic optimal FDS (Force-Directed Scheduling) [5, 6], a globally optimal ILP (Integer Linear Programming) [7], and List-based scheduling using DFG information [8, 9] have been proposed. Our algorithm is based on the List-based scheduling and consists of the following steps:

1. Calculate the mobilities of each operation using the ASAP/ALAP schedule. The *mobility* is defined as the difference between the ASAP time and ALAP time of the operation. Make a *ready-list* (priority queue), which is always sorted with respect to a *priority function.* The operation with a smaller mobility has a higher priority.
2. Iterate Step 3 and 4 to complete the scheduling of all operations.
3. Determine the operation schedule according to the priority of a list, avoiding the same initiation states among operations.
4. Update a list and delete scheduled data from it. ASAP time and ALAP time of operations whose predecessors or successors have been scheduled, are changed. Then, these changed values are used to re-calculate the mobilities.

The hardware cost reduction is tried by using available information from the rate-law function of the corresponding DFG, keeping the latency of the Solver Core. The framework of this algorithm is similar to that of FDS, but complicated calculation process such as *force* is not necessary in our method.

4 Evaluation

The algorithm was implemented in C++ (gcc 4.0.0) and the steps from DFG generation to Verilog-HDL file generation are automated. To evaluate the quality of our method, it was applied on 18 rate-law functions of the 33 SBML (System Biology Markup Language) pre-defined functions. The generated Verilog files were mapped on a XC2VP70-6FF1517 FPGA, using the Xilinx ISE 8.2i tool. Latencies of adders, multipliers, and dividers used in this implementation are 5, 5, and 27 clock cycles, respectively.

Table 1. Implementation results and comparison to basic scheduling (N: operation count, L: latency, F: frequency, T: throughput, S: slice count)

Function	N	Basic ASAP				Basic ALAP				Proposed			
		L	F(MHz)	T(Mrps)	S	L	F(MHz)	T(Mrps)	S	L	F(MHz)	T(Mrps)	S
UCTR	15	71	123.9	41.3	2859	71	125.1	41.7	3605	71	124.3	41.4	2664
UMAR	15	71	123.9	41.3	2859	71	125.1	41.7	3605	71	124.3	41.4	2647
UMR	15	71	123.9	41.3	2858	71	125.1	41.7	3605	71	124.3	41.4	2647
UNIR	15	71	123.9	41.3	2858	71	125.1	41.7	3605	71	124.3	41.4	2663
UCTI	10	70	141.0	70.5	3356	70	141.0	70.5	2716	69	141.5	70.7	3285
UMAI	10	70	141.0	70.5	3356	70	141.0	70.5	2716	69	141.5	70.7	2570
UMI	10	70	141.0	70.5	3356	70	141.0	70.5	2716	69	141.5	70.7	2569
UNII	10	70	141.0	70.5	3356	70	141.0	70.5	2716	69	141.5	70.7	3285
UUCI	8	70	139.3	69.6	2513	70	139.3	69.6	2044	70	139.3	69.6	2028
UUCR	13	71	124.6	41.5	2842	71	126.0	42.0	3574	71	124.9	41.6	2090
UAII	7	64	138.1	69.0	2380	64	139.0	69.5	1728	64	138.1	69.0	2395
UAR	12	70	127.2	42.4	3309	70	124.9	41.6	2764	69	125.3	41.7	2623
UCII	7	64	138.1	69.0	2380	64	139.0	69.5	1728	64	138.8	69.4	2395
UCIR	12	70	127.2	42.4	3309	70	124.9	41.6	2764	69	125.1	41.7	2623
ORDBBR	34	93	108.1	21.6	9476	93	121.5	24.3	15664	90	119.8	31.9	6519
ORDBUR	18	82	122.0	30.5	3077	82	124.1	31.0	2501	79	122.3	30.5	2349
ORDUBR	18	76	124.5	31.1	2600	76	122.9	30.7	2420	74	123.9	30.9	2890
PPBR	25	83	115.7	23.1	6255	83	121.9	24.3	11085	79	113.1	22.6	3903

Table 2. Results with modified ASAP/ALAP scheduling (L: latency, F: frequency, T: throughput, S: slice count)

Function	Modified ASAP				Modified ALAP			
	L	F(MHz)	T(Mrps)	S	L	F(MHz)	T(Mrps)	S
UCTR	71	123.8	41.2	2780	71	125.1	41.7	3621
UMAR	71	123.8	41.2	2780	71	125.1	41.7	3621
UMR	71	123.8	41.2	2780	71	125.1	41.7	3621
UNIR	71	123.8	41.2	2780	71	125.1	41.7	3621
UCTI	69	141.0	70.5	3372	69	141.0	70.5	2716
UMAI	69	141.0	70.5	2569	69	141.0	70.5	2716
UMI	69	141.0	70.5	2569	69	141.0	70.5	2716
UNII	69	141.0	70.5	3372	69	141.0	70.5	2716
UUCI	70	139.3	69.6	2044	70	139.3	69.6	2060
UUCR	71	126.3	42.1	2175	71	126.0	42.0	3590
UAII	64	138.1	69.0	2380	64	139.0	69.5	1728
UAR	69	126.3	42.1	3409	69	124.9	41.6	2749
UCII	64	138.1	69.0	2380	64	139.0	69.5	1728
UCIR	69	126.6	42.2	2596	69	124.9	41.6	2749
ORDBBR	90	109.6	21.9	6928	90	121.5	24.3	15573
ORDBUR	79	124.5	31.1	5147	79	124.1	31.0	2453
ORDUBR	74	122.8	30.7	2588	74	122.9	30.7	2404
PPBR	79	109.5	21.9	5942	79	121.7	24.3	11041

Table 1 shows the implementation results of *basic* ASAP/ALAP scheduling and the proposed scheduling. Here, *basic* means data-input timing is determined in a random order. In determining data-input timing for the proposed scheduling, the highest priority is given to the number of successors, and then the number of divider subtrees gets preference over the usage frequency. Compared to the ASAP/ALAP, the proposed algorithm achieved the best latency for every function, reflecting the effectiveness of our data-input scheduling policy. In terms of the frequency and throughput (reactions per second), the achieved performance was almost same, while the required slice count was reduced by 17.57 % on average, showing considerable trade-off. The average hardware reduction rate of 27.43 % was achieved, affecting calculation throughput only by 0.5%.

Furthermore, to evaluate the effect of arithmetic scheduling, we compared our algorithm to *modified* ASAP scheduling and *modified* ALAP scheduling. The results were summarized in Table 2. Here, *modified* means that data-input timing is determined in the same way as the proposed algorithm. Unlike the basic scheduling algorithms, any differences in latency were not observed for the modified algorithms and the proposed algorithm. This implies scheduling of data-input timing is quite important for achieving short latency. In terms of required slices, the proposed algorithm achieved the reduction rates of 11.37 % and 27.3 % on average compared to ASAP and ALAP, respectively. This reduction comes from arithmetic scheduling algorithm itself. In addition, the differences in the reduction rates between the basic and modified algorithms suggest that scheduling of data-input timing is also a significant factor to reduce the hardware costs.

5 Conclusion

In this paper, a priority assignment method for data-input to enable efficient arithmetic pipeline scheduling with input bandwidth constraints is proposed. Evaluation results with the SBML pre-defined functions revealed that the proposed method achieved shorter latency compared to randomly input ASAP and ALAP scheduling, reducing hardware costs by 17.57 % and by 27.43 % on average, respectively. Our future work includes further investigating of the trade-offs among the priority parameters with large scale graphs.

References

1. Y. Osana et al. “ReCSiP: An FPGA-based general-purpose biochemical simulator”, *Elect. and Communications in Japan,*, Part 2, Vol. 90, no. 7, pp. 1–10, Jul. 2007
2. M. Tomita et al. “E-Cell: software environment for whole cell simulation”, *Bioinformatics*, Vol. 15, no. 1, pp. 72–84, Jan. 1999
3. Ion I. Moraru et al. “The virtual cell: an integrated modeling environment for experimental for and computational cell biology”, *Annals of the New York Academy of Sciences*, Vol. 971, pp. 595–596, 2002
4. Robert A. Walker, Samit Chaudhuri, “Introduction to the Scheduling Problem”, *IEEE Design and Test of Computers*, pp. 60–69, Summer, 1995
5. P.G. Paulin, J. Knight, “Algorithm for High-Level Synthesis”, *IEEE Design & Test of Computers*, Vol. 6, No. 6, pp. 18-31, Dec. 1989
6. P.G. Paulin, J. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASICs”, *Trans. CAD*, Vol. 8, No. 6, pp. 661-679, Jun. 1989
7. Cheng-Tsung Hwang et al, “A Formal Approach to the Scheduling Problem in High Level Synthesis”, *IEEE Trans. CAD*, Vol. 10, No. 4, pp. 464–475, Apr. 1991
8. Sriram Govindarajan and Ranga Vemuri, “Cone-Based Clustering Heuristic for List-Scheduling Algorithms”, *Proc. EDTC*, Paris, France, pp. 456–462, 1997
9. Azeddien M. Sllame and Vladimir Drabek, “An Efficient List-Based Scheduling Algorithm for High-Level Synthesis”, *Proc. DSD*, pp. 316–323, 2002