Code Generation for STA Architecture

J. Guo, T. Limberg, E. Matus, B. Mennenga, R. Klemm, and G. Fettweis

Vodafone Chair Mobile Communication Systems, T.U.Dresden, Germany
{guojie, limberg, matus, mennenga, klemm, fettweis}@ifn.et.tu-dresden.de

Abstract. This paper presents a novel compiler backend which generates assembly code for Synchronous Transfer Architecture (STA). STA is a Very Long Instruction Word (VLIW) architecture and in addition it uses a non-orthogonal Instruction Set Architecture (ISA). Generating efficient code for this architecture needs highly optimizing techniques. The compiler backend presented in this paper is based on Integer Linear Programming (ILP). Experimental results show that the generated assembly code consumes much less execution time than the code generated by traditional ways, and the code generation can be accomplished in acceptable time.

1 Introduction

Code generation has been the focus of many research works. In order to generate efficient code for irregular architectures, Integer Linear Programming (ILP) modelling for code generation has been explored extensively in the recent past.

Kent Wilken et al. [1] developed an instruction scheduling model and efficient basic block partition. Timothy Kong et al. [4] developed a register allocation model for regular and irregular architectures. Because the interdependencies between the phases in code generation may lead to a significant decrease of code quality, Daniel Kästner et al. [5] built two sets of ILP formulations for phase-coupled code generation. Besides using some ideas of Wilken and Kästner, our ILP model is built in order to be aware of the STA features.

This paper is organized as follows: Section 2 gives a brief introduction to the STA features. Section 3 and 4 explain the compiler backend in detail. The results of our experimental evaluation are summarized in Section 5. Section 6 concludes and gives an outlook.

2 Synchronous Transfer Architecture (STA)

Figure 1 gives an overview over the STA concept. STA [6] processors are built up from modules, each with a set of input and output ports. The output ports are buffered. The buffer at the output holds the result of the last operation, until the next operation of the belonging module is executed. The data at an input port is selected from a set of connected output ports by a multiplexer. Thus, data can be obtained directly from an output register of connected FU, which

lowers the requirement for additional register strongly. This kind of data transfer will be called *direct data routing* (DDR) in the rest of this paper. Direct data routing can dramatically reduce the amount of required registers in a register file.

For the sake of generality and simplicity, register files and memory read or write ports are also implemented as STA modules. Each module behavior is fully qualified by an opcode and multiplexer control lines. The opcode controls the operation on the module. Multiplexer control lines select the input operands. The opcodes and multiplexer control lines for all modules are aligned in a VLIW.

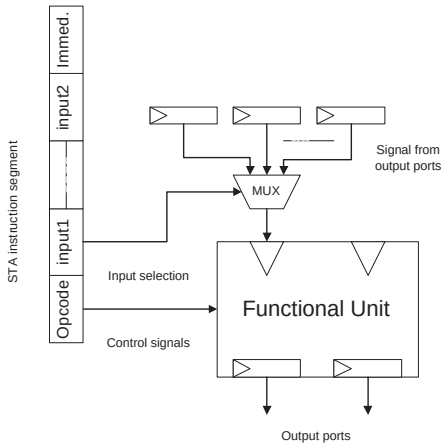

Fig. 1. STA modules

Different from common architectures, minimizing register access in STA can result in much better results. The operands can't be made available one single clock cycle after storing them. When a FU uses a value in hardware register, the value is transferred to the read port of register file in the first clock cycle. In the next cycle, the FU can select this read port as its input and does operation. Due to the limited number of register read ports, additional wait cycles may be introduced, if more operands need to be read than ports are available.

3 Analysis for Integer Linear Programming (ILP)

Our ILP-based compiler backend can be divided into several stages: Firstly, an ILP model is built, which includes a set of formulations for all possible states. In a second step the useful data (exact value of the variables in formulations) is created from Medium-Level Intermediate Representation (MIR). Each kind of data is used by one or more different formulations in the model. Then the solvable formulations are generated and the results will be found by using CPLEX (a software for solving different kinds of optimization problems). In a final step, a

program transforms the results of CPLEX into assembly code. The developed ILP model is restricted to basic block [7]. The proposed process of the data file generation consists of the following six phases:

3.1 Memory Spill Code Generation

Firstly, we made a global data flow analysis [7] to find the IN and OUT data for each basic block. Then all the registers are divided into two groups: group I is used to handle IN and OUT data and group II is used for register operations, which manipulate the temporary results within one basic block.

Our simulations with many different basic blocks have shown that *data direct routing* is mostly used in the results of CPLEX, so the number of register operations used for saving temporary results, which appear only in one basic block, is very small. Thus in our current implementation, we assigned 80% of all available registers to group I and the remaining 20% to group II. If the amount of registers in group I is not enough, memory spill code (load and store) will be generated in MIR. Otherwise, the remaining registers in addition to the registers of group II can be used in order to determine an optimal tradeoff between *direct data routing* or register access within each basic block.

3.2 MIR Analysis

Suppose instruction i_1 and i_2 are in the same basic block of MIR, and i_1 appears before i_2 and they may have one of the following dependence:

1. i_1 writes a location that i_2 reads (RAW)
2. i_1 reads a location that i_2 writes (WAR)
3. i_1 writes a location that i_2 writes (WAW)
4. i_1 reads a location that i_2 reads (RAR)

For guaranteeing the correctness of the assembly code, the first two kinds of dependencies are considered in the scheduling. The third dependence WAW is avoided by location renaming [7] in our MIR. Assume $\mathbb{I}$ is the set of all the instructions in the MIR, then these two kinds of dependence can be observed as:

1. Set $\mathbb{DDR\_U} \subset \mathbb{I} \times \mathbb{I}$ contains for all read after write instruction pairs (possible to be used for data direct routing of STA)
2. Set $\mathbb{U} \subset \mathbb{I} \times \mathbb{I}$ contains for all write after read instruction pairs (necessary to be considered in the scheduling)

In the rest of this paper we refer to these two coupled instructions in a pair of the first category as a *DDR pair*.

3.3 Inter-blocks Read and Write Analysis

Since register allocation is done for IN and OUT data in the first step, the following data sets are defined:

1. $\mathbb{IN}$ (the input data in registers come from other basic blocks or from itself in the last iteration)
2. $\mathbb{OUT}$ (the data in registers which will be as the input data in the other basic blocks or itself in the next iteration)
3. $\mathbb{U}_{\mathbb{IN}} \subset \mathbb{RI} \times \mathbb{I}$ (dependence between inter-block register read instructions and MIR instructions)
4. $\mathbb{U}_{\mathbb{OUT}} \subset \mathbb{I} \times \mathbb{WI}$ (dependence between MIR instructions and inter-block register write instructions)

3.4 Inner-blocks Read and Write Analysis

We assume there are always a pair of virtual read instruction $r_{u,v}$ and write instruction $w_{u,v}$ between DDR pair (i_u, i_v). There are two possibilities: only one instruction uses the result of i_u (Case 1) and more than one instructions use the result of i_u (Case 2).

In case 2, if we consider these *DDR pairs* independently, more than one write instructions may take place. Actually, it is enough to have only one write instruction, because the data from i_u exists already in the register after the first write instruction. In this case, only one register write instruction is generated in our compiler. If a true register write exists, a new read instruction is generated directly before each instruction which uses i_u's result in our compiler. Read instructions can be only executed on the read port. If the data is stored in the register's read port until last use, this read port cannot perform other read instructions for several clocks. It may decrease the instruction level parallelism.

3.5 Machine Resource Analysis

One of the key concepts in our model is the *execution time* of the instructions. One type of FUs have same execution time. Different operations which can be assigned to the same type of FU have the same execution time. The following parameters are predefined:

- E_u: the execution time of instruction i_u, $i_u \in \mathbb{I} \cup \mathbb{RI} \cup \mathbb{WI}$.
- $E^w_{u,v}$/ $E^r_{u,v}$: the execution time of the virtual write/ read in a *DDR pair* (i_u, i_v).

The machine resource is divided into three classes in our model. There are Functional Units (FUs), the read and write ports of register, and registers. Each instruction can be explicitly matched into one type of FU in our architecture. According to the type of FUs, we divided instructions in $\mathbb{I}$ into some FUi sets.

4 Integer Linear Programming(ILP) Model

Let C be the number of clock cycles needed for the execution of all the very long instruction words in a basic block, then our optimization model is explained as follows:

- **Objective:** minimizing C
- **Self Constraint** This constraint ensures each instruction is scheduled exactly once in the basic block scheduler.
- **Data Routing Constraint** It describes the state of each FU with the consideration of RAW dependence.
- **Machine Constraint** This constraint guarantees that the scheduling is performed without exceeding machine resources in each cycle.
- **Dependence Constraint** It describes the WAR dependence.

4.1 Self Constraint

For an instruction $i_u \in \mathbb{I}$, we define a series of binary variables $x(u,j), j \in \mathbb{J}$, where $\mathbb{J}$ is the set of all clock cycles: $1..C$. The value of $x(u,j)$ is 1 when i_u is scheduled to start in cycle j, otherwise 0.

Equation (1) guarantees that each instruction i_u must be started and can be only started once.

$$\sum_{j=1}^{C} x(u,j) = 1, \forall i_u \in \mathbb{I} \tag{1}$$

In the same way, we define the corresponding binary variables for inner-block read and write instructions: $r(u,v,j), w(u,v,j), (i_u,i_v) \in \mathbb{DDR\_U}, j \in \mathbb{J}$. Between each *DDR pair*, only one read and one write instruction may start.

4.2 Data Routing Constraint

Observing a *DDR pair* $(i_u,i_v) \in \mathbb{DDR\_U}$ as shown in Figure 2 and 3. At first, i_v uses the result of i_u, so i_v should start after i_u is finished. Because the data transfer in our STA architecture is synchronous, i_v can also use the result of i_u in the same cycle i_u is finished. The equation (2) describes this constraint.

$$\sum_{j=1}^{C} x(u,j) * j + E_u \leq \sum_{j=1}^{C} x(v,j) * j, \qquad (i_u,i_v) \in \mathbb{DDR\_U} \tag{2}$$

Secondly, the write instruction $w_{u,v}$ must start in the cycle when it can get the result of instruction i_u. The equation (3) describes this constraint.

$$\sum_{j=1}^{C} x(u,j) * j + E_u \leq \sum_{j=1}^{C} w(u,v,j) * j, \qquad (i_u,i_v) \in \mathbb{DDR\_U} \tag{3}$$

Thirdly, as explained in Section 3.3, the read instruction $r_{u,v}$ is placed directly before instruction i_v. Equation (4) is used to describe this constraint.

$$\sum_{j=1}^{C} r(u,v,j) * j + E^r_{u,v} = \sum_{j=1}^{C} x(v,j) * j, \qquad (i_u,i_v) \in \mathbb{DDR\_U} \tag{4}$$

With the above three constraints, there are only five possible cases according to the sequence of virtual write and read instructions:

1. $w_{u,v}$ is at least $E^w_{u,v}$ cycles before $r_{u,v}$ (Fig2.a)
2. $w_{u,v}$ is in the same cycle as i_v (Fig2.b)
3. $w_{u,v}$ appears after i_v (Fig3.a)
4. $w_{u,v}$ is in one of the cycle from the same cycle of $r_{u,v}$ to one cycle before i_v (Fig3.b)
5. $w_{u,v}$ is before $r_{u,v}$, but less than $E^w_{u,v}$ cycles (Fig3.c)

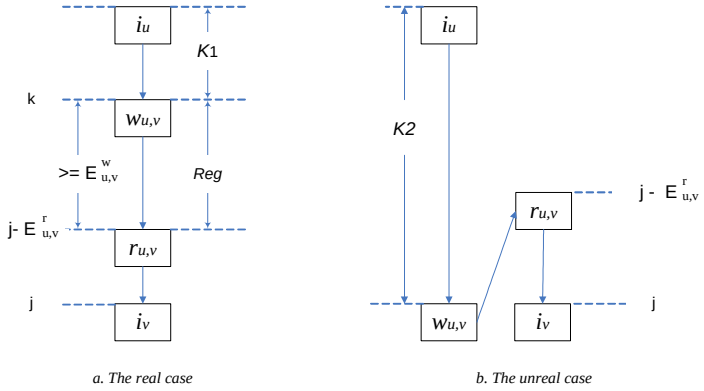

Fig. 2. The real and unreal case in our model

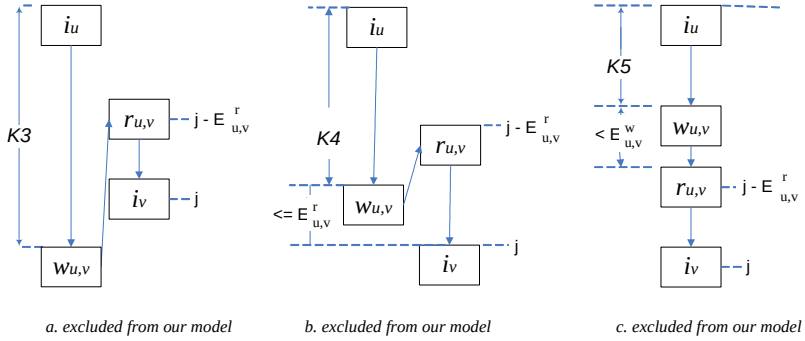

Fig. 3. The excluded cases in our model

Because a feasible write operation must save the data in the register before any read operation can access it, case 1 (Fig2.a) is considered as *real* and register

is used to transfer the result of i_u to i_v. Other cases are all defined as *unreal*, but only case 2 (Fig2.b) is allowed and *direct data routing* is used to transfer the result. Case 3, 4 and 5 (Fig3) are excluded for conciseness of the modelling.

For excluding case 3 (Fig3.a), we further define that the read instruction can appear at most $E^r_{u,v}$ cycles before the corresponding write instruction. Equation (5) is used to represent this constraint. Equation (4) has defined that $r_{u,v}$ is exactly $E^r_{u,v}$ cycles before i_v. So $w_{u,v}$ can not appear after i_v.

$$\sum_{j=1}^{C} w(u,v,j) * j - E^r_{u,v} \leq \sum_{j=1}^{C} r(u,v,j) * j, \qquad (i_u, i_v) \in \mathbb{DDR\_U} \tag{5}$$

Equation (6) and (7) are used to exclude case 4 (Fig3.b) and case 5 (Fig3.c) respectively.

$$w(u,v,j) + r(u,v,j+l) \leq 1, \quad (i_u, i_v) \in \mathbb{DDR\_U}, \forall j \in \mathbb{J}, l \in (0..E^r_{u,v} - 1) \tag{6}$$

$$w(u,v,j+l) + r(u,v,j) \leq 1, \quad (i_u, i_v) \in \mathbb{DDR\_U}, \forall j \in \mathbb{J}, l \in (0..E^w_{u,v} - 1) \tag{7}$$

Thus, the occupation of FU can be uniformly expressed with the remaining two cases by relative concise ILP equations. In $K1$ cycles (Fig 2.a), the corresponding FU is *occupied* until a register write instruction for saving the result of i_u appears. If $w_{u,v}$ appears in the same cycle as i_v (Fig 2.b), we define that the result of i_u is transferred to i_v by *direct data routing*, and the FU is occupied until i_v appears. Detailed analysis and other machine resource constraints are discussed in the Section 4.3. The formulations for the case that more than one instructions use the result of i_u are the extension of the above equations. They will also be explained together with the machine resource in the Section 4.3.

4.3 Machine Resource Constraint

The Constraints for Functional Units

There are two cases that one of the FU is occupied: an instruction is executed on this FU or the output register of this FU should be kept. We define a new set of binary variables $D_{link}(u,j)$, $j \in \mathbb{J}$ to represent the occupation of a FU. $D_{link}(u,j)$ takes the value of 1 if the FU is occupied in cycle j, otherwise 0. Because the coming instruction on the same FU has the same execution time of i_u and STA has pipeline structure, so this FU can be available for this instruction $E_u - 1$ cycles before write instruction takes place.

$$D_{link}(u,j) \geq \sum_{k=1}^{j} x(u,k) - \sum_{k=1}^{j+E_u-1} w(u,v,k) \quad \forall (i_u, i_v) \in \mathbb{DDR\_U}, \forall j \in \mathbb{J} \tag{8}$$

Equation (8) can be considered as a stair function with j as X-axis and the lower bound of $D_{link}(u,j)$ as Y-axis. $D_{link}(u,j)$ must take the value 1 from the cycle of i_u to $E_u - 1$ cycles before $w_{u,v}$ takes place, while not restricted in other cycles. In every cycle, the number of occupied functional units is bounded by the total available number N_{FUi} in the architecture. Thus, the equation of resource constraint for each type of FU can be derived:

$$\sum_{i_u \in FUi} D_{link}(u,j) \leq N_{FUi}, \forall j \in \mathbb{J} \tag{9}$$

The Constraints for Register Read and Write Ports

The access of register in each cycle is limited by the read and write ports. In Section 4.2, we have defined the *real* and *unreal* cases for the read and write instructions. Then, two sets of binary variables, $w_{real}(u,v,j)$ and $r_{real}(u,v,j), \forall j \in \mathbb{J}$, are defined to describe the usage of read and write ports in these two cases. The value 1 represents a real read/write operation in cycle j, and 0 means no register operation.

The equation (10) checks if a read instruction takes place before the corresponding write instruction. If so, $w_{real} \geq 0$, there is no restriction for this binary variable. The constraint $w_{real} \geq 1$ can only occur where $w(u,v,j) = 1$ and the read instruction appears after write.

$$w_{real}(u,v,j) \geq w(u,v,j) - \sum_{k=1}^{j-E^r_{u,v}} r(u,v,k), \forall j \in \mathbb{J}, \forall (i_u, i_v) \in \mathbb{DDR\_U} \tag{10}$$

In the same way, the equation (11) checks if a write instruction appears in the previous cycle of where the read instruction is.

$$r_{real}(u,v,j) \geq r(u,v,j) - \sum_{k=j+E^r_{u,v}}^{C} w(u,v,k), \forall j \in \mathbb{J}, \forall (i_u, i_v) \in \mathbb{DDR\_U} \tag{11}$$

If more than one instructions use the result of i_u, a set of binary variables $w_b(u,j), \forall j \in \mathbb{J}$ is introduced to trace the first write instruction (equation (12)), and the write instruction should still be placed after the instruction i_u (equation (13)).

$$\sum_{j=1}^{C} w_b(u,j) * j \leq \sum_{j=1}^{C} w_{real}(u,v,j) * j, \forall (i_u, i_v) \in \mathbb{DDR\_UB} \tag{12}$$

$$\sum_{j=1}^{C} x(u,j) * j + E_u \leq \sum_{j=1}^{C} w_b(u,j) * j, \forall i_u \in \mathbb{B} \tag{13}$$

In the above equations, $\mathbb{DDR\_UB}$ is defined as all DDR pairs in case 2 defined in Section 3.4 and $\mathbb{B}$ as the set of all instructions whose result is used by more than one instructions. We further define $\mathbb{DDR\_US}$ as the set of DDR pairs in case 1 in Section 3.4. Suppose N_{write} and N_{read} as the number of available register ports, then the register port constraints can be formulated with the following equations:

$$\sum_{(i_u.i_v)\in\mathbb{DDR\_US}} w_{real}(u,v,j) + \sum_{u\in\mathbb{B}} w_b(u,j) \leq N_{write}, \forall j \in \mathbb{J} \tag{14}$$

$$\sum_{(i_u.i_v)\in\mathbb{DDR\_U}} r_{real}(u,v,j) \leq N_{read}, \forall j \in \mathbb{J} \tag{15}$$

The Constraints for Registers

The following formula expresses a stair function expressed by the binary variables $reg(u,v,j)$, whose lower bound jumps from 0 to 1 in the cycle of the write instruction, and return to 0 in the cycle of read instruction.

$$reg(u,v,j) \geq \sum_{k=1}^{j} w(u,v,k) - \sum_{k=1}^{j} r(u,v,k), \forall j \in \mathbb{J}, \forall (i_u,i_v) \in \mathbb{DDR\_U} \tag{16}$$

$$reg(u,v,j) \geq 0, \forall j \in \mathbb{J}, \forall (i_u,i_v) \in \mathbb{DDR\_U} \tag{17}$$

The meaning is clear in the *real* case. For the *unreal* case, because the read instruction appears before write instruction, the right side of the formula (16) could be -1 in some cycles. But the $reg(u,v,j)$ is constrained to take non-negative values. Thus, it can represent the buffer occupation for both *real* and *unreal* cases.

The case 2 defined in Section 3.4 should also be considered here. We have introduced that after first write instruction is performed, the data exists in a register. It can be read again and again, and the hardware register can store other data as soon as the last read instruction takes place. A set of binary variables $r_b(u,j)$ is defined to trace the last read instruction of the branch:

$$\sum_{j=1}^{C} r_b(u,j) * j \geq \sum_{j=1}^{C} r_{real}(u,v,j) * j, \forall (i_u,i_v) \in \mathbb{DDR\_UB} \tag{18}$$

Thus, with another set of binary variables $reg_b(u,j)$, the register occupation in the case 2 can be calculated:

$$reg_b(u,j) \geq \sum_{k=1}^{j} w_b(u,v,k) - \sum_{k=1}^{j} r_b(u,v,k), \forall j \in \mathbb{J}, \forall (i_u,i_v) \in \mathbb{DDR\_UB} \tag{19}$$

$$reg_b(u,j) \geq 0, \forall j \in \mathbb{J}, \forall i_u \in \mathbb{B} \tag{20}$$

Assume $N_{register}$ is the available number of hardware registers (group II in Section 3.1). The constraint for registers is expressed in the following equation:

$$\sum_{(i_u,i_v)\in\mathbb{DDR\_US}} reg(u,v,j) + \sum_{i_u\in\mathbb{B}} reg_b(u,j) \leq N_{register}, \forall j \in \mathbb{J} \tag{21}$$

4.4 Dependence Constraint

We introduce this constraint in two categories: First, write after read dependence; Second, instructions in MIR and inter-block read and write instructions.

For the first category, assume instruction i_m reads a location before instruction i_n writes it. Equation (22) represents this dependence.

$$\sum_{j=1}^{C} x(m,j) * j + E_n - 1 \leq \sum_{j=1}^{C} x(n,j) * j, \qquad (i_m, i_n) \in \mathbb{U} \tag{22}$$

For the second category, assume inter-block read instruction y_p reads a data from a register location, then instruction i_m uses this data, $(y_p, i_m) \in \mathbb{U}_{\mathbb{IN}}$. For the instruction y_p, we also define a set of binary variables $y(p,j), y_p \in \mathbb{RI}, j \in \mathbb{J}$. The equation (23) expresses the modelling that y_p are placed directly before i_m. The reason is explained in Section 3.4.

$$\sum_{j=1}^{C} y(p,j) * j + E_p = \sum_{j=1}^{C} x(m,j) * j, \qquad (y_p, i_m) \in \mathbb{U}_{\mathbb{IN}} \tag{23}$$

Assume inter-block write instruction z_p writes the result of i_m to a register. Equation (24) represents their dependency.

$$\sum_{j=1}^{C} x(m,j) * j + E_m \leq \sum_{j=1}^{C} z(p,j) * j, \qquad (i_m, z_p) \in \mathbb{U}_{\mathbb{OUT}} \tag{24}$$

These inter block read and write instructions also occupy read/write ports, so the constraint equations (14) and (15) must be extended to accommodate the inter block instructions. Equation (25) and (26) describe the constraints. N_{read} and N_{write} the numbers of register read port and write port.

$$\sum_{(i_u.i_v) \in \mathbb{DDR\_US}} w_{real}(u,v,j) + \sum_{i_u \in \mathbb{B}} w_b(u,j) + \sum_{i_q \in \mathbb{WI}} z(q,j) \leq N_{write}, \forall j \in \mathbb{J} \tag{25}$$

$$\sum_{(i_u.i_v) \in \mathbb{DDR\_U}} r_{real}(u,v,j) + \sum_{i_p \in \mathbb{RI}} y(p,j) \leq N_{read}, \forall j \in \mathbb{J} \tag{26}$$

5 Experimental Results

We implemented all the formulations in our compiler by using one user licence of ILOG CPLEX (9.1 version) on one CPU. Our DSP has SIMD and VLIW architectural features [9]. The compiler center part performs vectorization for each application in MIR [10], then compiler backend generates corresponding assembly code with VLIW instruction set. Table 1 lists all the machine resource

Scalar Unit		Vector Unit		Scalar MEM		Vector MEM		Other Unit	
name	amount	name	amount	name	amount	name	amount	name	amount
Salu	4	Valu	1	Sld	1	Vldst	1	Icu	1
Smul	1	Vshift	1	Sst	1			Preg	16
Sshift	1	Vif	1					Seq	1
Sif	1	Vfpu	1					Decoder	1
Slogic	1	Vreg	8						
Sfpu	1								
Sreg	32								

Table 1. Machine Resource Table

in our experiments. The execution time of decoder is 0; sfpu, sld, sst, vfpu, vldst are 2 clock cycles; seq is 3 clock cycles; the rest FUs are 1 clock cycle.

Table 2 shows the performance of our code generator for some signal processing benchmarks. According to the different partitions of basic blocks, we did three groups of tests. Each basic block in three groups contains 10, 15 and 20 instructions respectively. E[cycles] shows the execution time of the ILP generated assembly code in clock cycles. F[%] is the ratio (ILP_cycles/trad_cycles*100%). T[s] is the solving time of the CPLEX in seconds. As the length of the basic block becomes larger, the solving time of the CPLEX has exponential increment.

The execution time of the assembly code can be reduced about 50% in our experiments. Furthermore, these assembly code use much less memory than the former code. ILP-based code generator uses output buffers of the FUs very efficiently, that leads to less registers' requirement and much less memory spill code.

benchmarks	group one			group two			group three		
name	E[cycles]	F[%]	T[s]	E[cycles]	F[%]	T[s]	E[cycles]	F[%]	T[s]
firparallel	114	65.1%	1.44	90	51.4%	2.9	82	46.9%	87.8
iirparallel	133	67.9%	1.73	105	53.6%	2.4	89	45.4%	19
firserial	75	64.1%	0.94	60	51.3%	2.5	55	47%	3.7
iirserial	62	54.9%	0.66	49	43.4%	5.1	43	38.1%	5.9
lmsparallel	1079	89.6%	19.7	904	75.1%	63.5	779	64.7%	842.1
lmsserial	149	50.5%	2.1	129	43.7%	7.9	103	34.9%	45
fft648	922	72.7%	29.7	723	56.9%	330.5	681	53.7%	1558.8
fft1288	1125	78.2%	35.1	890	61.8%	481.8	828	57.5%	1890.6
fft2568	1292	72.5%	44.3	1022	57.4%	555.8	951	53.4%	2092.1
dct2d88	767	59.2%	86.5	667	51.5%	249.8	607	46.9%	708.5

Table 2. Performance comparison between ILP and traditional code generator

6 Conclusion and Future Work

The advantage of this work is, that it can improve the instruction level parallelism with block partition in acceptable time. Furthermore, the code generation is very flexible: the number of machine resource and the optimality of the solutions can be easily adjusted by the users. The limitation of such ILP-based code generator is its scalability. Within acceptable time, it can only optimize the code in the small basic blocks, and the performance of such code generator is also related to the block partition. As the future work, we will make better block partition for our code generator, then some global heuristic algorithms will also be implemented.

References

1. Kent Wilken, Jack Liu, and Mark Heffernan. "Optimal Instruction Scheduling Using Integer Programming." Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, p121-133.
2. Krishnan Kailas, Kemal Ebcioglu and Ashok Agrawala. "CARS: A New Code Generation Framework for Clustered ILP Processors." Proceedings of the 7th International Symposium on High-Performance Computer Architecture, p133.
3. J.Hennessy and D.Patterson. "Computer Architecture, a Quantitative Approach." Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1996.
4. Timothy Kong and Kent Wilken. "Precise Register Allocation for Irregular Architectures." Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture.
5. Daniel Kästner. "Retargetable Postpass Optimisation by Integer Linear Programming", PHD Thesis, Universität des Saarlandes, Germany, Oktober 2000.
6. Gordon Cichon, P. Robelly, H. Seidel, E. Matus, M. Bronzel and Gerhard Fettweis. "Synchronous Transfer Architecture (STA)." SAMOS-04, p126-130, June, 2004.
7. Steven S. Muchnick. "Advanced Compiler Design Implementation." Morgan Kaufmann Publishers, 1997.
8. A.V.Aho, R.Sethi, and J.D.Ullman. "Compilers, Principles, Techniques and Tools." Addison-Wesley, Redding, MA, 1985.
9. Gordon Cichon, P. Robelly, H. Seidel, M. Bronzel and Gerhard Fettweis. "SAMIRA: A SIMD-DSP Architecture targeted to the Matlab source language." GSPx'04, USA, 27. -30. July 2004.
10. P. Robelly, G. Cichon, H. Seidel and Gerhard Fettweis. "Automatic code generation for SIMD DSP architectures: An algebraic approach." PARELEC'04, Dresden, Germany, 07. - 10. September 2004.