

Integrating Splitting Transformations into an ILP Instruction Scheduler

Morteza Fayyazi

Electrical and Computer Engineering
Northeastern University
Boston, MA

Norman Rubin
Chris Reeve

ATI Research Inc.
Marlborough, MA

Abstract

Instruction scheduling is an essential technique in a compiler that generates high quality code. This paper, presents an integer linear programming (ILP) based approach to local instruction scheduling for multiple-issue processors with arbitrary latencies and non-identical functional units. Then the ILP model is extended to use splitting transformations. Such transformations replace single operations with equivalent multiple instruction sequences, when use of the sequence would improve overall performance. The target architecture of our instruction scheduler is a graphics accelerator. This approach is compared to the heuristic approach used in a production compiler. The contributions of this paper are: (1) We study performance of the heuristic algorithms for instruction scheduling; (2) We introduce a novel integer linear programming approach for optimum instruction scheduling of multiple-issue processors that can split operations ; and (3) We provide experimental results of running different graphics applications with our optimal scheduler. The results show that an optimum instruction scheduler can improve measured performance by up to 8% in 12% of the programs (shaders) in our benchmarks.

1 Introduction

Graphics accelerators are specialized coprocessors used to speed up drawing graphics. These accelerators contain custom processing elements that achieve extremely high performance by SIMD parallel processing. A typical high end graphics processor (the ATI X850xtpe) has 16 processing elements, each of which can generate 4 floating point values per clock cycle. In graphics, processors are usually rated by the maximum number of pixels that can be drawn

per second. In those terms, the effective sustained processing power of the X850xtpe is 8.6G pixels per second [17]. In contrast, the Pentium 4 has an effective sustained processing power of at most 0.2G pixels per second, about 40 times slower.

Although graphics accelerators provide enormous processing power, the underlining DSP nature of the processor architecture makes it challenging for a compiler to utilize these chips efficiently.

1.1 Programming model

Graphics processors are programmed in an unusual style. Programmers do not write separate programs for each alternative graphics chip. Instead they generate small programs, called “shaders” in one of a small collection of virtual assembly languages (most common is DirectX Assembly language). Programs are generated in three main ways: (1) Programmers write in a high level language such as HLSL or CG, or (2) they use an artist oriented tool like Advanced Shading Language Interface (Ashli), or (3) they write directly in the virtual assembly language.

Shader programs tend to be small; A single instruction program can do useful work. For current generation graphics processors, a 50 instruction shader program is considered large. Many shaders contain only a single basic block, though some are quite complex. The virtual assembly languages used are similar to Java byte code. However, because small vector operations are common in graphics, there are built-in float vector data types.

There are two scenarios where shaders are compiled. On specialized game player consoles (such as the Xbox), shaders can be pre-compiled. The goal of a compiler for a console is to generate optimal code (if possible) in a reasonable time. Since the compile time for consoles is not critical, optimum algorithms

for code generation can be implemented in these compilers. In contrast, on a desktop computer, shaders are compiled on the fly, using a just-in-time (JIT) compiler. Hidden in the core of the graphics driver, there is a compiler that compiles shaders before running them on the accelerator.

Compilers in a graphics driver, need to execute quickly, since compilation time slows down the application, but they need to generate high performance code. Consequently driver based compilers like the production JIT shader compiler (SC) used by ATI, use highly tuned heuristic algorithms to find instruction schedules. In this paper, we will compare the results of an ILP scheduler to those produced by SC.

1.2 Target architecture

The graphics processor of interest to this paper is a custom chip, containing multiple processing elements. There is hardware support for multi-threading. Whenever a long latency operation is issued, (such as the use of a load), the processor will automatically switch to a different thread. The total number of threads in flight at any moment depends on the total number of resources used by those threads. Consequently, compilers may be able to improve performance by decreasing register usage, even if that could increase cycle length.

Each processing element is a two-issue machine containing two primary functional units, called the 'v' unit and the 's' unit. Informally, 'v' stands for vector and 's' for scalar. But this is not exactly accurate, since the v unit can evaluate certain complex operations like dot products, and the s unit can evaluate other complex operations like transcendental functions.

The 'v' and 's' units are quite different; The 'v' unit can perform certain vector operations producing up to 3 results per cycle. On the other hand, the 's' unit operates in a scalar mode producing one result per cycle. Under some cases, operations can be executed by either unit. The latency between two ALU operations can be greater than one. The instruction set includes a rich set of operations. For example, a three element dot product can be calculated in at least two ways; Either a single cycle dot product operation which runs on the 'v' unit, or a sequence of three 's' unit multiply-and-add operations can be used. Using the dot operation requires less operations (1 versus 3) but may take longer to execute, when the instruction scheduler can fit the scalar operations into unused schedule slots. In addition, there are a number of mini-alu units which can perform

a limited set of operations, without requiring either functional unit. Finally, because of resource restrictions, not all combinations of pairings are supported.

The ATI production compiler (SC) uses a complex set of heuristics to generate a schedule. Besides minimizing the schedule length, SC tries to minimize register usage, tries to place loads to take advantage of hardware supported multi-threading and tries to apply assorted transformations that move operations between the 'v' and 's' functional units. When the scheduler is started, assignments to mini-alus have already been made.

The instruction scheduler receives an intermediate representation of operations in a basic block in addition to a directed acyclic graph (DAG) showing data dependency among these operations. A node in the DAG represents an operation. A directed edge between two operations shows data dependency between those operations. The weight of an edge indicates the required latency between the execution time of those operations.

1.3 Prior work

Generated code is optimal when the program runs in the minimum number of cycles and requires the minimum number of registers. Unfortunately, it is not always possible to provide both minimum execution time and minimum number of registers.

The complexity of an instruction scheduler for a processor depends on the number of resources and the maximum latencies which occur among the instructions. J. Hennessy et al. [11] showed instruction scheduling is NP-complete for a single-issue processor with a maximum latency¹ of more than two cycles between the operations. Optimal scheduling is also NP-complete for multiple-issue processors [14]. There are some special cases where instruction scheduling is tractable [6, 7, 12, 13]. D. Bernstein et al. [6] proposed a polynomial time algorithm for scheduling instructions in a single-issue processor with a maximum latency of two machine cycles among instructions. E. G. Coffman et al. [7, 13] introduced polynomial time scheduling algorithms for both preemptive and non-preemptive jobs for an architecture with two identical processors and a maximum latency of one cycle among jobs. Complexity of instruction scheduling for any case more complicated than these limited cases is likely to be NP-complete.

¹Latency is defined as the difference between the cycle in which operation i executes and the first cycle in which data-dependent operation j can execute.

Because optimal instruction scheduling is considered intractable for most current processors, production compilers, such as SC, use a sub-optimal heuristic approach based on list scheduling.

Although optimal instruction scheduling for complex processors is known to be hard, there are some cases where it may be possible to solve many instances of these problems in a reasonable time. A local instruction scheduler is only responsible for scheduling instructions in a basic block. The size of basic blocks is small enough in many cases to optimally solve instruction scheduling using methods from combinatorial optimization. Prior work has used various combinatorial optimization approaches to optimally schedule instructions for complex processors. P. V. Beek et al. [10] used constraint logic programming to optimally schedule instructions for a single-issue processor with arbitrary latencies. C. Kebler et al. [5] used dynamic programming solution for code generation problems. Their algorithm is applicable for basic blocks with up to 20 instructions. Prior work has also used integer linear programming (ILP) to optimally schedule instructions [1, 2, 3, 4, 8, 9]. A well-designed ILP approach for instruction scheduling can be used in practice for many programs (i.e., SPEC benchmarks). Integer linear programming can also be used to optimally solve various other compiler optimization problems, including register allocation and array dependence analysis.

We have selected integer linear programming to solve instruction scheduling because ILP algorithms are sufficiently general to solve virtually any combinatorial optimization problem. ILP solutions have been shown to be more efficient and applicable for instruction scheduling than other combinatorial optimization techniques [1, 2, 3, 4, 8, 9]. Although ILP is NP-complete in general, linear programming (LP) can be solved efficiently in polynomial time. We use LP to find a lower bound for the number of cycles needed to schedule a block. LP can be achieved from an ILP model by relaxing integer variables. An ILP problem can be solved by using branch and bound techniques. The generic linear programming is formulated as follows:

$$\begin{aligned} & \text{Minimize } z = c^T x \text{ (objective function)} \\ & \text{Subject to: } Ax \leq b \\ & \text{where } x, c \in R^{n \times 1}, A \in R^{m \times n} \text{ and } b \in R^{m \times 1}. \end{aligned}$$

In ILP formulations, some variables are constrained to be integers.

In the next section, we introduce our ILP models for instruction scheduling. Section 3 provides our experimental results. Section 4 provides conclusions and describes future work.

2 ILP models for instruction scheduling

An optimum instruction scheduler for a processor determines the best order of instructions to minimize execution time of a program. For the two-issue VLIW structure, an instruction scheduler needs not only to find the best order of operations to minimize latencies but also pack operations into instructions to maximize resource utilization. An instruction includes a 'v' and an 's' operation. Each instruction may also include smaller operations (executed on a mini-alu) such as subtract, absolute, shift and negation. Operations can be divided into the following categories based on the resources that they need:

1. Operations that do not need any resource,
2. Operations that execute only on the 's' unit,
3. Operations that execute only on the 'v' unit,
4. Operations that execute on either unit (flexible), and
5. Operations that require both units.

It is possible for some 'v' operations in the shaders to be split into several 'v' and/or 's' operations, which makes optimum instruction scheduling even harder. In the following, we introduce an ILP formulation for instruction scheduling without splitting operations. Then, we introduce our general ILP formulation in which we consider splitting 'v' operations with single outputs into 's' operations.

2.1 ILP formulation without splitting operations

Many researchers have tried to apply ILP approaches to instruction scheduling. Similar to D. Kastner et al. [2] we use a time-indexed ILP formulation, where the decision variables are based on the time the operation is issued.

We speed up the performance of the ILP scheduler, by using SC to find an upper bound m for the number of cycles.

To produce a schedule for an n -operation basic block with an upper bound of m cycles, a binary variable x_{ic} is created for each (operation, cycle) pair. The variable x_{ic} represents the decision to schedule (1) or not schedule (0) operation i in clock cycle c . The objective function is selected to minimize overall

execution time of a basic block. The following is the ILP formulation:

$$\text{minimize } T \quad (1)$$

subject to:

$$x_{ic} \in \{0, 1\} \text{ for } i \in [1, n], c \in [1, m] \quad (2)$$

$$\sum_{c=1}^m x_{ic} = 1 \text{ for } i \in [1, n] \quad (3)$$

$$\sum_{c=1}^m c \times x_{ic} \leq T \text{ for } i \in [1, n] \quad (4)$$

$$\sum_{c=1}^m c \times x_{ic} + L_{ij} \leq \sum_{c=1}^m c \times x_{jc} \text{ for } i, j \in [1, n] \quad (5)$$

$$\sum_{i=1}^n (x_{ic} \times S_i) \leq 1 \text{ for } c \in [1, m] \quad (6)$$

$$\sum_{i=1}^n (x_{ic} \times V_i) \leq 1 \text{ for } c \in [1, m] \quad (7)$$

$$\sum_{i=1}^n (x_{ic} \times (S_i + V_i + E_i + 2 \times B_i)) \leq 2 \text{ for } c \in [1, m] \quad (8)$$

$$x_{ic} + x_{jc} \leq 2 - F_{ij} \text{ for } i, j \in [1, n], c \in [1, m] \quad (9)$$

Equation 1 shows the objective function and Equations 2 to 9 provide all constraints required to achieve a valid schedule. T and x are variables and S , V , E , B , F and L are constant data available before invoking the ILP solver. T is an integer variable representing the optimum number of cycles that all operations can be scheduled. S , V , E and B are arrays of n binary numbers showing the resource requirement of each operation. S_i is 1 if and only if operation i executes on an s unit. V_i is 1 if and only if operation i executes on a v unit. E_i is 1 if and only if operation i executes either on a s or a v unit. B_i is 1 if and only if operation i requires both units. F and L are $n \times n$ matrices. F_{ij} is a binary number showing that there would be a resource conflict between operations i and j if they were issued in the same cycle. L_{ij} is an integer number showing the latency between operations i and j . Parameter m is an upper bound for the number of cycles achieved from the list scheduler and n is the number of operations. Equation 3 guarantees that each operation is scheduled in one and only one cycle. Equation 4 forces T to have the cycle of the last operation in the schedule. Equation 5 provides the dependency constraint between operations. Equations 6, 7, 8 and 9 implement resource constraints.

We have used various techniques to simplify ILP formulations, resulting in fewer variables and constraints, thus reducing solution time. An operation's scheduling range can be reduced by finding upper and lower bounds for the cycle in which an operation can be scheduled. All variables residing outside of a valid range for an operation can be eliminated. To find a valid range for each operation i , we consider the number and type of predecessor and successor operations of i in DAG. We also consider the number of operations in the critical path from a root to i and from i to a leaf in DAG. A lower bound for the cycle in which an operation i can be scheduled is set by:

$$l_i = 1 + \max\{cr_i, ps_i, pv_i, (ps_i + pv_i + pf_i)/2\} \quad (10)$$

where ps_i , pv_i and pf_i are the number of predecessor s, v and flexible operations of i , respectively. cr_i is the length of the critical path from root(s) to i . An upper bound for the cycle in which an operation i can be scheduled is set by:

$$u_i = m - \max\{cl_i, ss_i, sv_i, (ss_i + sv_i + sf_i)/2\} \quad (11)$$

where ss_i , sv_i and sf_i are the number of successor s, v and flexible operations of i , respectively. cl_i is the length of the critical path from i to leaves.

We can find a lower bound for an ILP problem (with minimizing objective function) by relaxing integer constraints and finding an LP solution to the problem. We have used this lower bound to verify if the result of our list scheduler is optimum so we can skip running the ILP scheduler in the early stages.

We can summarize the steps used to schedule a basic block using ILP as follows:

1. Run the SC (list) scheduler to get an upper bound. Any optimal schedule must be no longer than the schedule generated by SC.
2. Generate the assorted matrix terms. Here, each operation is classified by resources required, and each pair of operations is checked to see if there would be a resource conflict if the pair are issued in the same cycle.
3. Solve the related LP problem to get a lower bound. A solution to the LP problem allows non-integer results, so that fractions of a single operation could be assigned to different cycles.
4. If the upper and lower bounds are equal, then the list scheduler produced an optimal solution and we are done.
5. Otherwise, solve the ILP problem.

Before scheduling, each operation is classified by possible resources. We call this the format of the operation. For example, a scalar add would have format 'v or s', indicating that it can execute in either functional unit. During scheduling, the operation is placed into a specific unit so that its format changes to 'v' or to 's'.

2.2 ILP formulation with splitting operations

Some vector operations receive vector inputs and produce a scalar output such as dot-product operations. It is possible to split these operations into a sequence of multiply-add scalar operations. A vector operation which has a vector output can not be split into scalar operations. Splitting vector operations can improve performance in the cases that there are not enough scalar operations in the block to fill the s unit.

The main decision variables in ILP formulation are x_{icr} and y_{if} where a value of 1 for x_{icr} means that operation i is executed in clock cycle c on execution unit r and a value of 1 for y_{if} means that operation i appeared in the schedule with the format f . Each execution unit r can be an s unit (S), a v unit (V), both s and v units (B) and none of the s or v units (N) for operations that do not need any resource. An operation format f defines the number of resources required by each operation. An operation format f is an index in a table of operation formats Γ . Each element Γ_{ifr} is an integer value which is the number of r resources that operation i requires in format f . Table Γ is available before invoking ILP. For instance, when operation i needs only an s unit (Table 1), Γ_{i1S} is one and Γ_{i1V} , Γ_{i1B} and Γ_{i1N} are zero. Table 2 shows the format table for a v operation. When operation i requires both s and v units (Table 3), Γ_{i1S} , Γ_{i1V} and Γ_{i1B} are one and Γ_{i1N} is zero. When operation i is flexible (Table 4), it can have the format of a s or v operation. Table 5 shows the format table for a dot-product operation (dp3). dp3 has two vector inputs and one scalar output. The inputs are vectors with three elements. The output is the result of the dot-product. A dp3 operation can be implemented as a single v operation or three s operations (i.e., one multiply and two multiply-add scalar operations). Table 5 shows two possible formats for the dp3 operation.

We also define two auxiliary integer variables l_i and u_i which are lower bound and upper bound cycles for operation i , respectively. The objective function is to minimize overall execution time of a basic block. The

followings are the ILP formulation:

$$\text{minimize } T \quad (12)$$

subject to:

$$x_{icr} \in \{0, 1\} \quad (13)$$

$$\text{for } i \in [1, n], c \in [1, m], r \in \{S, V, B, N\}$$

$$y_{if} \in \{0, 1\} \text{ for } i \in [1, n], f \in [1, p] \quad (14)$$

$$l_i, u_i \geq 0 \text{ integer for } i \in [1, n] \quad (15)$$

$$u_i \leq T \text{ for } i \in [1, n] \quad (16)$$

$$c \times x_{icr} \leq u_i \quad (17)$$

$$\text{for } i \in [1, n], c \in [1, m], r \in \{S, V, B, N\}$$

$$l_i \leq c \times x_{icr} + m \times (1 - x_{icr}) \quad (18)$$

$$\text{for } i \in [1, n], c \in [1, m], r \in \{S, V, B, N\}$$

$$\sum_{c=1}^m x_{icr} = \sum_{f=1}^p (y_{if} \times \Gamma_{ifr}) \quad (19)$$

$$\text{for } i \in [1, n], r \in \{S, V, B, N\}$$

$$\sum_{f=1}^p y_{if} = 1 \text{ for } i \in [1, n] \quad (20)$$

$$x_{icB} \leq x_{icr} \quad (21)$$

$$\text{for } i \in [1, n], c \in [1, m], r \in \{S, V\}$$

$$u_i + L_{ij} \leq l_j \text{ for } i, j \in [1, n] \quad (22)$$

$$\sum_{i=1}^n x_{icr} \leq 1 \text{ for } c \in [1, m], r \in \{S, V\} \quad (23)$$

where n is the number of operations, m is an upper bound for the number of cycles and p is the maximum number of formats for operations. Equations 13 and 14 define x_{icr} and y_{if} variables as binary numbers. Equation 15 defines l_i and u_i as positive integer numbers. Equation 16 defines the objective value T to be greater than or equal to the last cycle an operation executes. Equations 17 and 18 define upper bound cycle u_i and lower bound cycle l_i for each operation, respectively. Equation 19 guarantees that each operation receives the exact resources that it requires. Equation 20 guarantees that one and only one format is selected for each operation. Equation 21 is a constraint to reserve both s and v units for an operation that needs both resources ($x_{icB} = 1$). Equation 22 represents data dependency. And finally, Equation 23 provides resource constraints.

Table 1: S operation

f	S	V	B	N
1	1	0	0	0

Table 2: V operation

f	S	V	B	N
1	0	1	0	0

Table 3: S-V operation

f	S	V	B	N
1	1	1	1	0

Table 4: Flexible operation

f	S	V	B	N
1	1	0	0	0
2	0	1	0	0

Table 5: dp3 operation

f	S	V	B	N
1	0	1	0	0
2	3	0	0	0

3 Experimental results

We have implemented and verified both ILP formulations for instruction scheduling presented in this paper. We utilized the GNU Linear Programming Kit (GLPK) [16] for modeling and solving ILP formulations in our simulation platform. GLPK is an open source software that includes a modeling language and a solver for LP and ILP problems. GLPK supports the GNU MathProg language, which is a subset of AMPL [15] modeling language. We replaced the list scheduler in the SC compiler with our ILP instruction schedulers and compared the results. First, we provide results comparing the list scheduler with the ILP scheduler without splitting operations. These results include the number of instructions scheduled by each scheduler and the number of registers allocated for each shader program. We have also created a graphics driver utilizing ILP instruction scheduler to compare two schedulers on real execution time of graphics applications. As the benchmark, we have used shaders from real applications with a size of 4 to 74 operations in their intermediate representation format. The shaders are from three groups of shaders, *High_level*, *IDE* and *Hand* shaders. The *High_level* shaders were generated by a high level compilers such as HLSL or CG. The *IDE* shaders were

generated by an integrated development environment (IDE) called Ashli. The remaining group of shaders, *Hand* were written by hand directly in the virtual assembly language (most of these shaders can be found in commercial 3d games).

Table 6 represents our experimental results for these shaders. The first row shows the number of shader programs available in each benchmark. The second row shows the number of programs in which an optimum solution have been found in less than 1000 seconds. As shown in the third row, we have been able to verify optimality of the list scheduler in majority of the shaders (73% on average) by finding only an LP solution. Rows 4 and 5 compare the number of registers allocated by the list and ILP schedulers, respectively. The number of registers are almost the same for these schedulers. Rows 6 and 7 compare the number of instructions scheduled by the list and ILP schedulers, respectively. The percentage improvement by ILP is shown in the 8th row. ILP improved the number of instructions and the code size of the final program on an average of 1.25% for these benchmarks. The 9th row shows the percentage of the shaders in which ILP scheduler has improved the number of instructions. On average, ILP has improved about 12% of all shaders. The number of cycles that have been improved by the ILP scheduler varies from one cycle to five cycles for different shaders. Figure 1 represents the number of cycles improved by ILP for each shader in our benchmarks (485 shaders). On the X-axis we show the size of shaders in their intermediate format and on the Y-axis we plot the number of cycles improved by ILP scheduler. As the size of shaders increase, it is more likely that the list scheduler fails to find an optimum solution. Therefore, we expect to see more benefits of running ILP scheduler on larger shaders. Figure 2 shows percentage of the shaders improved by ILP scheduler for different sizes. On the X-axis we show five groups of shaders ordered by size. On the Y-axis we plot the percentage of shaders in the group where ILP has improved performance. For the shaders with less than 10 operations, the list scheduler finds an optimum solution. As the size of shaders increase, the ILP scheduler has a higher rate of improvement. For the largest group of the shaders with 41 to 74 operations, the ILP scheduler has improved 17.9% of the shaders.

We have implemented the ILP formulation with splitting of vector operations. This formulation has more variables and constraints than the original ILP formulations. Therefore, we are limited to smaller shaders from our benchmarks where compi-

lation takes less than 1000 seconds per shader. We have compiled 240 shaders of the *hand* benchmark with sizes from 5 to 42 operations using this ILP scheduler. Our experimental results show that the ILP scheduler with splitting dot-product operations improves the performance of the generated code for 2% of the shaders.

While minimizing cycle count is important, real applications often spend considerable time waiting for cache misses. To compare the ILP scheduler with the heuristic list scheduler and investigate the effect of cache misses for real-time graphics applications, we have created a driver that uses ILP for instruction scheduling. As the benchmark, we have selected 21 pixel shaders where the list scheduler finds a near-optimum solution. We have utilized hardware counters to measure the performance (number of frames per second) for these benchmarks. Each test rendered a spinning cube, where the shader was used to draw an image on each face of the cube. We ran each shader program for 10 seconds and collected the actual frame rates. Table 7 gives the results for each of these shaders. Relative frame rate is the percent that the ILP scheduler was faster than the list scheduler. Positive numbers indicate that the ILP was faster, negative numbers indicate that it was slower. Our results show a speedup of up to 8% for the ILP scheduler. For some cases, the ILP scheduler has lower performance than the list scheduler due to higher cache misses.

Given several independent consecutive load operations, the hardware can overlap cache accesses. The list scheduler tries to take advantage of this by a heuristic that groups *load* operations. The number of blocks of loads in the shader is called the *level*. Reducing the number of levels often produces less cache misses although it sometimes does worse. It is possible that adding new constraints to the ILP formulation to minimize the number of levels may improve the overall performance.

4 Conclusions

We developed two optimum instruction schedulers based on integer linear programming. The first, introduced a novel integer linear programming approach for optimum instruction scheduling of multiple-issue processors with non-identical resources and multiple-format instructions. The second allowed splitting while scheduling. We were able to optimally schedule a large percentage of real graphics applications. We provided experimental results of running different graphics applications with our optimum instruc-

Table 6: ILP scheduler versus list scheduler

	hlsl	ide	hand
Total # of shaders	90	110	368
Optimal found	84	60	341
Using LP solution	72%	65%	75%
list_reg	178	253	1256
ilp_reg	178	255	1264
list_cycles	457	1097	3112
ilp_cycles	439	1081	3089
Cycles improved	4%	1.5%	0.75%
Shaders improved	15%	15%	10.5%

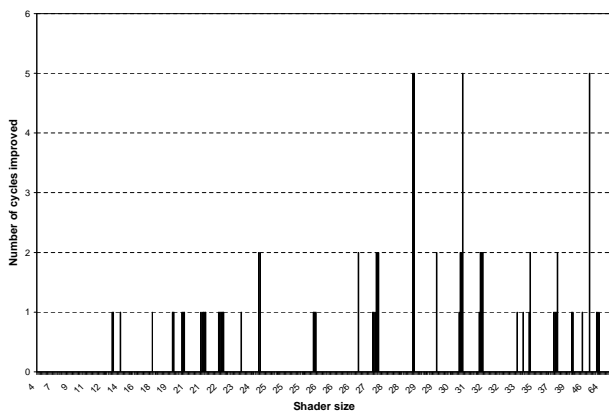


Table 7: measured performance

shader	length	list		ilp		relative frame rate
		reg	cycles	reg	cycles	
System.rfx.1094	13	2	7	2	6	-0.10%
3DMark03_GT4_0x00000004.ps	17	3	7	3	6	8.04%
System.rfx.1063	19	3	9	3	8	-0.16%
System.rfx.1097	19	2	8	2	7	0.00%
AquaMark3_0x0000000b.ps	20	5	10	5	9	-12.98%
3DMark03_GT2_0x00000005.ps	21	4	10	5	9	4.57%
DX9SDK_RT_HDR_IBL12_0x00000006.ps	21	4	10	4	9	1.71%
C1E6f_sumie_psh.dxf.dps11	21	4	9	4	8	1.63%
C1E6f_sumie_psh.dxf.dps14	21	4	9	4	8	1.63%
C1E6f_sumie_psh.dxf.dps20	21	4	9	4	8	1.57%
C7E6f_dispersion.cgf.cdx9ps2	22	5	8	5	7	-0.85%
C1E6f_sumie_psh.cgf.cdx9ps2	22	4	9	4	8	1.51%
C1E6f_sumie_psh.cgf.c8ps	23	4	9	4	8	1.57%
AquaMark3_0x0000000c.ps	24	6	13	6	11	-11.65%
System.rfx.1066	25	3	10	3	9	0.00%
C2E5f_fireFP.dxf2.dps20	25	3	12	4	11	5.96%
ATI_Demos_Caves_0x00000006.ps	27	7	14	7	13	4.18%
C7E6f_dispersion.dxf.dps20	27	6	12	6	11	-6.79%
potato_potat_program_0.psh	37	3	17	4	16	5.07%
ATI_Demos_CarPaint_0x00000014.ps	37	3	15	4	14	6.13%
ATI_Demos_CarPaint_0x00000015.ps	43	7	23	7	22	4.24%
AVERAGE	24	4.2	11.0	4.3	9.9	0.73%

tion schedulers. The results showed that a list scheduler provides high performance for smaller shaders and relatively good performance for larger shaders. To improve performance of the scheduler for larger shaders, other heuristic algorithms might be considered.

One challenge to future improvement of ILP-based schedulers is to consider the effect of scheduling on cache misses. An ILP scheduler that minimizes the rate of cache misses might outperform one that minimizes the number of instructions.

Another challenging approach to ILP-based schedulers would be to include register allocation while maintaining the number of constraints and variables low enough to solve the ILP problem in a reasonable time. In the multi-thread architecture of graphics accelerators with limited number of registers, it is essential to allocate minimum number of registers to provide high performance.

5 Acknowledgments

We would like to thank Gang Chen, Richard Bagley and Myron King from compiler group at ATI Research Inc. for their helpful comments.

References

- [1] M. Heffernan, J. Liu, K. Wilken, "Optimal instruction scheduling using integer programming," Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 121-133, June 2000.
- [2] D. Kastner, S. Winkel, "ILP-based instruction scheduling for IA-64," Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'01), pp. 145-154, 2001.
- [3] S. Chaudhuri, R. A. Walker, J. E. Mitchell, "Analyzing and Exploiting the structure of the constraints in the ILP approach to the scheduling problem," IEEE Transactions on VLSI Systems, Vol. 2, No. 4, pp. 456-471, December 1994.
- [4] S. Amarasinghe, D. R. Karger, W. Lee, V. S. Mirrokni, "A theoretical and practical approach to instruction scheduling on spatial architectures," Technical Report MIT-LCS-TM-635, MIT, September 2002.
- [5] C. Kebler, A. Bednarski, "Optimal integrated code generation for clustered VLIW architectures," Proceedings of the ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'02), pp. 102-111, 2002.
- [6] D. Bernstein, I. Gertner, "Scheduling expressions on a pipelined processor with a maximal delay of one cycle," ACM Transactions on Programming Languages and Systems, Vol. 11, No. 1, pp. 57-66, January 1989.

- [7] E. G. Coffman, J. Sethuraman, V. G. Timkovsky, "Ideal preemptive scheduling on two processors," *Acta Informatica* 39, pp. 597-612, 2003.
- [8] S. Touati, "Optimal acyclic fine-grain scheduling with cache effects for embedded and real time systems," *IEEE Hardware/Software Codesign (CODES)*, pp. 159-164, 2001.
- [9] R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, G. R. Gao, "Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures," *IEEE Transactions on Computers*, Vol. 52, No. 1, pp. 4-20, January 2003.
- [10] P. V. Beek, K. Wilken, "Fast optimal instruction scheduling for single-issue processors with arbitrary latencies," *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pp. 625-639, 2001.
- [11] J. Hennessy, T. Gross, "Postpass code optimization of pipeline constraints," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 422-488, July 1983.
- [12] S. M. Kurlander, T. A. Proebsting, C. N. Fischer, "Efficient instruction scheduling for delayed-load architectures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 17, No. 5, pp. 740-776, September 1995.
- [13] E. G. Coffman, R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica* 1, pp. 200-213, 1972.
- [14] D. Bernstein, M. Rodeh, I. Gertner, "On the complexity of scheduling problems for parallel/pipelined machines," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1308-1313, September 1989.
- [15] <http://www.ampl.com>
- [16] <http://www.gnu.org/software/glpk/glpk.html>
- [17] <http://www.ati.com/products/radeonx800/downloads.html>