Journal of Universal Computer Science, vol. 14, no. 21 (2008), 3465-3480
submitted: 16/4/08, accepted: 5/6/08, appeared: 1/12/08 © J.UCS

Instruction Scheduling Based on Subgraph Isomorphism for a High Performance Computer Processor

Ricardo Santos
(Dom Bosco Catholic University, Campo Grande, Brazil
ricr.santos@ucdb.br)

Rodolfo Azevedo
(State University of Campinas, Campinas, Brazil
rodolfo@ic.unicamp.br)

Guido Araujo
(State University of Campinas, Campinas, Brazil
guido@ic.unicamp.br)

Abstract: This paper[1] presents an instruction scheduling algorithm based on the Subgraph Isomorphism Problem. Given a Directed Acyclic Graph (DAG) G_1, our algorithm looks for a subgraph G_2' in a base graph G_2, such that G_2' is isomorphic to G_1. The base graph G_2 represents the arrangement of the processing elements of a high performance computer architecture named 2D-VLIW and G_2' is the set of those processing elements required to execute operations in G_1. We have compared this algorithm with a greedy list scheduling strategy using programs of the SPEC and MediaBench suites. In our experiments, the average Operation Per Cycle (OPC) and Operations Per Instruction (OPI) achieved by our algorithm are 1.45 and 1.40 times better than the OPC and OPI obtained by the list scheduling algorithm.

Key Words: instruction scheduling, subgraph isomorphism, 2D-VLIW
Category: D.3.m, C.1.3, I.2.5

1 Introduction

It is well-known that current high performance microprocessor architectures use hardware and software techniques to exploit the Instruction Level Parallelism (ILP) available in the applications. The proper scheduling of operations onto the hardware resources plays an important role in the final application performance. For architectures that schedule instructions at compile time, advanced compiler optimizations are essential to exploit the ILP available in programs.

Advanced compiler optimizations, including new instruction scheduling strategies, can look at large code regions to find out larger parallelism levels [Faraboschi et al. 2001]. In order to adopt large code regions at scheduling time, there is an increasing number of scheduling techniques based on graph

[1] A preliminary version of this paper was published in the Proceedings of the 12^{nd} Brazilian Symposium on Programming Languages, 2008.

matching [Maheswaran and Siegel 1998]. These proposals range from scheduling operations on multiprocessor systems, clustered VLIW architectures, and heterogeneous computing systems.

In this paper we address the instruction scheduling problem in a high performance computer architecture through a subgraph isomorphism approach. Instead of looking at single operations of a Directed Acyclic Graph (DAG) and allocating them to the hardware resources, our scheduling approach deals with two graphs: a DAG and a hardware resource graph. The goal is to match the DAG onto the hardware graph. This approach has shown to be very useful for architectures based on multiple processing elements once it easily captures the interconnection patterns of the DAGs and the available hardware resources at scheduling time. We propose a new instruction scheduling algorithm for a multiple functional-unit architecture named 2D-VLIW [Santos et al. 2006]. This architecture relies thoroughly on the compiler optimizations to provide speedup for the applications, since the compiler is the responsible for allocating and managing the hardware resources.

We have performed some experiments considering the Operations Per Cycle (OPC) and Operations Per Instruction (OPI) obtained by our algorithm. We compare the results of the subgraph isomorphism scheduling with a greedy list scheduling algorithm using programs of the SPEC (integer and float-point) [Henning 2000] and MediaBench [Lee et al. 1997] suites.

This paper is organized as follows: A description of the 2D-VLIW architecture is presented in Section 2. Section 3 discusses the Subgraph Isomorphism problem. Section 4 describes our instruction scheduling approach based on subgraph isomorphism. In addition, we present some heuristics to boost the subgraph isomorphism algorithm. The experiments and results are presented in Section 5. The final remarks and proposals for future work are described in Section 6.

2 The 2D-VLIW Architecture

2D-VLIW (**T**wo-**D**imensional **V**ery **L**ong **I**nstruction **W**ords) is a high performance computer architecture that exploits ILP through a bi-dimensional arrangement of the hardware resources and a pipeline datapath. In the 2D-VLIW architecture, the compiler is the unique responsible for assigning operations to the available resources. The architecture is called 2D-VLIW because it fetches large instructions, comprised of single operations, from the memory and executes these operations on a 2D-matrix of functional units through a pipeline. 2D-VLIW follows the EPIC terminology for operations and instructions: an operation corresponds to a RISC-like instruction and an instruction is a group of operations. The 2D-VLIW architectural arrangement is depicted in Figure 1.

Figure 1(a) sketches the datapath and its major architectural components. Figure 1(b) details the matrix of functional units (FUs) where four operations

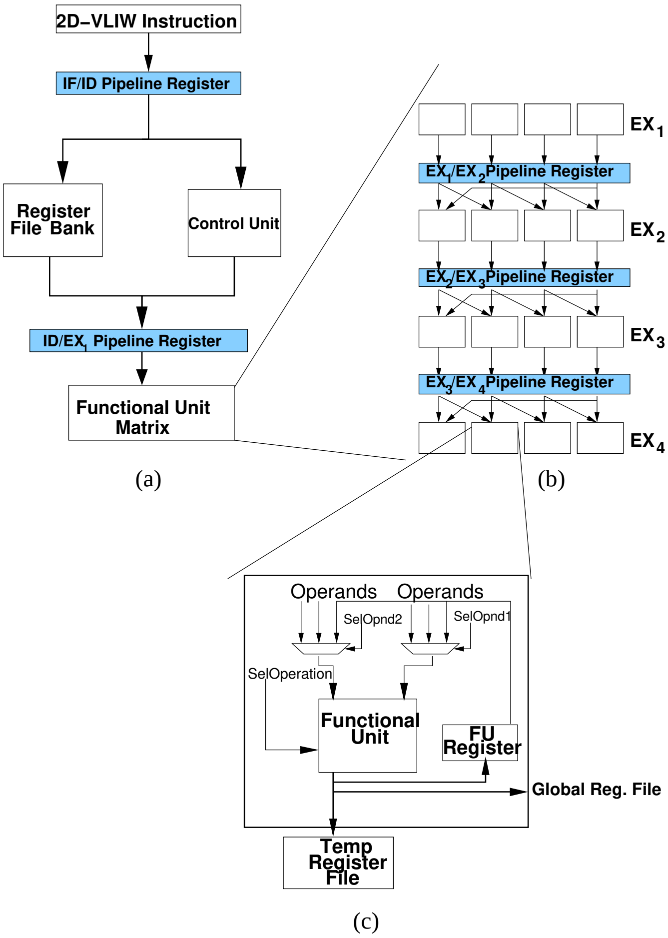
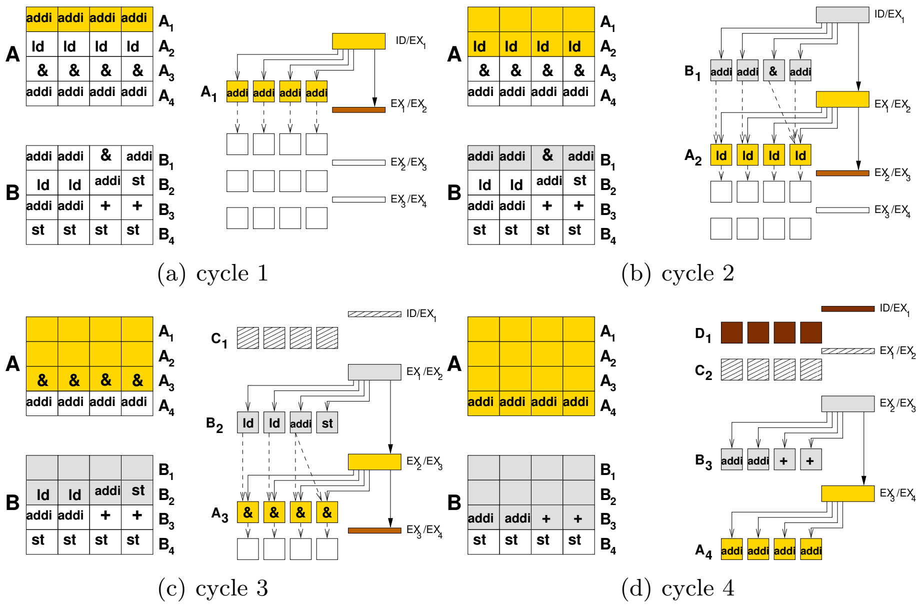


Figure 1: An overview of the 2D-VLIW datapath.

can be executed by four FUs at each execution stage EX_1, EX_2, EX_3, EX_4 of the pipeline. Figure 1(c) shows all logic blocks and signals inside a 2D-VLIW FU. Results from an FU may be written either into the *Temp Register File* (TRF) or the *Global Register File* (GRF). TRF is a small register bank containing 2 local registers dedicated to each FU. The GRF has 32 registers. The result of an FU is always written into an internal register called *FU Register* (FUR). *Operands* input data come from three possible sources: a GRF register, a TRF register, or from the FUR of the FU itself. The $SelOpnd1$, $SelOpnd2$ and $SelOperation$ input selection signals are available from the pipeline registers.

Program execution over the 2D-VLIW matrix is pipelined. At each clock cycle, one 2D-VLIW instruction (comprised of 16 single operations) is fetched from the memory and pushed into the pipeline stages. On the execution stages, the operations from this instruction are executed according to the number of FUs in each row. Assuming that the architecture has 16 functional units organized as a 4×4 matrix, a 2D-VLIW instruction can also be represented as a 4×4 operation matrix comprised of 16 operations. Figure 2 shows the execution of instructions A and B in the 2D-VLIW datapath. Since the decode and fetch stages work the same as in a standard processor, we start at the EX_1 execution stage. After the ID/EX_1 pipeline register has been filled in, the execution starts over the matrix.



(a) cycle 1 (b) cycle 2 (c) cycle 3 (d) cycle 4

Figure 2: The execution stages of two instructions on the 2D-VLIW architecture.

Figure 2(a) depicts the first execution cycle on the FU matrix. The first row receives data from the ID/EX_1 pipeline register. Four functional units from row A_1 execute operations addi, addi, addi, addi (instruction A). The dashed arrows indicate which FUs receive the results from these operations. At the second execution cycle, Figure 2(b), operations ld, ld, ld, ld from A_2 are running on the second row. At the same time, operations from B_1 start onto the first row. At the third execution stage, Figure 2(c), the EX_2/EX_3 pipeline register carries information to execute operations from A_3, the FUs in the second row are executing operations ld, ld, addi, st from B_2 and the first row (C_1) of a 2D-VLIW instruction C, starts its execution on the matrix. At the fourth execution stage, Figure 2(d), operations from A_4 are running on the fourth row, operations from B_3 are running on the third row, operations from C_2 are running on the second row and instruction D starts its execution.

As the unique responsible for the mapping of operations onto the hardware resources, the compiler plays an important role in the 2D-VLIW architecture. In order to have a compiler infrastructure generating code for the 2D-VLIW architecture, the Trimaran compiler [Chakrapani et al. 2004] has been adopted.

Trimaran does not have a scheduler that captures all 2D-VLIW features however. The scheduling techniques carried out in Trimaran are based on variations of the classical list scheduling. We have found out that approaches dealing with individual operations (like list scheduling does) or small code regions, do not draw a good performance on this architecture when compared to a scheduling looking at larger code regions and taking the functional units interconnection into account. Our scheduling proposal, on the other hand, picks up DAGs from programs and matches these DAGs onto the FU-matrix. The matching is performed by a subgraph isomorphism algorithm.

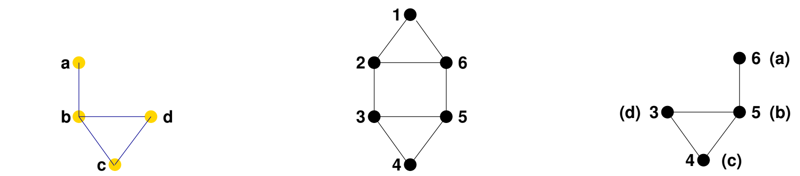
3 The Subgraph Isomorphism Problem in the 2D-VLIW Architecture

Subgraph isomorphism is a very general form of exact pattern matching and a common generalization of many important graph problems including finding Hamiltonian paths, cliques, matchings, girth and shortest paths [Eppstein 1999].

In the classical graph isomorphism problem, two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic, denoted by $G_1 \cong G_2$ if there is a bijection $\varphi : V_1 \to V_2$ such that, for every pair of vertices $v_i, v_j \in V_1$ holds that $(v_i, v_j) \in E_1$ if and only if $(\varphi(v_i), \varphi(v_j)) \in E_2$. In the subgraph isomorphism problem, graph $G_1 = (V_1, E_1)$ is isomorphic to graph $G_2 = (V_2, E_2)$ if there exists a subgraph of G_2, for example G_2', such that $G_1 \cong G_2'$. For certain choices of G_1 and G_2 there can be exponentially many occurrences so that listing all these occurrences is impractical thus leading to an $\mathcal{NP}$-complete decision problem [Garey and Johnson 1979]. Without loss of generality, lets call G_1 the input graph and G_2 the base graph.

Figure 3 shows an example of subgraph isomorphism. Figures 3(a) and 3(b) show the input graph and the base graph, respectively. The result of the mapping of edges and vertices from graph G_1 to graph G_2 is the graph in Figure 3(c). Notice that $\varphi(a) = 6$, $\varphi(b) = 5$, $\varphi(c) = 4$, $\varphi(d) = 3$.

Instruction scheduling has been one of the most complex tasks for a 2D-VLIW compiler mostly because the scheduler has to capture several hardware



(a) Input graph G_1 (b) Base graph G_2 (c) Graph G_2'

Figure 3: Example of Subgraph Isomorphism.

features such as FU-matrix size and topology, number of global and temporary registers and take all of them into account at the scheduling time. An instruction scheduling scheme based on subgraph isomorphism can minimize part of this complexity by representing the functional units and their interconnections through a base graph whereas the operations of the input DAG and their dependencies are represented by the input graph. Given an input DAG G_1 and the FU-matrix represented as a base graph G_2, the scheduler goal is to find a subgraph G_2' (represented as hardware resources inside the matrix) that matches to G_1.

The mapping of operations onto the FU-matrix leads to a complete 2D-VLIW instruction. Figure 4 depicts the 2D-VLIW instruction scheduling as a subgraph isomorphism problem. Notice that the functional units of the matrix and their interconnections are nodes and edges of the base graph. By looking at the scheduling result, we can realize that each position of the 2D-VLIW instruction has one operation which will be executed as the instruction goes down the FU-matrix.

4 A Subgraph Isomorphism Algorithm to Schedule 2D-VLIW Instructions

Since subgraph isomorphism problems were already proved as $\mathcal{NP}$-complete problems [Ullmann 1976], adopting efficient algorithms is a mandatory condition to use solutions to solve them. We have carried out a set of heuristics and used the main ideas from the VF subgraph isomorphism library [Cordella et al. 2001] to perform the 2D-VLIW scheduling based on the subgraph isomorphism strategy. The VF algorithm finds an isomorphism if there exist one and can determine the best isomorphism (optimal result). Further optimizations were added to this library to make it suitable for our purposes. For example, the original worst-case time complexity of VF algorithm is $\mathcal{O}(V!V)$, where V = #vertices of the input graph. One of our optimizations consists in determining a time constraint ($\leq$ 8 sec.) where the VF algorithm should find an isomorphism for each DAG.

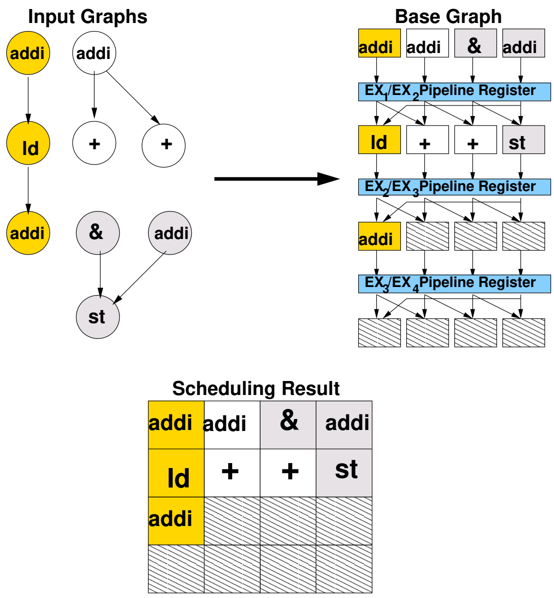


Figure 4: 2D-VLIW Instruction Scheduling Strategy.

If the isomorphism is not found within this time constraint, one optimization is performed to aid the scheduling. By performing the experiments, we could realize that these optimizations were indeed useful to schedule DAGs having more than a hundred vertices.

The inputs of the 2D-VLIW instruction scheduling strategy are DAGs generated by the Trimaran compiler and a base graph built in a specific procedure. The DAGs are passed to the scheduler on a hyperblock [Mahlke et al. 1992] basis, i.e., all the DAGs from a hyperblock i are scheduled before going to the next hyperblock p, $p > i$. Our algorithm takes these DAGs along with operation's latency information and performs the matching between the DAG and the base graph. Despite taking operations' latency and explicit data dependencies to schedule instructions, our algorithm also obeys some non-explicit data dependency such as saving passing parameters before procedure calls, and so on. Algorithm 1 outlines the 2D-VLIW instruction scheduling strategy.

First of all, we execute the topological_order procedure (Line 1) to perform a topological ordering in the input DAG G_1. Procedure subg_iso_sched (Line 2) finds a subgraph G_2', $G_2' \subseteq G_2$, isomorphic to G_1. This procedure uses the VF subgraph isomorphism library to find out subgraph G_2'. If subgraph G_2' is not found (Lines 3-8) the scheduler chooses one heuristic (Lines 4-6) and runs it over the input parameters. Variable tag (Line 4) acts as a heuristic selector.

Algorithm 1 2D-VLIW Subgraph Isomorphism Scheduling Algorithm

INPUT: An input graph G_1 and a base graph G_2
OUTPUT: Set of 2D-VLIW instructions
Sched(DAG: G_1**, BASE GRAPH:** G_2**)**
1) topological_order(G_1);
2) G_2' =subg_iso_sched(G_1, G_2, tag);
3) while ($G_2' == NULL$)
4)   switch(tag)
5)   case 1 : base_graph_resize(G_2);
6)   case 2 : DAG_stretch(G_1);
7) G_2' =subg_iso_sched(G_1, G_2, tag);
8) end while
9) create_2D-VLIW_instruction(G_2');

Heuristic base_graph_resize (Line 5) increases the base graph size in order to speed up the subgraph isomorphism procedure. The last heuristic, DAG_stretch (Line 6), transforms DAG G_1 into a more flexible graph so that it can be easily matched to G_2. Observe that only one heuristic is used after each unsuccessful execution of the subg_iso_sched procedure. Each heuristic is executed following a sequential order according to the numbers in Lines 5 and 6. After the scheduling is found, a 2D-VLIW instruction is created (Line 9). It should be clear that the time constraint is set in the subg_iso_sched procedure.

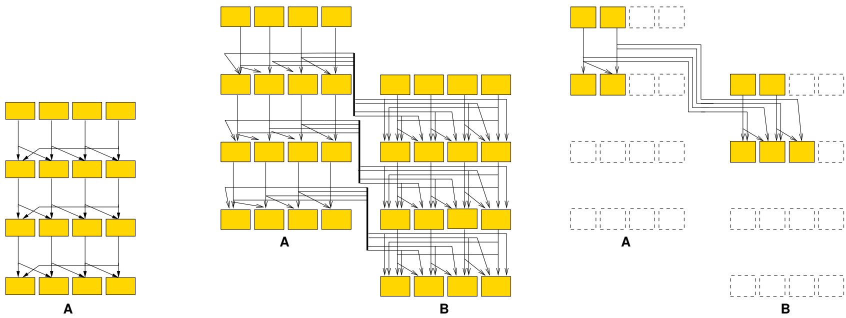
4.1 Topological Ordering Procedure

The topological ordering procedure provides the vertices ordering for the subg_iso_sched procedure. A bad vertex ordering can increase the number of backtracking steps of the isomorphism algorithm, resulting in a poor scheduling performance.

4.2 Heuristic 1: Base Graph Resizing

This heuristic is run when a base graph G_2 is not large enough to have a subgraph isomorphic to DAG G_1. The heuristic enlarges a base graph 2-fold, thereby forming up a new larger base graph. One can see this heuristic as a base graph "unrolling" method once the enlargement of the base graph is given by doubling (unrolling) its nodes and interconnections along the time, like in the software pipeline techniques [Aiken and Nicolau 1988]. Figure 5 exemplifies an unrolling of a single base graph, the FU-matrix in Figure 5(a), into a new larger base graph, the interconnected FU-matrices in Figure 5(b). For the sake of simplicity we left out the pipeline details. Looking at FU-matrix B, we can notice that the

FUs in row n from this matrix, $2 \leq n \leq 4$, have more edges than the FUs in the same row from FU-matrix A. These edges indicate that an FU from an unrolled matrix B can read values produced by FUs from matrix A and matrix B itself. Figure 5(c) shows the interconnection details from two nodes of FU-matrix A to two nodes of FU-matrix B.



(a) Single base graph (b) Larger base graph (c) Interconn. details between FU-matrix A and FU-matrix B

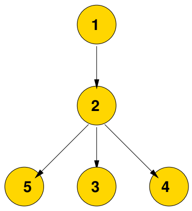
Figure 5: Example of base graph unrolling.

In Figure 6, one can observe that the input DAG in 6(a) cannot be scheduled onto a base graph comprised of just one FU-matrix. This scheduling cannot happen because node 2 has 3 descendants and the nodes of the base graph have only 2 descendants. The solution is to unroll the base graph to produce nodes with 3 descendants. Remember from Figure 5(b) that a base graph comprised of 2 FU-matrices has nodes with more than 2 descendants. After unrolling the single base graph, in Figure 6(b), we obtain the final successful scheduling in Figure 6(d).

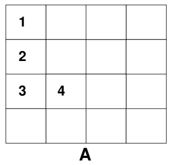
4.3 Heuristic 2: DAG Stretch

The DAG stretch heuristic acts as a mechanism to make the input DAG flexible enough to be allocated onto the 2D-VLIW FU-matrix.

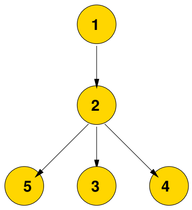
At scheduling time, the scheduler struggles to match input DAGs onto the base graph. If the input DAG is too complex such that the scheduler cannot find a subgraph isomorphic to the input DAG, the scheduling algorithm resorts to the DAG stretch heuristic. The heuristic converts single nodes of the input DAG into a new class of nodes named g-nodes in order to let the scheduler can use different base-graph regions to perform the scheduling. The single nodes are the nodes with the most number of descendants. This choice comes to



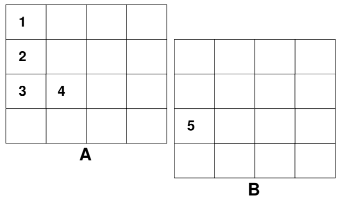
(a) Input DAG



(b) Single base graph



(c) Input DAG



(d) Larger base graph

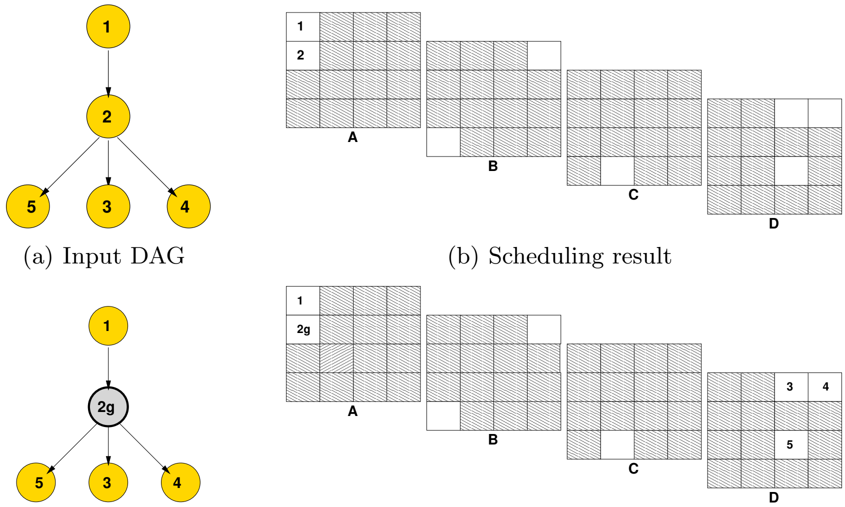
Figure 6: Base graph resize example.

the intuition that nodes with many descendants are most likely, the cause of backtrackings. Figure 7 outlines how this heuristic works. In Figure 7(a) node 2 has 3 descendants and the base graph, Figure 7(b), has enough available nodes to match the input DAG. However, the mapping of DAGs edges onto the FUs interconnections (TRFs) enables the subgraph isomorphism algorithm to explore only TRFs-connected nodes. Our solution is to convert node 2 into **2g** (global node) that allows its descendants to be scheduled onto any node of the base graph.

One can notice that the **g-node** makes the input DAG flexible enough enabling it to use other nodes of the base graph that were not initially available. Such flexibility is owed to the global registers usage since a global register data can be read from any functional unit of the FU-matrix through the pipeline registers. In other words, the conversion of node 2 into **2g** indicates that node **2g** writes its result to an available global register, hence making it possible to its descendants nodes to be scheduled onto any functional unit.

4.4 Subgraph Isomorphism Scheduling and Register Allocation

Register allocation is another compiler task that can take advantage of a subgraph isomorphism strategy. By including special nodes in the base graph, we can perform instruction scheduling and register allocation together in the 2D-VLIW architecture. These special nodes represent 2D-VLIW registers (global and temporaries) and they follow the 2D-VLIW interconnection as depicted in



(a) Input DAG (b) Scheduling result

(c) Input DAG with a g-node (d) Scheduling result after a g-node insertion

Figure 7: DAG stretch example.

Figure 1. Figure 8 depicts a base graph with basic nodes (FUs) and special nodes (temporary register banks).

By adopting an integrated approach, instruction scheduling and register allocation, the 2D-VLIW register allocator acts as follows:

1. Root nodes of the input graphs read values from the global register file while leaf nodes write their results to the global registers.

2. Internal nodes of the input graphs read and write from/to the temporary registers.

5 Experimental Results

This Section presents the performance results of the 2D-VLIW scheduling. We compare the results of our approach based on subgraph isomorphism with a greedy list scheduling algorithm. This list scheduling is described in Subsection 5.1. We have performed three experiments that measure the number of 2D-VLIW instructions per program, the number OPC enabled by each scheduling type, and the number of OPI. The results of the experiments are presented in Subsection 5.2. All the experiments were carried out using programs of the SPEC and MediaBench suites compiled with the Trimaran compiler infrastructure.

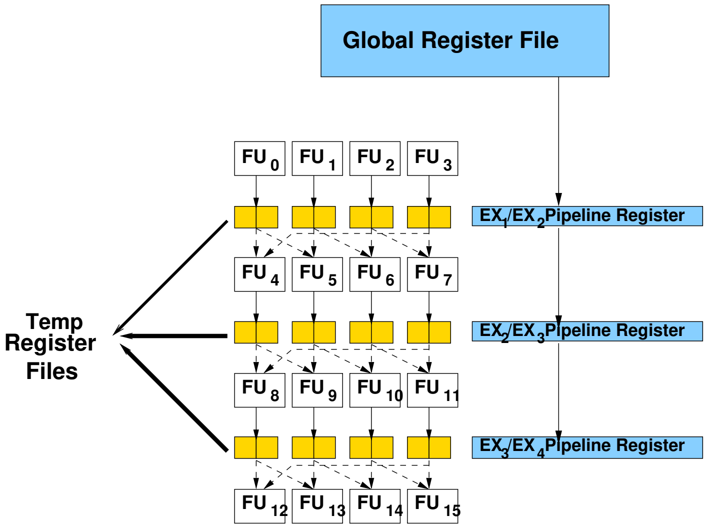


Figure 8: Temporary and global registers as nodes of the base graph.

5.1 A 2D-VLIW Greedy List Scheduling Algorithm

The 2D-VLIW list scheduling algorithm is a greedy list scheduling solution for the 2D-VLIW FU-matrix. As usual, it schedules each individual node according to its order in the ready set. Given an operation i to be scheduled, the algorithm tries to choose an available FU at the same matrix as its ancestors in order to increase the program OPI. If such FU is not available, the algorithm chooses the first available FU in the next matrix following the interconnection constraints of its parents. Like the isomorphism algorithm (discussed in Section 4), the list scheduling strategy backtracks whenever there is no possible scheduling for the current node. Despite its exponential time complexity, the backtracking procedure is a necessary step to make sure that the scheduling follows the DAG's nodes dependency.

Algorithm 2 outlines the main steps of the 2D-VLIW list scheduling strategy. Except for Lines 2, 5, and 7, the steps of the algorithm look like the same as in Algorithm 1. Observe that the DAG_Stretch heuristic can also be applied to the input parameters in Algorithm 2. Moreover, the same time constraints used in Algorithm 1 are also applied to Algorithm 2.

5.2 Results

The first experiment shows the number of 2D-VLIW instructions obtained by each scheduling algorithm and their respective scheduling time. The results in

Algorithm 2 Greedy List Scheduling Algorithm

INPUT: An input graph G_1 from a hyperblock
OUTPUT: Set of 2D-VLIW instructions
LS_Sched(DAG: G_1)
1) topological_order(G_1);
2) G'_2 =greedy_ls_sched(G_1);
3) while ($G'_2 == NULL$)
4) DAG_stretch(G_1);
5) G'_2 =greedy_ls_sched(G_1);
6) end while
7) create_2D-VLIW_instruction(G'_2);

Table 1 has a straight impact on the final code size and performance of the programs since programs that have more instructions will require more instruction fetch cycles to the memory. The last column indicates the improvement of the subgraph isomorphism scheduling time when compared to the list scheduling time. Columns 2 and 3 represent the number of instructions and the scheduling time (seconds) of the List Scheduling (LS) algorithm. Columns 4 and 5 show the same values (number of instructions and scheduling time) considering the Subgraph Isomorphism Scheduling (SIS) algorithm.

Table 1: Number of 2D-VLIW instructions and Scheduling Time of the List Scheduling (LS) and the Subgraph Isomorphism Scheduling (SIS).

Programs	**LS**		**SIS**		Improv.
176.gcc	158,300	401,573	108,786	48,967	(87%)
175.vpr	58,483	54,217	43,511	7,525	(86%)
181.mcf	9,689	9,243	7,782	1,264	(86%)
197.parser	71,514	86,912	45,410	6,666	(92%)
255.vortex	110,602	128,608	75,235	5,679	(95%)
256.bzip2	21,020	24,875	17,109	5,184	(79%)
300.twolf	99,688	141,371	84,344	23,843	(83%)
epic	8,732	634	1,953	151	(76%)
gsm.decode	10,557	1,831	9,136	1,113	(39%)
gsm.encode	13,852	5,034	11,810	1,885	(62%)
pegwit	14,128	10,634	11,159	2,171	(79%)
168.wupwise	9,336	6,869	6,666	1,096	(84%)
183.equake	10,787	5,150	7,243	655	(87%)

The results in Table 1 show that the isomorphism scheduling strategy leads to fewer 2D-VLIW instructions than the greedy list scheduling for all the evaluated programs. The isomorphism scheduler time performance were up to 95% better than the list scheduling. The scheduler performance gains of our strategy can be observed for all the programs (Column Improv.). We have also performed an experiment comparing the time performance of our scheduling strategy to the greedy list scheduling algorithm. The average time speedup of our scheduling over the greedy list scheduling algorithm is 8.21. The worst running performance of the list scheduling is owing to two reasons: 1) its local view of the scheduling process; 2) the backtracking procedures running every time a vertex does not match to the base architecture graph. Despite not being shown, we believe that a list scheduling algorithm that does not perform the backtracking, provides a better performance at the expense of a larger global register usage.

The next experiment compares the OPC and OPI achieved by both scheduling algorithms when considering all the programs. Unlike other approaches based on the program's kernels, we compute the OPC and OPI by the whole program code source. The values in Table 2 indicate the OPC and OPI (in parenthesis), respectively. Likewise Table 1, LS represents the results obtained by the List Scheduling algorithm and SIS represents the results of the Subgraph Isomorphism Scheduling algorithm. Moreover, we compute the average and standard deviation of the OPC and OPI in each algorithm.

Table 2: OPC and OPI

Program	**LS**	**SIS**
176.gcc	2.27 (6.17)	3.14 (8.99)
175.vpr	2.50 (2.34)	3.70 (3.15)
181.mcf	5.00 (4.99)	6.60 (6.22)
197.parser	2.20 (2.03)	3.59 (3.20)
255.vortex	1.09 (2.25)	3.41 (3.31)
256.bzip2	2.47 (2.47)	3.20 (3.03)
300.twolf	2.18 (1.93)	2.76 (2.28)
epic	2.61 (1.30)	4.49 (5.84)
g721.decode	2.23 (2.23)	3.52 (3.44)
g721.encode	2.24 (2.24)	3.48 (3.41)
gsm.decode	2.47 (2.47)	3.10 (2.86)
gsm.encode	2.55 (2.55)	3.20 (2.99)
pegwit	4.32 (4.30)	5.07 (5.44)
168.wupwise	2.23 (2.23)	3.38 (3.13)
179.art	2.36 (2.36)	3.51 (1.67)
183.equake	2.20 (2.30)	3.52 (3.28)
Average	**2.55 (2.76)**	**3.72 (3.89)**
St. Dev.	**0.9 (1.2)**	**0.9 (1.8)**

The isomorphism scheduling algorithm outperforms the greedy list scheduling for all programs. The average OPC and OPI of the subgraph isomorphism scheduling are about 1.5× better than the values produced by the list scheduling.

Actually, the maximum OPC and OPI values achieved by the subgraph isomorphism is up to 3.12× (program 255.vortex) better than the greedy list scheduling.

Another interesting result shows up when we compare the OPC achieved by a subset of the programs in Table 2 (programs 179.art, 256.bzip2, 183.equake, and 197.parser) with the results reported in [Coons et al. 2006] when using the same subset. The peak OPC (Annealed IPC) obtained in [Coons et al. 2006] for these programs is 6.4, 3.0, 2.7, and 2.4, leading to an average OPC of 3.62. For this same program subset, the average OPC achieved by our subgraph isomorphism scheduling is 3.45. Most important, the scheduling algorithm described in [Coons et al. 2006] does not consider compiler time constraints to achieve these peak performance. On the other hand, all our results were obtained according to time constraints added into our scheduling algorithm.

6 Conclusions and Future Work

A new instruction scheduling algorithm based on subgraph isomorphism theory was presented in this paper. The algorithm has been used as a basic scheduler for an architecture named 2D-VLIW. The algorithm considers DAGs of a program as input graphs and the functional units and their interconnections are represented by a base graph. The objective is to find a subgraph of the base graph which is isomorphic to the input graph.

By comparing the average OPC and OPI achieved by our scheduling technique to the greedy list scheduling algorithm, the subgraph isomorphism algorithm obtains 3.72 and 3.89, respectively, whereas the greedy list scheduling values are 2.55 and 2.72. All the programs have less 2D-VLIW instructions when scheduled with the subgraph isomorphism strategy. Interesting enough, our scheduling algorithm achieves a time speedup of 8.21 over the greedy list scheduling algorithm.

Future research is focused on plugging scheduling and register allocation together using the subgraph isomorphism strategy. We are also experimenting new parameters for the Base Graph Resize heuristic. Furthermore, we intend to evaluate this scheduling algorithm under other architectures based on multiple processing elements with dynamic or fully interconnecting schemes.

Acknowledgments

The authors thank Brazilian Research Agencies (CAPES and CNPq) for their financial support to the research in the Computer Systems Laboratory of the Institute of Computing. Ricardo Santos also thanks Dom Bosco Catholic University for its financial support in the 2D-VLIW project.

References

[Aiken and Nicolau 1988] Aiken, A., Nicolau, A.: "Limits on Multiple Instruction Issue"; Proceedings of the ACM Conf. on PLDI, ACM Press, 1988, 308-317.

[Chakrapani et al. 2004] Chakrapani, L. N., Gyllenhaal, J., Mei, W., Hwu, W., Mahlke, S. A., Palem, K. V., Rabba, R. M.: "Trimaran - An Infrastructure for Research in Instruction-Level Parallelism"; Lecture Notes in Computer Science, Springer-Verlag, 3602, 2004, 32-41.

[Coons et al. 2006] Coons, K. E., Chen, S., Kushwaha, S. K., Burger, D., McKinley, K. S.: "A Spatial Path Scheduling Algorithm for EDGE Architectures"; Proceedings of the 12^{th} ACM Int. Conf. on ASPLOS, ACM Press, October 2006, 40-51.

[Cordella et al. 2001] Cordella, L. P., Foggia, P., Sansona, C., Vento, M.: "An Improved Algorithm for Matching Large Graphs"; Proceedings of the 3^{rd} IAPR TC15 Workshop on Graph-Based Representations, Ischia-Italy, 2001, 149-159.

[Eppstein 1999] Eppstein, D.: "Subgraph Isomorphism in Planar Graphs and Related Problems"; Journal of Graph Algorithms and Applications, World Scientific Publishing, 3 (3), 1999, 1-27.

[Faraboschi et al. 2001] Faraboschi, P., Fisher, J. A., Young, and C.: "Instruction Scheduling for Instruction Level Parallel Processors"; Proceedings of the IEEE, IEEE Computer Society, 89 (11), November 2001, 1638-1658.

[Garey and Johnson 1979] Garey, M. R., Johnson, D. S.: "Computers and Intractability: A Guide to the Theory of NP-Completeness"; W. H. Freeman and Company, 1979.

[Henning 2000] Henning, J.: "SPEC CPU2000: Measuring CPU Performance in the New Millenium"; IEEE Computer, IEEE Computer Society, 33 (7), 2000, 28-35.

[Johnson 1974] Johnson, D. S.: "Fast Algorithms for Bin Packing"; Journal of Computer and System Sciences, Society for Industrial and Applied Mathematics, 8, 1974, 272-314.

[Lee et al. 1997] Lee, C., Potkonjak, M., Smith, W. H.: "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems"; Proceedings of the 30^{th} Int. Symp. on MICRO, IEEE Computer Society, December 1997, 330-335.

[Maheswaran and Siegel 1998] Maheswaran, M., Siegel, H. J.: "A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems"; Proceedings of the 7^{th} Heterogeneous Computing Workshop, IEEE Computer Society, 1998, 57-69.

[Mahlke et al. 1992] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., Bringmann,R. A.: "Effective Compiler Support for Predicated Execution Using the Hyperblock": Proceedings of the 25^{th} IEEE/ACM Int. Symp. on MICRO, IEEE Computer Society, December 1992, 45-54.

[Santos et al. 2006] Santos, R., Azevedo, R., Araujo, G.: "Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach"; Proceedings of the 13^{th} IEEE Int. Workshop on Reconfigurable Architectures, IEEE Computer Society, Rhodes Island-Greece, 2006.

[Ullmann 1976] Ullmann, J. R.: "An Algorithm for Subgraph Isomorphism"; Journal of the Association for Computing Machinery, ACM Press, 23 (1), 1976, 31-42.