Architecture Base Graphs: An Alternative Instruction Scheduling and Register Allocation Approach for High Performance Processor Architectures

Abstract This paper presents a unified algorithm that addresses the instruction scheduling and register allocation problems for high performance processor architectures. Our proposal is based on the subgraph isomorphism theory. Given a Directed Acyclic Graph (DAG) $G_1$, the algorithm looks for a subgraph $G'_2$ in an architecture base graph $G_2$, such that $G'_2$ is isomorphic to $G_1$. $G_1$ is a DAG extracted out from the user program in the compiler phase. The base graph $G_2$ is built upon the arrangement of the processing elements and user-visible registers of a computer processor. $G'_2$ is the set of those processing elements and registers required to execute operations in $G_1$. Our strategy focuses on code generation for constrained and irregular processor architectures exploiting ILP. We have performed experiments using this integrated approach in programs of the SPEC and MediaBench suites on an ILP processor and compared it to the scheduling and register allocation algorithms of the Trimaran compiler infrastructure. The results show that the isomorphism strategy provides better generated code (3%–85%) on kernels covering up to 90% of the application execution time.

Keywords D.3.4.a Code Generation, D.3.4.b Compilers, D.3.4.g Optimization

1 Introduction

A compiler infrastructure is a kind of system where different intermediate formats, input languages, and output platforms are putting together to generated a final code able to run on a target machine. To manage this huge infrastructure, sometimes it is necessary to compromise part of the infrastructure, splitting it or facing only small parts of a big problem at each time.

Instruction scheduling and register allocation are two phases of the compiler back-end that focus on the target architecture capabilities. Most of the time, they are managed as two different phases by the lack of infrastructure (software or model) to handle both in an integrated way. Surely, the best scheduling and register allocation is the one that fulfill all the architecture constraints and from these constraints, register spill is one of the most important to face. Looking from the opposite side, when facing a register pressure that probably will require a spill, there can be another equivalent instruction order that reduce some live-ranges, consequently reducing the register pressure.

There are some previous work on integrating instruction scheduling and register allocation algorithms [2, 8, 11, 21, 24, 26]. This paper presents an integrated approach for instruction scheduling and register allocation focusing on the matching of code regions to the available hardware resources. Our technique goes beyond the algorithm and also proposes a new way to characterize the processor architecture so that both instruction scheduling and register allocation occurs at the same time. This new representation is called Architecture Base Graph (ABG) and represents all the functional units and available registers from the architecture.

The architecture base graph can handle:

- Timing information: it is possible to provide simultaneous timing to different instructions, as happens in VLIW instructions while keeping one functional unit busy for more than one cycle, as in superscalars.
- Internal registers: it is possible to support temporary bypass registers and flag registers like in the x86 instruction set.
• Multiple register banks: it is possible to transparently move data between registers bank and this can be used to reduce spill pressures.

• Processing elements: it is possible to identify instruction-specific processing elements and their interconnection patterns.

The architecture base graph and the directed acyclic graph (DAG) - representing the program code - are the inputs to our proposal for an integrated scheduling and register allocation algorithm. We evaluated it using 12 kernels of programs from MediaBench, SPEC 2000 and SPEC 2006 benchmarks and got an improvement on the performance of the generated code ranging from 3% up to 85%.

This paper is organized as follows: Section 2 shows some instances where scheduling and register allocation can provide better results if they are handle together. Section 3 presents the basic definitions to be used along the paper. Section 4 presents our instruction scheduling and register allocation integrated algorithm. Section 4.2 presents the main concepts of the architecture base graph. Section 4.3 presents a case study of our technique on the HPL-PD processor. The experiments and results are presented in Section 5. Section 6 outlines the related research to this work. The final remarks and proposals for future work are described in Section 7.

2 Motivation

List scheduling [1] and graph coloring [4] are well-known algorithms for instruction scheduling and register allocation of back-end compilers for a wide range of current processor architectures. Despite several optimizations have been proposed for those algorithms, most of the implementations follow the classical approach. As an example, the Trinaran compiler infrastructure [5] adopts scheduling (list scheduling) and register allocation algorithms (vertex coloring) where memory operations are assigned to hardware resources in a conservative way. Another example is the LLVM [23] compiler which also adopts list scheduling as the basic scheduler and it adopts the linear scan algorithm [27] as the default register allocator. Basic approaches for scheduling and register allocation do not capture all the constraints of the target architecture. This is a problem for compiler infrastructures that intend to generate code for different processor architectures. Figure 1 shows an example where an instruction scheduler and register allocator cannot capture all the constraints of the architecture. Figure 1(a) represents a processor architecture comprised of four dedicated functional units using distributed register files. One can observe that FU3 (for load operations) can read operands from both register files but FU1 (for add operations) and FU2 (for and operations) can read only from GRF A. In order to read values from load operations, one move operation must be used to copy values from GRF B to GRF A. Figure 1(b) shows a directed acyclic graph (DAG) which is used as input to a greedy scheduler. Notice the result of that scheduler in Figure 1(c). Since the scheduler does not capture the constraints in advance, the two lw operations were scheduled one after the other. Furthermore, move operations must be inserted between the lw and the add. That situation is not captured by the scheduler so that it is necessary to have a specific phase to insert those moves in the DAG and schedule them. Figure 1(d) shows a DAG used by our integrated strategy for scheduling and register allocation. After the input DAG (Figure 1(b)) is first evaluated, our technique insert move vertices directly on the DAG. The modified DAG is matched to the base graph of the processor architecture using a subgraph isomorphism approach. Since the whole DAG is matched, the vertices are tested for a possible matching as soon as possible. This is why the first lw operation is scheduled together with the first add operation. Figure 1(e) shows the result of our technique. By the example, the subgraph isomorphism technique brings a better ILP and, as a result, a better performance for the generated code.
Fig. 1. Differences between classical and integrated algorithms for scheduling and register allocation.

Our algorithm looks for a scheduling where operations can be matched to the hardware resources (functional units and registers) as soon as possible according to the availability of the resources. In addition, the algorithm takes possible hazards (true data dependencies and memory hazards) between two operations into account at matching time. The example presented in Figure 1 exemplifies the situation where our algorithm can take advantage upon conservative algorithms.

3 Definitions

The following definitions will be necessary to describe and understand all the details behind the approach for scheduling and register allocation presented in this paper.

Definition 1 (Directed acyclic graph): A directed graph $G = (V, E)$, where $V$ is the set of vertices of $G$, $E$ is the set of edges of $G$ is named Directed Acyclic Graph (DAG), if it is acyclic and if and only if there are not strongly non-trivial connected components.

Definition 2 (Graph isomorphism): An isomorphism of graphs $G$ and $H$ is a bijection between the vertex set of $G$ and $H f : V(G) \rightarrow V(H)$ such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $H$.

Definition 3 (Subgraph isomorphism): A graph $G_1 = (V_1, E_1)$ is isomorphic to a subgraph $G'_2$ of a graph $G_2 = (V_2, E_2)$, denoted by $G_1 \cong G'_2 \subseteq G_2$, if there is an injection $\varphi : V_1 \rightarrow V_2$ such that, for every pair of vertices $u_i, v_j \in V_1$, if $(u_i, v_j) \in E_1$ then $(\varphi(u_i), \varphi(v_j)) \in E_2$.

Definition 4 (Base graph): A directed graph $G = (V, E)$ where $V$ represents the set of vertices of $G$, $E$ is the set of edges of $G$, is named Base Graph, if $V$ represents the set of functional units, processing elements, registers, and $E$ represents the set of interconnections between functional units and registers. More details on Base Graph creation will be given in Section 4.2.

Definition 5 (Subgraph isomorphism scheduling and register allocation): Given a DAG $G_1$ and a Base graph $G_2$, the subgraph isomorphism scheduling and register allocation is an isomorphism between $G_1$ and $G_2$ in order to obtain a subgraph $G'_2 \subseteq G_2$ representing the hardware resources necessary to execute $G_1$. 

Our algorithm looks for a scheduling where operations can be matched to the hardware resources (functional units and registers) as soon as possible according to the availability of the resources. In addition, the algorithm takes possible hazards (true data dependencies and memory hazards) between two operations into account at matching time. The example presented in Figure 1 exemplifies the situation where our algorithm can take advantage upon conservative algorithms.

3 Definitions

The following definitions will be necessary to describe and understand all the details behind the approach for scheduling and register allocation presented in this paper.

Definition 1 (Directed acyclic graph): A directed graph $G = (V, E)$, where $V$ is the set of vertices of $G$, $E$ is the set of edges of $G$ is named Directed Acyclic Graph (DAG), if it is acyclic and if and only if there are not strongly non-trivial connected components.

Definition 2 (Graph isomorphism): An isomorphism of graphs $G$ and $H$ is a bijection between the vertex set of $G$ and $H f : V(G) \rightarrow V(H)$ such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $H$.

Definition 3 (Subgraph isomorphism): A graph $G_1 = (V_1, E_1)$ is isomorphic to a subgraph $G'_2$ of a graph $G_2 = (V_2, E_2)$, denoted by $G_1 \cong G'_2 \subseteq G_2$, if there is an injection $\varphi : V_1 \rightarrow V_2$ such that, for every pair of vertices $u_i, v_j \in V_1$, if $(u_i, v_j) \in E_1$ then $(\varphi(u_i), \varphi(v_j)) \in E_2$.

Definition 4 (Base graph): A directed graph $G = (V, E)$ where $V$ represents the set of vertices of $G$, $E$ is the set of edges of $G$, is named Base Graph, if $V$ represents the set of functional units, processing elements, registers, and $E$ represents the set of interconnections between functional units and registers. More details on Base Graph creation will be given in Section 4.2.

Definition 5 (Subgraph isomorphism scheduling and register allocation): Given a DAG $G_1$ and a Base graph $G_2$, the subgraph isomorphism scheduling and register allocation is an isomorphism between $G_1$ and $G_2$ in order to obtain a subgraph $G'_2 \subseteq G_2$ representing the hardware resources necessary to execute $G_1$. 

Our algorithm looks for a scheduling where operations can be matched to the hardware resources (functional units and registers) as soon as possible according to the availability of the resources. In addition, the algorithm takes possible hazards (true data dependencies and memory hazards) between two operations into account at matching time. The example presented in Figure 1 exemplifies the situation where our algorithm can take advantage upon conservative algorithms.
4 An Integrated Approach for Instruction Scheduling and Register Allocation

It is already known that integrated approaches for register allocation and instruction scheduling can bring better performance and hardware usage for the generated code. Specifically, compilers targeted to code generation for constrained processor architectures could take advantage of unified approaches since for those architectures, it is necessary to consider hardware constraints in order to perform correct placement and routing of instructions.

Our proposal performs register allocation along with instruction scheduling by modeling the problem as a subgraph isomorphism problem [10,29]. Basically, we use DAGs of a program and the hardware resources (processing elements, registers and their interconnections) as inputs to our technique. The goal is to find a subgraph in the graph of hardware resources (base graph) which is isomorphic to the DAG of the program (input graph). Following subsections give a comprehensive description of the algorithm and how it works.

4.1 Subgraph Isomorphism for Instruction Scheduling and Register Allocation

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 [10]. Variations of the subgraph isomorphism have also been used to model practical problems as compiler optimizations, graph data [6], and integrated circuit testing [3].

Figure 2 shows an example of subgraph isomorphism. By using the two graphs of Figures 2(a) and 2(b) as inputs for a subgraph isomorphism algorithm, one possible result is the graph in Figure 2(c).

Since subgraph isomorphism problems were already proved as \( \mathcal{NP} \)-complete problems [29], for certain choices of inputs (input graph and base graph) there can be exponentially many occurrences so that listing all of them is impractical. Adopting efficient algorithms is a mandatory condition to use solutions to solve subgraph isomorphism problems.

We have carried out a set of heuristics and used the main ideas from the VF subgraph isomorphism library [7] to extending the algorithm proposed in [12,13] to perform scheduling and allocation. The VF algorithm finds an isomorphism if there exist one and can determine the best isomorphism (optimal result). The algorithm performs an exhaustive search in order to test all isomorphisms between the input graph and the base graph. Further optimizations were added to make the VF library suitable for our purposes.

In our technique, one of the input graphs is a Directed Acyclic Graph (DAG) of a program. The other one is named a Base Graph. To perform scheduling and register allocation, the goal of the subgraph isomorphism is to finds a subgraph isomorphic to the input DAG. Since DAGs of different sizes, geometries and weights come to our subgraph isomorphism technique, several optimizations were necessary in order to deal with those DAGs. One of the optimizations consists in splitting up a DAG whenever there are not available registers for allocation. Another optimization is to insert move nodes in the DAG to speed up the isomorphism or even to make it feasible. We have also added time constraints in the VF isomorphism algorithm to cut out unnecessary searches. By the
experiments, we could realize that these optimizations were indeed useful to schedule and allocate large DAGs (> 30 vertices). Figure 3 shows a sample of DAGs from kernels of the SPECint00 programs. Despite most of the DAGs are small (< 20 vertices), there are many large DAGs responsible for more than 80% of the execution time of our algorithm.

Integrating scheduling and allocation constitutes a new and challenging manner to deal with two activities which are conflicting sometimes. Spill code is one of these challenges. One approach to deal with spill codes by using our integrated algorithm is splitting up the DAG and call the subgraph isomorphism algorithm all over again to find an isomorphism between two (or more) new DAGs and the the base graph.

Algorithm 1 outlines the instruction scheduling and register allocation strategy.

**INPUT:** An input graph G1 and a base graph G2

**OUTPUT:** Set of instructions of the targeted architecture

Sched(DAG: G1, BASE GRAPH: G2)

1) topological_order(G1);
2) G2’=subg_iso_sched_alloc(G1,G2,tag);
3) while $(G2’==NULL)$
   4) switch(tag)
      5) case 1: base_graph_resize(G2);
      6) case 2: DAG_stretch(G1);
      7) case 3: DAG_split(G1);
      8) G2’=subg_iso_sched_alloc(G1,G2,tag);
   9) end while
10)create_targeted_architecture_instruction(G2’);

Algorithm 1. Subgraph Isomorphism Scheduling and Register Allocation Algorithm

First of all, we execute the topological_order procedure (Line 1) to perform a topological ordering in the input DAG G1. Procedure subg_iso_sched_alloc (Line 2) finds a subgraph G’2, G’2 ⊆ G2, isomorphic to G1. This procedure uses the VF subgraph isomorphism library to find out subgraph G’2. If subgraph G’2 is not found (Lines 3-9) the scheduler chooses one heuristic (Lines 5-7) and runs it over the input parameters. Variable tag (Line 4) acts as a heuristic selector. Heuristic base_graph_resize (Line 5) increases the base graph size in order to speed up the subgraph isomorphism procedure. Heuristic DAG_stretch (Line 6) transforms DAG G1 into a more flexible graph.
so that it can be easily matched to \( G_2 \). Heuristic DAG\_split(\( G_1 \)) (Line 7) changes register vertices of the DAG. Originally, all the registers vertices are featured as TRF (Temporary Register File) registers. If there are not enough TRF registers to perform the allocation (spill event), the heuristic splits up the DAG by modifying the TRF vertices to GRF (Global Register File) registers. If there are not enough GRF vertices a spill code is inserted, \( G_1 \) is splitted, and it should be rebuilt. After the matching is found, an instruction is created (Line 10).

As described, one of the input graphs for the subgraph isomorphism procedure are DAGs of the program. The other input graph is a graph that represents the available resources of the target processor. We call that new input graph an Architecture Base Graph (ABG). Next section describes how to build an architecture base graph and give some examples of ABGs in different processor architectures.

### 4.2 Architecture Base Graph

This subsection has two main objectives: first, to describe the Architecture Base Graph (ABG) and how it can be created for one architecture, and second to show some examples of ABG from non-trivial architectures.

Architecture Base Graph (ABG) is a graph that represents the available hardware resources at the time of scheduling and register allocation steps. Building ABGs correctly is a mandatory step for a successful scheduling and register allocation based on subgraph isomorphism. The ABG vertices are functional and storage units so that a register file is represented as a vertex and a single register in the pipeline can also be represented as a vertex. All functional units are represented as vertex also. Directed edges are used to represent data flow from one vertex to another. If a functional unit requires 2 input data, two edges must be at its input. The same happens for output values. The size of ABGs is defined according to the input DAGs. It is important to notice that ABGs’ height represents timing for scheduling and ABGs’ width represents amount of available hardware resources in parallel. Other architectural constraints of the functional units such as latencies and special purpose functional units (instruction specific) can also be represented by setting weights on the vertices (functional unit vertices) of the ABG.

According to Definition 3 in Section 3, the subgraph isomorphism procedure matches graph \( G_1 \) to graph \( G_2 \) in order to obtain a subgraph \( G'_2 \). In our technique, \( G_1 \) is characterized by the DAGs of input programs and \( G_2 \) is the ABG. Subgraph \( G'_2 \) is the set of necessary hardware elements to run each vertex (instruction of the source code) of the DAG at a specific time.

Considering the DAGs representation of the instructions of the program source, DAGs’ vertices are the opcodes of instructions, DAGs’ edges correspond to the source (producer) of operands. DAGs’ edges can carry a weight according to the execution cycle of an instruction. Since ABGs represent available hardware resources, those graphs have a minimum size (number of vertices and edges). On the other hand, to match to the DAGs’ geometry at scheduling and allocation time, an ABG could be “unrolled” in order to represent the available resources at time and to provide a subgraph isomorphic to the input DAG. In this way, the ABG is built based on the hardware resources and on the DAGs’ geometry. As an example of an ABG, Figure 4 illustrates a ABG creation based on an input DAG of height 3 (Figure 4(a)) and an processor architecture with two parallel functional units, and one global register file (Figure 4(b)).
Fig. 4. Example of ABG’s creation.

One can notice that creating an ABG just based on the architecture resources (Figure 4(b)) is not enough to match to the input DAG. By the example, it is necessary to unroll the original ABG three times in order to couple with the height of the input DAG. The ABG representing the unrolled hardware elements is shown in Figure 4(c) where is possible to observe that the global register file is characterized by a rectangular vertex while the functional units are represented by rounded vertices. The unrolling represents the resources usage along the time. If one look at the ABG as two axes: x and y; x axis means the resources available at the same time; y axis represents the available resources at each step of computation (one clock cycle). Observe that the ABG in Figure 4(c) is ready to act as an input to the subgraph isomorphism procedure since is large enough to have a subgraph isomorphic to the input DAG 4(a). In Figure 4(d) the black squares in the global registers represents the write registers used by the operations. In the example, the subgraph resultant is comprised of left rounded vertex at the top of Figure 4(d), the two rounded vertices at the middle of the Figure and the two rounded vertices at the bottom. Notice also that the DAG’s bypassing edge in Figure 4(a) between operation addi and operation and is represented by the edge between the first register file and the two last rounded vertices in Figure 4(d), which is only possible because the output register from addi is not written by any operation before the and, as expected. Since there is no edge between functional units, all edges of the ABG have the same weight. But it is important to notice that ABGs can also carry weight on the edges in order to drive the subgraph isomorphism procedure to schedule operations closer to their sources.

The following subsections presents some ABGs for distinct architectures ranging from VLIW to DSP showing how different number of functional units, register bank and interconnection varies between them and how to model the ABG in each case.

4.2.1 Base Graphs on Constrained Architectures

2D-VLIW (Two-Dimensional Very Long Instruction Words) [14, 15] is a constrained-interconnection computer architecture that exploits application performance 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 and operands 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. The 2D-VLIW
architectural arrangement is depicted in Figure 5.

Figure 5(a) sketches the datapath and its major architectural components. Figure 5(b) details the matrix of functional units (FUs) where four operations can be executed by four FUs at each execution stage $EX_1, EX_2, EX_3, EX_4$ of the pipeline. Figure 5(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 $SelOpnd_1$, $SelOpnd_2$ and $SelOperation$ input selection signals are available from the pipeline registers.

In Figure 6 a continuous edge $e_{ij} = (FU_i, TR_j)$ corresponds to a write port from $FU_i$ to register $TR_j$. A dashed edge represents a read port used by an FU. Each FU stores its results in only one temporary register but can read operands from two different temporary registers according to the interconnection.

Figure 7 shows an example of an ABG after 3 unrolling operations. For the sake of simplicity, the GRF and TRFs were left out.
Another example of constrained processor architecture is the HPL-PD [19] processor. HPL-PD is parametric processor architecture dedicated to research in ILP designed by HP Labs and its university partners in the middle of 90’s. Since then, HPL-PD has been used as a basic platform for hardware and software research based on instruction level parallelism. As a parametric architecture, the HPL-PD processor accepts different configurations of hardware elements in terms of organization and amount. The HPL-PD ISA is similar to that of a RISC-like load/store architecture, with standard integer, floating point, and memory operations. In addition, it provides a number of advanced features for enhancing and exploiting parallelism in programs. The basic architecture has functional units and hardware resources to support speculative and predicated execution, branch mechanisms, and several register files. Figure 8 sketches the HPL-PD architecture found in the Trimaran compiler.

Code generation for an architecture like HPL-PD is a challenge since there are dedicated functional units connected to the register banks. Those connections should remain on a base graph representation of HPL-PD so that specific vertices of the input DAGs can be scheduled to its correct functional unit and allocated to the right register file. Figure 9 shows the ABG for the HPL-PD architecture. For the sake of simplicity, the example shows an ABG of height 2 (two levels of functional units surrounded by register file banks according to the organization of the original architecture in Figure 8). The register files are represented by rectangles and the functional units are cycles in the ABG.

4.2.2 Examples of Architecture Base Graphs on DSPs and Media Processors

A Trimedia TM 1000 processor [17, 30] has 28 functional units, one 128 × 32 bits register file, five issue slots, five write ports to the register file, functional units with multi-cycle latencies. Figure 10 sketches an overview of the Trimedia TM 1000 processor. An ABG of the TM 1000 processor contains vertices representing the number of functional units working together at the same clock cycle (in parallel). Since only five units can run instructions at the same clock cycle, the corresponding ABG shows the interconnection between the register file and five functional units.
Figure 10 shows an example of an ABG for the TM 1000 processor. Notice that the ABG was built regarding a 3 times unrolling. The ABG represents the GRF as a squared vertex and the functional units as rounded vertices. Similar to the example in Section 4.2, the ABG for the TM 1000 architecture processor does not have weights on its edges since there are no edges between functional units. On the other hand, since each functional unit of the TM 1000 processor is dedicated to a set of instructions, the TM 1000 ABG has weights on its functional units vertices in order to represent that feature. For the sake of simplicity, we have left out some edges from the registers files to the functional units.

Another example of an ABG is shown for the TMS320c62x [18] DSP processor. A TMS320c62x is a DSP processor with eight functional units, two global register files with 15 × 32 bits each. Figure 12 depicts an overview of the TMS320c62x architecture.

Figure 13 shows the equivalent ABG for the TMS320c62x DSP processor considering a 3 times unrolling. The representation of GRF and FUs follows the same previous examples. As the ABG becomes larger (more unrollings), the edges are increased greater than the vertices so that the complexity of the ABG creation procedure is bounded.
to the edges. For some constrained architectures [16], the number of edges grow quadratically to the number of vertices.

Fig. 13. ABG (3 times unrolling) for the TMS320c62x DSP processor.

4.3 Case Study: Scheduling and Register Allocation for the HPL-PD Architecture

In this paper, we have adopted the HPL-PD processor architecture, presented in subsection 4.2.1, as the native processor to generate code by using our subgraph isomorphism strategy for scheduling and register allocation. The configuration and organization of our adopted HPL-PD architecture is found in the Trimaran compiler infrastructure [5].

The inputs of the HPL-PD instruction scheduling and register allocation strategy are DAGs generated by the Trimaran compiler. The HPL-PD architecture base graph is built using an in-house procedure that accepts DAGs as inputs.

After building the ABG for the input DAGs, those DAGs are passed to the scheduler on a hyperblock [25] 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 and edge’s latency information and performs the matching between the DAG and the base graph. Despite taking operations’ latency and explicit data dependencies, our algorithm also obeys some non-explicit data dependency such as saving passing parameters before procedure calls, and so on.

One example of our strategy considering the HPL-PD ABG is presented in Figure 14. Figure 14(a) shows the input DAG characterized with vertices representing operations (circles) and store points (rectangles). The operations vertices will be scheduled to their corresponding functional units vertices in the ABG. Similarly, store points will be matched to available register vertices in the ABG. It is important to observe that the match represents the register allocation procedure.

Fig. 14. Example of scheduling and register allocation on the HPL-PD architecture.
while all store point vertices were allocated to positions (registers) of the register file bank. Although there are memory operations and integer/logic operations, meaning that the lw operation must be scheduled on a memory functional unit vertex and the integer/logic operations must be scheduled on integer functional unit vertex, they share operands, thus forcing an allocation on a register file (GRF) which is shared by those functional units. We emphasize that Figure 14(b) is one possible result. One can notice that there are other schedulings and registers that could be used.

5 Experiments and Results

This Section presents the experimental results of the scheduling and register allocation algorithm. All the experiments were based on the HPL-PD processor architecture using a subset of programs of the SPEC2000, SPEC2006, and MediaBench suites. We have compared our results considering execution time of our integrated approach to the execution time of scheduling and register allocation algorithms carried out by the Trimaran compiler infrastructure. Additionally, we compare the scheduling cycles obtained by our strategy and the Trimaran compiler in order to clarify the effects of our approach to the user code. Table 1 presents the scheduling and register allocation performed by Trimaran (2nd and 3rd columns) and our integrated approach (4th and 5th columns).

<table>
<thead>
<tr>
<th>Programs</th>
<th>Trimaran Cycles</th>
<th>Trimaran Time (sec)</th>
<th>Subiso Cycles</th>
<th>Subiso Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>401.bzip2</td>
<td>274</td>
<td>0.42</td>
<td>214</td>
<td>0.38</td>
</tr>
<tr>
<td>429.mcf</td>
<td>252</td>
<td>0.21</td>
<td>194</td>
<td>0.26</td>
</tr>
<tr>
<td>443.milc</td>
<td>654</td>
<td>0.5</td>
<td>207</td>
<td>2.01</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>251</td>
<td>0.24</td>
<td>243</td>
<td>0.26</td>
</tr>
<tr>
<td>464.h264hef</td>
<td>269</td>
<td>0.19</td>
<td>174</td>
<td>0.33</td>
</tr>
<tr>
<td>475.libo</td>
<td>1948</td>
<td>1.71</td>
<td>284</td>
<td>3.11</td>
</tr>
<tr>
<td>175.vpr</td>
<td>400</td>
<td>0.42</td>
<td>865</td>
<td>1.69</td>
</tr>
<tr>
<td>176.gcc</td>
<td>683</td>
<td>1.15</td>
<td>496</td>
<td>0.65</td>
</tr>
<tr>
<td>197.parser</td>
<td>282</td>
<td>0.21</td>
<td>239</td>
<td>0.32</td>
</tr>
<tr>
<td>300.twolf</td>
<td>558</td>
<td>0.54</td>
<td>160</td>
<td>0.36</td>
</tr>
<tr>
<td>rasta</td>
<td>433</td>
<td>1.01</td>
<td>284</td>
<td>1.32</td>
</tr>
<tr>
<td>epic</td>
<td>555</td>
<td>0.84</td>
<td>380</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Table 1. Scheduling and register allocation results.

One can notice that our approach brings improvements in static scheduling cycles in 11 out of 12 evaluated programs. The reduction in cycles for the generated code ranges from 3%-85%. Most of the improvements were due to the spill cycles generated by the register allocator of Trimaran [20]. A few improvements were due to the arrangement of the scheduling. The Trimaran scheduler adopts a conservative approach to the memory operations. On one hand it brings a faster scheduling strategy as can be seen by comparig the 3rd and 5th columns in Table 1. On the other hand, it minimizes some memory optimizations at scheduling time. Trimaran adopts a cycle based scheduler and the operations are assigned to the resources soon they came to the scheduler. As an example, Figure 15 shows a DAG extracted out from a kernel (procedure update_bb) of the 175.vpr program. There are several roots nodes which can be scheduled as soon as possible. The Trimaran’s cycle based scheduler provides an instruction scheduling using 21 cycles. Our approach, on the other hand, schedules the same DAG by using 11 cycles.
Since Trimaran adopts a step-based code generation strategy by using pre-pass scheduling, register allocation, and post-pass scheduling, some false dependencies come up on the code. False dependencies minimize the register and functional units usage by the operations. Specifically, we have noticed that unrolled loops pose several false dependencies on using the same base register for array indexes on the Trimaran compiler. By using DAGs, our algorithm spread the indexes along the available registers in order to improve the ILP over time.

We have performed another interesting experiment to figure out whether there are improvements by considering the code generated by Trimaran. This new experiment rebuilds DAGs from the code after the post-pass scheduling. The extracted code is input to our strategy. Table 2 presents the results by considering the same metrics of Table 1 (scheduling cycles and execution time) and the same kernels of programs.

Once again, even after the Trimaran’s scheduler and register allocator procedures, our strategy has found some opportunities to reduce the scheduling cycles for the final code. Comparing the results in Table 2 to the Trimaran scheduling and register allocation values in Table 1, the improvements range from 3% up to 29%. Now, the improvements are only due to the better arrangement of the instructions by our technique. These improvements have most impact on regions like the one presented in Figure 15.

### 6 Related Work

In this Section we outline a number of research work proposing new integrated approaches for instruction scheduling and register allocation. In addition, most of the research work have focused on constrained architectures to generate code by adopting unified strategies.
for scheduling and allocation.

The Unified ReSource Allocation (URSA) is a unified approach targeted to VLIW processor architectures [2]. The URSA adopts a DAG representation of the program for performing both register allocation and instruction scheduling. The approach uses a DAG that represents resource reuse information to measure a program’s resource requirements. DAGs of the code are decomposed into set of chains to determine the resources usage and to maximize the performance. The technique consists of three major components: (1) a representation of the program and requirements for each resource, (2) algorithms to measure the resource requirements, and (3) transformations that reduce the resource requirements to levels supported by the target architecture. The goal is to modify the dependence DAG identifying the regions of the DAG that require reductions in resource requirements to meet the levels supported by the target machine. Contrary to others techniques, URSA performs resource assignment after all resources are allocated, reducing the interaction between the two allocation problems (instruction scheduling and register allocation). The authors do not present results of their technique.

[22] and [28] present the aspects of register allocation for irregular architectures. Concerning the register allocation program, classical techniques such as graph-coloring approaches are mostly used, but graph-coloring does not match the needs of register allocation for architectures with non-orthogonal instruction sets and small register files. In [22] was proposed an approach based on 0-1 integer programming (IP). The x86 architecture was chosen for experiments since it includes a large variety of register irregularities. The IP model was extended so that some register irregularities could be precisely modelled. The results showed that IP register allocation is well suited for irregular-register architecture. In [28] was proposed an other approach based on a heuristic for partitioned boolean quadratic optimization problem (PBQP) for the global allocation.

In [21] a progressive register allocator for irregular architectures is proposed. The authors represent the intricacies of irregular architectures by using a multi-commodity network flow (MCNF) model. The approach solves problems related to network flows in order to find feasible solutions and attempt to improve them. Experiments were performed using the x86 architecture replacing the local register allocation of the gcc compiler. The main idea is to reduce the register allocation problem to an MCNF problem thus obtaining a proper and progressive solution for register allocation.

[9] presented an integrated algorithm for instruction selection and register allocation but focused on instruction set with mixed instruction formats for 16- and 32-bit instructions on the ARCompact ISA. They got a good code reduction of 16.7% and also a performance gain of 17.7%. This paper differs from our work because they have an instruction set architecture with two representations of the same instruction and focused on selecting the best one for each case.

7 Conclusions

An integrated approach for instruction scheduling and register allocation was presented in this paper. Our approach focus on the matching of large code regions to the available hardware resources. Basically, the technique matches an input DAG to the Architecture Base Graph in order to find out the resources available to run the code.

Besides the algorithm, our technique proposes to look at the modelling of the hardware resources as an Architecture Base Graph (ABG). ABG is an useful way to describe the hardware resources and, further, expose all the resources to a scheduling and register allocation
algorithm. We have presented the application of ABGs over different processor architectures.

The experiments show the gains of the subgraph isomorphism strategy compared to the cycle scheduling and vertex colouring approaches used by the Trimaran compiler. Our technique provides a static scheduling up to 85% better than the Trimaran’s approaches. Future work is focused on implementing our integrated algorithm on the LLVM compiler infrastructure and evaluating it on large sets of programs.

References


