Dependence Analysis of VLIW Code for Non-Interlocked Pipelines

Ralf Dreesen
University of Paderborn
Department of Computer Science
rdreesen@upb.de

Thorsten Jungeblut
University of Paderborn
Heinz Nixdorf Institute
tj@hni.upb.de

Michael Thies
University of Paderborn
Department of Computer Science
mthies@upb.de

Uwe Kastens
University of Paderborn
Department of Computer Science
uwe@upb.de

Abstract

Data dependence analysis (DDA) on assembly code is a frequent problem in compilers and program analysis tools. The fundamentals of a DDA on code for simple processors are well understood.

We propose a DDA method, that is applicable for a wider range of processors. This includes VLIW processors and processors with delayed branches and delayed register accesses. For these architectures, the instruction order may no longer match the order of register accesses, which necessitates a new analysis technique. The result of our analysis method is an instruction dependence graph (IDG), which also contains information on minimal instruction distances. For the mentioned architectures and allocated registers, the IDG may be cyclic. We discuss this phenomenon and outline an algorithm to reschedule such IDGs. We successfully implemented the DDA method and a respective scheduler in our compiler for the CoreVA VLIW architecture.

1. Introduction

A data dependence analysis (DDA) yields the dependencies between instructions in a sequence of assembly code. It is typically used to compute instruction dependencies for a subsequent scheduling. The input for a DDA is a sequence of assembler instructions. The result is an instruction dependence graph (IDG), in which each node corresponds to an instruction and an edge represents a dependence. The IDG describes the order of instructions, that must be obeyed by the scheduler. If there is an edge $X \rightarrow Y$, the instruction X must be issued before Y.

For regular processors, data dependence analysis on sequential assembler code is rather simple. During a single top down pass, read and write accesses are tracked for each register. Then, edges are inserted into the IDG for consecutive read and write accesses on the same register. The resulting instruction dependence graph

ODES-8
April 25, 2010 - Toronto, Canada

Substantial parts of the research described in this paper were funded by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung — BMBF) registered there under grant numbers 01BU0661 (MxMobile) and 01BU0643 (Easy-C).

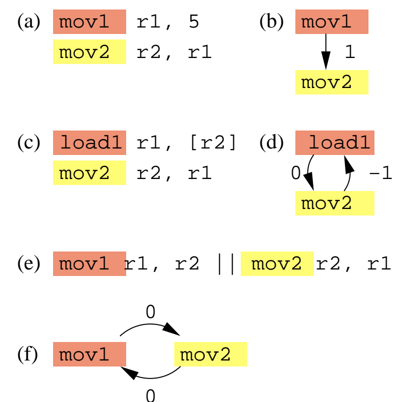

Figure 1. Examples of assembler code and dependence graphs.

(IDG) is a directed acyclic graph (DAG) for these regular processors. The example in Figure 1a shows two assembly instructions that write resp. read the same register `r1`. The IDG in 1b therefore contains a dependence edge. The edge is labeled with the distance 1, expressing that the second instruction must be issued at least one cycle after the first.

We propose a DDA on basic block level, which is suited for non-interlocked and partially-interlocked processors with arbitrary read and write latencies. For this class of processors, the order of assembly instruction may no longer match the order of register and memory accesses. The DDA can also be applied to code with allocated registers. Anti-dependencies, that originate in reuse of physical registers are detected and included in the IDG. The IDG can therefore be used for a rescheduling, that does not invalidate the register allocation. Assume that the `load` instruction in Figure 1c writes `r1` deferred by a latency of 2 cycles without interlocking. This means, that the result of the `load` instruction will appear only to instructions, that are issued at least two cycles after issuing `load`. The `mov` instruction will therefore read the old value of `r1`, not the result of the `load` instruction. Due to this anti-dependence, the `mov` instruction must logically precede the `load` instruction but may even be scheduled up to one cycle after the `load` instruction. This constraint is expressed by the edge labeled with distance -1 in Figure 1d. A second dependence edge is caused by an anti-

dependence via r2. The resulting IDG contains a cycle that allows two valid schedules of load and mov. The instructions may be scheduled as shown in Figure 1c or both instructions may be issued concurrently.

Our DDA method may also be applied on code for parallel processors with synchronous instruction execution, like VLIW. In Example 1e, both mov instructions are executed in parallel. The values in r1 and r2 are therefore swapped. During a rescheduling, the concurrent execution must be preserved. This constraint is expressed by the cycle in the instruction dependence graph in 1f. The edge from mov1 to mov2 originates from an anti-dependence via r2 and the other edge from an anti-dependence via r1.

Overview In the next section, we first relate our paper to existing work. In Section 3 the processor model, on which our method is based is introduced. The actual DDA method is described in Section 4. The DDA is split into 4 phases: computation of access lists (4.1), access dependence graphs (4.2), instruction dependence graphs (4.3) and finally the reduction of instruction dependence graphs (6). Properties of the instruction dependence graph are discussed in Section 5. After describing the DDA method, we show how to apply the method to analyze control dependencies (7) and data dependencies of memory accesses(8) . An instruction scheduler that is able to reorder cyclical dependent instructions is outlined in Section 9. The DDA method was evaluated using the CoreVA VLIW architecture [2, 4]. The results are presented in Section 10.

2. Related Work

In [7] Muchnick describes a standard data dependence analysis for assembly code. The complexity of the algorithm is $\mathcal{O}(n^2)$, where n is the number of instructions. The method can be applied to a linear, non-parallel sequences of assembly code and yields a data dependence graph that is guaranteed to be a DAG. The direction of an dependence always corresponds to the instruction order in the analyzed code. The decision, if two instructions are dependent is encapsulated in a *conflict* predicate. The implementation of this predicate is left open. Since the algorithm assumes non-parallel code and does not consider dependencies against instruction order, it can not be applied to the processor architectures listed in Section 10.1.

In [8] an overview of IDG construction algorithms is given. All presented algorithms assume the IDG to be a DAG. This is not true for processor architectures with special non-interlocked instruction latencies.

In [10] a data dependence analysis for ISAs with a 0-cycle read latency and an n-cycle write latency (like TI C6000 family) is presented. The proposed method transforms assembly code of the original ISA into assembly code of a modified ISA, where all instructions have a delay of 0 (i.e a write latency of 1). To preserve the program semantics, additional assembly instructions, artificial registers and predicated execution is added to the program. To derive the instruction dependence graph, classical analysis algorithms are applied on the modified code, which only contains 0-delay instructions. The result is a control dependence graph in SSA form, which only contains WR-dependencies (flow dependencies). The dependence graph does therefore not consider the original register allocation. The method is well suited to derive pure data flow information from the assembly code, but it is not applicable for rescheduling, since it discards the result of the previous register allocation.

In [1] a control flow analysis is presented for processors with delayed branches and non-canceled branches in delay slots of other branches. An example of such a processor is the C6000 family from Texas Instruments. For the analysis, the algorithms considers all execution paths of the program while keeping track of pending branches. If the processor architecture permits branches in delay slots of other branches, this CFA method can be used to derive the control flow graph prior to the data dependence analysis. If non-canceled branches in delay slots are not permitted, the simple control flow analysis described in Section 7 which builds on our DDA method may be used.

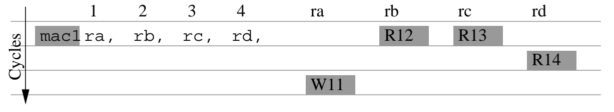

Figure 2. Read and write accesses of a multiply-accumulate instruction.

3. Processor Model

This section introduces the processor model for the proposed DDA method. The processor model is based on the processor specification language UPSLA [5]. We assume a parallel processor with synchronous instruction execution, like a VLIW processor. Each VLIW is composed of a set of instructions. Non-parallel processors are a special case of this processor class.

Each instruction reads a set of source registers and writes a set of destination registers. The timing for reading and writing registers is expressed as latency. The latency is a difference between two absolute cycles and thereby relative. In the following we will only focus on register accesses. The extension to memory accesses is covered in Section 8.

3.1 Latencies

Definition of Latency Read and write accesses may have non-uniform latencies. The *latency* expresses the delay of an access to a registers with respect to the issue cycle of an instruction. A read latency designates the delay of a register read access and a write latency the delay of a register write access. Note, that we regard accesses without interlocking. For example does an instruction reading a register r not wait for any previously issued instruction to write its result to r. Instead, the old value is read.

The *cycle* of an access is defined as the sum of the instruction issue cycle and the access latency. If C_X is the cycle of an instruction X causing an access G with latency L_G, the access cycle C_G is

$$C_G = C_X + L_G$$

In the following, X, Y will denote instructions and G, H will denote accesses. Accorindgly C_X, C_Y will be instruction issue cycles and C_G, C_H will be register access cycles.

Definition of Accesses The timing of read and write accesses is defined as follows: A value written by a write access G will be read by a read access H, if $C_G \leq C_H$ holds. Otherwise, H will read the previous value. Note, that the value written by G will even be read by H, if both accesses belong to the same cycle ($C_G = C_H$)[1].

For two write accesses G and H, the register will finally contain the value of H, if $C_G < C_H$ holds. Concurrent write accesses ($C_G = C_H$) are not allowed and will be detected by our DDA method.

To demonstrate latencies, Figure 2 visualizes the access latencies of the multiply-accumulate (mac) instruction, which is part of the CoreVA instruction set. In the figure, mac1 identifies the first mac instuction in an assembly listing. The mac instruction has four operands ra to rd. The instruction causes a write access W11 to the first operand ra. The access has a latency of two cycles, hence W11

[1] This semantic of concurrent read and write accesses is just a definition for our access model. It is not related to the resolution of concurrent accesses in hardware.

appears two rows below `mac1`. In the following, we denote read accesses `Rxy` and write accesses `Wxy`, where `x` is the instruction number and `y` the operand number of the instruction.

In contrast to the `mac` instruction, simple instructions like arithmetic and logical instructions typically have a latency of 1 for write accesses and a latency of 0 for read accesses. The result of an arithmetic instruction is read by a succeeding arithmetic instruction, but not by a concurrent one. Instructions with exceptional latencies include multiply, divide, load and store instructions.

4. Data Dependence Analysis Method

Due to non-uniform and non-interlocked read and write latencies, the order of register accesses may differ from the instruction order. Our method does therefore first extract the set of accesses from the instruction sequence. The accesses are grouped by the respective register that is accessed to form access lists for each register (Section 4.1). From each access list, an access dependence graph is derived (Section 4.2). By merging the access dependence graphs, an instruction dependence graph is created (Section 4.3). This instruction dependence graph is finally simplified by removing redundant edges (Section 6).

The example in Figure 3 shows the intermediate analysis results for a given instruction sequence. The figure will be used as a running example in the following sections.

4.1 Construction of the Access List

In this section, we describe the construction of access lists from a linear sequence of assembly code. Our method first extracts the register accesses from all instruction. These accesses are then grouped by the accessed register to form access lists. The access lists are sorted ascending by the access cycle as primary sort criterion and the access type as secondary criterion, where write accesses precede read accesses. This order corresponds to the dependence condition $C_G \leq C_H$ as defined in Section 3.1. As a result of grouping and sorting, there are only dependencies from an access to succeeding accesses of the same access list. This enables a fast derivation of dependencies as described in the next section.

Figure 3a shows a short sequence of assembly code. In the first cycle the `add` and `sub` instructions are issued in parallel. In the second cycle the `mac` instruction is issued and so on. The semantics and the latencies of the `mac` instruction are defined in Section 3. For the remaining instructions, the first operand is the destination register and the other operands are source registers. The latency is 0 for read accesses and 1 for write accesses. Figure 3b shows the resulting access lists for registers `r1` and `r2`. The access `R32` in cycle 2 of access list `r1` represents the read access of the the second operand of the `mac3` instruction (instruction number 3).

The following algorithm constructs an `accessList` for each register from the given `instructionList`. The runtime complexity of the algorithm is $\mathcal{O}(n \log n)$ where n is the number of instructions.

```
foreach instruction in instructionList:
 foreach access in accesses[instruction]
  addToAccessList( accessList[register[access]], access)
  cycle[access] := cycle[instruction]+latency[access]
foreach accessList in accessLists
 sort accessList by cycle[access] and type
```

4.2 Construction of the Access Dependence Graph

The output of the previous phase is one sorted access list for each register. This phase transforms each access list into an access dependence graph (ADG). A node in the ADG represents a register access and an edge expresses a dependence between two register accesses.

4.2.1 Dependencies

A dependence is either a WR-dependence (flow-dependence), a RW-dependence (anti-dependence) or a WW-dependence (output-dependence). If a write access W is followed by a read access R without another intervening write access, a WR-dependence edge is inserted into the ADG. Accordingly, a RW-dependence edge is inserted for a read access followed by a write access and a WW-dependence edge for a write access followed by another write access. Note, that code is in SSA form only contains WR-dependencies. However, after register allocation, all types of dependencies may show up.

WR and WW-dependencies To insert WR- and WW-dependencies, the accesses in the access list are visited top-down in access order. When a read access is visited, a WR-edge is inserted into the graph from the node of the last visited write access to the node of the current read access. If a write access is visited, a WW-dependence is inserted accordingly.

RW-dependencies To insert RW-dependencies, the accesses are visited bottom-up, starting with the last access cycle. When a read access is visited, a RW-edge is inserted into the access graph from the current read access to the last visited write access.

Figure 3c shows the access dependence graphs for `r1` and `r2`. Since the read access R22 is followed by the write access W11, there is a RW-dependence edge between these two nodes. There is a WR dependence edge between W11 and R42, as W11 is followed by R42 without an intervening write access. There is *no* edge between W11 and R52, since `r1` is written by W31 in-between.

4.2.2 Instruction Distances

So far, we described a method, which creates an access dependence graph for each register. The ADG specifies the order of accesses, that must be obeyed by the scheduler. For the dependence $G \rightarrow H$ of two accesses this means, that G must precede H with a distance of at least $D^{\min}_{GH}$, which is

$$D^{\min}_{GH} = \begin{cases} 0 & \text{for a WR-dependence} \\ 1 & \text{for a RW/WW-dependence} \end{cases}$$

The distance $D^{\min}_{GH}$ is derived from the definition of accesses in Section 3.1. The definition says, that $G \rightarrow H$ is a WR-dependence, if the following holds:

$$\begin{aligned} C_G &\leq C_H \\ \Leftrightarrow \underbrace{C_H - C_G}_{D_{GH}} &\geq 0 \\ \Leftrightarrow D^{\min}_{GH} &= 0 \end{aligned}$$

Accordingly, $G \rightarrow H$ is a WW- or RW-dependence, if the following holds:

$$\begin{aligned} C_G &< C_H \\ \Leftrightarrow \underbrace{C_H - C_G}_{D_{GH}} &\geq 1 \\ \Leftrightarrow D^{\min}_{GH} &= 1 \end{aligned}$$

As the scheduler allocates instructions rather than single accesses, these constraints on accesses must be transferred to constraints on instructions. The relation between minimal access distances and minimal instruction distances is illustrated in Figure 4. A dependence $G \rightarrow H$ results in a dependence $X \rightarrow Y$ of the according instructions. The minimal distance $D^{\min}_{XY(GH)}$ of $X \rightarrow Y$ resulting from $G \rightarrow H$ is

$$D^{\min}_{XY(GH)} = L_G - L_H + D^{\min}_{GH}$$

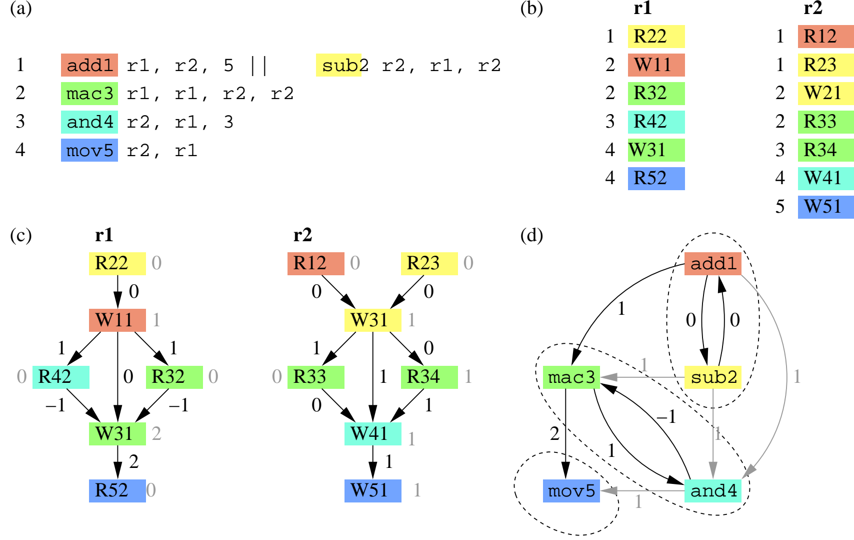

Figure 3. DDA Example: (a) Instruction Sequence, (b) Access Lists, (c) Access Dependence Graphs, (d) Instruction Dependence Graph (Reduced).

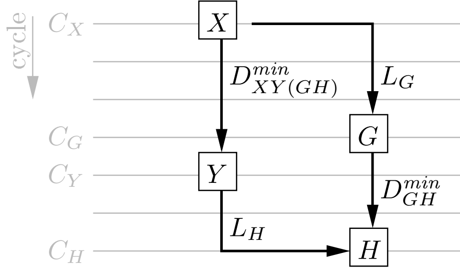

Figure 4. Relation between minimal access and instruction distances.

The distance $D^{\min}_{XY(GH)}$ can formally be derived from the latency definition in Section 3.1:

$$C_H = C_Y + L_H \wedge C_G = C_X + L_G$$
$$\Rightarrow \underbrace{C_X - C_Y}_{D_{XY(GH)}} = L_H - L_G + \underbrace{C_H - C_G}_{D_{GH}}$$
$$\Rightarrow D^{\min}_{XY(GH)} = L_G - L_H + D^{\min}_{GH}$$

Regarding the relation between the actual distance in the analyzed code and the minimal distances, the following holds:

$$D^{\text{code}}_{XY} \geq D^{\min}_{XY}$$
$$D^{\text{code}}_{GH} \geq D^{\min}_{GH}$$

In Figure 3c, accesses are annotated with their latency (gray), and edges are annotated with the minimal instruction distances $D^{\min}_{XY}$. For example, the distance between R22 and W11 is $L_{R22} - L_{W11} + 1 = 0$ and the distance between W31 and R52 is $L_{W31} - L_{R52} + 0 = 2$.

Putting it all together, the following algorithm creates an `accessGraph` for each `accessList`. The distance of an edge is contained in `distance[edge]`. The runtime complexity of this algorithm is $\mathcal{O}(n)$, where n is the number of assembly instructions.

```
foreach accessList:
 accessGraph = newAccessGraph(accessList)
 /* Create WR and WW dependencies */
 previousWrite := NULL
 foreach access in accessList:
  if previousWrite!=NULL
   edge := createEdge(accessGraph,previousWrite,access)
   distance[edge] := latency[previousWrite]-latency[access]+1
  if (type[access]=write)
   previousWrite := access
 /* Create RW dependencies */
 nextWrite := NULL
 foreach access in accessList in reverse order:
  if nextWrite!=NULL
   edge := createEdge(accessGraph,access,nextWrite)
   distance[edge] := latency[access]-latency[nextWrite]
  if (type[access]=write)
   nextWrite := access
```

4.3 Construction of the Instruction Dependence Graph

The instruction dependence graph is derived from the access dependence graphs, by merging all access nodes, that belong to the same instruction instance. Multiple edges between two nodes are also merged. The distance $D^{\min}_{XY}$ of a merged edge is the maximum of all edges $D_{XY(GH)}$ merged into it.

Figure 3d shows the merged IDG of the two ADGs in Figure 3c. The Nodes W11 and R12 are mapped to the `add` node. The edge between R22 and W11 is mapped to the edge from `sub` to `add`. The edges from W11 to R32 and from W11 to W31 are both mapped to the `add`→`mac` edge. The weight of this edge is 1, which is the maximum of the two originating edges.

The following algorithm constructs the instruction dependence graph (`instrGraph`). The runtime complexity of this algorithm is $\mathcal{O}(n)$, where n is the number of instructions. There are $\mathcal{O}(n)$ edges (see 5) and `lookupEdge` has a complexity of $\mathcal{O}(1)$, if we use an adjacency matrix for `instrGraph`.

```
instrGraph = newInstructionGraph()
foreach accessGraph:
 foreach edge in accessGraph:
  fromInstr := instruction[fromAccess[edge]]
  toInstr   := instruction[toAccess[edge]]
  instrEdge := lookupEdge(instrGraph,fromInstr,toInstr)
  if instrEdge=NIL then
   instrEdge := createEdge(instrGraph,fromInstr,toInstr)
   distance[instrEdge] := distance[edge]
  else
   distance[instrEdge] := max(distance[instrEdge],distance[edge])
```

5. Properties of the Instruction Dependence Graph

In the previous section, we presented an algorithm to construct the IDG. Based on this algorithm and previous definitions, we prove some important properties of the IDG in this section. These properties are use in run time considerations and the outlined scheduling algorithm in Section 9.

5.1 The IDG has $\mathcal{O}(n)$ edges

The number of edges is $\mathcal{O}(n)$, where n is the number of instructions. For typical processors, the number of accesses of an instruction is limited by a constant. The total number of accesses is therefore $\mathcal{O}(n)$. In the algorithm in 4.2, the `createEdge` function is called at most twice for each access. The total number of edges in the ADG is therefore $O(n)$. An edge in the ADG can result in at most one edge in the IDG. The number of edges in the IDG is therefore $\mathcal{O}(n)$.

5.2 The edge weight may be a positive or negative integer

The weight of an edge e in the IDG is the maximum of all edges in the ADGs, that are mapped to e. Let $G \rightarrow H$ be the ADG edge, that is mapped to x and that has maximum weight. The weight of $G \rightarrow H$ is given by $L_G - L_H + D^{\min}_{GH}$. Since L_G and L_H are independent positive integers, the difference can be any positive or negative integer.

5.3 The graph may contain cycles

The motivating example in Figure 1 presents two cycles in IDGs caused by concurrent execution and non-uniform latencies.

However, under certain conditions cycles will not show up. If the following two rules hold, IDG is guaranteed to be a DAG:

Rule 1: The code must be in SSA form.

Rule 2: All read accesses of an instruction X must precede all write accesses of X:

$$\forall X \in \text{Instrs}, G \in \text{Writes}(X), H \in \text{Reads}(X) : L_H < L_G$$

We prove by contradiction, that the IDG does not contain a cycle, if both rules hold: Suppose, that $(X_1, \ldots, X_n, X_{n+1})$ is a cycle in the IDG via instructions X_i and $X_{n+1} = X_1$. Since the code is in SSA form, the IDG only contains WR-dependencies. Let G_i be the write access of X_i and H_{i+1} be the read access of X_{i+1}, that cause the WR-dependence $X_i \rightarrow X_{i+1}$. According to Section 3.1 the condition $C_{G_i} \leq C_{H_{i+1}}$ must hold. Since H_i is a read access and G_i is a write access of instruction X_i, rule 2 says, that $L_{H_i} < L_{G_i}$ must hold. Thus,

$$\begin{aligned}
&\underbrace{L_{H_i} + C_{X_i}}_{C_{H_i}} < \underbrace{L_{G_i} + C_{X_i}}_{C_{G_i}} \wedge C_{G_i} \leq C_{H_{i+1}} \\
\Rightarrow & C_{H_i} < C_{G_i} \leq C_{H_{i+1}} \\
\Rightarrow & C_{H_1} < C_{H_2} < \ldots < C_{H_{n+1}} = C_{H_1}
\end{aligned}$$

must hold, which is contradictory. Thereby we can conclude, that the IDG does not contain a cycle if rule 1 and rule 2 hold.

Rule 2 holds for all processor architectures, that are known to us. However, Rule 1 may not hold after register allocation and therefore the IDG may contain cycles via anti-dependencies.

5.4 The sum of edge weights of each cycle is not greater than 0

Assume that the IDG contains a cycle $X_1, \ldots, X_n, X_{n+1}$ via instructions X_i and $X_{n+1} = X_1$. Let $D^{\text{code}}_{X_i, X_{i+1}}$ be the instruction distance between instructions X_i and X_{i+1} in the code, that is analyzed. The sum of these distances in the cycle is 0.

$$\sum_{i=1}^{n} D^{\text{code}}_{X_i, X_{i+1}} = 0$$

Since $D^{\min}_{XY} \leq D^{\text{code}}_{XY}$ holds (Section 4.2.2), the sum of edge weights (minimal distances) in the cycle is not greater than 0:

$$\sum_{i=1}^{n} D^{\min}_{X_i, X_{i+1}} \leq 0$$

6. Reduction of Instruction Dependence Graph

The IDG, which is constructed as described in the previous sections can be used as input for a scheduler. However, the graph typically contains many redundant edges, which may slow down certain schedulers and makes the IDG hard to read. In this section, we present an $O(n^3)$ Algorithm to remove all redundant edges.

To reduce the IDG, the algorithm successively removes edges from the graph, such that the maximum path between each pair of nodes is not changed. The following algorithm removes an edge $X \rightarrow Y$ and computes the new maximum path between X and Y using a modified Bellman-Ford algorithm. If there is no path or if the maximum path is smaller than the weight of the removed edge, the edge is re-inserted.

```
foreach edge in instrGraph:
 removeEdge(instrGraph,edge)
 maxPath := negatedBellmanFord(instrGraph, from[edge],to[edge])
 if maxPath=UNDEFINED or maxPath<distance[edge]
  insertEdge(instrGraph,edge)
```

To compute the maximum path between two nodes, the Bellman-Ford algorithm is applied on the graph with negated edge weights. The resulting shortest path corresponds to the longest path in the original graph. The Bellman-Ford algorithm can safely be applied, since the original graph does not contain positive cycles (Section 5) and the negated graph does therefore not contain negative cycles.

The complexity of our reduction algorithm is $\mathcal{O}(n^3)$. The graph has $\mathcal{O}(n)$ edges (Section 5), which results in a complexity of $\mathcal{O}(n^2)$ for the Bellman-Ford algorithm, which is executed $\mathcal{O}(n)$ times.

Alternatively, an algorithm based on Floyd-Warshall may be used with complexity $\mathcal{O}(n^3)$. This algorithm does not require the number of edges to be $O(n)$. Unfortunately, it only removes an edge, if there is a longer path (not one of equal length). The Dijkstra algorithm can not be used as a replacement for Bellman-Ford, since edge weights may be negative (Section 5).

7. Control Dependencies

The DDA described so far can be used to analyze data dependencies within a single basic block. The resulting IDG only contains data dependencies. We do now extend the DDA to insert also control dependencies into the IDG. The extended IDG can be applied to basic blocks and code regions with multiple in- and outputs. This approach is especially well suited for dependence analysis of code sequences resulting from linearization of basic blocks. This allows rescheduling of linearized code in a late compiler phase, even when control flow has been denormalized by optimizations.

To detect control dependencies, all target instructions of branches must be marked. There must also be a way, to recognize branch instructions. The association between branches and branch targets is not needed, as it does not affect control dependencies. Note, that the analysis does not construct a CFG, but inserts control dependencies into the IDG.

The DDA is applicable for achitectures with delayed branches. Control dependencies are correctly inserted between instructions

in delay slots and the corresponding branch. However, this method can only be applied under certain restrictions:

- There must not be branches in delay slots of branches.
- There are no pending accesses beyond the end of a basic block.

To analyze the control flow, we reduce the control flow analysis to a data dependence analysis. The control flow is mapped to read and write accesses on an artificial control dependence storage called CDEP in the following. All instructions read CDEP with a latency of 0. A write access on CDEP will therefore separate the instructions like a barrier. A branch or label causes a write access on CDEP. The write latency for a label is 0 and the write latency for a branch corresponds to the number of delay slots. As a result, the DDA inserts a RW dependence edges from each instruction to the next branch resp. label and a WR edge from the previous branch resp. label to the instruction. These control dependencies encloses the instructions between the preceding and succeeding branch resp. label, i.e. in their basic block.

Note, that this reduction considers delay slots of branches, such that instructions in delay slots logically precede the branch and the negative distance of the edge allows the scheduler to fill delay slots.

8. Memory Dependencies

In addition to highlevel alias information, the DDA may contribute dependency information about certain memory accesses. Therefore, each memory cell is treated like a register, i.e. there is an access list for each memory cell. A memory cell can be a single byte or a large memory region like an array. Memory regions, which are commonly accessed can be represented by a single cell. For a memory accesses with unknown addresses, all memory cells are conceptual accessed as a pessimistic estimate.

The runtime and memory requirement of the compiler is linear in the number of cells. It is therefore not practical to represent each byte of a large address space by a dedicated cell. To reduce the number of cells, the size of cells may be increased at the cost of precision. Alternatively, the analysis may be applied only for a small region of the memory.

In our compiler we analyse dependencies of accesses to the current stack frame. As a stack frame is rather small and the memory is word aligned, only few cells are required. The address of accesses to spill locations and compiler generated temporaries is statically known.

9. Scheduling

The IDG will typically be used as input for scheduling. Details of our scheduling heuristics are beyond the scope of this paper. However, we demonstrate, how to cope with cyclic IDGs, because such IDGs can not be scheduled with standard list scheduling techniques. List scheduling requires the IDG to be a DAG. We developed a hybrid scheduling approach, which schedules strongly connected components (SCC) of the graph using list scheduling [6]. For scheduling of the instructions within a SCC, we apply a backtracking approach.

```
sccGraph = createSccGraph(idg)
while sccNode := listSchedNext(sccGraph):
  btSched(sccNode)

function btSched(sccNode)
  if isEmpty(sccNode)
    return true
  instr = popInstr(sccNode)
  foreach pos in validPositions(instr):
    schedule(instr,pos)
    if btSched(sccNode)
      return true
    unschedule(instr)
  pushInstr(sccNode,instr)
  return false
```

The graph in Figure 3d consists of three SCCs (dashed ellipses). For a ASAP strategy, the SCCs are scheduled in the order `"add,sub"` followed by `"mac,and"` and `"mov"`. When the `"add,sub"` SCC is scheduled, the backtracking explores all valid schedules of `"add"` and `"sub"`. The result is an allocation of `add` and `sub` to the same execution cycle.

Backtracking algorithms usually have a high runtime complexity. However, the runtime of the SCC scheduling is acceptable, as SCCs are small. The set of valid schedules of a SCC with n instructions and a cycle weight of $-k$ is limited by:

$$\binom{n+k-1}{k}$$

Let $X_1, \ldots, X_n, X_{n+1} = X_1$ be an instruction cycle of weight $-k$. A valid schedule of these instructions corresponds to a set of distances:

$$S = \{D^{\text{code}}_{X_1,X_2}, D^{\text{code}}_{X_2,X_3}, \ldots, D^{\text{code}}_{X_n,X_1}\}$$

The number of valid schedules corresponds to the number of valid distance assignments S. For a distance assignment S the following constraints must hold according to Section 5.3:

$$\sum_{i=1}^{n} D^{\text{code}}_{X_i,X_{i+1}} = 0$$

$$D^{\text{code}}_{X_i,X_{i+1}} = D^{\text{min}}_{X_i,X_{i+1}} + v_i \qquad \text{with } v_i \in \mathbb{N}_0$$

This can be transformed into the following equation, which has the aforementioned number of solutions for $v_1, \ldots, v_n$:

$$\sum_{i=1}^{n} v_i = -\sum_{i=1}^{n} D^{\text{min}}_{X_i,X_{i+1}} = k \qquad \text{with } v_i \in \mathbb{N}_0$$

In practice 99% of the SCCs contain only a single node. The remaining SCCs contain not more than 4 instructions ($n \leq 4$). The weight of the cycle is typically 0 or 1 ($k \leq 1$). For $n = 4, k = 1$, the number of schedules is limited by 4. For exceptionally large SCCs with $n = 6, k = 4$, the solution space does not contain more than 630 schedules. In practice, the backtracking algorithm has almost no effect on the run time of the SCC scheduler.

10. Evaluation

This section first gives a short overview of processor architectures, that require a sophisticated data dependence analysis, as proposed in this paper. We then evaluate our analysis method using the CoreVA processor architecture in Section 10.2.

10.1 Architectures

As discussed in the previous sections, there are multiple processor features, that prevent the use of a simple data dependence analysis and may cause cyclic dependencies in the IDG:

- Delayed Branches (control hazards)
- Non-uniform and non-interlocked data latencies (data hazards)
- VLIW (concurrent accesses)

The following processor architectures have at least one of these features and therefore require a sophisticated DDA method.

The first member of the MIPS [3] processor family, the R2000 has one load delay slot and one branch delay slot. In our terminology, the load has a write latency of two cycles for the destination register and the branch has a write latency of two cycles on the

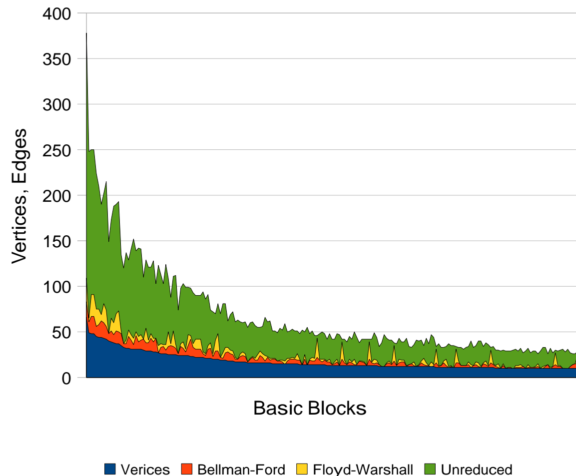

Figure 5. Number of edges for large basic blocks.

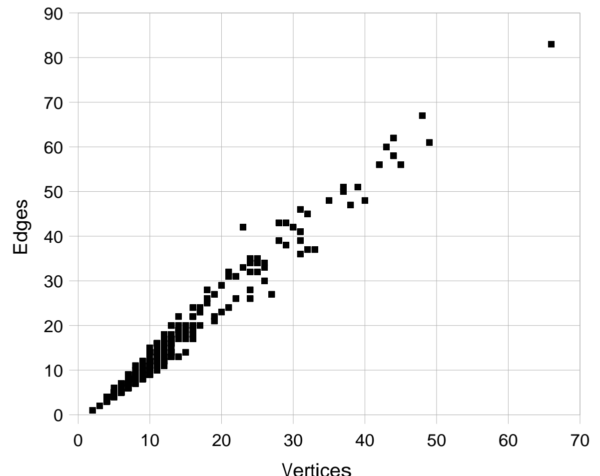

Figure 6. Correlation between number of vertices and reduced edges.

CDEP storage. Especially, the load instruction may lead to to IDG cycles as shown in Figure 1c.

DSPs belonging to the C6000 family from Texas Instruments[9] can execute up to 8 instructions in parallel using VLIW. The pipeline of the DSP is non-interlocked. Register write latencies range from 1 cycle for simple arithmetic instructions to 2 or 4 cycles for multiplications, 5 cycled for loads and 6 cycles for a branches. These instructions with higher latency and the VLIW execution may lead to a cyclic IDG.

The CoreVA architecture [2, 4] is a 4 issue VLIW processor with a 6 stage pipeline and two dedicated division step and multiply accumulate units. The pipeline is partially interlocked. The load, multiply-accumulate and branch instructions have a write latency of 2 cycles. The accumulator of the `mac` instruction (Figure 2) is read with a latency of one cycle.

10.2 Data Dependence Analysis

In the CoreVA compiler, cyclic instruction dependencies may show up after register allocation. By assigning multiple virtual registers to the same physical register, additional anti dependencies (RW dependencies) are introduced. These dependencies may lead to a cyclic IDG.

Frequency of Cycles Our evaluation has shown, that in average 0.1% of the instructions are cyclic dependent. Cyclic instruction dependencies are rare, but they frequently show up during compilation. The compiler must therefore be able to handle such code. The IDG of the EEMBC TCP benchmark for example contains 8 cycles (0.26% of the instructions) of size 2. An arithmetic test benchmark of our suite causes 3 cycles of size 2 and 1 cycle of size 3.

Instruction Dependence Graph The results of the edge reduction are shown in Figure 5 and 6. The charts shows the results for the EEMBC TCP benchmark. As results of other programs are similar, we only focus on this benchmark in the following. Chart 5 shows the number of edges (y-axis) for each basic block (x-axis). The basic blocks are sorted by the number of instructions in decreasing order from left to right. Since most basic blocks are very small, the chart shows only the quarter of the largest basic blocks. The blue area in the chart represents the number of vertices, i.e. the number of instructions in the basic block.

Edge Reduction The green area represents the number of edges before edge reduction, the yellow area after the Floyd-Warshall based reduction and the red area after the Bellman-Ford based reduction. It can be seen, that the number of edges is reduced by about two-thirds. Compared to the Floyd-Warshall based algorithm, the Bellman-Ford based algorithm produces slightly better results, because it also removes edges of same weight. In average, a basic block consists of 9 vertices and 26 edges. After reduction, the number of edges drops to 11 edges for the Floyd-Warshall resp. 9.5 for the Bellman-Ford based edge-reduction.

Edges per vertex Chart Figure 6 shows the correlation between the number of vertices and the number of edges after a Bellman-Ford based reduction. The correlation coefficient is 0.983, hence the relationship between the number of vertices and edges can be regarded linear. The number of edges per vertex is approximately 1.1. The number of edges is surprisingly low, because after register allocation, instructions are highly dependent. For example, in case of a total instruction order, the IDG would only contain one edge per vertex.

11. Conclusion

We presented a generic data dependence analysis method, which is aware of VLIW and non-interlocked read and write latencies. The method is applied on assembler code and yields a instruction dependence graph. Edges in the graph are annotated with minimal instruction distances, as required for subsequent scheduling. We discussed several properties of the graph and the schedulability of cyclically dependent instructions. Based on these results, we outlined a hybrid scheduling algorithm, which is as efficient as list scheduling, but can also cope with cyclic graphs. The evaluation using the CoreVA architecture has shown, that cyclic dependent instructions are rare, but frequently show up after register allocation. In addition, we presented an algorithm to reduce the number of edges in the IDG to about 1.1 times the number of instructions.

References

[1] Nerina Bermudo, Andreas Krall, and Nigel Horspool. Control flow graph reconstruction for assembly language programs with delayed instructions. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 107–118, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Ralf Dreesen, Thorsten Jungeblut, Michael Thies, Mario Porrmann, Uwe Kastens, and Ulrich Rückert. A synchronization method for

register traces of pipelined processors. In *Analysis, Architectures and Modelling of Embedded Systems*, pages 207–217. Springer Boston, 2009.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[4] Thorsten Jungeblut, Christoph Puttmann, Ralf Dreesen, Mario Porrmann, Michael Thies, Ulrich Rückert, and Uwe Kastens. Resource efficiency of hardware extensions of a 4-issue VLIW processor for elliptic curve cryptography. *Advances in Radio Science*, 2010.

[5] Uwe Kastens, Dinh Khoi Le, Adrian Slowik, and Michael Thies. Feedback driven instruction-set extension. In *Proceedings of ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, Washington, D.C., USA, June 2004.

[6] Sanjay M. Krishamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Not.*, 25(7):97–106, 1990.

[7] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[8] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, pages 93–102, New York, NY, USA, 1991. ACM.

[9] Texas Instruments. *TMS320C64x/C64x+ DSP CPU and Instruction Set*, 10 2008. Literature Number: SPRU732H.

[10] David Zaretsky, Gaurav Mittal, Robert P. Dick, and Prith Banerjee. Generation of control and data flow graphs from scheduled and pipelined assembly code. In *LCPC*, pages 76–90, 2005.