Proceedings of ITC-CSCC 2000, Pusan, Korea

NHS: A Novel Hybrid Scheduling for ILP

Song Pei You, Masahiro Sowa
University of Electro-Communications
Chofugaoka 1-5-1,Chofu-shi,
Tokyo, 182 8585

Tel: +81-424-43-5635, Fax: +81-424-43-5851
E-mail: {sou, sowa}@sowa.is.uec.ac.jp

Abstract: This paper presents a new scheduling method for ILP processing called NHS(Novel Hybrid Scheduling). It concerns not only exploiting as much ILP as possible like other state-of-the-art scheduling scheme, but also choosing the most important instructions among many ready-to-execute instructions to processors in order to reduce the execution time under limited hardware resource. At the heart of NHS is a conception called CCP(Complex Critical Path), an extension of CP(Critical Path). By using CCP, compiler not only can get a global information of the whole program to extract ILP, but also can collecting data dependence information and control flow information. The paper also presents the simulation results, to date, of our attempts to study the NHS scheduling method. The results indicate good potential for this scheduling method.

1. Introduction

Higher levels of performance benefit from improvements in semiconductor technology, which increase both circuit speed and circuit density. Further speedups must come, primarily, from some form of parallelism. ILP results from a set of processor and compiler techniques that speed up execution by causing individual RISC-style operations to execute in parallel. ILP-based systems take a conventional high-level language program written for sequential processors and use compiler technology and hardware to automatically exploit program parallelism.

Most of state-of-the-art ILP architectures have concentrated their efforts on exploiting as much ILP as possible. Especially EPIC processor - IA-64, it parallel executes program by data dependency, does not predict correct path of branch, executes both path of branch before conditional instruction. So there is a limitation in this system: that is the assumption of unlimited processors (functional units), they try to find out all the parallelism by compiler, and have enough processors to execute these instructions in parallel. But in fact, predication let CPU always execute instructions it is going to throw away. Furthermore, a program's parallelism is not same. If the number of parallelism is smaller than the number of processors, some processors will be idle. On the contrast, the remained instructions have to be executed at another new cycle, that creates idle processors also. Of course if compiler knows the number of processors, it can schedule the program more efficiently, but this creates capability problem.

This paper introduces a different scheduling method of ILP processing, called NHS.

2. Theory

The instruction section has highlighted the problems of state-of-the-art scheduling schemes. We introduce NHS, with a view to solve these problems.

2.1 Philosophy and basics of NHS

NHS has two parts, static scheduling part and dynamic scheduling part. In static scheduling stage, NHS is similar in concept to EPIC in that both predicative execute branch instructions, and allow the compiler to explicitly group (reorder) instructions for parallel execution. In addition, NHS assigns priority to instructions, collecting global information, and so on. While in dynamic scheduling stage, NHS selects most important instruction to issue, and decide next ready-to-execute instruction according to issued instructions.

NHS method does not allow a processor to remain idle as long as there are ready-to-execute instructions. Executable instructions are assigned to available processors to exploit as much of the parallelism of a program as possible.

A fundamental question that needs to be answered at this point is: **on what basis should instructions be assigned priority?** In numeric problem, CP(Critical Path) is used in assigning priority in list scheduling. In nonnumeric problem, the most important scheduling problem caused by the introduction of branch and loop, is due to the alteration of control flow and their impact on CP calculation. Any heuristic that uses level numbers or CPL(Critical Path Length) faces this problem. CPL of a node is the length includes all successor nodes along the longest path from this node to exit node. Unfortunately, in nonnumeric problem, the really successor nodes do not remain constant. The really successor nodes varies depending on the execution result of condition instruction. As for loop, the path would be tremendously increased with the execution of iteration. Since the CP is only available in numeric program, we propose to find out an extension CP which is the most probable CP for the nonnumeric program. Several extensions of CP have been studied for different purpose. An important difference in CCP over other extensions of CP lies in the fact that not only data dependence but control dependences have also been considered while identifying the CCP. Because the importance of an instruction is decided by instructions which depends on it in both data flow, and control flow.

Before present the algorithm of computing CCP, two DAG used in analyzing data dependence and control dependence, DFG and CFG are introduced.

DFG is also known as Data-precedence graph or data dependence DAG(DDD).

A DFG G=(V, E), where V is the set of v nodes, representing the instructions of program, E is the set of e directed edges, representing data dependence between the nodes.

The *CFG = (V, Ec)* of a program is defined in the same way as a DFG except that edges express control dependence rather than data dependence.

A control edge (u, v) $\in$ Ec implies that there exists some execution instance of the program that includes the task u and v.

The algorithm of computing CCP is shown following:

Note that, the CCP proposed in the thesis is not an optimal method, only an heuristic approach.

CCP (DFG , CFG)

input: DFG ; CFG
output an array CCPL[1. .n] assigned with the CCP length of each instruction.

```
Begin
 forall t∈Leaf_Node_Instruction_Set do CCPL[t]← 1;
 repeat
     repeat
         CCPL[t] ← the longest CCP length of all
Immediate_Successor_Instruction_of_DFG[t] + 1;
     until all node which entire
Immediate_Successor_Instruction_of_DFG are marked
     repeat
         CCPL[t] ←the longest CCP length of all
Immediate_Successor_Instruction_of_CFG[t] + 1;
     until all node which entire
Immediate_Successor_Instruction_of_CFG are marked
 until all nodes are marked
End
```

We have synthesized data- and control-dependence to a certain extent when we calculate CCPL in previous algorithm. However there are still some problems we have to consider about.

To simplify explain, we divide program in four parts. Part **A** includes the instructions before Branch_Conditonal_Instruction; part **B** includes the instructions in the branch body and are on the true path; part **C** includes the instructions in the branch body and are on the false path; part **D** includes remained instructions.

First, the overhead of speculative execution, as present in introduction. Therefore what we have to do next is let the priority of In_Branch_Instructions higher than Out_Branch_Instructions when they have the same CCPL. In another word, part **A** should have higher priority than part **B** and **C**.

Second consider the part **B** and **C**. Due to compiler won't know which part should be executed, even though maybe one of them have better CCPL, part **B** and **C** should have the same chance to be assigned to PE when they are ready-to-execute. Thus, after sorting **B** and **C** in decreasing order of CCPL respectively, we order **B**, and **C** alternately.

Finally, **D**, due to the same reason as **A**, should be ordered at last.

PRIORITY (CFG , CCPL)

input: CFG ; CCPL
output an array Priority[1. .n] assigned with the priority of each instruction.

```
Begin
  t ← the first Conditional_Instruction
  repeat
    let A be the set of part A of t.
          SORT (A)
    let B be the set of part B of t.
          PRIORITY (B)
    let C be the set of part C of t.
          PRIORITY (C)
    let D be the set of part D of t.
          PRIORITY (D)
    MERGE(T,A,B,C,D)
  Until all Conditional_Instruction has been searched
  SORT(T)
  PRIORI[..] ← {T}
End
```

MERGE(E,A,B,C,D)

input: four sets of instructions with their CCPL:A,B,C,D
output: a set of instructions with their CCPL:E

```
Begin
  s ← {B} ◎{C}
  E ← {A}∪{s}∪{D}
End
```

SORT(A): order A in decreasing magnitude of CCPL. (Because it is a quite sample procedure, no details are given here)

Now, we have presented the method of computing priority for each instruction. As list scheduling, the next job of NHS is repeatedly execute the following four steps until the program is finished.

1. Select from the ready-to-execute instruction list the instruction with the highest priority for scheduling.
2. Select a processor to accommodate this instruction.
3. Execute this instruction.
4. Decide next ready-to-execute instruction according to issued instructions.

2.2 Static Scheduling of NHS

In this section the details of compile-time technique, in another word static part of NHS is given.

To reduce hardware overhead, NHS does by statically scheduling as much of the work as possible, followed by dynamic scheduling "corrections" as the program executes.

The role of static scheduling in NHS is :

1. exploiting ILP
2. calculating priority for instructions

Fig.2 A Block Diagram of a Multi-Processor that Incorporates NHS

3. reordering the program
4. collecting data dependency information
5. collecting control dependency information
6. code generation

1, 2, and 3 has been present in previous section.

Because scheduling hardware(searching engine) need to identify ready-to-execute instructions from the instruction cache at runtime, the time required to search must be limited, otherwise, it would make the system impractical. It is very important, therefore, to reorder the instruction to shorten the distance between Sussessor_Instruction and Precedence_Instruction.

All the instructions in the same BLOCK of DFG is data-dependent, can be executed concurrently. And always the execution of previous block results in making the subsequent block be ready-to-execute. So, in NHS, instructions are reordered by block.

As presented previous, one difficulty in NHS is that compiler need to direct hardware execution. That is we need some extra bits in instruction's format for the textural description of the dynamic nature of program.

Due to in NHS, the ready-to-execute instructions are selected to execute in parallel at runtime according to their priorities, as well as the outcome of Branch_Conditonal_Instruction. So, for letting dynamic scheduler know, which instructions will become ready-to-execute instruction according to the current assigning of instructions to PEs(Processing Elements), NHS appends a list of Succssor_Instruction to each instruction. Each time an instruction is fetched by scheduling hardware ,fetched instructions have to provide information (i.e. Data_Dependent_Information) to dynamic scheduler about next ready-to-execute instructions.

On another hand, as predicate execution of branch instruction, in NHS, the results of Branch_Conditonal_Instruction have to be broadcast to dynamic scheduling hardware units, and when PEs speculative execute a In_Branch_Instruction, the result will be written in corresponding parts of Temp_Memory. In another word, PEs need to know which control flow operations belong to.

Collecting both data-dependence information (DDI) and control-flow information (CFI)are a straightforward jobs, they can be done at the same time as generate DFG and CFG. The algorithms of collecting CFI and DDI are omitted in this paper.

The final part of static scheduling of NHS is to synthesize all the information generated from previous part, generate new instruction. Fig. 1 shows the format of generated new instruction by compiler, as following:

Operation	DDI	CFI	Priority	Valid

Fig. 1: Instruction Format

Operation is a unit of computation, such as an addition, memory load, or branch, which would be referred to as an instruction in the context of a sequential architecture. A NHS instruction is an operations with many other information generated by compiler to guide dynamic scheduling.

2.3 Dynamic Scheduling of NHS

In this section, we present how dynamic part of NHS accomplished at runtime, by hardware.

Fig. 2 gives a block diagram of a multi-processor that incorporates NHS. By this design we can assure that fetch engine can match the processor's execution, and can prove the practicality of the scheduling algorithm we proposed.

To start this section, let's image that the generated codes of program have been stored in the memory, and been ready to be dynamically assigned to processors to be executed.

Due to limited space, the explain of formats of memories and caches are omitted in the paper.

First, the function of hardware units are presented:

We divide dynamic scheduling mechanism of NHS

into seven major components: a Fetch Unit (**FU**), a Ready-to-execute Instruction Buffer (**RIB**), a Scheduling Unit (**SU**), a Scheduling Buffer (**SB**), a Reference Buffer (**RB**), a Temporary Memory (**TM**), and a conventional instruction cache (**IC**).

The functions of ***FU*** are:

1. Check the valid bits of RIB, as long as there is instruction in RIB, which valid bit is '0', FU fetches an instruction with valid='1' from IC to RIB.
2. According to the DDI of fetched instructions of RIB by SU; fetch instruction which valid bit is '1' from IC to RIB.

While, the functions of ***SU*** are:

1. Fetching the operations of instructions which have the highest priorities from IC to SB
2. Resetting the valid bit of instruction in IC be '1', according to the execution of BI.

PEs in NHS, are needed not only to execute the instruction assigned by SU, but also to broadcast the result of the execution of BI, deal with TM.

RB , stores the execution result of BI, broadcast the result of BI by setting the valid bit to '0' of corresponding instruction in Memory, IC and RIB; and modifying TM.

In the following, we enumerates the steps involved in program execution on a NHS.

Program Execution Flow:

1. Initial block from memory to IC, according to RB;
2. FU fetch ready-to-execute instructions from IC to RIB;
3. Scheduling Unit (SU) searches RIB, fetches instructions which have a higher priority, put them into Scheduler Buffer (SB), and meantime, reset the instruction's valid bit which have fetched to SB.
4. PEs execute the instructions from SB, and check CFI. If the executing instruction is BI, PE issues the result to RB. If the executing instruction is IBI, and the BI which depends on hasn't been executed yet, PE reads operand from TM, and writes result to TM. Whenever PEs finish current work (we have assumed that the execution time of every instruction is same), there are ready instructions assigned by SU with the highest priority which selected from the IC by SU.
5. RB broadcasts the result forwarded by PEs to RIB, IC and Memory, set the corresponding instruction's valid bit be '0', and removes the result of mis-executed path of IBI of TM, and, copies the results of correct path from to M.

3. Simulation and Performance Measurement

The performance of NHS was compared with another scheduling method, named BG. BG schedules in the same way as NHS except that BG doesn't use CCP. In another word, SH forwards ready-to-execute instructions to PEs randomly from IC. This scheduling methods also can be considered as the best case of IA-64's scheduling methods when IA-64's compiler doesn't know the number of processor.

Due to time restriction, two random graphs, DFG and CFG were generated in stead of real program.

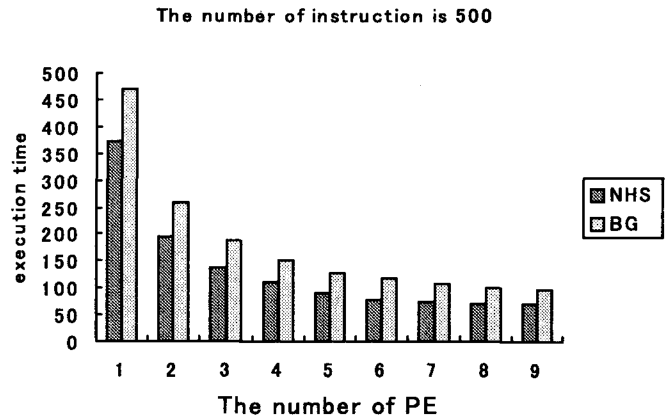
To generate these two graphs, the number of nodes and edges are decided first. In DFG, we randomly assign successor nodes and predecessor nodes for each node. In CFG, we randomly decide which node is BI node, and the length of TP and FP is also decided randomly. The outcome of BI is decided randomly as well.

We tried to study the effect of the number of the nodes, and edges on the speedup. For that purpose, we generated graphs with 500 nodes and 1000 nodes. The results from the simulation show that the speedup is not affected by the change of the size of the input graphs.

Tab. 1 shows the execution cycles scheduled by NHS and BG on a program which has 500 instructions with PEs number range from 1 to 9.

PEn	1	2	3	4	5	6	7	8	9
NHS	374	196	139	111	91	76	74	72	71
BG	470	259	188	151	126	116	109	101	97

Tab. 1 Simulation Result of a Program with 500 Instructions



The percentage of speedup ranges from 23.5% to 31.2%

4. Conclusion

This paper proposed a scheduling technique NHS for nonnumeric program onto multiple PEs.

To support this NHS algorithm, high-level hardware organization is designed. By this design we can assure that fetch engine can match the processor's execution.

Measurements showed that this scheduling algorithm could compete in performance with other state-of-the-art scheduling scheme, and it is also economical in terms of the number of processors used.

References

[1] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conference Proceeding, 30, pp.483- 485, 1967.

[2] J. A. Fisher; B. R. Rau, "Instruction-Level Parallel Processing", Science, Volume 253, Number 5025, September 13, 1991, pages 1233-1241.

[3] D. W. Anderson, F. J. Sparacio, T. M. Tomasulo, IBM J, Res. Develop. 11. 8 1967.