REDUCING THE NUMBER OF INSTRUCTIONS

M. Gusev, A. Misev, G.Popovski
Faculty of Natural Sciences, St. Cyril and Methodius University
Arhimedova bb, PO Box 162, 1000 Skopje, Macedonia
{marjan, infolab, pope}@pmf.ukim.edu.mk
P. Mitrevski
Faculty of Technical Engineering, St. Clement Ohridski University
"Ivo Lola Ribar" bb, PO Box 99, 7000 Bitola, Macedonia
pece.mitrevski@uklo.edu.mk

Abstract: *The purpose of this article is to reduce the number of instructions while executing in processor. We analyse memory address dependent instructions and eliminate the address generation processing if the address was previously calculated. For standard RISC, VLIW and in-order superscalar processor we introduce a solution where the RMI (Reduction of Memory Instructions) Algorithm is performed in the compile stage and address dependent instructions do not enter the processor at all. For out-of-order superscalar processors we introduce two solutions, the first one when these instructions are not issued at all and the second solution when these instructions are issued only in a reservation station without execution unit. All these solutions improve the behaviour of the processor for at least 10% since the processor does not executes these instructions.*

Keywords: memory, superscalar, processor, register renaming, shelving, out of order, simulation

1. Introduction

A lot of papers concern about the distribution of instructions in typical RISC or ILP processor [1,2,3,5,6,8]. A lot of authors analysed the memory address dependencies [2,3,6,7,9] and a complete survey of memory address instruction distribution can be found in [4].
Following the conclusion that on average 39% of loads do not need to access memory without speculation and that distribution of loads is 25% of all dynamically instructions we end up that 10% of all dynamically executed instructions can be skipped without any processing and access to memory. Similar scheme is capable of killing stores that access the same memory address for another 20% of store instructions.
In this paper we deal with memory access instructions and their elimination in the instruction sequence. It is shown in [4] that memory address dependent instructions we deal are between 10% and 20%. So the process of execution of an ordinary program results with smaller execution times for more then 10%. The extra hardware and software to be included to achieve this speedup is a realisation of simple tables, indexed search and simple comparison logic that does not delay the execution of ordinary instructions.

2. Motivation

A typical RISC like sequence consists of variable fetch instruction, instruction that computes the result and variable store instruction. These are the L/S architecture design approaches. Consider for example the high level instructions to increment or decrement the counter or instructions that are used as sum or product of array elements. One can see that very common are cases when a memory variable is loaded, then its value is changed and it is stored back in memory. For further analysis lets

consider the high level language instruction inc(counter) in Pascal like language or ++counter in C like language.
If the counter address is assembled as sum of offset and base register R1 then typical sequence representing the previous high level language instruction is
load R6, offset(R1)
add R6, R6, 4
store offset(R1), R6
Lets see what really happens in the arithmetic logic unit. Whenever we analyse a typical RISC, VLIW, in-order superscalar or out-of-order superscalar processor the execution units compute the memory address for first instruction. Then the actual load is performed in load buffer. The integer ALU then executes the add instruction. The third instruction is executed similar to the first one, the memory address is generated and then the store buffer realises the memory access.
The question is why to compute the same memory address twice. Two steps before processing of the third instruction the ALU has been used for the same thing! There is no need for store instruction in this case. The address is already computed and the result to be stored comes out after the processing of the second instruction. However, think about small complexity of the RISC like processing, pipeline etc.
The solution is very obvious. We introduce a cache memory to keep the generated addresses. Whenever a memory access instruction is to be executed we check if the address has been stored in this small cache memory. Then avoid executing this instruction, just forward the memory address and data to the load or store buffers to finish the real memory access. These instructions do not enter the pipeline and do not occupy the resources. We gain in overall speedup of processing.

3. Reduction of Memory Instructions (RMI) Algorithm

Looking the design of an ordinary RISC, or even at the design of VLIW and out-of-order superscalar processor one can conclude that memory access instruction is executed like an ordinary instruction. This means that in RISC and VLIW processor it is pipelined through the IF, OF/ID, EX stages (Instruction Fetch, Operand Fetch/Instruction Decode and the Execution stages). These stages are used for address generation and then the generated address is forwarded to the write buffer where the real memory access is realised.
In out-of-order superscalar processor it is pipelined through the F, D/I/RR, OF/D, E, C, WB stages (Fetch, Decode/Issue/Register Renaming, Operand Fetch/Dispatch, Execution, Completition /Retirement, Write Back). Once again this instruction is processed for address generation and then it activates the write or load buffer to finish the real memory access.
As one can see in all the processors models there are 3 to 6 stages for address generation processing. Then the entire job is transferred to buffers to finish the access process of memory.
What we discovered is that the process of address generation is activity that is already performed in previous memory access instruction. So if a memory access instruction comes in instruction sequence and if the address was already calculated then it is not necessary to do the same process again. Our research showed that all these L/S (load/store) processor architectures where the address was previously calculated and repeatedly used are frequent. A typical L/S architecture executes an instruction sequence that loads a memory variable, calculates a value and then stores this value to the same memory position. Actually the compiler produces such a code while transforming a high-level language code in typical L/S sequence. This means that almost all high language instructions are compiled in similar sequences where the same memory location is accessed more than once.

The idea of Reducing the Memory Instructions (RMI) algorithm is based on elimination of memory address dependent instructions in the address generation part if the addresses where previously calculated.

To realise the idea not to include the processing address calculation for memory access instructions means not to add too much additional hardware and processing logic. Actually we just use the fact not to include just the part of the address generation, since it was done in some previous instruction. The calculated address is kept in a special table. As the instructions enter the processor, the store instruction that has already calculated address in previous instruction does not enter the processor. It does not occupy the instruction fetch and decode stages, it does not require any operand fetch and it does not activate the ALU for address generation. Actually the previous instruction that generates a result to be stored in memory is associated with a special identification mark. This means that after the execution stage of calculating the result it forwards the result to the write buffer along with the already calculated address from the previous load instruction. Similar happens within the load instruction if the address is already calculated. Then it is forwarded to the load buffer where the actual memory access is realised.

4. Memory Address Dependencies

In this article we are concerned about the Memory Address Dependencies (MAD). We do not compare address dependencies with data dependencies identified as real data dependence, input and output data dependencies and antidependencies corresponding to RAW (Read After Write), RAR (Read after Read) and WAW (Write After Write) WAR (Write After Read). Here we analyse address dependencies i.e. dependencies in instructions that calculate addresses sent to memory as input data. These addresses are usually calculated in special Address Generation Units (AGU).

Four MAD (Memory Access Dependencies) can be identified by the sequence of load and store instruction: LAL (Load After Load), LAS (Load After Store), SAS (Store After Store), SAL (Store After Load). LAL dependence should not occur in a typical code produced by the compiler. Actually the same memory variable is loaded twice. Similar to this case LAS identifies the case when the variable is saved in the memory and afterwards it is once again loaded in a register. The first two address dependencies LAL and LAS occur in cases when there are not enough registers in the register file. So the memory variables repeatedly have to be loaded in the registry file. This is a common case while working with vector data and arrays. SAS dependence is identified whenever a variable is calculated and stored. Then once again the variable is calculated and stored to the same position. This technique can also be used for keeping the memory space for variables small enough. The last case SAL is very frequent in L/S architecture. A typical high-level language instruction consists of loading a memory variable, computing a new value and then store to the same position. A lot of examples can show this behaviour like counters, sum or product variables etc.

5. Additional hardware

The main idea in this article is to eliminate the part of memory access instruction that concerns the address generation since this stage is already done. The table where the calculated addresses are kept is a special buffer called Generated Address Cache (GAC). This buffer is positioned after the execution stage in the processor. The idea of implementing a GAC is similar to the idea to implement BTAC (Branch Target Address Cache). In this cache we store the addresses of the next instructions if branch is taken. So the processor is released of the address generation process by calculating a new target address since this process was already done in previous instructions.

GAC is accessed in two ways. Whenever a load or a store instruction calculates an address, this address is written in the GAC as a special entry after the execution stage. Whenever it is required, the GAC is searched by the index and the generated address is forwarded to the write buffer.

The realisation of this cache is associated with realisation of a simple logic that discovers if the address of the store instruction is already performed within the previous memory access instruction. Then this control logic adds a special mark to the previous arithmetic instruction that calculates a result that has to be stored with this store instruction. Then the store instruction is eliminated from the instruction sequence. This control logic is realised by simple table with entry associated to each logic register, a simple comparator that compares the entry in this table with the actual store instruction, and a logic that puts a special mark to the arithmetic instruction that produces the result. This control logic is very simple and can be integrated very easy. In the case of RISC and VLIW processor models our research showed that the position of this control logic is in the prefetch/ predecode stage, but a compiler can very efficiently solve it so the elimination of the store instruction is realised while producing the assembler code. In the case of out-of-order processing in a superscalar processor there is no need for compiler intervention. Everything is realised with this simple logic in the issue stage and the GAC is stored after the execution stage before the completition stage. While the arithmetic instruction enters the reservation station it is associated with a special mark that means the result should be also written in the write buffer along with the calculated address from the GAC. The situation with the load instruction is less complex since the hardware does not need to look for an arithmetic instruction to put a special mark. The control logic only detects whether the address has been previously generated. If there is a match it proceeds the address and register identification to the load buffer. This instruction is not issued and not executed at all.

6. RMI realization in out-of-order superscalar processor

Unlike standard RISC, VLIW and in-order superscalar processors, the implementation of this idea in out-of-order superscalar processors is simpler and is all realised in hardware. Superscalar processors have already implemented different techniques that help the realisation of RMI, such as Register Renaming and Shelving.

We introduce two different approaches to implement this idea.

A. without issuing a MAD instruction.

B. with issuing a MAD instruction for special processing.

The first approach eliminates the execution of a memory address dependent instruction whenever the address has already been calculated. The second approach does not eliminate the memory address dependent instruction, but it does not use an execution unit, it only uses in a Special Store Reservation Station (SS RS). Both approaches speedup the process of memory accesses since we do not calculate the same address if it has already been calculated.

7. Case A RMI realisation in out-of-order superscalar processors

A typical out-of-order processor consists of Instruction Decode/Register Renaming stage, Instruction Cache and Register File, separate Reservation Stations for L/S (Load/Store) Unit, FX (Integer) Unit, FP (Floating Point) Unit and B (Branch) Unit. After the execution Completition Unit processes the instructions and the results are sent via Common Data Bus to the register file and reservation stations. Memory accesses are realised via corresponding load and store buffers. The new hardware for Case A RMI realisation is presented in bold lines. It consists of DLT (Dependence Lookup Table), GAC (Generated Address Cache), SFC (Store Forward Control) and LFC (Load Forward Control).

Case A RMI realisation is based on 4 additional hardware tables. The first table called Dependence Lookup Table (DLT) contains information about the data dependence relations between the instructions used to detect memory address dependencies MAD. Register and Offset values are identified whenever a new entry is assigned for a store or load instruction. The Process Counter

field is used as a counter that denotes when this entry should not be replaced by another entry. Practically it means that the corresponding value in GAC should be used.
The Valid Bit is attached to GAC table. Its access is not indexed by register address but by Entry ID assigned by DLT.
The Queues for Load Forward Control (LFC) and Store Forward Control (SFC) are realised with Entry ID and Value for the registers.
The idea behind the algorithm for Case A RMI implementation in out-of-order superscalar processor is to maintain the tables in the instruction decode stage. We assume that the register renaming is also performed and we deal with the renamed physical registers. The DLT is matched if there is any entry that has equal register and offset. If such entry exists in the DLT, then the address is already calculated and stored in GAC.
Greater number of entries in the GAC and DLT improve the behaviour of this idea. Entries in both tables are not removed immediately after instruction completition. Instead, they are kept for consecutive use. The policy for removing an entry can be LRU, but considering the Process Counter.

8. Case B RMI realisation in superscalar processor with out-of order execution

Instead of eliminating the store instruction, the second approach forwards it in a special RS called Special Store Reservation Station (SS RS). The execution path of the store instruction is shortened. This approach relies on the concept of shelving i.e. the instructions are buffered before executing their operation. The original hardware realisation for out-of-order execution is same to the explanation for Case A realisation and the new hardware introduced is DLT (Dependence Lookup Table) and SS RS (Special Store Reservation Station).
The entry number of memory access instruction determined by DLT is stored in the 'Entry ID' field and the store instruction is identified with its ROB ID in the issue stage. The value to be stored in memory by the store instruction is stored in the Value field and the Result Valid bit marks its availability. If the Result Valid bit is set, the value is available. Otherwise, the register number is stored in the Value field of the entry in order to update the real value via the Common Data Bus. The Address field of the entry is updated by the corresponding load instruction when it generates the memory address. The Valid Bit field marks the availability of the address. When a load instruction opens the entry, it resets the Valid Bit field to its default value 0 and when the address is calculated, it sets the Valid Bit field to 1.
The layout of DLT is the similar to Case A realisation, but in this case we do not use counter Process Counter field, we use a single bit field Process Bit.
The algorithm for the additional processing of load and store instructions in the decode stage is similar to the previous one. The detection logic for SAL dependencies is analogous to Case A realization except the part of assigning the DLT and SS RS entries. New corresponding entries are assigned to each load instruction both in DLT and SS RS. When a store instruction is detected in the decode stage it is checked within DLT if there is an entry with the same memory reference opened by previous load instruction. This means that it uses the same base address register and the same offset as the previous load instruction. Then it sets the Process Bit and it is issued to the SS RS along with its ROB ID in Re-Order Buffer. Otherwise, the processing of the store instruction proceeds normally i.e. it is issued to the standard L/S RS.
Assigned entries to each load instruction in the SS RS and DLT remain occupied until a store instruction retrieves the generated address, when it resets the Process Bit field. Also these entries are reserved until an instruction that modifies the address base register forwards the store instruction. According to the fact that DLT is limited with fixed number of entries then LRU algorithm should be used to make space for new entries.

9. Conclusion

In this article we introduce RMI algorithm to reduce the number of instructions while executing in processor. The program structure is not changed at all. Only the memory access instructions that are address dependent of some previous instructions are eliminated since the address required in the current instruction was already calculated by some previous instruction. We introduce solutions that cover all types of today's processors. The offered solution for RISC, VLIW and in-order superscalar processor are realised partly by software and partly by hardware. The compiler eliminates address dependent instructions and attaches a special mark and address field to the existing arithmetic instructions. These instructions carry the information that the result should be written to memory.

There are two solutions for out-of-order superscalar processors. The Case A solutions do not issue address dependent instruction, instead it forwards the memory access to the corresponding Load or Store Forward Control that checks when the address and the data are ready to be sent or received from corresponding buffer. The Case B solution issues address dependent instruction in a special reservation station. Then no address generation is executed it uses as a standard reservation station that accepts data and forwards control when all the conditions are fulfilled.

In all offered solution GAC is a Generated Address Cache where memory addresses are kept. These addresses will be used for subsequent memory accesses. It can be noted that case B uses simpler logic and generates less bus traffic. The simpler implementation of GAC and the use of standard shelving technique can be taken as advantages for case B.

Case A offers reusability of calculated addresses, which in turn is its main advantage over case B. The store instructions can be completely eliminated in the decode stage, and, although the store instruction execution path is shortened in case B, case A does not require ROB entry for the store instruction.

10. References

1. Austin,T., Sohi,G.S., (1992), *Dynamic Dependency Analysis of Ordinary Programs,* Proc.19th Int.Conf. on Computer Architecture, ISCA-19.
2. Calder,B., Krintz,C., John,S., Austin,T. (1998), *Cache-Concious Data Placement,* Proc 8th Int.Conf. ASPLOS-8.
3. Gonzales,J., Gonzales,A., (1997), *Speculative Execution Via Address Prediction and Data Prefetching,*Proc.11th Int.Conf.Supercomputing, pp.196-203.
4. Gusev,M, Misev.A and Popovski.G (1999), *Memory Address Dependencies,* ITI-99 pp.191-196.
5. Hennessy,J.L., Patterson,D.A., (1996), *Computer Architecture: A Quantitative Approach*, sec. edition, Morgan Kaufmann Publishers, San Francisco, California.
6. Mochovos,A., G.S.Sohi (1997), *Streamlining Inter-operation Memory Communication via Data Dependence Prediction,* Proc.30th Ann.Int.Symp. on Microarchitecture, MICRO-30.
7. Reinman,G., Calder,B. (1998), *Predictive Techniques for Aggressive Load Speculation,* Proc.31st Ann.Int.Symp. on Microarchitecture, MICRO-31.
8. Sima D. at al. (1997), *Advanced Computer Architectures: A Design Space Approach*, Addison Wesley Longman, Harlow, England.
9. Tyson,G., Austin,T. (1997), *Improving the Accuracy and Performance of Memory Communication Through Renaming*, Proc.30th Ann.Int.Symp. on Microarchitecture, MICRO-30.