FUNCTIONAL UNIT USAGE BASED THREAD SELECTION IN A SIMULTANEOUS MULTITHREADED PROCESSOR

Deepak Babu M Iyappan Babu , Lakshmi Narayanan Bairavasundaram,
Madhu Saravana Sibi Govindan and Ranjani Parthasarathi
blnarayanan@hotmail.com
School of Computer Science and Engineering, College of Engineering,
Anna University, Chennai, India.

Abstract. This paper proposes and evaluates a new mechanism for thread selection in simultaneous multithreaded processors that is based on functional unit(FU) usage information. The performance of any processor depends on the set of dependences that it can manage. In a multithreaded architecture there is an opportunity to manage structural dependences more effectively than in conventional superscalar processors. Minimizing the competition for FUs will greatly improve the performance of multithreaded processors. This can be achieved by means of proper thread selection techniques. In this paper, a framework that aids thread selection based on FU usage information, is discussed. The results show that this thread selection mechanism can improve performance significantly.

1 Introduction

The performance of any processor depends on the set of dependences that it can manage effectively. Over the years, various mechanisms have been developed for dependence management. With the advent of multithreaded architectures, dependence management has become easier due to availability of more parallelism. But, the demand for hardware resources has increased. In order for the processor to cater efficiently to multiple threads, it would be useful to consider resource conflicts between instructions from different threads. This need is greater for simultaneous multithreaded processors, since they issue instructions from multiple threads in the same cycle. Similar to the operating system's interest in maintaining a good job mix, the processor is now interested in maintaining a good mix of instructions. One way to achieve this is for the processor to exploit the choice available during instruction fetch. To aid this, a good thread selection mechanism should be in place. This paper aims at proposing one such mechanism of thread selection using functional unit (FU) usage. The following sections deal with this mechanism in detail. Section2 describes background and related work. Section3 presents the motivation. Section4 proposes schemes and discusses implementation issues. Section5 presents experimental methodology and results. Section6 gives a conclusion.

2 Background and Related Work

Dependences - data and control - limit the exploitation of instruction level parallelism (ILP) in processors. This is especially so in superscalar processors, where multiple instructions are issued in a single cycle. Hence, a considerable amount of research has been carried out in the area of dependence management to improve processor performance[1].

Data dependences are of two types : true and false. False data dependences: anti and output dependences are removed using register renaming, a process of allocating different hardware registers to an architectural register. True data dependences are managed with the help of queues where instructions wait for their operands to become available. The same structure is used to wait for FUs. Control dependences are managed with the help of branch prediction.

Multithreaded processors add another dimension to dependence management by bringing in instruction fetch from multiple threads. The advantage in this approach is that the latencies of true dependences can be covered more effectively. Thus thread-level parallelism is used to make up for lack of instruction-level parallelism.

Simultaneous multithreading(SMT) [2] combines the best features of multithreading and superscalar architectures. Like a superscalar, SMT can exploit instruction-level parallelism in one thread by issuing multiple instructions each cycle. Like a multithreaded processor, it can hide long latency operations by executing instructions from different threads. The difference is that it can do both at the same time, that is, in the same cycle.

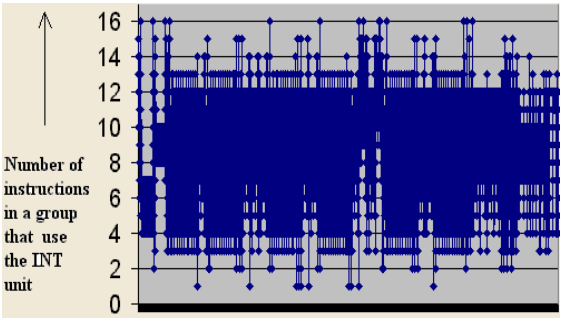
The main issue in SMT is effective thread scheduling and selection. While scheduling of threads from the job mix may be handled by the operating system, selection of threads to be fetched is handled at the microarchitecture level.One technique for jobscheduling called Symbiotic Jobscheduling [3] collects information about different schedules and selects a suitable schedule for different threads.

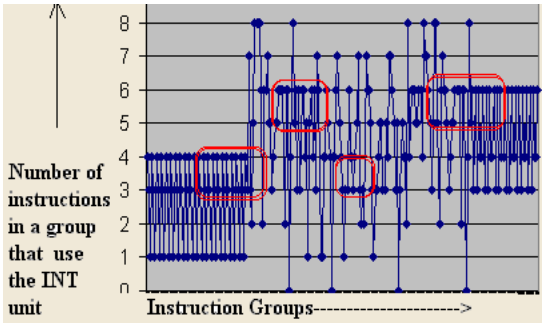
A number of techniques[2] have been used for thread selection. The ***Icount*** feedback technique gives the highest priority to the threads that have the least number of instructions in the decode, renaming, and queue pipeline stages. Another technique minimizes branch mispredictions by giving priority to threads with the fewest outstanding branches. Yet another technique minimizes load delays by giving priority to threads with the fewest outstanding on-chip cache misses. Of these the Icount technique has been found to give better results [2].

This paper aims at improving thread selection by having minimal structural dependence as one of the criteria for selection.

3 Motivation

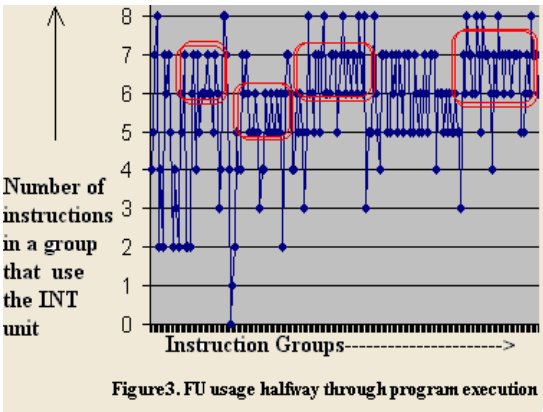
The motivation for this idea comes from an analysis of FU usage and queue fills. The usage of FUs during the execution of a number of programs has been analyzed using the SimpleScalar toolset[5]. Figure 1 shows the usage of a particular FU(Integer ALU) during the execution of a typical program - the **hydro2d** program, that is a part of the SPEC95 benchmark suite. The number of instructions in every group of 16 that uses the FU has been

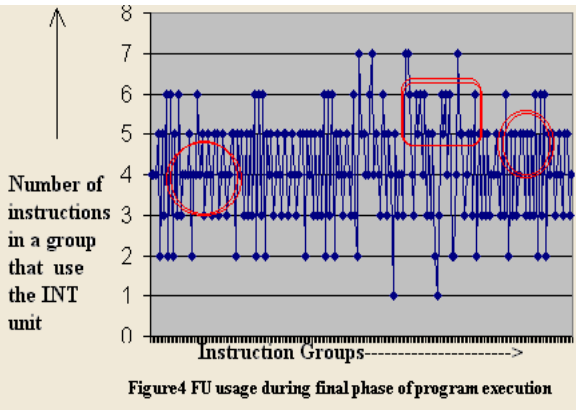




plotted (for a total of 166739 instructions). The dark shaded region indicates that the FU usage varies considerably throughout the execution of the program. Though the thread can be classified as integer-intensive, the exact FU requirements at any point cannot be determined. Hence, it may not be feasible for the operating system to aid the architecture in thread selection. Therefore, a closer examination of the behavior is required to look for alternatives.

Figure 2. takes a closer look at FU usage for every 8 instructions during the initial phase of program execution (for a total of 1600 instructions). Figure 3. shows the details of FU usage for every 8 instructions halfway through program execution (for a total of 1600 instructions). Figure 4. shows the details of FU usage for every 8 instructions during the final phase of program execution (for a total of 1600 instructions).





The circles in the figures highlight bunches of instructions where the FU usage remains constant. This indicates that the usage of FUs can be specified on a per bunch basis. Interestingly, the compiler is in a position to identify these bunches. If the processor were provided with this information beforehand, it would be able to estimate future FU usage for different selections of threads. It can then decide the most advantageous selection of threads based on available FU resources. That is, the combination with the least number of conflicts for Fus can be selected. Thus, a suitable compiler-architecture interaction mechanism could be designed, through which valuable information for thread selection could be conveyed.

4 Proposed Schemes and implementation issues

Based on the reasoning given above, two schemes are proposed. Both these schemes use the framework discussed below. There are two issues to be considered in such a framework : how the compiler relays the information and how the processor uses it.

The hardware viewpoint is considered first. The processor has to maintain expected FU usage information for each thread. During a fetch cycle, it can consult this information for the purpose of thread selection. It would be reasonable to consider other factors also and have a multi-level selection of threads. For example, the Icount method may be used to do a preliminary selection and then the FU usage information may be employed to do the final selection. The sum of FU usages of every possible pair of selected threads and current resource usage is computed. The pair with the least sum is selected for fetching.

The usage information has to be conveyed by the compiler suitably. Special instructions can be used for this purpose. If the future FU usage is different from the current usage, the compiler inserts instructions to indicate the new usage. The degree of change that merits the insertion of an instruction can be decided with heuristics based on FU latencies.

Two schemes are considered within the above framework. In the first scheme, the compiler indicates the usage information for the set of **N** instructions that may be fetched in the next cycle. If the FU usage is different from the current usage, a functional unit usage (***FUusage***) instruction of the form < ***CHSTUSAGE*** usageFU$_1$,usageFU$_2$,.... > is inserted, where usageFU$_i$ denotes future usage for FU **i**. The processor then loads the information into a table which has an entry < ***usageFU$_1$,usageFU$_2$,...***> for each thread. It is this table that is consulted during every fetch.

An important implementation issue here is with respect to mispredicted branches. When a branch is mispredicted and fetch proceeds along the wrong path, a ***CHSTUSAGE*** instruction in this path may modify the table erroneously. When the misprediction is detected, the instruction is squashed. In order for table correctness to be maintained, the table has to be restored to its original state. So a copy of the table is always maintained. This copy gets updated whenever a ***CHSTUSAGE*** instruction commits. On detecting a misprediction, the current table is restored to the correct state using the copy.

It may be noted that in the above scheme, the ***CHSTUSAGE*** instruction has to be executed repeatedly, if the FU requirements change inside a loop. To overcome this, a second scheme is proposed in which an instruction of the form < ***STUSAGE*** **PC**, ***usageFU$_1$,usageFU$_2$...***> is inserted. Here **PC** denotes the pc from which FU **i** has the specified usage. To support this scheme, the processor has to maintain an associative table for each thread with entries of the form < PC, FU usages >. When this instruction is executed, a new entry is added to the table.

Replacement of entries in the associative table is an issue to be considered because of table size constraints. An LRU replacement policy can be followed. Another issue is with regard to the instruction size. In a fixed instruction size processor, it may not be feasible to specify many FU usages along with the PC in the same instruction. So, an instruction of the form < ***STUSAGE*** **PC**, ***usageFU$_X$***> is more realistic, where **X** is a single FU. This would imply that a greater number of instructions may be required to specify usage changes.

However, it can be noted that this scheme has an extra advantage in that the compiler can schedule these instructions anywhere in the program.

5 Experimental Methodology and Results

The results of some preliminary investigation of the schemes is presented below. There are two parameters to be considered: overheads caused by the additional instructions and the performance improvement obtainable.

To determine the overheads , the number of ***FUusage*** instructions that are inserted and the number of times these ***FUusage*** instructions are executed has been studied. SPEC95 benchmarks running on SimpleScalar have been used for this purpose. The functional units considered are Integer ALU, Integer multiply, Integer divide, FP ALU, FP multiply, FP square root, Load unit and Store unit. In a ***FUusage*** instruction, the usage is specified for a set of **N** instructions. The numbers have been obtained for two values of **N** : 8 and 16 . Tables 1 and 2 show the number of instructions inserted and the number executed for Scheme1 with N = 8 and N = 16 respectively. Tables 3 and 4 show the number of instructions inserted for Scheme2 with N = 8 and N = 16 respectively.

Benchmark	Number of instructions	Number of CHSTUSAGE instructions inserted	Number of instructions executed	Number of CHSTUSAGE instructions executed
mgrid	21768	1428 (6.5%)	945152	68368 (7.2%)
hydro2d	26590	1704 (6.4%)	165976	13831 (8.3%)
cc1	270846	16110 (5.9%)	6138664	397671 (6.4%)

Table1 Number of instructions inserted , executed for Scheme1 , N = 8.

Benchmark	Number of instructions	Number of CHSTUSAGE instructions inserted	Number of instructions executed	Number of CHSTUSAGE instructions executed
mgrid	21768	1004 (4.6%)	1250720	60798 (4.8%)
hydro2d	26590	1200 (4.5%)	166739	8496 (5.0%)
cc1	270846	11552 (4.2%)	6138664	262954 (4.2%)

Table2 Number of instructions inserted , executed for Scheme1 , N = 16.

Benchmark	Number of instructions	Number of STUSAGE instructions inserted
mgrid	21768	2324 (10.6%)
hydro2d	26590	2844 (10.6%)
cc1	270846	25418 (9.3%)

Table3 Number of instructions inserted for Scheme2 , N = 8

Benchmark	Number of instructions	Number of STUSAGE instructions inserted
mgrid	21768	1969 (9.0%)
hydro2d	26590	2528 (9.5%)
cc1	270846	21256 (7.8%)

Table4 Number of instructions inserted for Scheme2 , N = 16

The heuristics used for inserting instructions are as follows**:** An instruction is inserted if the change in FU usage is greater than a tolerance value for each FU. The tolerance values

used are 2,1,1,1,0,0,0,1,1 for the Integer ALU, Integer multiply, Integer divide, FP ALU, FP multiply, FP divide, FP square root, Load unit and Store unit respectively. It is evident from the tables that Scheme2 inserts almost double the number of instructions as Scheme1. It may also be noted that for N = 16, the number of instructions inserted is approximately 70% of that of N = 8.

The performance of Scheme1 with instruction group size, N = 8 has been studied using the SMTSIM simulator [4]. Six threads from simple programs have been executed simultaneously and the following performance parameters have been measured : average usage of instruction queue, number of instruction queue conflicts, number of INT unit

Simulator Configuration	average size of inst Q	No of inst Q conflicts	No of INT unit conflicts	No of LD/ST unit conflicts	% of useful instns executed
Original SMTSIM	**7.42**	**234**	**1**	**77**	**83**
SMTSIM with FU usage info based thread selection	**7.22**	**231**	**0**	**75**	**82**

Table5 Comparison of performance parameters between unmodified SMTSIM and SMTSIM with FUusage based thread selection

conflicts, number of load/store unit conflicts and percentage of useful instructions executed. The values obtained are compared with those obtained by the unmodified SMTSIM.

From the values obtained, it is seen that when FU usage based thread selection is performed, a marginal improvement is obtained. There is a very small dip in the number of useful instructions executed. The unmodified SMTSIM uses a 6-issue Integer unit(each capable of performing all operations) and 4 issue FP unit(each capable of performing all operations). In the modified simulator, the same configuration has been used. In a processor which is not so rich in resources, each unit will typically perform a specific operation. In this scenario, it is expected that the proposed thread selection mechanism would perform much better than the unmodified SMTSIM. This is to be explored further.

6 Conclusion

In this paper, the motivation for a new thread selection mechanism based on FU usage has been presented. A framework for implementing the mechanism has been discussed and two schemes based on the framework have been proposed. An analysis of the performance of these schemes shows considerable promise. Work is being done to evaluate the first scheme under hardware resource restrictions. The second scheme is also being explored. The framework can be extended to include other resources such as registers and cache for further improvement.

7 References

[1] John L Hennessy and David A Patterson, Computer Architecture : A Quantitative Approach, Second Edition ,Morgan Kaufmann publishers 1995.

[2] Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm and Dean M. Tullsen , Simultaneous Multithreading: A Platform for Next-generation Processors, IEEE Micro September/October 1997.

[3] Allan Snavely, Dean M. Tullsen, Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, ASPLOS-IX November 2000.

[4] SMTSIM simulator - http://www-cse.ucsd.edu/users/tullsen/smtsim.html

[5] SimpleScalar toolset – http://www.simplescalar.org