An Enhanced Trace Scheduler for SPARC Processors

Spiros Kalogeropulos

Sun Microsystems, 16 Network Circle MS: UMPK16-203,
Menlo Park CA 94025, USA.
spiros.kalogeropulos@sun.com

Abstract. In this paper an enhanced trace scheduler implementation is described which targets processors with moderate support for parallelism and medium size register file such as the *SPARC* processors *UltraSPARC(rtm) II* and *UltraSPARC(rtm) III*. The enhanced trace scheduler is a global instruction scheduler, which identifies and exploits the available instruction level parallelism in a routine, contributing to performance improvement for UltraSPARC processor systems. The enhanced trace scheduler is part of the Sun Forte 6 update 2 product compilers for *UltraSPARC II* and *UltraSPARC III* processors. The enhanced trace scheduler, in exploiting the available instruction level parallelism, attempts to address issues such as register pressure, speculation and the amount of compensation code.

1 Introduction

The *SPARC* processors *UltraSPARC II* and *UltraSPARC III* are in-order issue superscalar processors which provide moderate support for parallelism by issuing and executing up to four instructions per clock cycle. The SPARC processors rely on the compiler to produce an instruction schedule that extracts and exploits the available instruction level parallelism in a routine to maximize the instruction issue rate and the parallelism of memory operations by issuing prefetches and loads as early as possible.

The process of detecting and scheduling the available fine grain parallelism is usually applied on the control flow graph of a routine. Instruction schedulers use single basic blocks for detecting and scheduling the available fine grain parallelism. However, single basic blocks contain insufficient instruction level parallelism; therefore higher performance is achieved by exploiting instruction level parallelism in consecutive basic blocks.

Several global instruction scheduling techniques have been described in the literature, which perform instruction scheduling beyond the basic blocks boundaries. Trace scheduling [1], [2] selects a frequently executed sequence of contiguous basic blocks from one path in the control flow, called trace, on which instruction scheduling is performed. Most of the global scheduling techniques [1], [2], [3] target wide issue processors with a big register file and hardware support

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 597–602, 2003.

for large amount of parallelism. The enhanced trace scheduling technique described in this paper targets the *UltraSPARC II* and *UltraSPARC III* processors which have a medium size architectural and physical register file of 32 integer registers, 32 floating point registers and support for a moderate amount of parallelism. Therefore, the performance improvement due to the exploitation of the available parallelism involves successfully addressing issues such as register pressure, speculation and the amount of compensation code. For instance, aggressive code motion might cause register spilling if the register pressure is high, which negates the benefit of utilizing the parallelism. Furthermore, aggressive speculation could hurt the performance of a program if the latency of the speculated instructions is not hidden by using idle resources of the processor.

In the following section, we present an overview of the enhanced trace scheduler. Subsequently, we describe briefly a flexible trace formation scheme. Next, the scheduling of traces and dealing with register pressure is discussed. Finally, results are presented which evaluate the enhanced trace scheduler work.

2 An Overview of the Enhanced Trace Scheduler

The enhanced trace scheduler phase could be invoked twice. The first invocation is before the register allocation during a phase named as early instruction scheduling. The second invocation could be after register allocation during the late instruction scheduling phase. The early instruction scheduling is applied on a compiler intermediate representation where values are kept in an unlimited number of virtual registers. On the other hand, the late instruction scheduling is applied on the same compiler intermediate representation after the register allocation where values reside in actual machine register.

The enhanced trace scheduler performs the following actions repetitively using the control flow of a routine until it cannot form a trace:

- It forms a trace and annotates the trace with the following information:
  - Data flow information for control flow edges entering, exiting the trace for virtual registers which are accessed but will not be renamed.
  - Information regarding the original basic block of each instruction in the trace and its properties such as execution probabilities.
- It invokes the trace instruction scheduler which addresses issues such as register pressure, speculative code motion.
- It introduces partial renaming, compensation code. Compensation code is introduced in a similar way to [1], [2].

3 Trace Formation

The enhanced trace scheduler targets a wide variety of applications with diverse enhancing performance requirements. For instance, in database applications the main requirement for performance improvement is to improve the concurrency in memory operations, since the portion of run time spent on memory operations is

high. The enhanced trace scheduler tackles the above problem by being flexible in the trace formation and by producing traces which match the performance characteristics and potential of different applications.

The trace formation phase may form three different kinds of traces, called *hot* trace, *warm* trace and *long* trace, by applying different edge execution probabilities thresholds during the trace formation.

After determining the basic blocks that will be part of the trace using a technique similar to [1], [2], the next step is to collapse all the basic blocks into a trace block. During the collapsing of the basic blocks a pseudo instruction called *join* instruction is inserted at the beginning of each basic block. The *join* instruction represents the notion of the basic block in the trace and the data flow information for control flow edges entering and exiting the trace. Furthermore, a mapping of the instructions in each basic block to its *join* instruction is maintained before and after scheduling the instructions in the trace. This information will enable us to identify the instructions that have moved from their original block after scheduling the instructions in the trace.

4 Scheduling Traces

Our approach to schedule instructions in a trace is different from other trace scheduling techniques [1], [2]. In the enhanced trace scheduler first the available parallelism is exposed. We improve the available parallelism by ignoring the data dependencies due to virtual register accesses and memory accesses in the off-trace control flow paths which enter or exit a trace and employing virtual register partial renaming on the hoisted instructions. Subsequently, its usefulness is evaluated by using a cost benefit analysis scheme which calculates the impact of speculation, compensation code and register pressure. Finally, the instructions are scheduled in the trace.

The instruction scheduler exposes the available parallelism by viewing the trace as one big basic block when it builds the necessary information for scheduling such as the DAG (directed acyclic graph). However, the instruction scheduler during scheduling views the basic blocks in the trace distinctively. Therefore the instruction scheduler is able to analyze the code motion and distinguish between a speculative code motion and a non speculative code motion. The above analysis is facilitated by maintaining information about the current basic block under scheduling and using the information of the *join* pseudo instructions. At the beginning of the scheduling, the current basic block is the first block in the trace, when all its instructions are scheduled, then the next block in the trace becomes the current basic block.

Before the scheduling commences, the instruction scheduler builds a DAG for the whole trace, where the nodes are instructions in the trace and the edges represent the data dependencies between the instructions. After building the DAG the instruction scheduler maintains height information for each instruction in the trace. The height of an instruction in a DAG is the estimated execution time of the longest path from the instruction to the exit of the DAG. Height information

is one of the factors for prioritizing the scheduling of ready for execution instructions. Furthermore, for each instruction in the trace, the information about its original block is maintained. The above information helps the instruction scheduler to find the estimated execution probability of the instruction. The execution probability is used by the cost benefit analysis scheme to determine the usefulness of the available parallelism.

The instruction scheduler is list driven and uses the DAG to build a ready list of instructions in the trace which are available for execution. Subsequently, a cost benefit analysis scheme is employed to evaluate the instructions which may belong to different basic blocks using the following heuristics:

- Earliest instruction issue time.
- A speculative instruction which blocks the further execution of instructions in the pipeline until its execution is completed, for instance a divide instruction, gets low priority.
- An instruction which will cause an amount of compensation code above a certain threshold when moved from its original block gets low priority.
- An instruction which belongs to a basic block, other than the current basic block, with execution probability relative to the head of the trace less than a certain threshold gets low priority.
- If an instruction belongs to a basic block, other than the current basic block, with low execution frequency then it may move only to a control equivalent block.

After determining the useful parallelism, instructions are prioritized further using the following main criteria: 1) Impact on register pressure. 2) Impact on machine resources. 3) Critical path height reduction.

After considering all the above criteria the most suitable instruction is scheduled and is removed from the list of the available for execution instructions.

4.1 Dealing with Register Pressure

Our approach to deal with register pressure is to exploit the available parallelism while the hardware register availability is high and focus on the register pressure impact of the instructions in the trace when the hardware register availability is close to zero.

During the scheduling of each instruction in the trace, we maintain information about the number of live virtual registers and an estimate of the available hardware registers. When our estimation shows that the availability of hardware registers drops close to zero, then the instruction scheduler for the trace gives highest priority to instructions which, if they were scheduled would free a used virtual register or not define a new virtual register. In this way we try to reduce the number of spills.

5 Results

In this section, we present results of experiments conducted to measure the performance improvement when the enhanced trace scheduler is invoked on the

benchmarks SPEC2000 INT. The compilation of the benchmarks used profile feedback where the training data set is different from the actual testing data set used by the benchmarks. The evaluation of the enhanced trace scheduler was achieved by using the peak configuration file, used in Sun's SPEC submissions, to establish the peak SPEC2000 INT numbers. Then new peak SPEC2000 INT numbers were produced by enabling the enhanced trace scheduler.

By using the peak options without enabling the enhanced trace scheduler, the scheduling of the instructions is done by a basic block scheduler invoked before and after the register allocation. However, before the first time the basic block scheduler is invoked, a global code motion phase that redistributes the available parallelism similar to [4] is applied. The main objective of the above phase is to reduce the critical path of a basic block and its machine resource usage by trying to move some of its instructions to other basic blocks without increasing their height and utilizing the available hardware resources.

Figure 1 shows the performance improvement when the SPEC2000 INT benchmarks are compiled with an additional sequence of options enabling the enhanced trace scheduler on top of the peak flags. The enhanced trace scheduler is invoked after the global code motion phase. The binaries run on a 450 MHz *UltraSPARC II* processor with 4 Mbyte *L2* cache, and a 1050 MHz *UltraSPARC III* processor with 8 Mbytes *L2* cache.

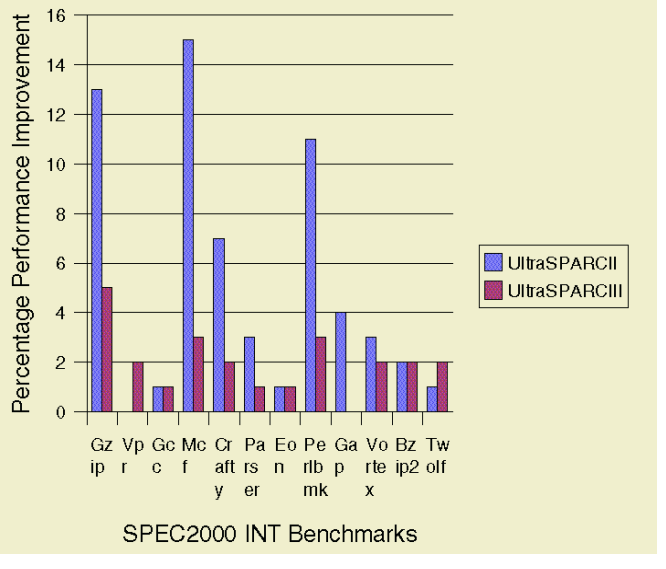

Fig. 1. Performance Improvement on *UltraSPARC II* and *UltraSPARC III* processors

The difference in performance improvement between *UltraSPARC II* and *UltraSPARC III* is due to mainly the different processor characteristics and the opportunities provided by the processors to hide the latency of integer load

instructions missing the *L1* data cache. The *UltraSPARC II* processor has a 16 Kbyte direct mapped *L1* data cache in comparison to *UltraSPARC III*'s 64 Kbytes 4 way associative *L1* data cache. Therefore there are a lot more *L1* cache misses when the SPEC2000 INT benchmarks run on *UltraSPARC II* in comparison to *UltraSPARC III*. The loads which miss the *L1* cache in *UltraSPARC II* stall the pipeline only when an instruction using the value provided by the load executes. The enhanced trace scheduler has the potential to hide the 9 cycles latency of the loads which miss *L1* cache in *UltraSPARC II*, by moving the loads far away from the instructions that use the value provided by the loads and achieve performance improvement. However, in the case of *UltraSPARC III* a load instruction missing the *L1* cache stalls the pipeline till the load instruction completes execution, therefore the code motion cannot hide the latency of the above load. Furthermore, in both processors when a load instruction hits the *L1* data cache the latency for using the load's value is 2-3 cycles. The enhanced trace scheduler can hide the above load latency by scheduling the load instruction and its data dependent instructions at an optimal cycle distance.

6 Conclusions

Most of the global scheduling techniques target processors with hardware features supporting large amount of parallelism where register pressure, speculation and compensation code is not a big issue. However, exploiting successfully the available parallelism for processors with small or medium size register files and moderate support for parallelism, such as *UltraSPARC* processors, involves successfully addressing the issues of speculation, register pressure and compensation code.

Our approach in the enhanced trace scheduler involves a flexible trace formation scheme according to the performance needs of different applications. A partial register renaming scheme is employed to improve the instruction level parallelism. Finally, the usefulness of the available parallelism is determined by using a cost benefit analysis scheme which takes into consideration the impact of speculation, compensation code and register pressure.

References

1. Lowney G., Freudenberger S., Karzes T., and et al. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing, Volume 7. p. 51–142*, 1993.
2. Ellis J. Bulldog: A compiler for VLIW Architectures. *Ph.D. thesis, MIT*, 1986.
3. Hwu W. and et al. The Superblock: An Effective Structure for VLIW and Superscalar Compilation. *The Journal of Supercomputing, Volume 7. p. 229–248*, 1993.
4. Gupta R. and Soffa M. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Transactions on Software Engineering, Volume 16. p. 421–431*, 1990.
5. Moon S. and Ebcioglu K. Parallezing Non-Numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems, Volume 16. p. 1–40*, 1997.