Multithreaded Architectural Support for Speculative Trace Scheduling in VLIW Processors

Manvi Agarwal and S.K. Nandy
CADL, SERC,
Indian Institute of Science,
Bangalore, INDIA
{manvi@rishi.,nandy@}serc.iisc.ernet.in

J.v. Eijndhoven and S. Balakrishanan
Philips Research Laboratories,
Eindhoven, The Netherlands
{jos.van.eijndhoven,srinivasan.balakrishnan}
@philips.com

Abstract

VLIW processors are statically scheduled processors and their performance depends on the quality of schedules generated by the compiler's scheduler. We propose a multithreaded architectural support for speculative trace scheduling in VLIW processors. In this multithreaded architecture the next most probable trace is speculatively executed, overlapping the stall cycles of the processor during cache misses and page faults. Switching between traces is achieved with the help of special hardware units viz. Operation State Buffers and Trace Buffers. We observe an 8.39% reduction in the overall misprediction penalty as compared to that incurred when the stall cycles due to cache misses alone are not overlapped.
.

1. Introduction

In a Very Long Instruction Word (VLIW) processor the micro-architecture of the processor is exposed to the compiler and it generates schedules to exploit maximum *Instruction Level Parallelism (ILP)* present in the code. Two main methods of scheduling in VLIW processors are: *basic block scheduling* and *extended basic block scheduling*. Basic block scheduling is limited in its scope of exploiting ILP because of small size of basic blocks. (4-5 interdependent operations on an average in each basic block.) In extended basic block scheduling, groups of basic blocks are scheduled as a single unit. Extended basic block scheduling can be categorized into following: trace scheduling [8], superblock scheduling [15], hyperblock scheduling [13] and decision tree scheduling [10]. All these scheduling schemes suffer from the drawback of issue slot wastage as explained in [9]. In [9] we proposed a new scheduling scheme- *speculative trcae scheduling* for VLIW processors which ensures minimal issue slot wastage. The traces are executed speculatively based on their probability of execution. The penalty paid on a misprediction is the length of the trace. Misprediction penalty depends on the frequency of mispredictions and our speculative trace scheduling scheme minimizes this frequency of mispredictions by scheduling traces based on their probability of execution. Our scheduling scheme gives a performance of 1.41x as compared to decision tree scheduling of TriMedia (TriMedia SDE version 2.0 was used for simulation purposes). The results reported in [9] are reproduced here (Table 1) for sake of completeness. These results assume necessary hardware to roll back to the previous checkpoint state of the processor on a misprediction. Misprediction penalty is the length of incorrect trace and additional cycles required for roll-back operation. Mispredicitons penalty incurred on a misprediction can be reduced if the stall cycles of the processor during cache miss or page faults are overlapped with the execution of the next most probable trace. We propose a multithreaded architectural support for this purpose in this paper. The multithreaded architectural support reduces the overall penalty suffered on a misprediction by overlapping the stall cycles of the processor during page faults and/or cache misses with the execution of the next probable trace.

Table 1:Perfomance improvement of Speculative Traces (predicted traces) relative to Decision Trees (delayed branches) in TriMedia

Benchmarks	Predicted Traces vs Decision Trees
008.espresso	1.5387
022.li	1.3631
023.eqntott	1.4009
072.sc	1.3677
Average	1.4170

The rest of the paper is organized as follows: section 2 gives details of related work in multithreaded architectures. In section 3 we explain our multithreaded architectural support for speculative trace scheduling with simulation results given in section 4. We summarize contribution of the work in section 5 and draw conclusions.

2. Multithreaded Architecture

Traditionally, at the time of exceptions such as page faults and cache misses, processor stalls until the data is available which is clearly an overhead since cycles are wasted. This overhead can be reduced considerably if the stall cycles of the processor are overlapped with the execution of the next probable trace. Once cache misses/page faults are handled, processor resumes execution of the original trace. On a trace misprediction, processor starts the execution of the next probable trace which is already ahead in its execution and takes fewer cycles to complete. This reduces the effective misprediction penalty suffered because it is already ahead in execution to some extent during the stall cycles. We propose a multithreaded architectural support to achieve this end in VLIW processors.

2.1. Related Work

Multithreading is the technology that allows several threads to execute simultaneously in a processor. Main objective of multithreading is to maximize processor utilization in the absence of enough ILP to fully exploit and utilize processor resources. Multithreading also help in hiding processor stall cycles on a cache miss, page fault and branch misprediction. Even if one thread suffers for example, a cache miss, other threads continue to execute while the cache miss is handled. In normal processor architectures, processor resources (functional units) are wasted in stalling and waiting for the data, while the cache miss is handled. Some of the notable research works in the area of multithreading are: [1], [2], [11] and [14]. Two main types of multithreading as differentiated by Dean Tullsen in [2] are *Fine Grain Multithreading* and *Simultaneous multithreading*. Fine grain multithreading allows only one thread to issue operations in one cycle and attempts to maximize the utilization of all functional units available in the processor architecture. In Simultaneous multithreading more than one thread can issue operations to the functional units simultaneously in a single cycle.

Threaded Multiple Path Execution (TME) [14], *Simultaneous Subordinate Multithreading (SSMT)* [11] and *Speculative Data-Driven Multithreading (DDMT)* [1] have been proposed to enhance the performance of a single thread by initiating micro threads from the parent thread.

In TME [14], idle context threads of a multithreaded processor are used to spawn and execute multiple alternate paths of the primary thread. An architecture platform is proposed in the paper [14], which has the ability to execute multiple paths of the same thread as well as different threads simultaneously. Whenever a hardware context is available an alternate path from the primary thread is spawned at a suitable checkpoint (branch) which executes simultaneously with the primary thread. When the branch is resolved the thread with mispredicted path is squashed and the hardware context is made available for alternate path of the primary thread. If primary thread turns out to be mispredicted then primary thread is squashed and the correct alternate path is made the primary thread. From this new primary thread alternate path is spawned whenever a hardware context is available to execute it.

In SSMT [11], several microthreads are micro-coded and kept stored in a special microRAM in the architecture. These microthreads are spawned at certain predefined events in the processor. Each of this microthread is associated with a microcontext and it does not affect the cache behavior of the primary thread because this microthread is stored in a microRAM. These microthreads assist the primary thread to improve the branch prediction accuracy, cache hit rate and prefetch effectiveness.

In DDMT [1] data driven threads (DDT) are spawned speculatively to hide stall cycles of the processor during cache misses and branch mispredictions. DDTs are formed with the help of algorithm proposed in the paper and these DDTs execute speculatively with the primary thread. In DDMT long latency load operations and frequently mispredicted branches are termed as critical operations. Critical operations are identified in the primary thread and then these critical operations serve as the trigger points for the DDT. DDTs are executed speculatively, consequently, results do not change the architecture state of the processor. The architecture state of the processor is changed when results of these DDTs are integrated into the parent thread.

3. Proposed Architecture Support

All the schemes mentioned above employ multithreading for superscalar processors and aim at enhancing the performance of the main control flow by executing micro threads from the same primary thread or alternate paths. In TME, alternate paths of the primary threads are executed simultaneously with the primary thread. The scheme, which we propose in this paper, is

different from TME in the sense that the next probable trace is executed only during cache miss or a page fault.

3.1. Mathematical Model

We propose a multithreaded architectural support for speculative trace execution on VLIW architectures. To keep the hardware architecture simple complying with the VLIW philosophy (hardware simplicity) we execute the next probable trace of the application during a cache miss or a page fault when the processor has to stay idle for multiple cycles till the data is available. Several paths are not executed simultaneously. Execution of the next probable trace helps hide the stall cycles of the processor by doing useful work from the next probable trace. SSMT and DDMT are different from our scheme because these schemes enhance performance of the primary thread by spawning microthreads from within the primary thread.

Our speculative traces scheduling scheme [9] splits the decision tree into its corresponding traces. Each trace is annotated with its probability of execution obtained from profile information gathered through prior runs of the application. Consider a tree split into its "*n*" corresponding traces. Trace 1 is the trace with the highest probability of execution. Let "L_i" denote the length of each trace "*i*", "E_i" the execution count, "p_i" the probability of execution, "s_i" the total number of stalls suffered and "*R*" the next PC misprediction rate of dynamic branch predictor. The misprediction penalty (MP) suffered when the stall cycles of the processor are not overlapped with the execution of the next probable trace is given by:

$$MP_{WithoutStallUse} = \sum_{\forall trees} \sum_{i=1}^{n} E_i * L_i * p_i * R \qquad (1)$$

Misprediction penalty when the stall cycles of the processor during cache misses and page faults are overlapped with the execution of the next probable trace is given by:

$$MP_{WithStallUse} = \sum_{\forall trees} \sum_{i=1}^{n} E_i * (L_i - s_{i-1}) * p_i * R \qquad (2)$$

The reduction in the misprediction penalty by overlapping the stall cycles of the processor with useful execution of the next probable trace is given by:

$$PenaltyReduction = MP_{WithStallUse} - MP_{WithoutStallUse} \qquad (3)$$

Reduction in misprediction penalty when the stall cycles of a trace are overlapped with the execution in the next probable trace is illustrated in figure 1. Figure 1(a) shows execution of two traces when stall cycles of the processor during cache miss and page faults are not overlapped with execution of next trace. The misprediction penalty suffered is the whole length of trace T1. Figure 1(b) shows the execution of the same trace T1 (of figure 1(a)) when stall cycles of the processor are overlapped with the execution of the next probable trace T2. The penalty incurred on trace misprediction is reduced in this case.

3.2. Details of Hardware Support

The hardware support proposed by us is set to context of a typical VLIW processor as shown in figure **2**. Architectural support that facilitates multithreaded execution of traces is highlighted as shaded regions in figure **2**. When the execution of the application starts, multiple traces are stored in the trace buffers. Each trace is associated with a hardware thread. The trace buffers are located in the stage following the decode unit (figure **2**) so the traces stored in them can be issued to the functional units immediately on a stall. The trace with the highest probability of execution is issued for execution first. During execution, it may incur a cache miss or a page fault leading to processor stalls. In our scheme, whenever processor incurs stalls due to cache misses and/or page faults, it starts execution of the next probable trace while the cache miss and/or page faults (exceptions) are handled in the background. Once the processor returns from these exceptions, execution is switched back to the original trace. To achieve this functionality we make use of two hardware units, Operation state buffer (OSB) and Controller in our architecture as shown in shaded region of figure **2**. The block diagram of the OSB is shown in figure 3. It is used to send "exception and switch to

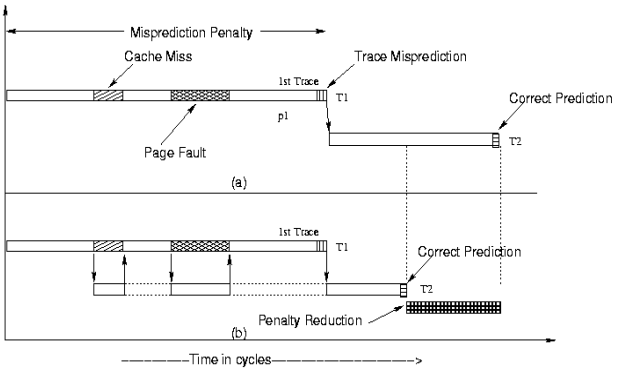

Figure 1: Reduction in misprediction penalty

another trace" signal to the Controller. Tom Conte et al has proposed a structure named Control State Buffer for fast interrupt handling in VLIW processors in [4]. The OSB used in our architecture though similar to the one proposed in [4], it has been modified and adapted for speculative scheduling. It has been enhanced with Trace Label entry and a Trace Pointer to identify the trace of the operation. OSB is a circular buffer with Head and Tail pointers. A special pointer is present which points to the excepted trace. This pointer is named the Trace Pointer.

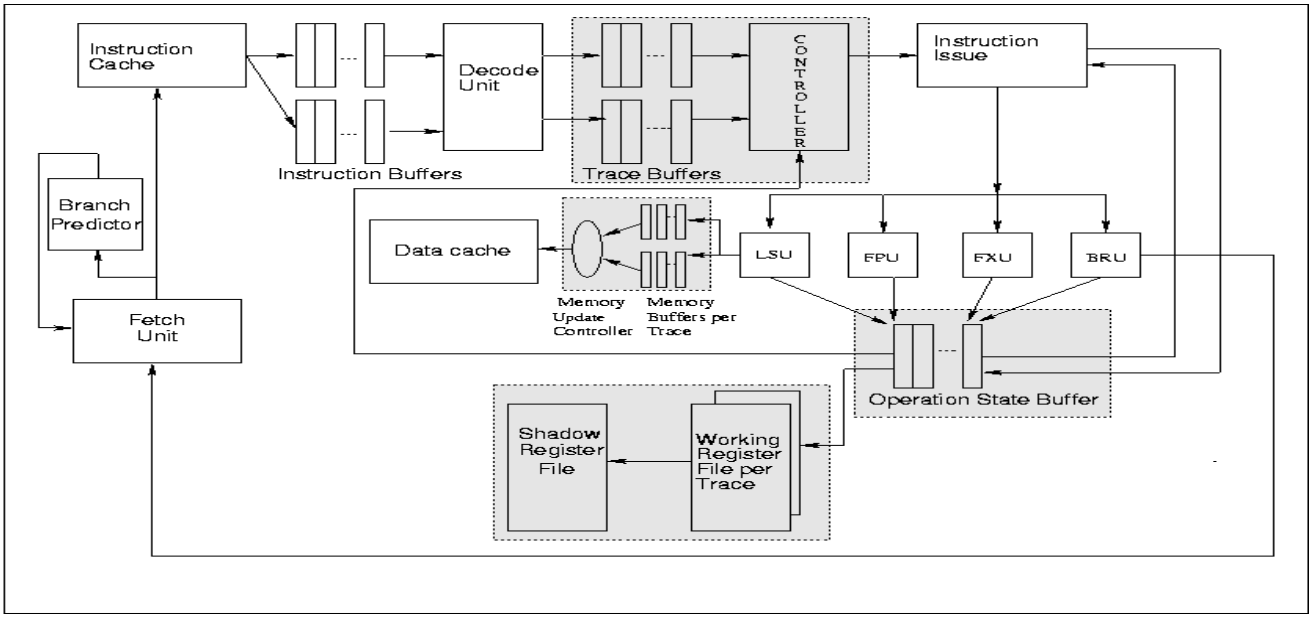

Figure 2: Block diagram of the architecture support proposed

Trace pointer maintains a list of traces, which are under execution in the processor. The list of trace labels as maintained by the trace pointer is shown in figure 4. OSB stores the PC value and the trace label of the instruction under execution in the processor. The other entries in the OSB are execute bit and except bit for each operation that gets executed in the machine. When the system initiates, head, tail and the trace pointers point to the first location of the OSB. When a VLIW instruction is issued its PC value and trace label is written into the OSB on the location pointed to by the tail pointer. The tail pointer is

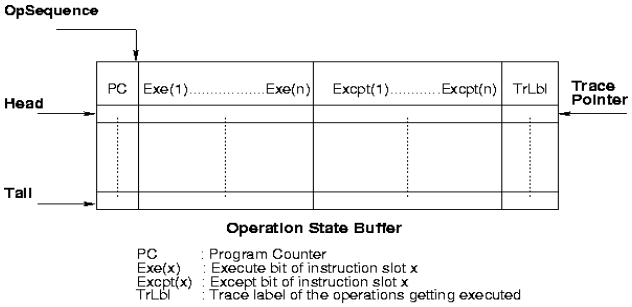

Figure 3: Operation state buffer

then incremented to point to the next location. The OSB address of the entry is communicated back to the issue unit and then each operation in the VLIW instruction is issued with an attached buffer address and an operation identifier. During execution of operations, corresponding execute bit and except bits for each operation are set according to the execution status of the operation. If operation suffers a stall during execution, the corresponding except bit is set and in the next cycle this is communicated back to the controller to start the execution of the next trace. If the operation executes successfully then the corresponding execute bit is set and the results are updated in the register file. At the start of each cycle, the entry pointed to by the head pointer is checked. If the execution of all the operations corresponding to the entry pointed to by the head pointer is complete with no exceptions incurred, the results are updated in the working set of registers and the head pointer is incremented to point to the next entry. When the head pointer is incremented, previous entry is cleared and

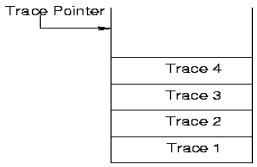

Figure 4: Stack of traces maintained by the Trace pointer.

made available to the next instruction that is issued to execute. The trace label of the entry cleared is also removed from the list maintained by the trace pointer.

On an exception i.e. when any except bit of an entry is set an exception is raised and a signal is sent to the controller to switch to another trace. As soon as the exception is raised, all the entries of the OSB corresponding to the excepted trace are checked and the results of the operations, which have completed their execution, are forwarded to the register file. The operations, which are midway in their execution process,

are saved and issued when the processor returns from cache miss or a page fault. The trace pointer is updated to point to the excepted trace and the execution of the next probable trace is started which is present in the trace buffer. As soon as the exception is handled the processor resumes execution of the original trace. To resume execution of the original trace, processor suspends execution of the current trace and switches back to the trace pointed to by the trace pointer, which is the original trace. Additional cycles have to be accounted for switching traces as separate hardware threads and it depends on the hardware implementation. A register file is associated with each trace, which we call as working register file. A set of shadow registers is also maintained which is common to all traces. Shadow registers are used to retrieve the state of the processor on a trace misprediction. Shadow registers maintain the state of the processor at correctly executed trace boundaries. A trace uses working set of registers to compute the operations included in it. When the decision points at the tail of the trace assert correct prediction, its working set of registers is committed into the set of shadow registers. After the trace is committed all the other traces (of the same decision tree) that are currently in the execution stage are flushed and the execution of the next trace of the next decision tree is initiated. On a trace misprediction, contents of the set of working registers of the corresponding trace are discarded. A signal is sent to the controller, from the OSB, to switch to the next probable trace (which is already ahead in its execution) and the entries of the OSB with the mispredicted trace label are discarded. The entry of the mispredicted trace is also removed from the list maintained by the trace pointer. To take care of the memory writes, a set of memory buffers is maintained as shown in figure 2. These memory buffers are present for each trace and memory writes are done to these buffers only. Subsequent memory read operations for the writes are done from the memory buffers. If the trace is found to be correctly predicted then the memory buffers are committed to the memory. On a misprediction, these memory buffers are discarded and the execution of the next most probable trace proceeds. As the execution of this trace is already in progress using stall cycles of the (previous) most probable trace, it is ahead in its execution and thus effective misprediction penalty is reduced.

4. Simulation and Results

Simulations were carried on Specint92 benchmarks and compiled using TriMedia compiler with optimiztion level –O3 and then simulated on the TriMedia simulator. Simulations report the statistics of the number of data cache stalls incurred for each decision tree. With the help of these statistics, the performance is evaluated using equations 1, 2 and 3. The statistics of the data cache stalls obtained from the simulations are plugged into the mathematical model equations to estimate the performance gain that can be obtained by the multithreaded architecture proposed in this work. Simulation results are given in Table 2. As is evident from the results, there is a reduction in the overall misprediction penalty when the stall cycles of the processor are overlapped with the execution of the next probable trace. The performance gain varies among the three benchmarks that are simulated. Highest performance gain is achieved in 023.eqntott because this application does not show much data locality and the total number of data cache misses is very high in this application. Least reduction in penalty is shown by 022.li because of the fact that the total number of cache misses in this application is very low. Overall we see a net performance gain.

This scheme can considerably increase the performance of the applications which show large number of cache misses. In certain media application where the cache stall cycles are as high as 20-40% of the application execution time the net performance can be improved considerably. The performance gain reported in this paper is conservative than that can actually be obtained by overlapping the stall cycles of the processor on cache misses/page faults because we have not simulated the overlapping of page faults in our work. Only cache misses have been considered to evaluate the advantage in the misprediction penalty achieved by overlapping the stall cycles of the processor with the execution of the next probable trace.

4. Conclusion

The performance of the VLIW processors can be improved considerably by dividing the application into multiple traces and using dynamic branch prediction for scheduling. By speculatively scheduling traces based on their probability of execution, the performance obtained by us is approximately 1.41 times the original TriMedia performance. A considerable reduction in the misprediction penalty was achieved by overlapping the stall cycles of the processor by useful execution in the next probable trace during cache misses alone. We proposed a multitreaded architectural support in this paper that enhances VLIW architectures to utilize the stall cycles. An average reduction of approximately 8.39% was observed in the overall misprediction penalty as compared to the penalty incurred when the stall cycles due to cache misses are not overlapped with execution of the next probable trace. Our results are conservative

because we have considered only cache misses in our simulations and ignored stall cycles due to page faults. The overall average performance is approximately 1.44 times the original TriMedia performance.

References

[1] Amir Roth et al., "Speculative Data-Driven Multithreading", In *Proceedings of the 7th International Conference on High Performance Computer Architecture (HPCA -7)*, pp. 192-202, Monterry, Mexico, Jan. 22-24, 2001.

[2] Dean M. Tullsen et al., "Simultaneous Multithreading: Maximising On-Chip Parallelism", In *Proceedings of 22nd International Symposium on Computer Architecture (ISCA-22)*, pp.392-403, Santa Margherita, Italy, June 1995.

[3] Dean M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreaded Processor", In *Proceedings of 23rd International Symposium on Computer Architecture (ISCA-23)*, pp. 191-202, Philadelphia, PA, USA, May 1996.

[4] E. Ozer et al., "A Fast Interrupt Handling Scheme for VLIW Processors", In *Proceedings of the 1998 International Conference of Parallel Architecture and Compilation Techniques (PACT'98)*, pp. 136-141, Paris, France, Oct. 1998.

[5] Gurindar S. Sohi et al., "Speculative Multithreaded Processors", *IEEE Computer*, pp. 66-73, April 2001.

[6] Jan Hoogerbrugge, "Dynamic Branch Prediction for a VLIW Processor", In *Proceedings of the 2000 International Conference on Parallel Architecture and Compiler Techniques (PACT'00)*, pp. 207-216, Philadelphia, PA, Oct. 2000.

[7] Jan Hoogerbrugge et al., "Instruction Scheduling for TriMedia", *Journal of Instruction Level Parallelism*, 1, 1999.

[8] John R. Ellis, "*BULLDOG: A Compiler for VLIW Architectures"*, ACM Doctoral Dissertation Awards, MIT Press, Cambridge, Massachusettes, 1986.

[9] Manvi Agarwal et al., "Speculative Trace Scheduling in VLIW Processors", Accepted for Publication in *International Conference on Computer Design (ICCD'02)*, Freiburg, Germany, September 16 - 18, 2002.

[10] Peter Y. T. Hsu et al., "Highly Concurrent Scalar Processing", In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA-13)*, 14(2):386-395, Tokyo, June, 1986.

[11] Robert S. Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)", In *Proceedings of International Symposium on Computer Architecture*, 27(2):186-195, Atlanta, Georgia, May 1999.

[12] Sanjeev Banerjia et al., "Treegion Scheduling for Highly Parallel Processors", In *Proceedings of Euro-Par'97*, pp. 1074-1078, Passau, Germany, Aug. 1997.

[13] Scott A. Mahalke et al., "Effective Compiler Support for Predicated Execution using the Hyperblock", In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 45 - 54, Portland, Oregon, USA, Dec.1-4, 1992.

[14] Steven Wallace et al., "Threaded Multiple Path Execution", In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 238-249, Barcelona, Spain, 1998.

[15] W. W. Hwu et al., "The Superblock: An Effective Structure for VLIW and Superscalar Compilation", *Journal of Supercomputing*, pp. 229 - 248, 1993.

Benchmarks	Misprediction Penalty With/Without Stall Use (Cycles)	Penalty Reduction (%)	Performance of Stall Use vs Default TriMedia
008.espresso	20880786.507	7.707	1.5475
	22624617.2036		
022.li	63933184.221	0.157	1.3632
	64034219.9727		
023.eqntott	2651488.671	17.31	1.4191
	32067224.795		
Average		8.391	1.4433

Table 2: Reduction in branch misprediction penalty achieved by overlapping the stall cycles of processor during cache misses and page faults with the execution of the next probable trace.