A Parallel Formulation for the Simulation of a Generic Branch Predictor

L. Curi-Quintal[1,2] and J. Cadenas[1]
[1]School of Systems Engineering, University of Reading, Reading RG6 6AY, UK
[2]FMAT - Universidad Autonoma de Yucatan, Merida, Mexico
l.f.curiquintal@pgr.reading.ac.uk, o.cadenas@reading.ac.uk

Abstract - *A parallel formulation for the simulation of a branch prediction algorithm is presented. This parallel formulation identifies independent tasks in the algorithm which can be executed concurrently. The parallel implementation is based on the multithreading model and two parallel programming platforms: POSIX threads and Cilk++. Improvement in execution performance by up to 7 times is observed for a generic 2-bit predictor in a 12-core multiprocessor system.*

Keywords: branch predictor simulator, parallel formulation, multi-threading, Cilk++.

1 Introduction

A branch predictor is an algorithm implemented in hardware which is intended to improve the performance of instruction execution in the pipeline of modern microprocessors. This algorithm predicts whether a branch instruction should be taken (T) or not-taken (NT) by the microprocessor, based on stored history of execution of previous branches [1]. Prediction accuracy is inherent to the prediction algorithm and is frequently evaluated and quantified by computer simulation, using branch traces as input data. These branch traces contain a set of branch addresses and outcomes which represent every branch instruction seen by a processor when executing a computing task.

Typically, branch traces are collected from the execution of well-known benchmark programs, and stored in very large files with millions of addresses and outcomes. The simulation process executes a software model of the prediction algorithm and relates obtained results with observed outcomes. The analysis computes a prediction to each branch in the trace, compares this prediction with the actual outcome observed, and keeps a global tally of the number of comparison matches, referred to as hits. Behaviour and accuracy analysis of a branch prediction algorithm implies the execution of a batch process for a number of trace files, each one subject to different parameter values. This analysis requires a significant computational time.

In this paper, a parallel formulation of a generic branch predictor algorithm is described. This parallel formulation exploits the inherent parallelism of the algorithm and reduces the execution time of the analysis by simulation of a branch predictor using multiprocessor systems. The implementation of the parallel formulation has three main steps. Firstly, a concurrent classification of the branch traces, according to the branch address, is performed. Secondly, the execution of the predictor algorithm on each class of address proceeds concurrently. Thirdly, a final step to tally global hits is computed. Results show an improvement in execution performance by up to 7 times when this implementation is evaluated on a 12-core system.

2 Generic Branch Prediction Algorithm

Prediction algorithms are basically defined by two processes: a prediction function and an update procedure. The prediction function makes a prediction (T or NT) for a branch instruction based on information stored from previous branches. The update procedure modifies the recorded history of branches based on the prediction and the actual outcome of the branch. A Branch Prediction Buffer (BPB) maintains information of previous branches as a table. BPB is indexed by a hash function of the branch instruction address. BPB typically contains a set of bits indicating whether the branch was recently taken or not [1].

Fig. 1 describes an algorithm of a generic branch predictor simulator, using a BPB table with 2^S entries, with $S > 0$. Each entry in BPB table stores history for a branch instruction address.

Input arrays *A* and *O* in Fig. 1 contain the branch traces for the simulation. Variable *hits* stores the tally on the correct predictions. Every branch instruction in the trace is related with one element of the BPB table and evaluated by the prediction function (lines 2-3) making a Boolean decision as either taken (T) or not-taken (NT). This value is compared with the recorded outcome (line 4), and matches are stored in *hits* (line5). Finally, BPB table is updated (line 7) at the corresponding entry for the branch instruction address.

```
Input:
   A ← array of addresses
   O ← array of outcomes
Output:
   hits: correct predictions

1. for i=1 to N do
2.  entry ← $A_i$ mod $2^S$
3.  p ← prediction(entry,BPB)
4.  if (p=$O_i$) then
5.      hits ← hits + 1
6.   end if
7.   update(entry,p,$O_i$,BPB)
8. end for
```

Figure 1. Pseudocode of a generic branch predictor simulator algorithm.

2.1 Parallel formulation

The pattern access to BPB table determines the behaviour of the algorithm in Fig.1: each branch instruction accesses only the entry in BPB corresponding to its address. Therefore, data decomposition can be applied sequentially over branch instruction addresses to classify outcomes. This classification creates a list of arrays, where each item in the list corresponds to outcomes for a specific address value in BPB. Thus, each array in the list can be processed by an independent predictor task, and all these tasks can be executed concurrently, although every individual task performs sequentially.

An example of this data decomposition is shown in Fig. 2. Arrays *A* and *O* represent the input traces. Array *A* contains a set of 13 hash values of branch instruction addresses, for a BPB table with 8 entries (entry values between 0 and 7). Array *O* contains its corresponding set of outcomes (T or NT).

i	1	2	3	4	5	6	7	8	9	10	11	12	13
A	5	6	2	4	6	1	7	1	5	3	5	3	7
O	T	T	NT	NT	T	T	T	NT	NT	NT	T	T	NT

LO

0	1	2	3	4	5	6	7
	T	NT	NT	NT	T	T	T
	NT		T		NT	T	NT
					T		

Figure 2. Example of data decomposition of input traces.

The input data is arranged in a list of arrays, *LO*. Each item in list *LO* corresponds to an entry value in BTB table, and contains an array of outcomes ordered by appearance in the trace. For example, for an entry value of 5, *LO* item has three outcomes (T, NT, T) corresponding to the outcomes of elements 1, 9 and 11 in the input trace.

Fig. 3 describes the parallel formulation for the algorithm presented in Fig. 1. In this parallel formulation, outcomes of the branch traces in array *O* are classified in a list of 2^S arrays (*LO*) according to the value of its corresponding address (lines 1-4). Concurrent tasks process *LO* items. Each task *k* executes sequentially the predictor algorithm on the array elements (lines 5-13) and registers correct predictions in $lhits_k$. After all tasks conclude, all *lhits* are accumulated in a global tally (*hits*).

```
Input:
   A ← array of addresses
   O ← array of outcomes
Output:
   hits: correct predictions

1.  for i=1 to N do
2. entry ← $A_i$ mod $2^S$
3. $LO_{entry}$.append($O_i$)
4.  end for
5.  for each k in[0..$2^S$-1] do parallel
6. while ($LO_k$.get($O_{ki}$)) do
7.    p ← prediction(k, BPB)
8.    if (p=$O_{ki}$) then
9.   $lhits_k$ = $lhits_k$ + 1
10.   endif
11.   update(k,p,$O_{ki}$,BPB)
12.  end while
13. end foreach
14. for k=0 to $2^S$-1 do
15.  hits = hits + $lhits_k$
16. end for
```

Figure 3. Pseudocode of the parallel formulation for the simulator of a generic branch predictor.

3 Parallel implementation

The parallel formulation algorithm in Fig. 3 was implemented using a shared memory model for concurrent tasks in multiprocessor systems called multithreading [2]. Multithreading is a parallel programming model that allows concurrent execution of multiple threads in the same process. A thread is a sequence of instructions within a process that can be scheduled for independent execution with other threads. Every program has one main thread. This thread can perform all the tasks by itself, or create other threads with defined subtasks. These subtasks should be designed to encapsulate functionality in order to exploit the inherent concurrency in the algorithm. Thread synchronization operations influence on the overall execution time performance [3]. Two multithreading programming platforms were used to compare performance of parallel implementations: POSIX threads and Cilk++.

3.1 POSIX threads

POSIX threads (*pthreads*) API (Application Program Interface) is a standard approved by IEEE for thread management. *Pthreads* is a portable threading library designed to provide a consistent programming interface across different operating systems platforms. Their main functions focus on thread creation, destruction and synchronization. The most common functions for thread synchronization are the mutual exclusion (*mutex*) locks and barriers. [4]

The *pthreads* version for parallel simulation of the branch predictor algorithm used in this paper encapsulates the classification of the outcomes and the predictor procedure. In the classification step, input arrays are divided in sub-arrays and each thread classifies a sub-array in a local version of *LO* list. After this classification is completed, corresponding arrays from all local *LO* lists are orderly merged in global arrays to create a global *LO* list. Then, the predictor function is executed concurrently, where each thread sequentially processes one item of the global *LO* list at a time. Finally, every thread updates the global tally of correct predictions with its own local tally.

In this implementation, barrier synchronization is used at the end of the input classification and at the end of merging local *LO*s. *Mutex* synchronization is used for concurrent update of the tally of correct predictions.

3.2 Cilk++

Cilk++ is a language extension for programming languages C/C++. Three statements make up the main part of the extension; *cilk_spawn*, *cilk_sync* and *cilk_for*. *cilk_spawn* and *cilk_for* are used to create parallel tasks, either with complete functions or with loop iterations. The *cilk_sync* statement is a local barrier, and is used to synchronize parallel tasks created by *cilk_spawn*. Aditionally, Cilk++ includes a library for mutex locks. Locking tends to be used much less frequently than in other parallel environments, such as *pthreads*, because all protocols for control synchronization are handled by the Cilk++ runtime system. The Cilk++ runtime system is based on a work-stealing scheduler using threads. This is a dynamic load-balancing scheduler and improves the utilization rate of processing units in a system [5].

The Cilk++ version for parallel simulation of branch predictor algorithm used in this paper implements three stages: input array classification, merging of local *LO* and prediction with verification. Every stage is implemented with a *cilk_for* statement. *cilk_for* is a replacement for the conventional C++ *for* statement and executes loop iterations in parallel. Cilk++ compiler converts a *cilk_for* loop into a divide-and-conquer recursive function encapsulating the loop body. This strategy benefits Cilk++ scheduler performance [6]. After the three *cilk_for* loops, the sum of all the partial tally of correct predictions is performed sequentially.

4 Results

Both parallel implementations were executed on an Intel 12-core (dual 6-core Xeon X5690, 3.47GHz) desktop system with Scientific Linux 6.0 (release 2.6.32-131). *Pthreads* version was compiled with gcc 4.4.4 (Red Hat 4.4.4-13), and Cilk++ version was compiled with cilk++ (GCC) 4.2.4 (Cilk Arts build 8503). Branch traces were collected from SPEC 2000 benchmark programs, with sizes of 10 million and 30 million of traces. BPB table was set at 4096 entries. We specifically exercised a 2-bit branch predictor algorithm [1].

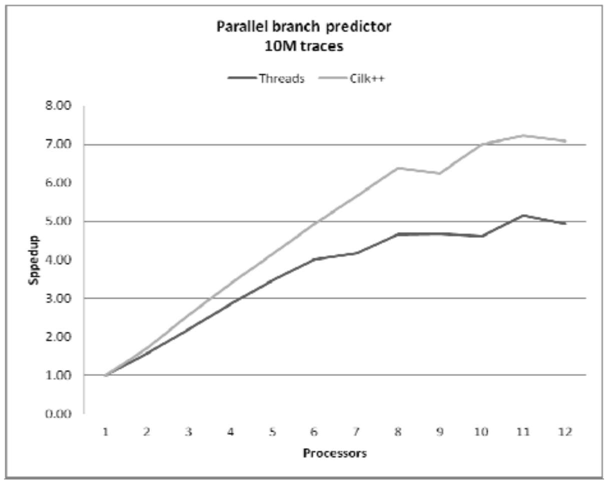

Figure 4. Speedups of the simulation of the parallel branch predictor in a 12-core system with 10 million traces.

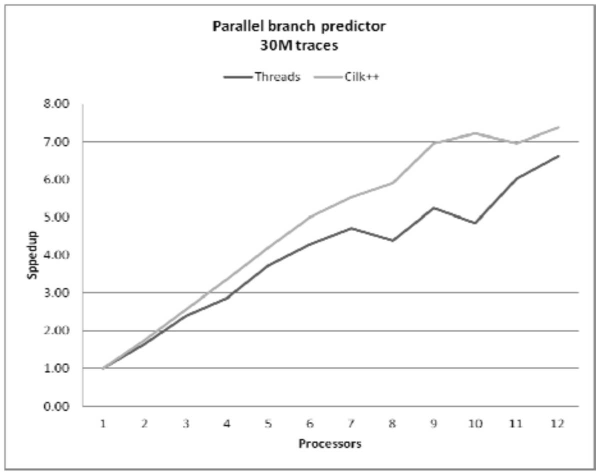

Figure 5. Speedups of the simulation of the parallel branch predictor in a 12-core system with 30 million traces.

Fig. 4 and Fig. 5 show the execution time speedup for *pthreads* and Cilk++ versions with input traces of 10 million and 30 million of elements, respectively.

On each parallel execution, one thread was created for each executing core. The execution time was calculated by the average of 50 executions of every version of the program (sequential, *pthreads* parallel and Cilk++ parallel).

5 Discussion

The Cilk++ implementation shows better performance than the *pthread* implementation for both sizes of branch traces files. Speedup in both implementations exhibits a sustained improvement on the first group of 6-cores. However, this improvement starts to decline with the second group of 6-cores. This behaviour is due to both the distribution of the data on cache memory of each processor and a slower external buffer that interconnects processors and memory.

The speedup tendency is very similar for Cilk++ implementation with both sizes of input traces. However, for *pthreads* implementation, with 10 million input traces, speedup starts to decline when the number of threads increases. With 30 million traces this tendency is not observed. This behaviour reflects how the cost of creating and synchronizing threads influences the execution performance.

The increasing speedup in performance suggests that concurrent independent tasks have been effectively identified from the sequential algorithm.

6 Conclusions

A parallel formulation for the simulation of a 2-bit branch prediction algorithm has been proposed. This parallel formulation was implemented, based on the multithreading model, using *pthreads* and Cilk++ programming platforms. Execution time performance exhibits an improvement with an incremental tendency by up to 7 times for a 12-core multiprocessor system. These results suggest that the parallel formulation effectively identifies inherent parallel tasks in the algorithm.

Future work is aimed at using the parallel formulation of the simulation of the 2-bit branch predictor with other branch predictor algorithms to analyze its performance.

7 Acknowledgements

This work was supported by PROMEP and UADY.

8 References

[1] Patterson, D.A., and J.L. Hennessy. *Computer Organization and Design: The Hardware /Software Interface, Fourth Edition*. Morgan Kaufmann Publishers, 2009.

[2] Rauber, T. and G. Runger. *Parallel Programming: for Multicore and Cluster Systems. First Edition*. Springer, 2010.

[3] Akhter, S. and J. Roberts. *Multi-Core Programming. Increasing Performance through Software Multi-threading*. Intel Press, 2006.

[4] Butenhof, D. *Programming with POSIX Threads.* Addison-Wesley Professional, 1997.

[5] Leiserson, C. E. *The Cilk++ concurrency platform.* Proceedings of the 46th Annual Design Automation Conference, 2009 (DAC '09), pp.522-527.

[6] Intel. *Intel Cilk++ SDK Programmer's Guide*. Document Num.: 322581-001US. Intel Corp, 2009.