Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction*

Jakob Engblom†
Dept. of Information Technology, Uppsala University
P.O. Box 337, SE-751 05 Uppsala, Sweden
jakob@it.uu.se / http://user.it.uu.se/~jakob

Abstract

This paper investigates how dynamic branch prediction in a microprocessor affects the predictability of execution time for software running on that processor. By means of experiments on a number of real processors employing various forms of branch prediction, we evaluate the impact of branch predictors on execution time predictability.

The results indicate that dynamic branch predictors give a high and hard-to-predict variation in the execution time of even very simple loops, and that the execution time effects of branch mispredictions can be very large relative to the execution time of regular instructions. We have observed some cases where executing more iterations of a loop actually take less time than executing fewer iterations, due to the effect of dynamic branch predictors.

We conclude that current dynamic branch predictions schemes are not suitable for use in real-time systems where execution time predictability is desired.

1. Introduction

When designing and verifying real-time systems, it is often required or desireable to predict the worst-case timing of pieces of software running on the system [5]. Knowing the execution time properties of your code is one of the most important parts of real-time systems development, and failing to ascertain the timing can be a quick way to system failure [6, 8, 25].

Determining the execution time of a program involves understanding and analyzing both the flow of the program and the timing of the processor the program is running on. To make this process possible, it is necessary that the hardware used is amenable to analysis, and that it is reasonably predictable in its behavior. A modern microprocessor employs many complex mechanisms to increase its performance, like pipelining, caching, parallel execution of instructions, out-of-order execution, and branch prediction. Of these, the prediction and analysis of branch prediction have received relatively little research interest [4, 20].

To our knowledge, however, no work has been performed to quantify how the branch predictors used in current processors actually affect the predictability of the execution time of a program. This paper attempts to remedy this, using a simple experiment as described in Section 3. We investigate the behavior of the Intel Pentium III (Section 6) and Pentium 4 (Section 9), AMD Athlon (Section 7), and Sun UltraSparc II (Section 5) and UltraSparc III (Section 8) processors. We also give a short introduction to branch prediction in Section 2, and finally, Section 10 contains our conclusions.

2. Branch Prediction Techniques

Branch prediction is the name given to a collection of techniques that are intended to reduce the penalty of executing branch instructions. The problem is that in a pipelined processor, the outcome of a conditional branch can only be determined quite late in the pipeline. Unless some guess is made as to where the program will continue, the pipeline will be stalled until the branch is decided since it does not know where to fetch the next instruction. As pipelines are getting

* This paper was presented at the Ninth Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), proceedings published by IEEE Computer Society. The Symposium was held in Washington, D.C., May 27-30, 2003, after being moved due to SARS from Toronto, Canada, where it was originally supposed to have been held. The proceedings still list the location as Toronto. This version of the paper corrects some typos found in the printed versions.

† Jakob is currently employed at Virtutech Inc. in Stockholm, Sweden (www.virtutech.com), and holds an adjunct professor position at Uppsala University.

deeper to support higher clock frequencies, branch prediction is getting more advanced to compensate [12].

From an average performance perspective, branch prediction is quite successful. The most advanced techniques currently in use obtain about 95% in prediction accuracy (the actual accuracy depends on the program executed and can vary quite significantly).

However, from a real-time systems perspective, it is interesting to investigate what the use of branch prediction means for the *predictability* of a program's execution time. Advanced branch prediction is considered generally bad for execution-time predictability [6, 11, 23], and we would like to check this assumption by experiments.

The simplest form of branch prediction is to continue fetching instructions sequentially beyond a branch. If the branch falls through, no time will be lost, but if it is taken, a time penalty is paid. This technique is used in simple embedded processors that have short pipelines (minimizing the penalty) and where the extra gates for more aggressive schemes cannot be motivated. Examples of such processors are the NEC V850 and ARM7 [3, 22].

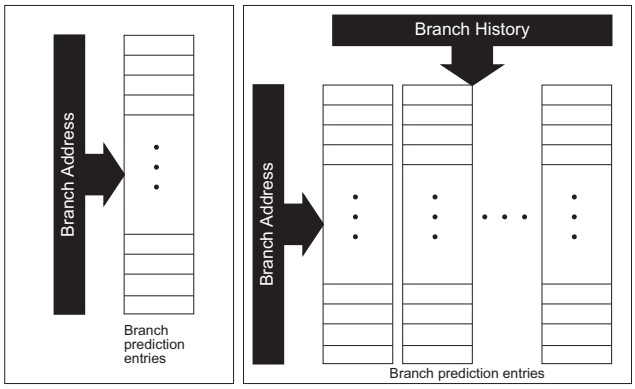
Since most of the execution time of a typical program can be assumed to be spent in loops, a simple improvement is to assume that all backwards branches are taken. This gives us the BTFN scheme, where backwards branches are assumed taken and forward branches assumed not taken. This simple scheme gives an accuracy of 65% to 70% on the EEMBC embedded benchmarks [17]. A twist to this scheme is to allow the compiler to set a bit in an instruction to direct the processor to predict it as taken or not taken. This technique can bring prediction accuracy up to 75% [10], and is found on many processors, including the PowerPC, UltraSparc and Intel's Pentium 4 [21, 26, 15].

Obviously, these *static* techniques do not make the execution time of a program harder to analyze. It is possible to inspect a branch instruction in a program and easily determine how it is going to be predicted and thus its execution time in its taken and not-taken paths. However, to reach higher accuracy than 75%, it is necessary to employ *dynamic* branch prediction techniques.

The simplest form of dynamic predictor is the *single-level* predictor, illustrated in Figure 1(a). In such branch predictors, a branch history table tracks the previous behavior of the branches in the program. For each branch, a one- or two-bit counter tracks if it has been taken or not taken previously [10, 28]. Note that only a subset of the branches in a program can be tracked, since the branch history table has a finite size. In some cases, the branch prediction bits are stored in the instruction cache, which makes the analysis more complex since it is necessary to track which instructions are loaded into and evicted from the cache [26].

Colin and Puaut [4] demonstrated that the Pentium branch predictor (which uses a stand-alone branch history table using two-bit counters to predict branch behavior) can be successfully analyzed and predicted in isolation from cache and pipelining effects. This is expected, since the predictor behavior is local: a branch only affects its own prediction.

Most recent processor designs for the desktop and server field use *two-level* branch prediction techniques, see Figure 1(b). In two-level predictors, the recent history of taken and not-taken branches is tracked (in order to detect patterns in how branches behave or are correlated with each other). This history is then used together with the branch address to perform a lookup into a branch prediction table, which makes the final decision to predict a branch as taken or not [10, 20, 28]. The branch prediction table typically contains two-bit counters.



(a) One-level dynamic branch prediction **(b)** Two-level dynamic branch prediction

Figure 1. Branch prediction techniques

It is obvious that two-level branch predictors are much harder to analyze for execution time estimation, since they rely on the history of execution. The history can be local (kept per branch) or global (a single history is used for all branches). Local predictors are easier to analyze than global predictors.

Mitra and Roychoudhury have made some progress in static analysis for global history-based two-level branch prediction schemes like `GAs` and `gshare` [20]. However, no integration with other analyses like cache and pipeline has been demonstrated, and the scalability of their approach is questionable since they build a large constraint system for the entire program.

3. Experimental Setup

The setup of our experiment is very simple, and is derived from microbenchmark code used when trying

```
for{k=1; k<32; k++) {
  starttimer();
  for(n=0; n < 10000000; n++)    // OUTER LOOP
    {
      for(i=0; i < k; i++)  // INNER LOOP
        {
          __nop();   // Some compiler-dependent way to get a nop
        }
    }
  stoptimer();
  recordtime();
}
```

Figure 2. Code used in the experiment

to measure the timing of a memory hierarchy. The C code is shown in Figure 2. The result of compiling this code is typically an inner loop of three or four instructions (depending on the architecture), with an outer loop containing about four instructions before and after the inner loop.

The entire loop nest fits comfortably in the instruction cache, and all variables are kept in registers, so we can safely assume that the memory system does not influence the results. By having a very large iteration count for the outer loop, the total execution time is large enough to be measurable. Interference by other tasks executing on the machine is minimized by executing the benchmark many times and taking an average. Furthermore, task switches should have a comparatively small effect on a tight loop nest like this (since caches and pipelines refill very quickly).

It is clear that the expected result, in the absence of branch prediction, is that the total execution time for the outer loop should be the greatest for $\texttt{k} = 31$, and the least for $\texttt{k} = 1$, as seen in Figure 3.

If we divide the total execution time by `k`, we should get a monotonically lower value, since the overhead of the outer loop is amortized over more executions of the inner loop (as seen in Figure 4). However, with dynamic branch predictors, this is not the case.

In all graphs in this paper, we use normalized execution times to make the relative magnitude of the changes in execution time clearer. In graphs showing the total execution time (like Figure 3 and Figure 5), the time for executing with $k = 1$ corresponds to 1.0. This baseline means that the relative increase in total execution time from $k = 1$ to $k = 31$ will vary. In graphs showing the execution time per iteration (like Figure 4 and Figure 6), the execution time per iteration for $k = 31$ corresponds to 1.0.

4. V850E

As a base case for our investigation, we use the V850E processor from NEC [22]. This processor simply keeps fetching instructions sequentially beyond a branch. If the branch is taken, it has to squash two instructions in its pipeline, incurring a two-cycle penalty.

On this processor, we get the expected result as described above: the total execution time increases monotonically (as shown in Figure 3), and the time per iteration decreases smoothly from $\texttt{k} = 1$ to $\texttt{k} = 31$, as shown in Figure 4.

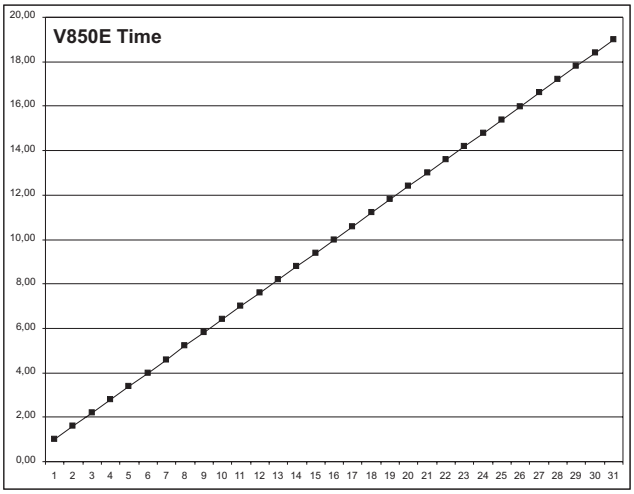


Figure 3. V850E, total execution time

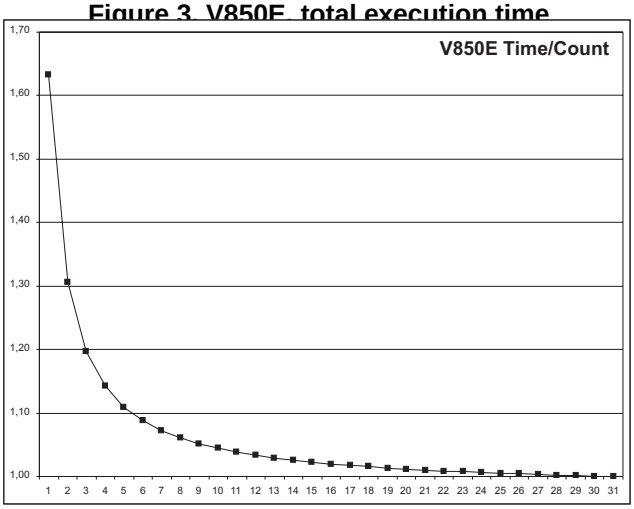


Figure 4. V850E, execution time per iteration

On this processor it is easy to predict the execution time, since we can assume that iterating more iterations of a loop takes more time, and the time for each instruction and branch is statically known.

5. UltraSparc II

The UltraSparc II uses a simple one-level branch predictor, with two bits of information per branch stored in the instruction cache. The penalty for a misprediction is four clock cycles, and the branch prediction success rate is about 87% for integer programs and 93% for floating-point programs [26].

As seen from Figure 5, the total execution time increases monotonically with increasing number of itera-

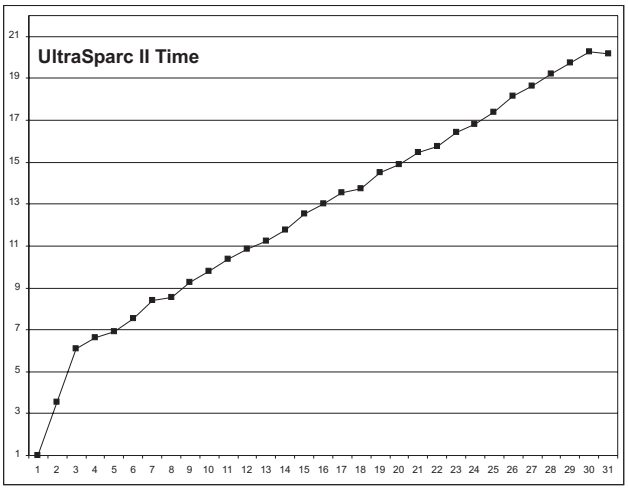


Figure 5. UltraSparc II, total execution time

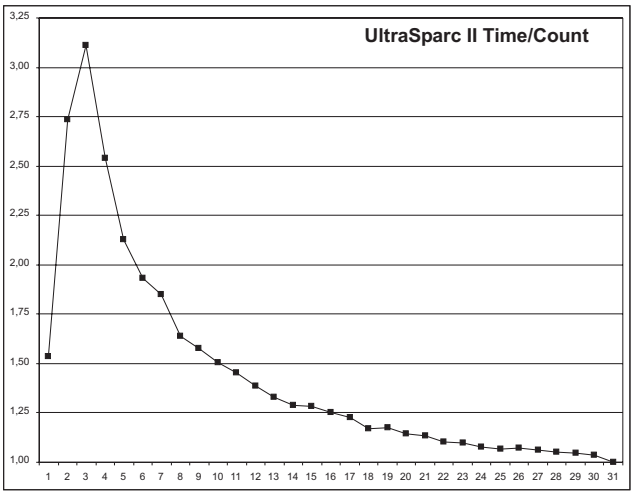


Figure 6. UltraSparc II, time per iteration

tions of the inner loop[1]. The curve is significantly less smooth than the one for the V850E, however.

Figure 6 is more interesting, since it shows that the time per iteration in the inner loop increases at first, doubling the time between $k = 1$ and $k = 3$ iterations, and then starts to decrease rapidly. The good behavior for $k = 1$ is due to the fact that branches are assumed to be not taken when first encountered [2].

This branch predictor is similar to the Pentium branch predictor that was analyzed in [4], and is clearly analyzable. In the specific case of the UltraSparc II, however, the location of the branch prediction information in the instruction cache makes the analysis more complex. This type of branch predictor is now starting to be found even in embedded processors like the ARM11 [2].

[1]The only exception is a small quirk at the last datapoint, which can be explained as a measurement error.

[2]The effect of assuming the opposite can be seen clearly in Figure 14.

6. Pentium III

The Pentium III employs a two-level dynamic branch predictor, since it needs to keep a very deep pipeline filled. The predictor is a Yeh and Patt `PAp`-style predictor [28].

The branch predictor table contains 512 entries, organized as a 16-way set-associative cache with the branch address used for indexing. For each entry in the table, there is a four-bit shift register that tracks the outcome of the last four instances of this branch (i.e. a local branch history). This four-bit value is used to select a 2-bit saturating counter [9, 23].

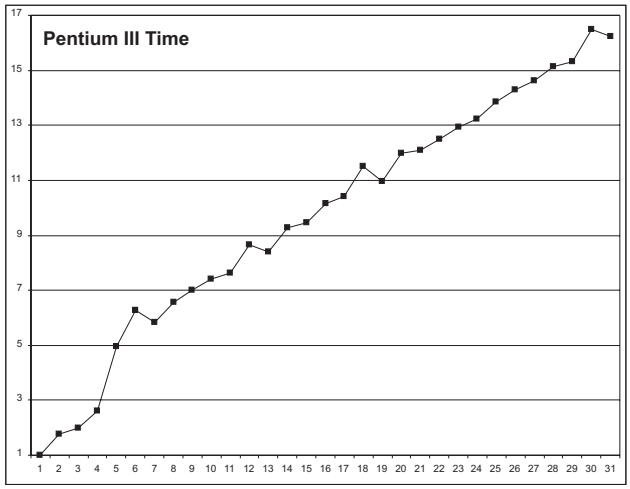


Figure 7. Pentium III, total execution time

On this processor, the execution time for the inner loop varies quite significantly, and is very dissimilar from the graphs obtained for the V850E. Figure 8 shows this variation clearly; the big jump between $k = 4$ and $k = 5$ agrees with the Pentium III performance optimization guidelines, where it is guaranteed that loops executing less than four iterations should have 100% correct prediction (while no such guarantees exist for loops iterating more than four times) [14].

Even more interesting is the total execution time numbers, as shown in Figure 7. Note that at three points along the curve, the execution time goes down as we execute more iterations of the inner loop. This happens with some regularity at $k = 6$, $k = 12$, and $k = 18$ iterations in the inner loop. We call this phenomenon an *inversion*, as executing more instructions take less time than executing fewer instructions.

The inversion effect is caused by the branch predictor. We note that the penalty for mispredicting a branch is between 10 and 26 cycles on the Pentium III (in the benign case that all code is in the cache) [14], which is definitely greater than the cost of executing

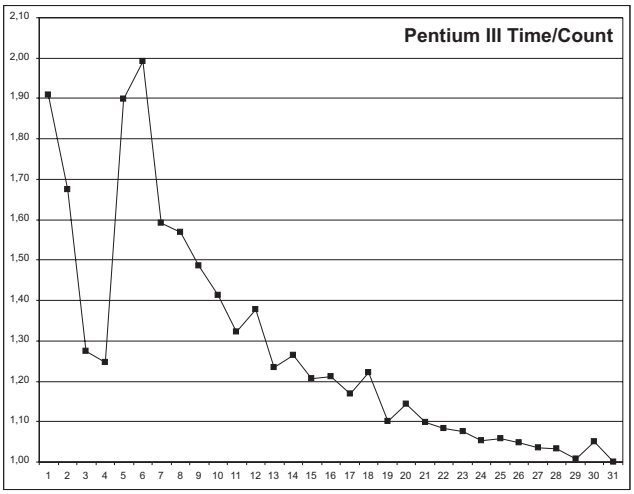


Figure 8. Pentium III, time per loop iteration

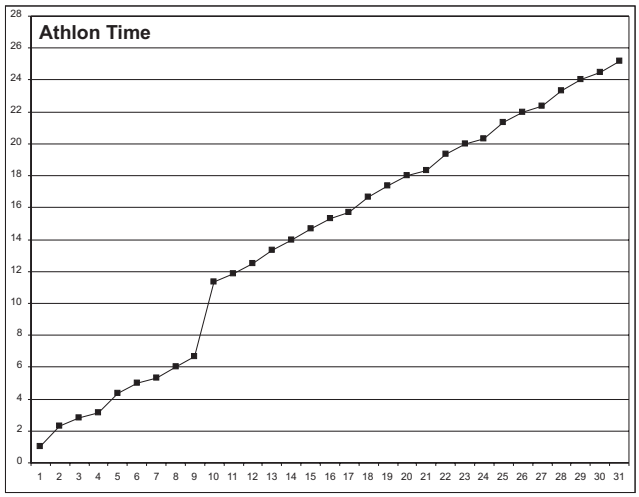


Figure 9. Athlon, total execution time

one more iteration of the loop body. Thus, if the branch prediction algorithm works better for $k = 7$ than for $k = 6$, the result is an inversion.

7. Athlon

The AMD Athlon processor is a newer design than the Intel Pentium III, and it uses a two-level global branch prediction scheme. It contains an eight-bit global history (a shift register containing the taken/not-taken results of the last eight branches executed) that is used together with four bits of the instruction address to select a two-bit saturating prediction counter from a table containing 2048 entries [9, 16]. The minimal branch penalty is 10 cycles [1]. There are some limitations on the location and density of branches: each 16-byte block in the L1 instruction cache can only hold two conditional branches that are correctly predicted. If more branches are present, mispredictions will result [16] (which is not a problem for our loop nest).

As seen in Figure 9, the total execution time increases monotonically (if unevenly), as we increase the number of iterations of the inner loop. There is a big jump at $k = 9$, which corresponds to a big jump in time per iteration as can be seen in Figure 10.

Considering the per-iteration execution times shown in Figure 10, the behavior is more complex that the Pentium III. Especially noticeable is the jump at $k = 9$, which indicates that some kind of unfavorable effect occurs in the branch predictor after the branch history register wraps around. A similar but smaller effect can be seen for the UltraSparc III in Figure 12.

8. UltraSparc III

The UltraSparc III has a global two-level branch predictor that combines 14 bits of the branch address

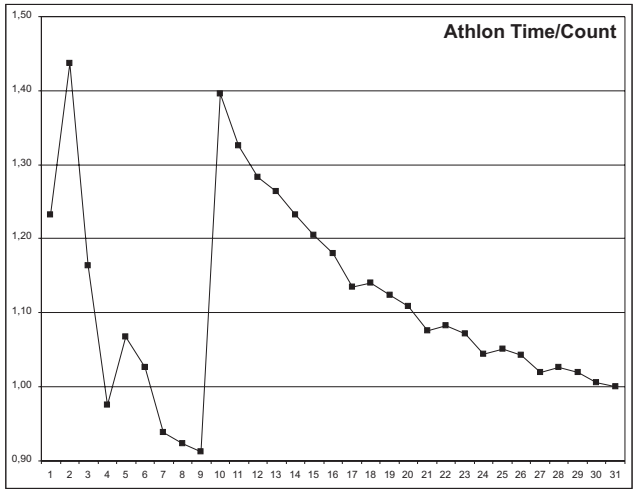


Figure 10. Athlon, time per loop iteration

with 12 bits of global branch history using an undocumented hash function to obtain an index into a branch history table. The branch history table contains 16384 entries, each which is a two-bit saturating counter. Sun reports a prediction accuracy of about 95% on the Spec95 benchmark suite. The branch misprediction penalty is seven cycles for taken branches and less for fall-through, which is significantly less than the Atlon, Pentium III, and Pentium 4 processors [24].

Looking at the total execution time, shown in Figure 11, we see a rather smooth curve that indicates that the branch prediction is working quite well. However, there are two points of inversion in the graph: at $k = 16$, and $k = 2$. The effect at $k = 2$ is quite logical as the default prediction is that an unknown backwards branch is taken. Thus, the inner loop will always be mispredicted for $k = 1$, which can match the execution time of one extra loop iteration.

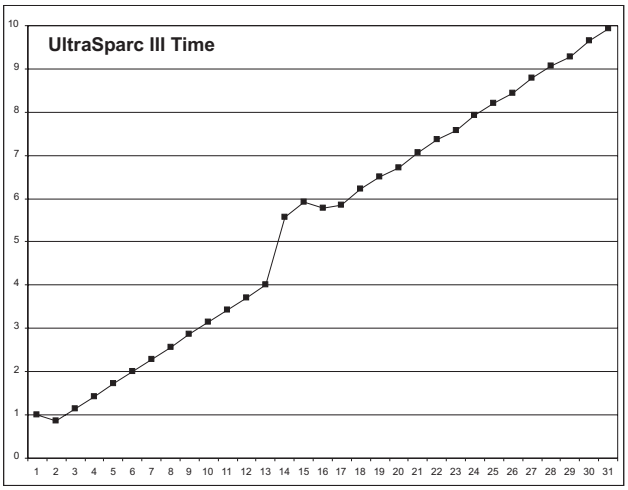


Figure 11. UltraSparc III, total execution time

The time per iteration, shown in Figure 12, is also a rather smooth curve. However, there is a bump at $k = 14$, which seems to indicate that just like on the Athlon, something happens just after the wrap-around of the global history register. This correlates with the

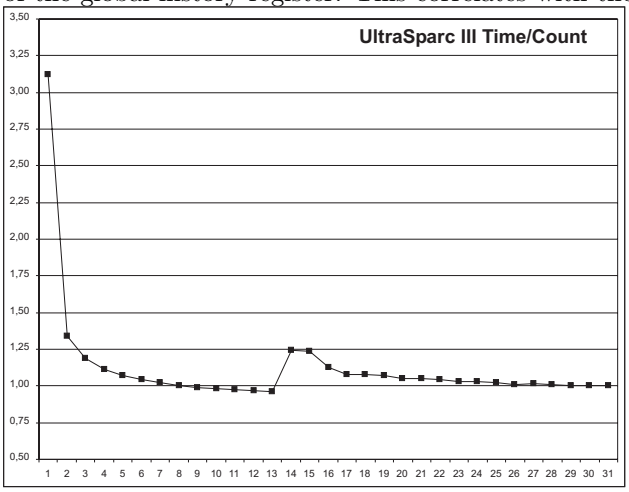


Figure 12. UltraSparc III, time per iteration

Analyzing execution time on this processor is quite tricky, despite the rather smooth curves. The inversions in the execution time graph indicate that surprises could be lurking just around the corner. For example, extrapolating from $k = 1$ to $k = 13$, you would expect a lower total execution than is the case for $k = 14$.

An additional issue with a branch predictor like that on the UltraSparc III and Athlon is that they map *sets* of branches to a single branch prediction table entry [28, 10]. Thus, several branches spread across the program will interact with each other. Determining the set of potential interaction candidates is tricky, since it depends on both the address of branches and the recent history of taken and not-taken branches.

9. Pentium 4

The Pentium 4 has by far the most complex architecture of any of the processors in this investigation. The branch prediction mechanism employed is not documented, but some facts about the general structure have been released by Intel [13].

Built for maximum clock frequency, the Pentium 4 employs a *trace cache* instead of a traditional instruction cache. The trace cache integrates branch prediction into the instruction cache by storing traces of instructions that have previously been executed in sequence, including branches. The same cache line can include both a branch and its target, with a zero penalty for executing the branch along the predicted direction[3]. There is a "small" branch predictor in the processor that is used to direct fetching of operations from the trace cache.

The trace cache is fed by a front-end that fetches instructions from the level 2 cache Here, a second branch predictor with a branch prediction table size of 4096 entries is employed to predict where to fetch instructions next from the L2 cache. This branch predictor uses some form of two-level technique that depends on the branch history, and is supposed to be more advanced than the techniques used on the Pentium III and Athlon processors. The history is continuously fed from the execution back-end to the front-end branch predictor to keep it up-to-date with the program behavior. For our experiments, the front-end predictor should have a minimal impact since our small loop should be kept in the trace cache at all times.

The "misprediction pipeline" is documented as being twice as long as for the Pentium III, which should indicate that the minimal branch misprediction penalty is at least 20 cycles. Branch misprediction penalties when missing the trace cache are even higher, since that requires a fetch from the L2 cache. Overall, the misprediction rate is about one-third lower than for the Pentium III [13].

The results for the total execution time for the Pentium 4 is shown in Figure 13. Even compared to the Pentium III, Athlon, and UltraSparc III, this graph is extraordinarily uneven.

We note that we have inversions at several points. Both $k = 2$ and $k = 3$ take less time than $k = 1$, which is consistent with the fact that the default guess

[3]With a regular cache, even a successfully predicted branch will suffer a small penalty from having to redirect the fetching of instructions to another line in the cache. Techniques used to minimize this penalty in other processors includes storing copies of the *instructions* found at the branch target in the branch prediction tables.

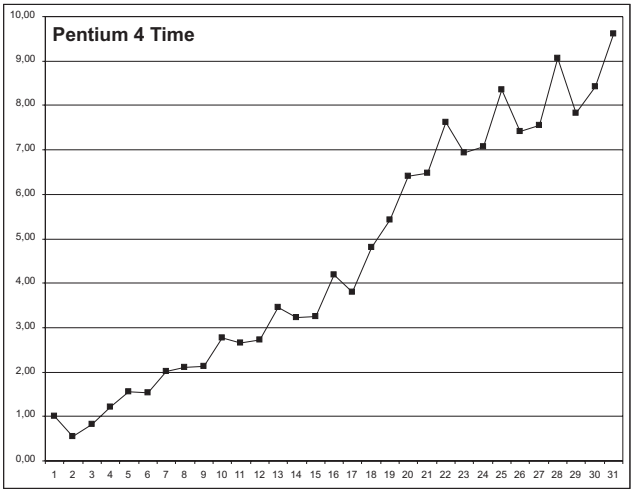


Figure 13. Pentium 4, total execution time

of the branch predictor is that a backwards branches are taken [15] (and the horrendous penalty for a branch misprediction even in the trace cache). We also note a series of inversion spikes $k = 23$, $k = 26$, and $k = 29$ where the size of the inversion is much larger than for

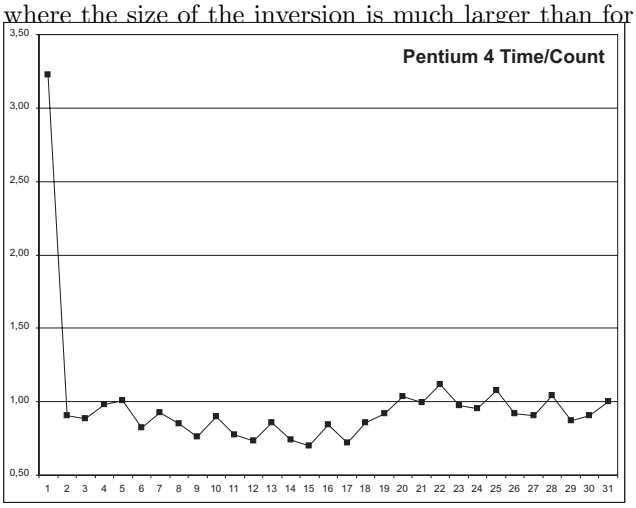


Figure 14. Pentium 4, time per loop iteration

Looking at the time per iteration plotted in Figure 14, we can see that it is much higher for $k = 1$ compared to all other k. The times vary in a pattern that is very unlike any other processor in this investigation, which can only be attributed to the unique form of branch prediction employed.

The Pentium 4 is supposed to be able to predict inner loops executing up to 16 iterations perfectly [15], and this seems consistent with our measurements, since the time per iteration increases significantly beyond $k = 16$.

The branch predictor on the Pentium 4 makes it necessary to investigate at least a range of possible loop iteration counts for a loop to find its worst-case execution time, since there are many inversions. Judging by our experiments, branches and their prediction and misprediction time dominate the execution time of the Pentium 4 (as long as instructions can be kept in the cache).

10. Discussion

From the measurements in the previous sections, it is clear that the effect of branch predictors on the execution time of a program can be very significant.

Accounting for branch predictors using global histories in static timing analysis requires global analysis, since the outcome of each and every branch can affect how the next branch is predicted. Even for the very simple loop structure used in this experiment, the branch prediction algorithms have some problems obtaining good predictions, and the execution time per iteration varies greatly.

Simpler one-level branch prediction techniques like the one used on the UltraSparc II can be predictable if they are suitably implemented (i.e. with a branch history table separate from the instruction cache).

A processor without branch prediction, like the V850E, has a very regular behavior where more instructions executed means a greater total execution time, while on the Pentium III, Pentium 4, and UltraSparc III, increasing the number of loop iterations can actually decrease the execution time. This makes WCET analysis more complex, since we cannot assume that iterating a loop for the maximal number of iterations comprise the worst case, and is a case analogous to timing anomalies [5, 19], requiring WCET analysis to consider very many different scenarios in order to find the worst case. In essence, loops have to be completely unrolled and all possible execution paths examined to find the worst case.

Thus, we note that while modern two-level branch prediction techniques work well on average, their behavior is less desirable when we seek to determine the worst-case execution time of a program.

We further note that this paper has only addressed the pipeline timing aspects of branch prediction, since the code used is small enough to fit comfortably in the instruction cache of the examined processors. For larger programs, the branch predictor might interact with the instruction cache. Often, cache fetches are performed speculatively based on the branch prediction, changing the instruction cache state. This can have significant effects on the execution time predictability [7, 11].

In conclusion, we feel that we can safely state that modern two-level branch prediction schemes are not suitable in circumstances where execution time predictability is sought. Static schemes (either BTFN or

compiler-directed) are the best choice, while one-level predictors fill a middle ground where predictability depends on the implementation details.

For processors where dynamic branch prediction can be turned off (which is quite common [27, 21]), turn it off and rely on static prediction instead (BTFN or compiler-directed). This will lose some performance but gain execution time predictability.

Isolated experiments on the PowerPC 440 indicate that this could cost up to 30% of the execution, but also that for some programs the execution time is actually lower when not using dynamic branch prediction [18]. Which again demonstrates the unpredictability of branch prediction.

Overall, our findings reinforce the well-known rule of thumb that simpler is better when it comes to building analyzable and predictable real-time systems. Building real-time systems requires hardware that is suitable, and predictable timing behavior is not something that can be introduced at the software level if the hardware itself is unpredictable.

In the future, we plan to extend these experiments by investigating how the effect of branch prediction is affected by using larger loop bodies.

Acknowledgements: We thank Stefan Petters for helping us with the measurements on the Athlon processor, and Professor Erik Hagersten for providing the initial impetus for the experiment. Thanks also to the anonymous reviewers who provided valuable feedback on the paper and helped improve the quality of the final text.

References

[1] AMD. *AMD Athlon Processor: x86 Code Optimization Guide*, February 2002. Publication no: 22007 K.

[2] ARM Ltd. *The ARM11 Microprocessor and ARM PrimeXsys Platform*, October 2002. White paper found at `http://www.arm.com/armtech/ARM11`.

[3] ARM Ltd. *ARM 7TDMI Data Sheet*, August 1995. Document no. DDI 0029E.

[4] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, 18(2/3):249–274, May 2000.

[5] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, April 2002. Acta Universitatis Upsaliensis, Dissertations from the Faculty of Science and Technology 36, `http://publications.uu.se/theses/`.

[6] Jennifer Eyre. The Digital Signal Processor Derby. *IEEE Spectrum*, 38(6), June 2001.

[7] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proc. First International Workshop on Embedded Software (EMSOFT 2001), LNCS 2211*. Springer-Verlag, October 2001.

[8] Jack Ganssle. Really Real-Time Systems. In *Proc. Embedded Systems Conference San Fransisco (ESC SF) 2001*, April 2001.

[9] Dayong Gu, Olivier Zendra, and Karel Driesen. The Impact of Branch Prediction on Control Structures for Dynamic Dispatch in Java. Technical Report Number 4547, Institut National de Recherche en Informatique et Automatique (INRIA), September 2002. INRIA-Lorraine/LORIA, `www.loria.fr`.

[10] Linley Gwennap. New Algorithm Improves Branch Prediction. *Microprocessor Report*, 9(4), December 1995.

[11] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems*, 2003. Accepted for publication.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2^{nd} edition, 1996. ISBN 1-55860-329-8.

[13] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1 2001.

[14] Intel. *Intel Architecture Optimization Reference Manual*, 1999. Document Number: 245127-001.

[15] Intel. *Intel Pentium 4 and Intel Xeon Processor Optimization*, 2001. Document Number: 248966-04.

[16] Andreas Kaiser. K7 Branch Prediction. `http://www.s.netic.de/ak/`, December 1999.

[17] Markus Levy. Exploring the ARM1026EJ-S Pipeline. *Microprocessor Report*, April 30, 2002.

[18] Markus Levy. Benchmarks Reveal Design Tradeoffs. *Microprocessor Report*, February 10, 2003.

[19] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. 20^{th} IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.

[20] Tulika Mitra and Abhik Roychoudhury. A Framework to Model Branch Prediction for WCET Analysis. Presented at the WCET Workshop held in conjunction with Euromicro Conference on Real-Time Systems, June 2002.

[21] Motorola Inc. *MPC750 RISC Microprocessor Family User's Manual*, December 2001. Document MPC750UM/D.

[22] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[23] Stefan Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, August 2002.

[24] Peter Song. UltraSparc-3 Aims at MP Servers. *Microprocessor Report*, October 27, 1997.

[25] David B. Stewart. Twenty-Five Most Common Mistakes with Real-Time Software Development. In *Proc. Embedded Systems Conference San Fransisco (ESC SF) 2001*, April 2001.

[26] Sun Microsystems. *UltraSPARC-IIi User's Manual*, 1997. Part No: 805-0087-01.

[27] Sun Microsystems. *UltraSPARC-III Cu User's Manual*, May 2002.

[28] Tse-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proc. of the 20th International Symposium on Computer Architecture (ISCA'93)*, pages 257–266, May 1993.