Branch Prediction and the Performance of Interpreters - Don't Trust Folklore

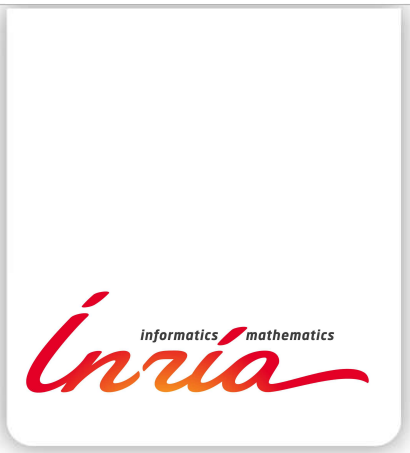
Erven Rohou, Bharath Narasimha Swamy, André Seznec

Branch Prediction and the Performance of Interpreters – Don't Trust Folklore

Erven Rohou, Bharath Narasimha Swamy, André Seznec

RESEARCH REPORT
N° 8405
November 2013
Project-Teams ALF

ISSN 0249-6399 ISRN INRIA/RR--8405--FR+ENG



Branch Prediction and the Performance of Interpreters – Don't Trust Folklore

Erven Rohou, Bharath Narasimha Swamy, André Seznec

Project-Teams ALF

Research Report n° 8405 — November 2013 — 23 pages

Abstract: Interpreters have been used in many contexts. They provide portability and ease of development at the expense of performance. The literature of the past decade covers analysis of why interpreters are slow, and many software techniques to improve them. A large proportion of these works focuses on the dispatch loop, and in particular on the implementation of the switch statement: typically an indirect branch instruction. Conventional wisdom attributes a significant penalty to this branch, due to its high misprediction rate. We revisit this assumption, considering current interpreters, and modern predictors. Using both hardware counters and simulation, we show that the accuracy of indirect branch prediction is no longer critical for interpreters. We also compare the characteristics of these interpreters and analyze why the indirect branch is less important than before.

Key-words: branch prediction, interpreters, performance

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Prédiction de branchement et performance des interpréteurs – Méfiez-vous des idées reçues

Résumé : Les langages interprétés sont utilisés dans des contextes variés. Ils fournissent portabilité et facilité de développement au prix de la performance. La littérature de la dernière décennie s'est intéressée aux raisons de la lenteur des interpréteurs, et à des techniques d'optimisation. Une proportion certaine de ces travaux se concentre sur la boucle de dispatch, et en particulier sur l'implémentation du switch qui est typiquement une instruction de saut indirect. Selon une idée reçue, ce saut est à l'origine d'une forte pénalité, en raison d'une mauvaise prédiction de branchement. Nous réévaluons cette hypothèse, avec des interpréteurs actuels et des prédicteurs modernes. Nous montrons à l'aide des compteurs de performance matériels et de simulation que la précision du prédicteur de branchement n'est plus un problème pour les interpréteurs. Nous comparons aussi les caractéristiques de ces interpréteurs et nous analysons pourquoi les sauts indirects sont moins importants que par le passé.

Mots-clés : prédiction de branchement, interpréteurs, performance

1 Introduction

Interpreters go back to the infancy of computer science. At some point, just-in-time (JIT) compilation technology matured enough to deliver better performance, and was made popular by Java [6]. Writing a JIT compiler is a complicated task. Conversely, interpreters provide ease of implementation, and portability, at the expense of execution speed.

Interpreters are still widely used. They are much easier to develop, maintain, and port applications on new architectures. Some languages used by domain scientists are executed mainly through interpreters, e.g. R, Python, Matlab... Some properties of widely adopted languages, such as dynamic typing, also make it more difficult to develop efficient JIT compilers, and dynamic features are heavily used in real applications [24]. On lower-end systems, where short time-to-market is key, it may not be commercially viable to develop JIT compilers, and these systems rely on interpreters.

Scientists from both CERN and Fermilab report [22] that "many of LHC experiments' algorithms are both designed and used in interpreters". As another example, the need for an interpreter is also one of the three reasons motivating the choice of Jython for the data analysis software of the Herschel Space Observatory [36]. Scientists at CERN also developed an interpreter for C/C++ [7].

Although they are designed for portability, interpreters are often large and complex codes. Part of this is due to the need for performance. The core of an interpreter is an infinite loop that reads the next bytecode, decodes it, and performs the appropriate action. Naive decoding implemented in C consists in a large `switch` statement (see Figure 1 (a)), that gets translated into a jump table and an indirect jump. Conventional wisdom states that this indirect jump incurs a major performance degradation on deeply pipelined architectures because it is hardly predictable. A large body of work has addressed this issue (see Section 6 for an overview).

The contributions of this paper are the following.

- We revisit the performance of `switch`-based interpreters, focusing on the impact the indirect branch instruction, on modern architectures (Nehalem and Sandy Bridge) and current interpreted languages (Python, Javascript, CLI). We show that the performance of the predictors and the characteristics of interpreters make the indirect branch much less critical than before.

- We evaluate the impact of a state-of-the-art branch predictor proposed in the literature on the same interpreters, and we show that the trend observed with existing hardware is confirmed and could be amplified.

The rest of this paper is organized as follows. Section 2 motivates our work: it analyzes in more details the performance of `switch`-based interpreters, it introduces *jump threading*, and measures its impact using current interpreters. Section 3 reviews the evolution of branch prediction over the last decades, and presents the state-of-the-art branch predictors TAGE for conditional branches and ITTAGE [31] for indirect branches. Section 4 presents experimental setup. In Section 5, we present our experimental results and our findings on branch prediction impact on interpreter performance. Section 6 reviews related work. We conclude in Section 7.

Terminology This work is concerned with interpreters executing *bytecodes*. This bytecode can be generated statically before actual execution (as in the Java model), just before execution (as in Python, which loads a `pyc` file when already present, and generates it when missing), or at runtime (as in Javascript). What is important is that the execution system is not concerned with parsing, the code is already available in a simple intermediate representation.

```
1  while (1) {
2    opc = *vpc++;
3    switch(opc) {
4      case ADD:
5      x = pop(stack);
6      y = pop(stack);
7      push(stack,x+y);
8      break;
9
10     case SUB:
11       ...
12   }
13 }
```

(a) C source code

```
1  loop:
2    add     0x2,esi
3    movzwl  (esi),edi
4    cmp     0x299,edi
5    ja      <loop>
6    mov     0x..(,edi,4),eax
7    jmp     *eax
8    ...
9    mov     -0x10(ebx),eax
10   add     eax,-0x20(ebx)
11   add     0xfffffff0,ebx
12   jmp     <loop>
```

(b) x86 assembly

Figure 1: Main loop of naive interpreter

Throughout this paper, the word *bytecode* refers to a virtual instruction (including operands), while *opcode* is the numerical value of the bytecode (usually encoded as a C enum type). *Instructions* refer to native instructions of the processor. The *virtual program counter* is the pointer to the current executed bytecode. *Dispatch* is the set of instructions needed to fetch the next bytecode, decode it (i.e. figure out that it is e.g. an `add`) and jump to the chunk of code that implements its semantics, i.e. the *payload* (in this example the code that performs the addition of the appropriate values).

2 Performance of Interpreters

Quoting Cramer et al. [6], *Interpreting bytecodes is slow.* There are several reasons to this. An interpreter is basically an infinite loop that fetches, decodes, and executes bytecodes, one after the other. Figure 1 (a) illustrates a very simplified interpreter written in C, still retaining some key characteristics. Lines 1 and 13 implement the infinite loop. Line 2 fetches the next bytecode from the address stored in virtual program counter `vpc`, and increments `vpc` (in case the bytecode is a jump, `vpc` must be handled separately). Decoding is typically implemented as a large `switch` statement, starting at line 3.

Many interpreted languages implement an evaluation stack. This is the case of Java, CLI, Python among others (Dalvik is a notable exception [9]). Lines 5–7 illustrate how an `add` is implemented.

The principal overhead of interpreters comes from the execution of the dispatch loop. Every bytecode typically requires ten instructions when compiled directly from standard C (as measured on our own interpreter compiled for x86, described in Section 5.3). See Figure 1 (b) for a possible translation of the main loop to x86. This compares to the single native instruction needed for most bytecodes when the bytecode is JIT compiled. Additional costs come from the implementation of the `switch` statement. All compilers we tried (GCC, icc, LLVM) generate a jump table and an indirect jump instruction (line 7 of Figure 1 (b)). This jump has hundreds of potential targets, and has been previously reported to be difficult to predict [12, 11, 18], resulting typically in an additional 20 cycle penalty. Finally, operands must be retrieved from the evaluation stack, and results stored back to it (lines 9–10) and the stack adjusted (line 11), while native code would have operands in registers in most cases (when not spilled by the register

```
void* labels[] = { &&ADD, &&SUB... };
...
goto *labels[*vpc++];

ADD:
   x = pop(stack);
   y = pop(stack);
   push(stack, x+y);
   goto *labels[*vpc++];

SUB:
   ...
   goto *labels[*vpc++];
```

Figure 2: Token threading, using a GNU extension

allocator).

A minor overhead consists in two instructions that compare the opcode value read from memory with the range of valid bytecodes before accessing the jump table (lines 4–5). By construction of a valid interpreter, values must be within the valid range, but a compiler does not have enough information to prove this. However, any simple branch prediction will correctly predict this branch.

This paper addresses the part of the overhead due to the indirect branches. We revisit previous work on the predictability of the branch instructions in interpreters, and the techniques proposed to address this cost. Other optimizations related to optimizing the dispatch loop are briefly reviewed in Section 6.

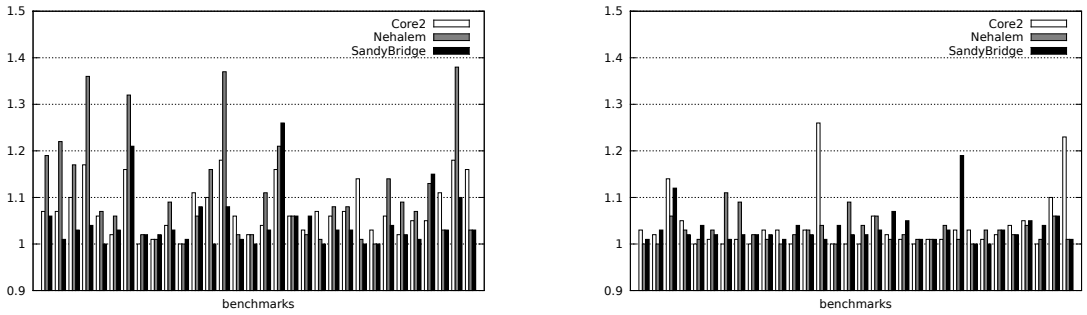
2.1 Jump threading

As mentioned, a significant part of the overhead of the dispatch loop is thought to come from the poorly predicted indirect jump that implements the `switch` statement. *Jump threading* is the name of an optimization that addresses this cost. It basically bypasses the mechanism of the `switch`, and jumps from one `case` entry to the next. Figure 2 illustrates how this can be written. Jump threading, though, cannot be implemented in standard C. It is commonly implemented with the GNU extension named *Labels as Values*[1]. And while many compilers now support this extensions (in particular, we checked GCC, icc, LLVM), older versions and proprietary, processor specific compilers may not support it.

The intuition behind the optimization derives from increased branch correlation: firstly, a single indirect jump with many targets is now replaced by many jumps; secondly, each jump is more likely to capture a repeating sequence, simply because application bytecode has patterns (e.g. a compare is often followed by a jump).

Many interpreters check if this extension is available in the compiler to decide whether to exploit it, or to revert to the classical `switch`-based implementation. Examples include Javascript and Python, discussed in this paper. Previous work [12] reports that it is also the case for Ocaml, YAP and Prolog. This double implementation, however, results in cumbersome code, `#ifdefs`, as well as the need to disable several code transformations that could de-optimize it (the source code of Python mentions global common subexpression elimination and cross-jumping).

[1] Alternatively, a few lines of inline assembly can be used, at the expense of portability.



(a) Speedup due to threaded code in Python-3 (b) Speedup due to opcode prediction in Python-2

Figure 3: Speedups in Python

2.2 Motivation Example

Current versions of Python-3 and Javascript automatically take advantage of threaded code when supported by the compiler. The implementation consists in two versions (plain `switch` and threaded code), one of them being selected at compile time, based on compiler support for the *Labels as Values* extension. Threaded code can be easily disabled though the `configure` script or a `#define`.

In 2001, Ertl and Gregg [11] observed that:

> "for current branch predictors, threaded code interpreters cause fewer mispredictions, and are almost twice as fast as switch based interpreters on modern superscalar architectures".

The current source code of Python-3 also says:

> "At the time of this writing, the threaded code version is up to 15-20% faster than the normal switch version, depending on the compiler and the CPU architecture."

The version revision on the Python repository let us track this comment back to January 2009.

We experimented with Python-3.3.2, both with and without threaded code, and the *Unladen Swallow* benchmarks (selecting only the benchmarks compatible with Python 2 and Python 3, with flag `-b 2n3`). We experimented with a recent Sandy Bridge machine, a slightly older Nehalem, as well as an older Core2. Figure 3 (a) shows the performance improvement due to threaded code.

Nehalem shows a few outstanding speedups in the 30 %–40 % range, as well as Sandy Bridge to a lesser extent, but the average speedups (geomean of individual speedups) for Core2, Nehalem, and Sandy Bridge are respectively 7.2 %, 10.1 % and 4.2 %, with a few outstanding values for each microarchitecture. The impact of threaded code is less on Sandy Bridge than on Nehalem.

Python-2 also supports a limited version of threading, implemented in standard C, aimed at the most frequent pairs of successive opcodes. Nine pairs of opcodes are identified and hardcoded in the dispatch loop. The speedups are reported in Figure 3 (b). Respective average speedups are 3.9 %, 2.8 % and 3.2 % for Core2, Nehalem, and Sandy Bridge. Outstanding values mostly apply to the Core2 architecture.

As illustrated by this simple experiment, the speedup brought by jump threading on modern hardware is much less it used to be. And a better branch prediction is not the single factor contributing to the speedup. As already observed by Piumarta and Riccardi [23] threaded code also executes fewer instructions because part of the dispatch loop is bypassed. Figure 4 illustrates,

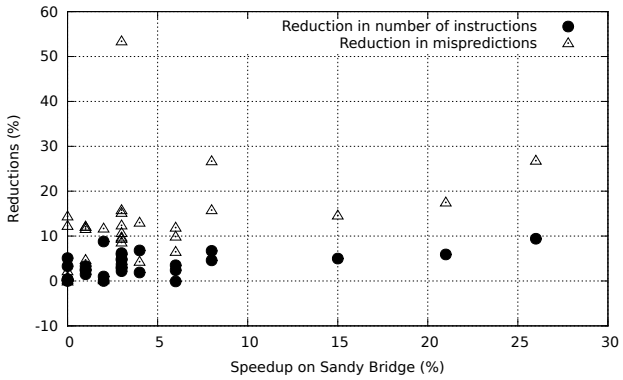


Figure 4: Impact of jump threading: speedup vs reduction in number of instructions executed and branch mispredictions

for each of the above mentioned Python benchmarks, how speedups on Sandy Bridge relates to reductions in number of executed instructions and mispredicted branch instructions. It shows that both contribute to the observed speedup.

2.3 Revisiting Conventional Wisdom

Conventional wisdom considers that the indirect branch that drives the dispatch loop of a `switch`-based interpreter is inherently difficult to predict. Much effort has been devoted – in the literature and in actual source code – to improving its predictability and reducing its overhead.

This was certainly true in the past, and we review related work in Section 6. However, branch predictors have significantly evolved, and they achieve much better performance. In this paper, we study the properties of current interpreters and state-of-the-art branch prediction, and we show that the behavior of the main indirect branch is now a minor issue.

3 (Indirect) Branch Predictors

3.1 State-of-the-art

Many proposals have been introduced for improving the accuracy of conditional branch prediction during the two past decades, e.g. two-level branch prediction [37], hybrid predictors [19], de-aliased predictors [16, 33, 20], multiple history length use [30], and more recently perceptron-inspired predictors [15] and geometric history length predictors [27, 31]. All these propositions have influenced the design of the branch predictors embedded in nowadays state-of-art processors.

While effective hardware predictors probably combine several prediction schemes (a global history component, a loop predictor and maybe a local history predictor), TAGE [31, 29] is generally considered as the state-of-the-art in global history based conditional branch prediction. TAGE features several (partially) tagged predictor tables. The tables are indexed with increasing global history length, the set of history lengths forming a geometric series. The prediction is given by the longest hitting table. TAGE predictors featuring maximum history length of several hundreds of bits can be implemented in real hardware at an affordable storage cost. Therefore TAGE is able to capture correlation between branches distant by several hundreds or even thousands of instructions.

For a long time, indirect branch targets were naively predicted by the branch target buffer, i.e. the target of the last occurrence of the branch was predicted. However the accuracy of conditional branch predictors is becoming higher and higher. The penalties for a misprediction on a conditional branch or on an indirect branch are in the same range. Therefore even on an application featuring a moderate amount of indirect branches, the misprediction penalty contribution of indirect branches may be very significant if one neglects the indirect branch prediction. Particularly on applications featuring `switch`es with many case statements, e.g. interpreters, the accuracy of this naive prediction is quite low. To limit indirect branch mispredictions, Chang et al. [4] propose to leverage the global (conditional) branch history to predict the indirect branch targets, i.e. a gshare-like indexed table is used to store the indirect branch targets. However Driesen and Holzle [8] point out that many indirect branches are correctly predicted by a simple PC-based table, since at execution time they feature a single dynamic target. They proposed the cascaded indirect branch predictor to associate a PC-based table (might be the branch target buffer) with a tagged (PC+global branch history) indexed table.

More recently, Seznec and Michaud [31] derived ITTAGE from their TAGE predictor. Instead of simple conditional branch directions, ITTAGE stores the complete target in tagged tables indexed with increasing history lengths which form a geometric series. As for TAGE, the hitting table featuring the longest history length provides the prediction. At the recent 3rd championship on branch prediction in 2011[2], TAGE-based (resp. ITTAGE-based) predictors were shown to outperform other conditional branch predictors (resp. indirect jump predictors).

3.2 Intuition of ITTAGE on interpreters

TAGE performs very well at predicting the behavior of conditional branches that exhibit repetitive patterns and very long patterns. Typically when a given (maybe very long) sequence of length L branches before the current program counter was always biased in a direction in the past, then TAGE – provided it features sufficient number of entries – will correctly predict the branch, independently of the minimum history *le* needed to discriminate between the effective biased path and another path. This minimum path is captured by one of the tables indexed with history longer than *le*. With TAGE, the outcomes of branches correlated with close branches are captured by short history length tables, and the outcomes of branches correlated with very distant branches are captured by long history length tables. This optimizes the application footprint on the predictor. The same applies for indirect branches.

Two possibilities were explored for ITTAGE in previous studies: global history of all branches (conditional, indirects, returns...) [31] or global history of only indirect branches and calls [28]. None of the solutions was clearly better than the other on the overall tested suites of benchmarks. However on applications such as interpreters where indirect branches form the skeleton of the application control path, we privilege using global history of only indirect branches and calls. We provide below the explanation of why ITTAGE should be able to predict with a very high accuracy the main `switch` in the interpreter loop.

When considering interpreters, the executed path is essentially the main loop around the execution of each bytecode. When running on the succession of basic block bytecodes, the execution pattern seen by the `switch` reflects the control path in the interpreted application: in practice the history of the recent targets of the jump is the history of bytecode opcodes. For instance, if this history is -load load add load mul store add- and if this sequence is unique in the bytecode of the interpreted application, then the next bytecode opcode is also uniquely determined. This history is in some sense a signature of the virtual program counter, it determines the next virtual program counter.

[2] `http://www.jilp.org/jwac-2/`

When running interpreters, ITTAGE is able to capture such patterns and even very long patterns spanning over several bytecode basic blocks, i.e. to "predict" the virtual program counter. Bytecode branches present the particularity to feature several possible successors. However, if the interpreted application is control-flow predictable, the history also captures the control-flow history of the interpreted application. Therefore ITTAGE will even predict correctly the successor of the bytecode branches.

4 Experimental Setup

This section presents our experimental setup, detailing in particular the chosen interpreters and benchmarks. We discuss how data is collected for actual hardware and simulation, and we make sure that both approaches are coherent.

4.1 Interpreters and Benchmarks

We experimented with `switch`-based (no threading) interpreters for three different input languages: Javascript, Python, and the Common Language Infrastructure (or CLI, a.k.a. the core of .NET), and several inputs for each interpreter. We used *Sunspider 1.0* for Javascript, the *Unladen Swallow Benchmarks* for Python, and *SPEC 2000* (*train* input set) for CLI. See Table 1 for an exhaustive list of benchmarks.

We used Python 3.3.2, the latest release of the 3.x series at the time of this writing. The motivation experiment of Section 2 also uses Python 2.7.5, the latest release of the 2.x series. Unladen Swallow benchmarks were run with the flag `--rigorous`. We excluded `iterative_count`, `spectral_norm` and `threaded_count` from the Unladen Swallow suite because they were not properly handled by our measurement setup.

Javascript experiments rely on *SpiderMonkey 1.8.5*. We modified the loop bounds of some of the Javascript Sunspider benchmarks to get longer runtimes, and reduce timing inaccuracies.

We used GCC4CLI [5] to compile the SPEC 2000 benchmarks. GCC4CLI is a port of GCC that generates CLI bytecode from the C language. The CLI interpreter is a proprietary virtual machine that executes applications written in the CLI format. Most of standard C is supported by the compiler and interpreter, however a few features are missing, such as UNIX signals, setjmp, or some POSIX system calls. This explains why a few benchmarks are missing (namely: `176.gcc`, `253.perlbmk`, `254.gap`, `255.vortex`, `300.twolf`). This is also the reason for not using SPEC 2006: more unsupported C features are used, and neither C++ nor Fortran are supported.

Most of the benchmarks are run to completion (including hundreds of hours of CPU for the simulation). We stopped the TAGE simulation at a trillion instructions for `hexiom2` and `164.gzip`, `177.mesa`, and `188.ammp`.

With the exception of Python on the Core2 machine, the interpreters are compiled with Intel icc version 13, using flag `-xHost` that targets the highest ISA and processor available on the compilation host machine. Because compilation of Python on the Core2 failed [3], we used GCC with the similar flag `-march=native`. Build systems are otherwise unmodified.

Some compilers force the alignment of each case entry to a cache line, presumably in an attempt to fit short entries to a single line, thus improving the performance. The downside is that many more targets of the indirect branch alias in the predictor because fewer bits can be used to disambiguate them. Visual inspection confirmed that this is not the case in our setup. McCandless and Gregg [18] previously reported this phenomenon and further developed a technique that modifies the alignment of individual case entries to improve the overall performance.

[3] A static_assert is triggered when checking the size of a data structure.

Table 1: Benchmarks

Python	regex_effbot	crypto-sha1
call_method	regex_v8	date-format-tofte
call_method_slots	richards	date-format-xparb
call_method_unknown	silent_logging	math-cordic
call_simple	simple_logging	math-partial-sums
chaos	telco	math-spectral-norm
django_v2	unpack_sequence	regexp-dna
fannkuch		string-base64
fastpickle	Javascript	string-fasta
fastunpickle	3d-cube	string-tagcloud
float	3d-morph	string-unpack-code
formatted_logging	3d-raytrace	string-validate-input
go	access-binary-trees	CLI
hexiom2	access-fannkuch	164.gzip
json_dump_v2	access-nbody	175.vpr
json_load	access-nsieve	177.mesa
meteor_contest	bitops-3bit-bits-in-byte	179.art
nbody	bitops-bits-in-byte	181.mcf
nqueens	bitops-bitwise-and	183.equake
pathlib	bitops-nsieve-bits	186.crafty
pidigits	controlflow-recursive	188.ammp
raytrace	crypto-aes	197.parser
regex_compile	crypto-md5	256.bzip2

We tried to manually change the alignment of the case entries in various ways, and observed no difference in performance.

4.2 Branch Predictors

We experimented with both commercially available hardware and recent proposals in the literature. Section 4.2.3 discusses the coherence of actual and simulated results.

4.2.1 Existing Hardware – Performance Counters

Branch prediction data is collected from the PMU (performance monitoring unit) on actual Nehalem (Intel Xeon CPU W3550 3.07 GHz) and Sandy Bridge (Intel Core i7-2620M 2.70 GHz) architectures running Linux. Both provide counters for cycles, retired instructions, retired branch instructions, mispredicted branch instructions, but one cannot discriminate conditional and indirect branches.

We relied on the Tiptop tool [25] to collect performance data from the PMU. Events are collected per process (not machine wide) on an otherwise unloaded workstation. Only user-land events are collected (see also discussion in Section 4.2.3).

Unfortunately, neither architecture has hardware counters for *retired* indirect jumps [14]. Sandy Bridge has a counter for "speculative and retired indirect branches". It turns out that non-retired indirect branches are rare. On the one hand, we know that the number of retired indirect branches is at least equal to the number of executed bytecodes. On the other hand, the value provided by the counter may overestimate the number of retired indirect branches in case of wrong path execution. That is:

$$n_{bytecodes} \leq n_{retired} \leq n_{speculative}$$

Table 2: Branch predictor parameters

Parameter	TAGE	ITTAGE 2	ITTAGE 1
min history length	5	2	2
max history length	75	80	80
num tables (N)	5	8	8
num entries table T_0	4096	256	512
num entries tables $T_1 - T_{N-1}$	1024	64	128
storage (kilobytes)	8 KB	6.31 KB	12.62 KB

or equivalently:

$$1 \leq \frac{n_{retired}}{n_{bytecodes}} \leq \frac{n_{speculative}}{n_{bytecodes}}$$

where $n_{speculative}$ is directly read from the PMU, and $n_{bytecodes}$ is easily obtained from the interpreter statistics.

In most cases (column `ind/bc` of Tables 3, 4, 5), the upper bound is very close to 1, guaranteeing that non retired indirect branches are negligible. In the remaining cases, we counted the number of indirect branches with a pintool [17] and confirmed that the PMU counter is a good estimate of retired indirect branches.

4.2.2 TAGE – Simulation

We also experimented with a state-of-the-art branch predictor from the literature: TAGE and ITTAGE [31]. In this case, the performance is provided through simulation of traces produced by Pin [17]. We used two (TAGE+ITTAGE) predictor configurations. In our reported results TAGE1 assumes a 8KB TAGE and a 12.62 KB ITTAGE, TAGE2 assumes a 8KB TAGE and a 6.31 KB ITTAGE. Table 2 summarizes their parameters.

4.2.3 Coherence of Measurements

Our experiments involve different tools and methodologies, namely the PMU collected by the Tiptop tool [25] on existing hardware, as well as results of simulations driven by traces obtained thanks to Pin [17]. This section is about confirming that these tools lead to comparable instruction counts, therefore experiment numbers are comparable. Potential sources of discrepancies include the following:

- non determinism inherent to the PMU [35] or the system/software stack [21];
- the x86 architecture provides a `rep` instruction prefix. The Intel PMU counts prefixed instructions as 1 (i.e. one instruction made of many micro-ops), while pintools may count each instance separately;
- Pin can only capture events in user mode, while the PMU has the capability to monitor also kernel mode events;
- tiptop starts counting a bit earlier than Pin: the former starts right before the `execvp` system call, while the latter starts when the loader is invoked. This difference is constant and negligible in respect of the running times of our benchmarks;
- applications under the control of Pin sometimes execute more instructions in the function `dl_relocate_symbol`. Because Pin links with the application, more symbols exist in the

executable, and the resolution may require more work. This happens only once for each executed symbol, and is also negligible for our benchmarks.

Since Pin can only trace user mode events, we configured the PMU in user mode only as well. To quantify the impact of ignoring kernel events, we ran the benchmarks in both modes and we measured the number of retired instructions as well as the instruction mix (loads, stores, and jumps). Not surprisingly for an interpreter, the difference remains under one percentage point. For branch instructions, it is even below 0.05 percentage point.

The main loop of interpreters is identical of Nehalem and Sandy Bridge, even though we instructed the compiler to generate specialized code. The average variation of the number of executed instructions, due to slightly different releases of the operating system and libraries, is also on the order of 1 %.

Finally, PMU and Pin also report instruction counts within 1 %.

5 Experimental Results

This section first presents results for each interpreter separately. We then present cross-interpreter discussions.

For each interpreter, we present results in a table (Tables 3, 4, 5). The left part reports general characteristics of the interpreter. Dynamic events are collected on Sandy Bridge, but are very similar to Nehalem. For each benchmark, the tables show the number of executed bytecodes (in millions), the number of native instructions (in billions), the overall measured performance reported as IPC (instructions per cycle), the number of instructions per bytecode, the fraction of branch instructions, the fraction of indirect branch instructions, and finally the number of indirect branch instructions per bytecode.

The second half of the tables report on the performance of each architecture in terms of branch prediction for each benchmark. The first three columns show the number of mispredictions per thousand instructions (MPKI) respectively for Nehalem, Sandy Bridge, and TAGE1 and TAGE2.

5.1 Python

Python is implemented in C. The syntax of the Python language differs slightly between versions 2 and 3 (a converter is provided), and so does the bytecode definition. Still, both have a similar set of roughly 110 bytecodes.

The dispatch loop is identical on Nehalem and Sandy Bridge (with the exception of stack offsets), it consists in 24 instructions for bytecodes without arguments, and 6 additional instructions to handle an argument. These instructions check for exit and tracing conditions, locating and loading the next opcode, accessing the jump table, and directing control to the corresponding address. A few instructions detect pending exceptions, and handle objects allocation/deallocation. Only one indirect branch is part of the dispatch loop.

Table 3 report our results with the Python interpreter. The performance (measure as IPC) on Sandy Bridge is fairly good, showing that no serious problem (such as cache misses or branch misprediction) is degrading performance.

It takes 120 to 150 instructions to execute a bytecode. Considering the overhead of the dispatch, about 100 instructions are needed to execute the payload of a bytecode. This rather high number is due to dynamic typing. A simple `add` must check the types of the arguments (numbers or strings), and even in the case of integers, an overflow can occur, requiring special treatment.

Table 3: Python characteristics and branch prediction for Nehalem (Neh.), Sandy Bridge (SB) and TAGE (T1 and T2)

benchmark	Mbc	Gins	IPC	ins/bc	br	ind	ind/bc	MPKI			
								Neh.	SB	T 1	T 2
call_method	6137	771	2.2	125.6	20 %	0.8 %	1.01	4.8	0.6	0.067	0.067
call_meth_slots	6137	766	2.18	124.8	20 %	0.8 %	1.02	5.9	0.7	0.068	0.068
call_meth_unk	7300	803	2.22	110.0	20 %	0.9 %	1.00	4.2	0.3	0.058	0.058
call_simple	5123	613	2.33	119.6	19 %	0.8 %	1.00	4.1	0.5	0.086	0.086
chaos	1196	162	1.55	135.4	22 %	0.8 %	1.07	18.4	4.4	0.680	2.548
django_v2	1451	332	1.44	228.7	21 %	0.6 %	1.33	15.9	3.9	0.529	1.829
fannkuch	7693	747	1.67	97.1	23 %	1.3 %	1.23	18.4	6.1	0.578	0.592
fastpickle	34	351	1.89	10277	22 %	0.5 %	54	19	2.6	2.258	2.290
fastunpickle	25	278	1.77	11320	22 %	0.8 %	91	16.5	3	2.365	2.673
float	1280	180	1.73	140.6	22 %	0.8 %	1.15	12	3	0.364	0.365
formatted_logging	750	125	1.13	166.9	22 %	0.7 %	1.20	17.5	5.4	0.633	4.220
go	2972	344	1.6	115.7	21 %	0.9 %	1.01	14	5.2	1.121	1.979
hexiom2	33350	3674	1.86	110.2	21 %	1.0 %	1.08	11.9	2.9	0.563	0.832
json_dump_v2	5042	1656	1.62	328.5	23 %	0.8 %	2.61	17.2	3.5	0.827	0.859
json_load	195	271	1.81	1391	26 %	0.5 %	6.76	15.7	3.3	3.074	3.198
meteor_contest	868	106	1.52	122.3	22 %	0.9 %	1.05	16.7	6.9	3.507	3.519
nbody	2703	184	1.73	68.2	23 %	1.5 %	1.00	13.8	5.9	0.700	0.701
nqueens	1296	152	1.46	117.0	22 %	0.9 %	1.06	16.5	3.9	0.549	0.549
pathlib	836	188	1.08	225.3	22 %	0.6 %	1.45	16.5	4.7	0.397	0.633
pidigits	94	223	2.37	2366	7 %	0.1 %	1.42	1.2	0.5	0.356	0.363
raytrace	4662	766	1.66	164.3	21 %	0.7 %	1.12	15.2	3.6	0.577	1.017
regex_compile	1938	214	1.52	110.6	21 %	1.0 %	1.05	15.1	5.3	1.588	2.257
regex_effbot	7	52	2.53	6977	22 %	2.1 %	146	6.1	0.1	0.026	0.027
regex_v8	31	37	1.79	1182	22 %	1.7 %	21	9.8	1.1	0.506	0.534
richards	961	108	1.59	111.9	21 %	0.9 %	1.02	13.2	5.8	0.824	1.518
silent_logging	353	53	1.83	148.8	21 %	0.7 %	1.08	10.3	3.7	0.035	0.035
simple_logging	731	120	1.14	164.4	22 %	0.7 %	1.19	17.4	5.3	0.669	4.748
telco	34	6	5.90	174.6	21 %	1.3 %	2.35	15.6	5.7	1.143	1.150
unpack_seq	966	38	2.04	39.5	21 %	2.6 %	1.01	8.9	4.3	0.056	0.057

In a few cases, the number is much higher, as for the `fastpickle` or `regex` benchmarks. This is because they apply heavier processing implemented in native libraries for performance. In the case of `fastpickle`, the benchmark serializes objects by calling a dedicated routine.

There is generally a single indirect branch per bytecode. Values significantly larger than 1 are correlated with a high number of instructions per bytecode, revealing that the execution has left the interpreter main loop proper to execute a dedicated routine.

Figure 5 plots the MPKI for each benchmark and branch predictor. It is clear that the Sandy Bridge predictor significantly outperforms Nehalem's, and the same applies for TAGE when compared to Sandy Bridge.

In practice, one can note that when the average payload is around 120 to 150 instructions **and** there are no or very few indirect branches apart the main `switch`, i.e., ind/bc ≤ 1.02, then TAGE+ITTAGE predicts the interpreter quasi-perfectly. When the average payload is larger or some extra indirect branches are encountered then misprediction rate of TAGE+ITTAGE becomes higher and may become in the same order as the one of Sandy Bridge.

5.2 Javascript

SpiderMonkey Javascript is implemented in C++. We compiled it without JIT support, and we manually removed the detection of *Labels as Values*. The bytecode consists in 244 entries. The dispatch loop consists in 16 instructions, significantly shorter that Python.

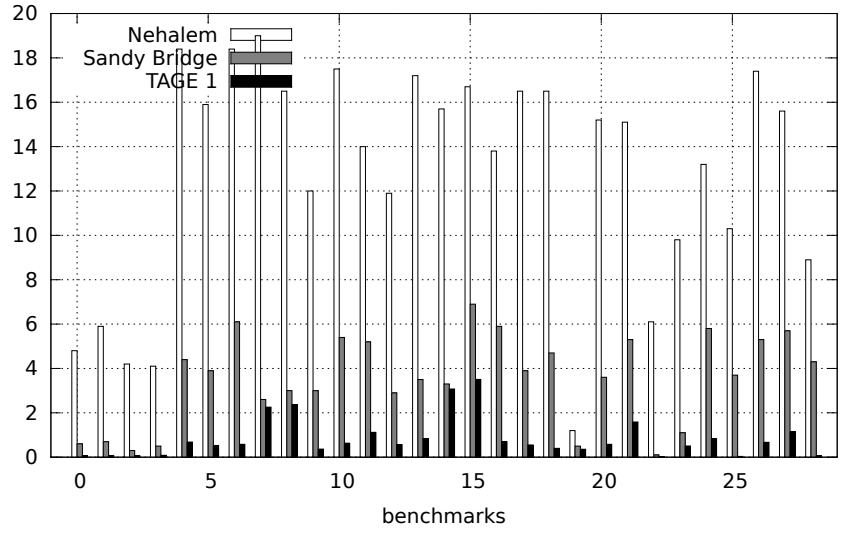


Figure 5: Python MPKI for Nehalem, Sandy Bridge, TAGE 1

Table 4 reports the characteristics of the Javascript interpreter. With the exception of `dna`, indirect branches come from the `switch` statement. `dna` also has a outstanding number of instructions per bytecode. Excluding `dna`, it takes on average in the order of 60 instructions per bytecode.

Table 4 also reports on the performance of the branch predictors, and Figure 6 illustrates the respective MPKI. As for Python, TAGE consistently outperforms Sandy Bridge, which also outperforms Nehalem.

As for the Python interpreters, with the exception of three outliers (the already mentioned `dna`, `bits-in-bytes` and `unpack-code`), TAGE predict quasi perfectly the interpreter. On `unpack`, the average payload is larger than the usual 40 to 80 instructions payload. On `bits-in-bytes`, the accuracy loss is partly associated with the unpredictability of bytecode branches in the interpreted application: `bits-in-bytes` counts the non-zero bits in a byte.

5.3 CLI

The CLI interpreter is written in standard C, hence dispatch is implemented with a `switch` statement. The internal IR consists in 478 bytecodes. This IR resembles the CLI bytecode from which it is derived. CLI operators are not typed, the same `add` (for example) applies to all integer and floating point types. The standard, though, requires that types can be statically derived to prove the correctness of the program before execution. The interpreter IR specializes the operators with the computed types to remove some burden from the interpreter execute loop. This explains the rather large number of different bytecodes. As per Brunthaler's definition [3], the CLI interpreter is a low abstraction level interpreter.

The dispatch loop consists in only seven instructions, illustrated on Figure 7. This is possible because each opcode is very low level (strongly typed, and derived from C operators), and there is no support for higher abstractions such as garbage collection, or exceptions.

Table 5 reports on the characteristics of the CLI interpreter and the behavior of the predictors. The number of speculative indirect jumps is between 1.01 and 1.07 bytecodes. In fact, most of

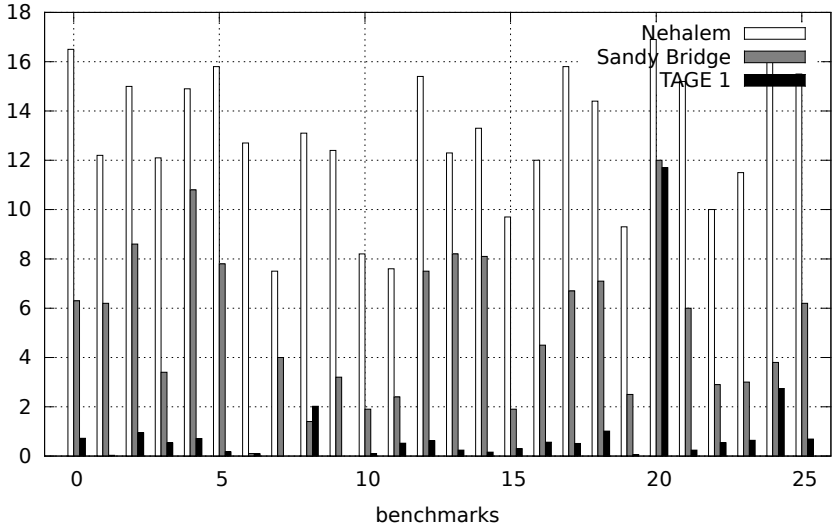


Figure 6: Javascript MPKI for Nehalem, Sandy Bridge, TAGE 1

Table 4: Javascript characteristics and branch prediction for Nehalem (Neh.), Sandy Bridge (SB) and TAGE

benchmark	Mbc	Gins	IPC	ins/bc	br	ind	ind/bc	MPKI			
								Neh.	SB	TAGE 1	TAGE 2
cube	728	33	1.53	44.7	19 %	2.2 %	1.003	16.5	6.3	0.72	1.20
morph	624	28	1.6	45.2	19 %	2.2 %	1.000	12.2	6.2	0.03	0.03
raytrace	4668	27	1.28	56.9	20 %	1.8 %	1.005	15	8.6	0.94	2.19
binary-trees	430	37	1.74	85.3	21 %	1.2 %	1.029	12.1	3.4	0.53	0.53
fannkuch	866	32	1.32	36.9	18 %	2.7 %	1.000	14.9	10.8	0.71	0.70
nbody	597	32	1.47	53.8	21 %	1.9 %	1.001	15.8	7.8	0.17	0.17
nsieve	1093	40	2	36.5	18 %	2.7 %	1.002	12.7	0.1	0.09	0.09
3bit-bits-in-byte	1307	52	1.91	39.4	16 %	2.5 %	1.000	7.5	4	0.00	0.00
bits-in-byte	1085	39	2.35	36.4	16 %	2.7 %	1.000	13.1	1.4	2.02	2.00
bitwise-and	660	37	1.92	56.7	21 %	1.8 %	1.000	12.4	3.2	0.00	0.00
nsieve-bits	1210	44	2.17	36.7	17 %	2.7 %	1.000	8.2	1.9	0.10	0.10
recursive	749	47	1.91	62.4	19 %	1.6 %	1.000	7.6	2.4	0.52	0.54
aes	536	22	1.42	41.7	18 %	2.4 %	1.007	15.4	7.5	0.62	0.86
md5	638	32	1.36	50.5	18 %	2.0 %	1.000	12.3	8.2	0.24	0.24
sha1	675	34	1.44	49.9	18 %	2.0 %	1.000	13.3	8.1	0.16	0.16
format-tofte	330	18	1.86	55.9	19 %	1.8 %	1.02	9.7	1.9	0.30	0.32
format-xparb	312	30	1.43	95.3	19 %	1.1 %	1.032	12	4.5	0.56	0.57
cordic	654	27	1.46	40.6	18 %	2.5 %	1.000	15.8	6.7	0.51	0.51
partial-sums	474	33	1.45	70.1	21 %	1.4 %	1.001	14.4	7.1	1.01	1.01
spectral-norm	872	42	1.95	47.9	19 %	2.1 %	1.001	9.3	2.5	0.06	0.06
dna	0.24	19	1.26	79177	27 %	0.0 %	11.8	16.9	12	11.70	11.70
base64	560	29	1.52	51.6	18 %	1.9 %	1.001	15.2	6	0.24	0.24
fasta	607	36	1.86	59.2	19 %	1.7 %	1.001	10	2.9	0.54	0.54
tagcloud	683	44	1.77	64.1	17 %	1.6 %	1.031	11.5	3	0.64	0.64
unpack-code	158	33	1.78	209.8	20 %	0.5 %	1.022	16.1	3.8	2.74	2.75
validate-input	524	32	1.47	61.3	19 %	1.6 %	1.003	15.5	6.2	0.69	0.69

```
1  loop:
2       cmp     0x299,edi
3       ja      <loop>
4       jmp     *eax
5       ...
6       mov     -0x10(ebx),eax
7       add     eax,-0x20(ebx)
8       add     0xfffffff0 ,ebx
9       add     0x2,esi
10      movzwl  (esi),edi
11      mov     0x...(,edi,4),eax
12      jmp     <loop>
```

Figure 7: Main loop of the CLI interpreter (dispatch + ADD bytecode)

the code is interpreted, even libraries such as libc and libm. Execution goes to native code at a cut-point similar to a native libc does a system call. The short loop is also the reason why the fraction of indirect branch instructions is higher than Javascript or Python.

The compiler could not achieve stack caching[4], probably due to the limited number of registers. Figure 7 illustrates the x86 assembly code of the dispatch loop, as well as entry for the ADD bytecode. The dispatch consists in seven instructions. Two of them (lines 2 and 3) perform the useless range check that the compiler could not remove. Note the instructions at line 9–11: the compiler chose to replicate the code typically found at the top of the infinite loop and move it at the bottom of each case entry (compare with Figure 1).

Across all benchmarks, 21 instructions are needed to execute one bytecode. Nehalem, Sandy Bridge and TAGE are ranked in the same order as for the other two interpreters.

However, with the CLI interpreter, the accuracy of TAGE+ITTAGE is often relatively poor, i.e. misprediction rate exceeds 1 MPKI. We remarked that in these cases the accuracy of the smaller ITTAGE (TAGE2) is much lower than the one with the larger ITTAGE (TAGE1). Therefore the relatively low accuracy seems to be associated with interpreter footprint issues on the ITTAGE predictor. To confirm this observation, we run an extra simulation TAGE3 where the ITTAGE predictor is 50 KB. A 50KB ITTAGE predictor allows to reduce the misprediction rate to the "normal" rate except for `crafty` which would still need a larger ITTAGE. The very large footprint required by the interpreter on the ITTAGE predictor is also associated with the huge number of possible targets (478) in the main `switch` of the interpreter.

5.4 Discussion

Generally all three interpreters run at quite high performance with median values of 1.7, 1.6 and 1.2 IPC for respectively Python, Javascript and CLI on Sandy Bridge.

Our experiments clearly show that between two recent generations of Intel processors, Nehalem and Sandy Bridge, the improvement on the branch prediction accuracy on interpreters is dramatic for Python and Javascript. Our experiments simulating TAGE and ITTAGE show that as long as the payload in the bytecode remains limited and do not feature significant amount of extra indirect branches then the misprediction rate on the interpreter execution can be even become insignificant (less than 0.5 MPKI).

[4] despite a number of attempts...

Table 5: CLI characteristics and branch prediction for Nehalem (Neh.), Sandy Bridge (SB) and TAGE (T1 to T3)

benchmark	Gbc	Gins	IPC	ins/bc	br	ind	ind/bc	MPKI				
								Neh.	SB	T 1	T 2	T 3
164.gzip	78.8	1667.2	1.23	21.2	23 %	4.8 %	1.02	18.7	14.5	0.64	1.58	0.62
175.vpr	18.4	400.9	1	21.8	23 %	4.7 %	1.03	24.8	21.9	6.49	13.62	1.08
177.mesa	118.6	3177.1	1.3	26.8	23 %	4.0 %	1.07	13.8	11.1	0.21	0.59	0.20
179.art	4.7	58.5	1.49	12.5	26 %	8.0 %	1.00	7.3	9.3	0.38	0.38	0.37
181.mcf	13.3	181.2	1.19	13.7	25 %	7.4 %	1.01	17.5	15.1	1.33	2.09	1.08
183.equake	40.5	726.1	1.09	17.9	24 %	5.7 %	1.02	20.8	19	0.47	0.68	0.43
186.crafty	35.8	1047.3	1.03	29.2	22 %	3.5 %	1.04	21.1	19.2	11.87	16.31	4.01
188.ammp	91.3	1665.9	1.24	18.3	24 %	5.7 %	1.04	19.5	14.4	0.39	1.14	0.30
197.parser	12.6	447.5	1.24	35.4	22 %	3.0 %	1.06	14.4	10.8	1.01	2.75	0.70
256.bzip2	28.3	460.8	1.32	16.3	24 %	6.2 %	1.02	17.5	12.7	1.55	2.15	0.44

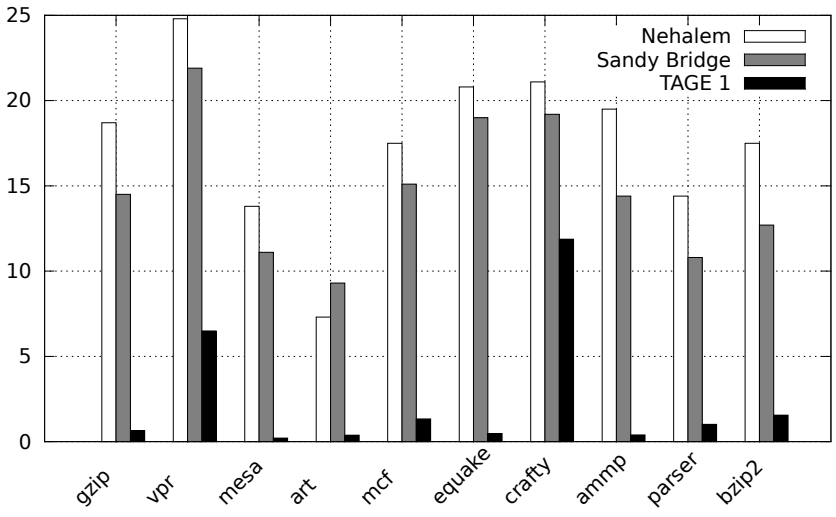


Figure 8: CLI MPKI for Nehalem, Sandy Bridge, and TAGE 1

Therefore, the predictability of branches, both indirect and conditional, should not be considered as an issue anymore for Javascript and Python interpreters.

For the CLI interpreter, the results on Nehalem and Sandy Bridge are much more mitigated, with Sandy Bridge only reducing the misprediction rate by at most 25 % and often much less, e.g. on `art`, it even loses accuracy. Without information on the predictor design in Sandy Bridge, but with the feedback of the simulation of TAGE, we suspect that Sandy Bridge predictor has not the capacity to retain the overall footprint needed by the CLI interpreter. The TAGE+ITTAGE predictor would be able retain this footprint provided that one uses a very large ITTAGE component.

5.5 Folklore on "hard to predict" branches in Interpreters

The indirect branch of the dispatch loop in each interpreter is generally considered as the one that is very hard to predict. Simulation allows us to observe the individual behavior of specific branch instructions. We measured the misprediction ratio of the indirect branch of the dispatch loop in each interpreter. The source code of Python refers to two "hard to predict" branches. The first is the indirect branch that implements the `switch` statement. The second comes for the macro `HAS_ARG` that checks whether an opcode has an argument. We also considered this conditional branch. Table 6 reports the misprediction numbers for these branches for all benchmarks.

For Javascript, the indirect branch appeared as quite easy to predict with misprediction rates generally lower than 1% with the 12 KB ITTAGE, and a single outlier benchmark, the already mentioned `bits-in-byte` at 3.8 %; not that dramatic if one considers that we are predicting complete target with 244 different possibilities!

On Python, `HAS_ARG` turns out to be very easily predicted by the conditional branch predictor TAGE. Indirect jumps are also very well predicted for most benchmarks, even in most cases by the 6 KB ITTAGE. Most of those that are badly predicted with 6 KB ITTAGE are well handled by the 12 KB configuration, see for example the Python `chaos`, `django-v2`, `formatted-log`, `go`.

With the CLI interpreter, the main indirect branch suffer from a rather high misprediction rate when executing `vpr` and `crafty` (and `bzip2` to some extent) with a 12 KB ITTAGE. But a 50 KB ITTAGE predictor strictly reduce this misprediction rate except for crafty which would need an even larger ITTAGE predictor.

Therefore the folklore on the unpredictability of the indirect branch in the dispatch loop is rather unjustified: this indirect branch is very predictable.

6 Other Related Work

This paper is about the interaction of interpreters with branch predictors. The most relevant work on branch prediction is covered by Section 3. This section reviews literature related to other techniques that aim at improving the performance of interpreters.

The overhead of interpreters compared to native code derives mostly from two sources: the management of the evaluation stack and the dispatch loop.

Very recent work by Savrun-Yeniçeri et al. [26] still references the original work of Ertl and Gregg [12]. Their approach, however, is very different: they consider *host-VM targeted* interpreters, i.e. interpreters for languages such as Python or Javascript implemented on top of the Java VM. Performance results are difficult to compare with ours.

Vitale and Abdelrahman [34] eliminate the dispatch overhead with a technique called *catenation*. It consists in copying and combining at run-time sequences of native code produced when the interpreter was compiled. Half the benchmarks, however, run slower, because of the induced code bloat, and the instruction cache behavior degradation. They report the `switch` dispatch to

Python	indirect		ARG	Javascript			CLI			
	IT 1	IT 2	TAGE		IT 1	IT 2		IT 1	IT 2	IT 3
call-meth	0.827 %	0.827 %	0.000 %	cube	0.777 %	2.873 %	164.gzip	0.612 %	2.698 %	0.569 %
call-meth-slots	0.827 %	0.827 %	0.000 %	morph	0.029 %	0.029 %	175.vpr	12.527 %	27.812 %	0.905 %
call-meth-unk	0.626 %	0.626 %	0.000 %	raytrace	1.014 %	8.149 %	177.mesa	0.050 %	1.026 %	0.019 %
call-simple	0.991 %	0.991 %	0.000 %	binary-trees	1.243 %	1.232 %	179.art	0.077 %	0.083 %	0.075 %
chaos	0.165 %	21.362 %	3.456 %	fannkuch	1.063 %	1.051 %	181.mcf	1.020 %	1.994 %	0.681 %
django-v2	0.308 %	22.421 %	0.372 %	nbody	0.452 %	0.452 %	183.equake	0.185 %	0.541 %	0.113 %
fannkuch	0.652 %	0.718 %	0.001 %	nsieve	0.148 %	0.149 %	186.crafty	32.405 %	45.311 %	9.688 %
fastpickle	0.478 %	0.634 %	0.042 %	3bit-bits-in-byte	0.008 %	0.008 %	188.ammp	0.382 %	1.752 %	0.222 %
fastunpickle	0.723 %	3.730 %	0.060 %	bits-in-byte	3.827 %	3.732 %	197.parser	1.881 %	7.939 %	0.786 %
float	0.008 %	0.010 %	0.821 %	bitwise-and	0.000 %	0.000 %	256.bzip2	2.190 %	3.102 %	0.470 %
formatted-log	0.136 %	48.212 %	1.216 %	nsieve-bits	0.146 %	0.146 %				
go	4.407 %	13.521 %	0.980 %	recursive	1.123 %	1.214 %				
hexiom2	1.749 %	4.510 %	1.042 %	aes	0.754 %	1.738 %				
json-dump-v2	0.015 %	0.200 %	0.841 %	md5	0.281 %	0.288 %				
json-load	2.580 %	3.676 %	1.630 %	sha1	0.183 %	0.188 %				
meteor-contest	0.460 %	0.558 %	0.583 %	format-tofte	0.529 %	0.588 %				
nbody	0.002 %	0.003 %	0.758 %	format-xparb	0.482 %	0.612 %				
nqueens	0.518 %	0.526 %	0.357 %	cordic	0.459 %	0.459 %				
pathlib	0.104 %	3.635 %	0.965 %	partial-sums	0.000 %	0.000 %				
pidigits	1.719 %	3.169 %	2.958 %	spectral-norm	0.089 %	0.089 %				
raytrace	0.873 %	6.691 %	3.861 %	dna	0.424 %	2.155 %				
regex-compile	9.852 %	16.284 %	0.589 %	base64	0.293 %	0.293 %				
regex-effbot	1.311 %	1.608 %	0.177 %	fasta	1.210 %	1.233 %				
regex-v8	1.678 %	2.374 %	0.112 %	tagcloud	0.368 %	0.376 %				
richards	0.420 %	6.775 %	1.602 %	unpack-code	0.216 %	0.334 %				
silent-logging	0.316 %	0.321 %	0.004 %	validate-input	0.713 %	0.719 %				
simple-logging	0.030 %	53.552 %	1.197 %							
telco	0.264 %	0.342 %	0.044 %							
unpack-seq	0.027 %	0.029 %	0.001 %							

Table 6: (IT)TAGE misprediction results for “hard” branch, TAGE 1 and TAGE 2

be 12 instructions and 19 cycles on an UltraSparc-III processor, and on average 100 cycles per bytecode for the factorial function written in Tcl.

McCandless and Gregg [18] propose to optimize code at the assembly level to eliminate interference between targets of the indirect branch. Two techniques are developed:forcing alignment, and reordering of entries. The hardware used for experiments is a Core2 processor. We claim that modern branch predictors are quite insensitive to target placement.

6.1 Stack Caching and Registers

With the notable exception of the Dalvik virtual machine [9], most current interpreters perform their computations on an evaluation stack. Values are stored in a data structure which resides in memory (recall Figure 1 for illustration). Ertl proposed *stack caching* [10] to force the top of the stack into registers. Together with Gregg, they later proposed to combine it with dynamic superinstructions [13] for additional performance. Stack caching is orthogonal to the behavior of the branch predictor. While it could decrease the number of cycles of the interpreter loop, and hence increase the relative impact of a misprediction, this data will typically hit in the L1 cache and aggressive out-of-order architectures are less likely to benefit, especially for rather long loops, and the already reasonably good performance (IPC). Register allocators have a hard time keeping the relevant values in registers [5] because of the size and complexity of the main interpreter loop. Stack caching also adds significant complexity to the code base.

As an alternative to stack caching, some virtual machines are register-based. Shi et al. [32] show that they are more efficient when sophisticated translation and optimizations are applied. This is also orthogonal to the implementation of the dispatch loop.

6.2 Superinstructions and Replication

Sequences of bycodes are not random. Some pairs of instructions are more frequent than others (e.g. a compare is often followed by a branch). Superinstructions [23] consist in such sequences of frequently occurring tuples of bytecode. New opcodes are defined, whose payloads are the combination of the payloads of the tuple they come from. The overhead of the dispatch loop is unmodified but the gain comes from a reduced number of iterations of the loop, hence a reduced average cost. Ertl and Gregg [12] discuss static and dynamic superinstructions.

Replication, also proposed by Ertl and Gregg, consists in generating many opcodes for the same payload, specializing each occurrence, in order to maximize the performance of branch target buffers. Modern predictors no longer need it to capture patterns in applications.

6.3 Jump Threading

We discussed jump threading in general terms in previous sections. To be more precise, several versions of threading have been proposed: token threading (illustrated in Figure 2), direct threading [1], inline threading [23], or context threading [2]. All forms of threading require extensions to ANSI C. Some also require limited forms of dynamic code generation and walk away from portability and ease of development.

[5] Even with the help of the GNU extension *register asm*, or manually allocating a few top-of-stack elements in local variable.

7 Conclusion

Despite mature JIT compilation technology, interpreters are very much alive. They provide ease of development and portability. Unfortunately, this is at the expense of performance: interpreters are slow. Many studies have investigated ways to improve interpreters, and many design points have been proposed. Because many studies go back when branch predictors were not very aggressive, folklore has retained that a high misprediction rate of an indirect jump is one of the main reasons for the inefficiency of `switch`-based interpreters. In this paper, we shed new light on this claim, considering current interpreters and state-of-the-art branch predictors. We show that the predictor of Sandy Bridge improves over the previous generation Nehalem, and that literature state-of-the-art ITTAGE would provide yet another significant improvement. ITTAGE performs specially well when the flow of execution stays in the main interpretation loop, i.e. when bytecode payloads remains simple. Finally we confirmed that the few cases where ITTAGE performs poorly are due to footprint issues, and not inherent to the prediction scheme.

References

[1] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6), 1973.

[2] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *CGO*, pages 15–26, 2005.

[3] S. Brunthaler. Virtual-machine abstraction and optimization techniques. *Electronic Notes in Theoretical Computer Science*, 253(5):3–14, 2009.

[4] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *ISCA*, pages 274–283, 1997.

[5] R. Costa, A. C. Ornstein, and E. Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, July 2007.

[6] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *Micro, IEEE*, 17(3):36–43, May/Jun.

[7] J. W. Davidson and J. V. Gresh. Cint: a RISC interpreter for the C programming language. In *Papers of the Symposium on Interpreters and interpretive techniques*, SIGPLAN '87, pages 189–198. ACM, 1987.

[8] K. Driesen and U. Hölzle. Accurate indirect branch prediction. In *ISCA*, pages 167–178, 1998.

[9] D. Ehringer. The Dalvik virtual machine architecture. Retrieved from `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf` on Nov 5 2013, 2010.

[10] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 315–327. ACM, 1995.

[11] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par 2001 Parallel Processing*, pages 403–413. Springer, 2001.

[12] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *PLDI*, pages 278–288. ACM, 2003.

[13] M. A. Ertl and D. Gregg. Combining stack caching with dynamic superinstructions. In *Workshop on Interpreters, Virtual Machines and Emulators*, pages 7–14. ACM, 2004.

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2013.

[15] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 197–206. IEEE, 2001.

[16] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *MICRO*, pages 4–13, 1997.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM, 2005.

[18] J. McCandless and D. Gregg. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM TACO*, 8(4):24:1–24:20, Jan. 2012.

[19] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, 1993.

[20] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. *SIGARCH Comput. Archit. News*, 25(2):292–303, May 1997.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276. ACM, 2009.

[22] A. Naumann and P. Canal. The role of interpreters in high performance computing. In *Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research*, page 65, 2008.

[23] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5):291–300, May 1998.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.

[25] E. Rohou. Tiptop: Hardware Performance Counters for the Masses. Technical Report RR-7789, INRIA, Nov. 2011.

[26] G. Savrun-Yeniçeri, W. Zhang, H. Zhang, C. Li, S. Brunthaler, P. Larsen, and M. Franz. Efficient interpreter optimizations for the JVM. In *International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 113–123. ACM, 2013.

[27] A. Seznec. Analysis of the O-GEometric History Length Branch Predictor. In *ISCA*, pages 394–405, 2005.

[28] A. Seznec. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*. JILP, June 2011.

[29] A. Seznec. A new case for the TAGE branch predictor. In *MICRO*, pages 117–127, 2011.

[30] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *ISCA*, pages 295–306. IEEE, 2002.

[31] A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism*, 8:1–23, 2006.

[32] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM TACO*, 4(4):2:1–2:36, Jan. 2008.

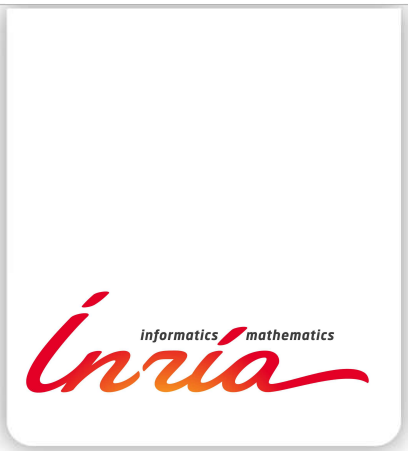
[33] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *ISCA*, pages 284–291, 1997.

[34] B. Vitale and T. S. Abdelrahman. Catenation and specialization for tcl virtual machine performance. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, IVME '04, pages 42–50. ACM, 2004.

[35] V. M. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *3rd Workshop on Functionality of Hardware Performance Monitoring*, Dec. 2010.

[36] E. Wieprecht, J. Brumfit, J. Bakker, N. de Candussio, S. Guest, R. Huygen, A. de Jonge, J. J. Matthiew, S. Osterhage, S. Ott, H. Siddiqui, B. Vandenbussche, W. de Meester, M. Wetzstein, E. Wiezorrek, and P. Zaal. The HERSCHEL/PACS Common Software System as Data Reduction System. *Astronomical Data Analysis Software and Systems (ADASS) XIII*, July 2004.

[37] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO*, pages 51–61. ACM, 1991.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399