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

Inria, France

first.last@inria.fr

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. Folklore attributes a significant penalty to this branch, due to its high misprediction rate. We revisit this assumption, considering state-of-the-art branch predictors and the three most recent Intel processor generations on current interpreters. Using both hardware counters on Haswell, the latest Intel processor generation, and simulation of the ITTAGE, we show that the accuracy of indirect branch prediction is no longer critical for interpreters. We further compare the characteristics of these interpreters and analyze why the indirect branch is less important than before.

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, though, is a complicated task. Conversely, interpreters provide ease of implementation, and portability, at the expense of 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. These dynamic features turn out to be heavily used in real applications [25]. On lower-end systems, where short time-to-market is key, JIT compilers may also not be commercially viable, and they rely on interpreters.

Scientists from both CERN and Fermilab report [23] 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 (see Section 6 for related work).

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 the most recent Intel processor generations(Nehalem, Sandy Bridge and Haswell) and current interpreted languages (Python, Javascript, CLI). Our experiments and measures show that on the latest processor generation, the performance of the predictors and the characteristics of interpreters make the indirect branch much less critical than before. The global branch misprediction rate observed when executing interpreters drops from a dramatic 12-20 MPKI range on Nehalem to a only 0.5-2 MPKI range on Haswell.
- We evaluate the performance of a state-of-the-art indirect branch predictor, ITTAGE [31], proposed in the literature on the same interpreters, and we show that, when executing interpreters, the branch prediction accuracy observed on Haswell and on ITTAGE are in the same range.

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. Section 7 concludes.

```
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 }
```

(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

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. 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 (figure out that it is e.g. an `add`) and jump to the chunk of code that implements its semantics, i.e. the *payload* (the code that adds of the appropriate values).

2. Performance of Interpreters

Quoting Cramer et al. [6], *Interpreting bytecodes is slow*. 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 (Dalvik is a notable exception [9]). Lines 5–7 illustrate an `add`

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 5.2.3). 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 [11, 12, 19], 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 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, inline assembly can be used, at the expense of portability.

```
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

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."

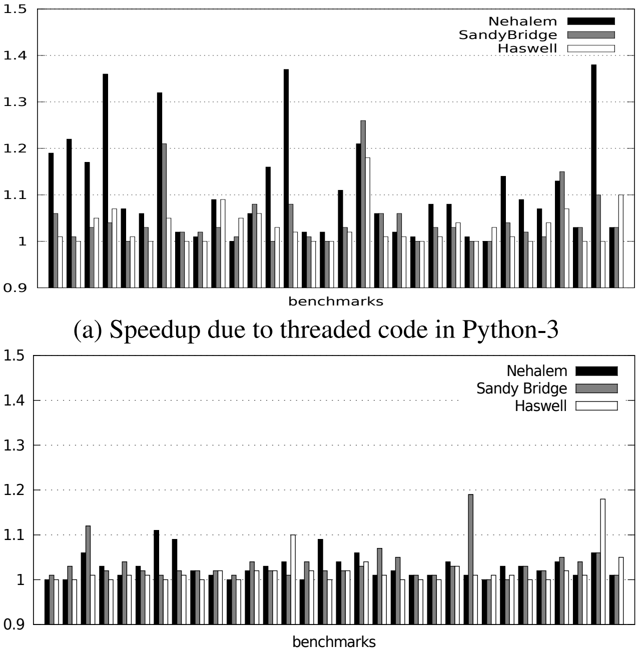
We tracked this comment back to January 2009.

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`). Figure 3 (a) shows the performance improvement due to threaded code on three microarchitectures: Nehalem, Sandy Bridge, and Haswell.

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 Nehalem, Sandy Bridge, and Haswell are respectively 10.1 %, 4.2 %, and 2.8 % with a few outstanding values for each microarchitecture. The benefits of threaded code decreases with each new generation of microarchitecture.

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 hard-coded in the dispatch loop. The speedups are reported in Figure 3 (b). Respective averages are 2.8 %, 3.2 % and 1.8 % for Nehalem, Sandy Bridge, and Haswell.

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 [24] threaded code also executes fewer instructions because part of the dispatch loop is bypassed. For example, on Python-3, we measured the average reduction in number of instructions to be on average 3.3 %, in the same range as the performance gain on the three tested architectures.



(a) Speedup due to threaded code in Python-3

(b) Speedup due to opcode prediction in Python-2

Figure 3. Speedups in Python

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 [20], de-aliased predictors [16, 21, 33], multiple history length use [30], and more recently perceptron-inspired predictors [15] and geometric history length predictors [28, 31]. All these propositions have influenced the design of the predictors embedded in 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 [29, 31] 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 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.

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 opcodes. For instance, if this history is –load load add load mul store add– and if this sequence is unique, then the next 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.

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. Branches bytecodes 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 branch bytecodes.

4. Experimental Setup

This section details our interpreters and benchmarks. We discuss how we collect data for actual hardware and simulation, and we make sure that both approaches are consistent.

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 (CLI, aka .NET), and several inputs for each interpreter. Javascript benchmarks consist in Google's *octane* suite[3] as of Feb 2014, and Mozilla's *kraken*[4]. For Python, we used the *Unladen Swallow Benchmarks*. Finally, we used a subset of *SPEC 2000* (*train* input set) for CLI. All benchmarks are run to completion (including hundreds of hours of CPU for the simulation). See Table 1 for an exhaustive list.

We used Python 3.3.2. The motivation example of Section 2 also uses Python 2.7.5. Unladen Swallow benchmarks were run with the flag `--rigorous`. We excluded `iterative_count`, `spectral_norm` and `threaded_count`

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

[3] http://code.google.com/p/octane-benchmark

[4] http://krakenbenchmark.mozilla.org/kraken-1.1/

Table 1. Benchmarks

Python
call_method
call_method_slots
call_method_unknown
call_simple
chaos
django_v2
fannkuch
fastpickle
fastunpickle
float
formatted_logging
go
hexiom2
json_dump_v2
json_load
meteor_contest
nbody
nqueens
pathlib
pidigits
raytrace
regex_compile
regex_effbot
regex_v8
richards
silent_logging
simple_logging
telco
unpack_sequence

Javascript (kraken)
ai-astar
audio-beat-detection
audio-dft
audio-fft
audio-oscillator
imaging-darkroom
imaging-desaturate
imaging-gaussian-blur
json-parse-financial
json-stringify-tinderbox
crypto-aes
crypto-ccm
crypto-pbkdf2
crypto-sha256-iterative

Javascript (octane)
box2d
code-load
crypto
deltablue
earley-boyer
gbemu
mandreel
navier-stokes
pdf
raytrace
regexp
richards
splay
typescript
zlib

CLI
164.gzip
175.vpr
177.mesa
179.art
181.mcf
183.equake
186.crafty
188.ammp
197.parser
256.bzip2

from the suite because they were not properly handled by our measurement setup.

Javascript experiments rely on *SpiderMonkey 1.8.5*

We used GCC4CLI [5] to compile the SPEC 2000 benchmarks. It is a port of GCC that generates CLI from C. 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.

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.

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 [19] reported this phenomenon and developed a technique that modifies the alignment of individual case entries to improve the overall performance. We manually changed the alignment of the 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.

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

4.2.1 Existing Hardware – Performance Counters

Branch prediction data is collected from the PMU (performance monitoring unit) on actual Nehalem (Xeon W3550 3.07 GHz), Sandy Bridge (Core i7-2620M 2.70 GHz), and Haswell (Core i7-4770 3.40 GHz) architectures running Linux. Both provide counters for cycles, retired instructions, retired branch instructions, and mispredicted branch instructions. We relied on Tiptop [26] to collect 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, but for "speculative and retired indirect branches" [14]. 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: $n_{bytecodes} \leq n_{retired} \leq n_{speculative}$ 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]. The performance is provided through simulation of traces produced by Pin [17]. We used two (TAGE+ITTAGE) configurations. Both have 8 KB TAGE. TAGE1 assumes a 12.62 KB ITTAGE, TAGE2 assumes a 6.31 KB ITTAGE (see Table 2).

4.2.3 Coherence of Measurements

Our experiments involve different tools and methodologies, namely the PMU collected by the Tiptop tool [26] on existing hardware, as well as results of simulations driven by traces obtained using Pin [17]. This section is about confirming that these tools lead to comparable instruction counts, therefore experiment numbers are comparable. Potential discrepancies include the following:

- non determinism inherent to the PMU [35] or the system/software stack [22];
- the x86 instruction set provides a `rep` prefix. The PMU counts prefixed instructions as one (made of many microops), while pintools may count each 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 our running times;
- 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 only traces user mode events, we configured the PMU correspondingly. To quantify the impact of 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 jumps, it is even below 0.05 percentage point.

The main loop of interpreters is identical on all architectures, 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 counts within 1 %.

5. Experimental Results

5.1 Overview

Figures 4, 5 and 6 illustrate the branch misprediction rates measured in our experiments on respectively Python, Javascript and CLI interpreters. The branch misprediction rates are measured in MPKI, misprediction per kilo instructions. MPKI is generally considered as a quite illustrative metric for branch predictors, since it allows to get at a first glance a very rough estimation of the lost cycles per kiloinstructions: $\frac{lost\_cycles}{KI} = MPKI \times average\_penalty.$

For the sake simplicity and for a rough analysis, we will assume on the considered architectures an average penalty of 20 cycles. Our measures clearly show that, on Nehalem, on most benchmarks, 12 to 16 MPKI are encountered, that is about 240 to 320 cycles are lost every 1 kiloinstructions. On the next processor generation Sandy Bridge, the misprediction rate is much lower: generally about 4 to 8 MPKI on Javascript applications for instance, i.e. decreasing the global penalty to 80 to 160 cycles every Kiloinstructions. On the most recent processor generation Haswell, the misprediction rate further drops to 0.5 to 2 MPKI in most cases, that is a loss of 10 to 40 cycles every kiloinstructions. Interestingly, the misprediction rates simulated assuming at TAGE + ITTAGE branch predictor scheme are in the same range as the misprediction rates measured on Haswell. This rough analysis illustrates that, in the execution time of interpreted applications, total branch misprediction penalty has gone from a major component on Nehalem to only a small fraction on Haswell.

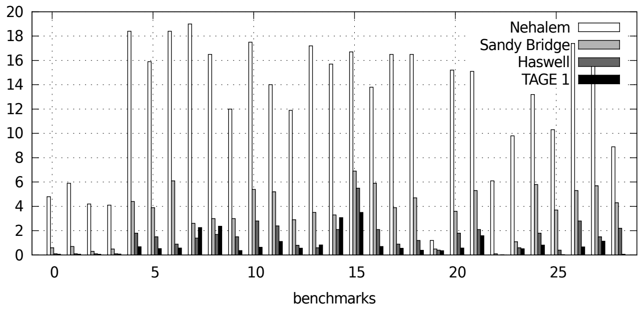


Figure 4. Python MPKI for all predictors

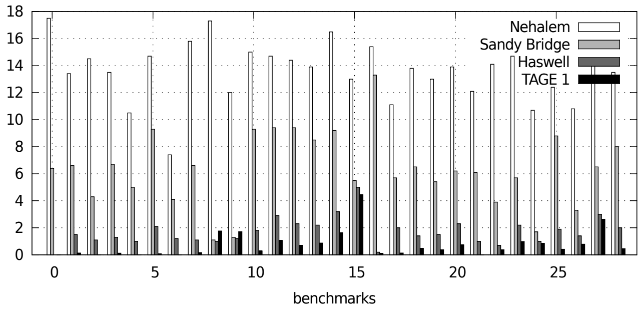


Figure 5. Javascript MPKI for all predictors

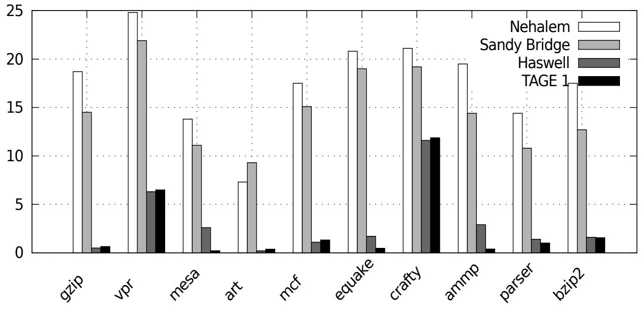


Figure 6. CLI MPKI for all predictors

The rough analysis presented above can be refined with models using performance monitoring counters. Intel [18] describes such methodology to compute wasted instruction issue slots in the processor front-end. We relied on Andi Kleen's implementation `pmu-tools`[5] and backported the formulas. Unfortunately, only Sandy Bridge and Haswell are supported. In the front-end, issue slots can be wasted in several cases: branch misprediction, but also memory ordering violations, self modifying code (SMC), and AVX-

[5] `https://github.com/andikleen/pmu-tools`

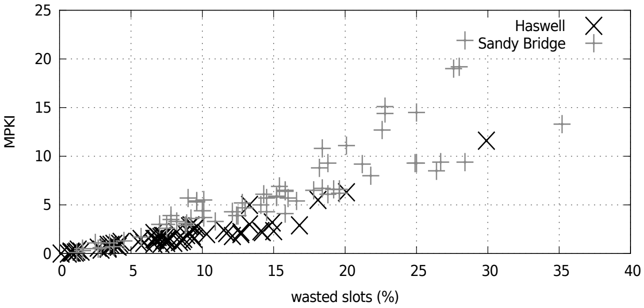


Figure 7. Correlation between MPKI and lost slots

related events. We confirmed that AVX never occurs and SMC is negligible in our experiments. Moreover, apart a very few cases, the number of memory ordering violations is two to three orders of magnitude less than the number of branch mispredictions. Nearly all wasted issue slots can be attributed to branch mispredictions.

Figure 7 shows, for all our benchmarks, how MPKI relates to wasted slots. Lower MPKI correlates with fewer wasted slots. On average Haswell wastes 7.8 % of the instruction slots due to branch mispredictions, while Sandy Bridge wastes 14.5 %. Better branch prediction on Haswell results in 50 % fewer wasted slots.

This confirms that, on last generation Intel processors, Haswell, branch misprediction penalty has only a limited impact on interpreted applications execution time.

In the remainder of this section, we present a detailed analysis of the measures and simulations for each interpreter separately. We also debunk the folklore on the "hard to predict" branches in the interpreter loop.

5.2 Detailed Analysis

We present results for each interpreter (Tables 3, 4, 5). The left part reports general characteristics of the interpreter. 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 executed 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 (MPKI) for each benchmark on Nehalem, Sandy Bridge, Haswell, TAGE1 and TAGE2.

Generally all three interpreters run at quite high performance with median values of 1.5, 1.4, 1.2 IPC for respectively Python, Javascript and CLI on Nehalem, 1.7, 1.5 and 1.2 on Sandy Bridge, and 2.4, 2.4 and 2.2 on Haswell.

Our experiments clearly show that between three recent generations of Intel processors, the improvement on the branch prediction accuracy on interpreters is dramatic for Python and Javascript. Our simulations of 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 can be even become insignificant (less than 0.5 MPKI). For CLI, the results on Nehalem and Sandy Bridge are much more mitigated than Python and Javascript, with Sandy Bridge only reducing the misprediction rate by at most 25 % and often much less, e.g. on `art`, it even loses accuracy. Haswell, however, predicts much better, achieving results close to TAGE+ITTAGE.

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

5.2.1 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 opcodes.

The dispatch loop is identical on Nehalem, Sandy Bridge and Haswell (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, 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 (measured as IPC) on Nehalem and Sandy Bridge is fairly good, showing that no serious problem (such as cache misses or branch misprediction) is degrading performance. It is even better on Haswell, with a median value of 2.4 IPC, and up to 3.4 IPC.

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.

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 4 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 Haswell and 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., $\frac{ind}{bc} \leq 1.02$, 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 Haswell.

5.2.2 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.

Table 4 reports the characteristics of the Javascript interpreter. With the exception of `code-load` in octane, and `parse-financial` and `stringify-tinderbox` in kraken, indirect branches come from the `switch` statement. These benchmarks also have a outstanding number of instructions per bytecode. Excluding them, 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 5 illustrates the respective MPKI. As for Python, Haswell and TAGE consistently outperform Sandy Bridge, which also outperforms Nehalem.

As for the Python interpreters, with the exception of three outliers, TAGE predict quasi perfectly the interpreter.

5.2.3 CLI

The CLI interpreter is written in standard C, hence dispatch is implemented with a `switch` statement. The internal IR consists in 478 opcodes. 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 opcodes. 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 8. 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 the code is interpreted, even libraries such as libc and libm. Execution goes to native code at a cutpoint 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.

```
 1  loop:
 2        cmp     0x1de,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 8. Dispatch loop of the CLI interpreter (+ADD)

The compiler could not achieve stack caching [6], probably due to the limited number of registers. Figure 8 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. Hawell's predictor is comparable to TAGE, with occasional wins for TAGE (2.6 vs 0.21 MPKI on 177.mesa) and for Haswell (0.2 vs 0.38 on 179.art).

However, with the CLI interpreter, the accuracy of TAGE +ITTAGE is particularly poor on `vpr` and `crafty`, 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.

5.3 Folklore on "hard to predict" branches

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. Moreover 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. For Python, we also considered this conditional

[6] despite a number of attempts...

Table 3. Python characteristics and branch prediction for Nehalem (Neh.), Sandy Bridge (SB), Haswell (Has.) and TAGE

benchmark	Mbc	Gins	IPC			ins/bc	br	ind	ind/bc	MPKI				
			Neh.	SB	Has.					Neh.	SB	Has.	TAGE 1	TAGE 2
call_method	6137	771	1.84	2.2	2.93	125.6	20 %	0.8 %	1.01	4.8	0.6	0.1	0.067	0.067
call_method_slots	6137	766	1.87	2.18	2.90	124.8	20 %	0.8 %	1.02	5.9	0.7	0.1	0.068	0.068
call_method_unknown	7300	803	1.99	2.22	2.88	110.0	20 %	0.9 %	1.00	4.2	0.3	0.1	0.058	0.058
call_simple	5123	613	1.89	2.33	3.12	119.6	19 %	0.8 %	1.00	4.1	0.5	0.1	0.086	0.086
chaos	1196	162	1.34	1.55	2.21	135.4	22 %	0.8 %	1.07	18.4	4.4	1.8	0.680	2.548
django_v2	1451	332	1.22	1.44	2.10	228.7	21 %	0.6 %	1.33	15.9	3.9	1.5	0.529	1.829
fannkuch	7693	747	1.52	1.67	2.44	97.1	23 %	1.3 %	1.23	18.4	6.1	0.9	0.578	0.592
fastpickle	34	351	1.62	1.89	2.63	10277	22 %	0.5 %	54	19	2.6	1.4	2.258	2.290
fastunpickle	25	278	1.47	1.77	2.06	11320	22 %	0.8 %	91	16.5	3	1.7	2.365	2.673
float	1280	180	1.67	1.73	2.27	140.6	22 %	0.8 %	1.15	12	3	1.5	0.364	0.365
formatted_logging	750	125	0.89	1.13	1.74	166.9	22 %	0.7 %	1.20	17.5	5.4	2.8	0.633	4.220
go	2972	344	1.43	1.6	2.20	115.7	21 %	0.9 %	1.01	14	5.2	2.4	1.121	1.979
hexiom2	33350	3674	1.72	1.86	2.52	110.2	21 %	1.0 %	1.08	11.9	2.9	0.8	0.563	0.832
json_dump_v2	5042	1656	1.48	1.62	2.44	328.5	23 %	0.8 %	2.61	17.2	3.5	0.6	0.827	0.859
json_load	195	271	1.57	1.81	2.50	1391	26 %	0.5 %	6.76	15.7	3.3	2.1	3.074	3.198
meteor_contest	868	106	1.36	1.52	1.89	122.3	22 %	0.9 %	1.05	16.7	6.9	5.5	3.507	3.519
nbody	2703	184	1.78	1.73	2.45	68.2	23 %	1.5 %	1.00	13.8	5.9	2.1	0.700	0.701
nqueens	1296	152	1.34	1.46	2.35	117.0	22 %	0.9 %	1.06	16.5	3.9	0.9	0.549	0.549
pathlib	836	188	0.79	1.08	1.88	225.3	22 %	0.6 %	1.45	16.5	4.7	1.2	0.397	0.633
pidigits	94	223	2.33	2.37	2.76	2366	7 %	0.1 %	1.42	1.2	0.5	0.4	0.356	0.363
raytrace	4662	766	1.40	1.66	2.21	164.3	21 %	0.7 %	1.12	15.2	3.6	1.8	0.577	1.017
regex_compile	1938	214	1.33	1.52	2.17	110.6	21 %	1.0 %	1.05	15.1	5.3	2.1	1.588	2.257
regex_effbot	7	52	2.45	2.53	3.35	6977	22 %	2.1 %	146	6.1	0.1	0.0	0.026	0.027
regex_v8	31	37	1.79	1.79	2.39	1182	22 %	1.7 %	21	9.8	1.1	0.6	0.506	0.534
richards	961	108	1.39	1.59	2.27	111.9	21 %	0.9 %	1.02	13.2	5.8	1.8	0.824	1.518
silent_logging	353	53	1.75	1.83	2.59	148.8	21 %	0.7 %	1.08	10.3	3.7	0.4	0.035	0.035
simple_logging	731	120	0.93	1.14	1.78	164.4	22 %	0.7 %	1.19	17.4	5.3	2.8	0.669	4.748
telco	34	6	1.28	1.28	2.24	174.6	21 %	1.3 %	2.35	15.6	5.7	1.5	1.143	1.150
unpack_seq	966	38	1.90	2.04	2.86	39.5	21 %	2.6 %	1.01	8.9	4.3	2.2	0.056	0.057
average										12.8	3.5	1.4	0.8	1.3

branch. Table 6 reports the misprediction numbers for these branches for all benchmarks for the three sizes of ITTAGE predictors, 6 KB, 12 KB and 50 KB.

On Python, the indirect jumps are most often very well predicted for most benchmarks, even by the 6 KB ITTAGE. However, in several cases the prediction of indirect jump is poor (see for example the Python `chaos`, `django-v2`, `formatted-log`, `go`). However, these cases except go are in practice near perfectly predicted by the 12 KB configuration: that is in practice the footprint of the Python application on the indirect jump predictor is too large for the 6 KB configuration, but fits the 12 KB configuration. go needs an even larger predictor as illustrated by the results on the 50 KB configuration. `HAS_ARG` turns out to be very easily predicted by the conditional branch predictor TAGE at the exceptions of the same few outliers with 1% to 4% mispredictions.

For Javascript, the indirect branch also appears as quite easy to predict with misprediction rates generally lower than 1% with the 12 KB ITTAGE. More outliers than for Python are encountered, particularly `code-load` and `type_script`. However these outliers are all amenable to low misprediction rates with a 50 KB predictor at the exception of `code-load`. However, `code-load` executes more than 4,000 instructions per bytecode on average (see Table 4) and therefore the predictability of the indirect jump in the interpreter dispatch loop has a very limited impact on the overall performance.

With the CLI interpreter, the main indirect branch suffers 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 reduces 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 provided the usage of a large enough efficient indirect jump predictor.

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. 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. [27] 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 bench-

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

benchmark	Mbc	Gins	IPC			ins/bc	br	ind	ind/bc	MPKI				
			Neh.	SB	Has.					Neh.	SB	Has.	T1	T2
kraken														
ai-astar	5713	296	1.55	1.55	2.36	51.8	20 %	1.9 %	1.002	17.5	6.4	0.0	0.01	0.01
audio-beat-detection	4567	195	1.42	1.59	2.56	42.7	19 %	2.3 %	1.002	13.4	6.6	1.5	0.14	0.16
audio-dft	3311	169	1.62	1.58	2.57	51.0	20 %	2.0 %	1.004	14.5	4.3	1.1	0.01	0.01
audio-fft	4459	189	1.42	1.59	2.65	42.4	19 %	2.4 %	1.002	13.5	6.7	1.3	0.12	0.12
audio-oscillator	2541	162	1.69	1.61	2.61	63.8	21 %	1.6 %	1.033	10.5	5.0	1.0	0.01	0.01
imaging-darkroom	4387	234	1.40	1.33	2.31	53.3	20 %	1.9 %	1.022	14.7	9.3	2.1	0.07	0.08
imaging-desaturate	8117	368	1.71	1.73	2.72	45.3	19 %	2.2 %	1.007	7.4	4.1	1.2	0.01	0.01
imaging-gaussian-blur	24490	1278	1.60	1.59	2.70	52.2	19 %	1.9 %	1.060	15.8	6.6	1.1	0.17	0.17
json-parse-financial	0.12	6	1.77	2.15	2.53	50000	21 %	1.2 %	569	17.3	1.1	1.0	1.76	1.77
json-stringify-tinderbox	0.30	4	2.09	2.27	2.84	13333	24 %	0.2 %	23.7	12.0	1.3	1.2	1.71	1.71
crypto-aes	1679	68	1.41	1.40	2.51	40.5	17 %	2.5 %	1.008	15.0	9.3	1.8	0.29	2.13
crypto-ccm	1034	43	1.40	1.39	2.31	41.6	17 %	2.4 %	1.016	14.7	9.4	2.9	1.07	1.62
crypto-pbkdf2	3592	139	1.29	1.34	2.42	38.7	16 %	2.6 %	1.006	14.4	9.4	2.3	0.71	1.24
crypto-sha256-iterative	1136	45	1.36	1.41	2.54	39.6	16 %	2.5 %	1.011	13.9	8.5	2.2	0.87	1.01
octane														
box2d	1958	131	1.26	1.30	2.01	66.9	20 %	1.6 %	1.016	16.5	9.2	3.2	1.64	2.46
code-load	2.8	12	1.31	1.33	1.70	4286	21 %	0.3 %	14.3	13.0	5.5	5.0	4.44	4.53
crypto	39480	1543	1.34	1.20	2.91	39.1	17 %	2.6 %	1.001	15.4	13.3	0.2	0.11	0.2
deltablue	9443	824	1.43	1.46	2.13	87.3	21 %	1.1 %	1.012	11.1	5.7	2.0	0.13	0.44
earley-boyer	12113	819	1.38	1.46	2.19	67.6	20 %	1.6 %	1.072	13.8	6.5	1.4	0.48	1.15
gbemu	1928	122	1.60	1.65	2.23	63.3	20 %	1.7 %	1.092	13.0	5.4	1.5	0.37	0.53
mandreel	3582	165	1.60	1.57	2.36	46.1	18 %	2.4 %	1.090	13.9	6.2	2.3	0.74	1.29
navier-stokes	10586	563	1.61	1.54	2.77	53.2	20 %	1.9 %	1.037	12.1	6.1	1.0	0.01	0.01
pdf	977	54	1.67	1.71	2.66	55.3	19 %	2.0 %	1.074	14.1	3.9	0.7	0.37	0.54
raytrace	4984	482	1.35	1.34	1.99	96.7	21 %	1.1 %	1.100	14.7	5.7	2.2	0.99	2.47
regexp	595	75	1.67	1.78	2.34	126.1	21 %	1.2 %	1.561	10.7	1.7	1.0	0.85	0.9
richards	12523	1029	1.38	1.25	2.26	82.2	21 %	1.3 %	1.000	12.4	8.8	1.9	0.42	0.71
splay	783	83	1.50	1.57	2.09	106.0	19 %	1.1 %	1.152	10.8	3.3	1.4	0.79	1.13
typescript	1263	120	1.26	1.31	1.85	95.0	20 %	1.1 %	1.079	14.2	6.5	3.0	2.64	3.86
zlib	41051	2106	1.47	1.45	2.47	51.3	18 %	2.2 %	1.138	13.5	8.0	2.0	0.46	1.40
average										13.6	6.3	1.7	0.7	1.1

Table 5. CLI characteristics and branch prediction for Nehalem (Neh.), Sandy Bridge (SB), Haswell (Has.) and TAGE

benchmark	Gbc	Gins	IPC			ins/bc	br	ind	ind/bc	MPKI					
			Neh.	SB	Has.					Neh.	SB	Has.	T1	T2	T3
164.gzip	78.8	1667.2	1.20	1.23	2.55	21.2	23 %	4.8 %	1.02	18.7	14.5	0.5	0.64	1.58	0.62
175.vpr	18.4	400.9	0.97	1	1.80	21.8	23 %	4.7 %	1.03	24.8	21.9	6.3	6.49	13.62	1.08
177.mesa	118.6	3177.1	1.32	1.3	2.13	26.8	23 %	4.0 %	1.07	13.8	11.1	2.6	0.21	0.59	0.20
179.art	4.7	58.5	1.64	1.49	2.70	12.5	26 %	8.0 %	1.00	7.3	9.3	0.2	0.38	0.38	0.37
181.mcf	13.3	181.2	1.13	1.19	2.19	13.7	25 %	7.4 %	1.01	17.5	15.1	1.1	1.33	2.09	1.08
183.equake	40.5	726.1	1.09	1.09	2.34	17.9	24 %	5.7 %	1.02	20.8	19	1.7	0.47	0.68	0.43
186.crafty	35.8	1047.3	1.02	1.03	1.42	29.2	22 %	3.5 %	1.04	21.1	19.2	11.6	11.87	16.31	4.01
188.ammp	91.3	1665.9	1.15	1.24	2.18	18.3	24 %	5.7 %	1.04	19.5	14.4	2.9	0.39	1.14	0.30
197.parser	12.6	447.5	1.19	1.24	2.18	35.4	22 %	3.0 %	1.06	14.4	10.8	1.4	1.01	2.75	0.70
256.bzip2	28.3	460.8	1.16	1.32	2.29	16.3	24 %	6.2 %	1.02	17.5	12.7	1.6	1.55	2.15	0.44
average										17.5	14.8	3.0	2.4	4.1	0.9

marks, however, run slower, because of the induced code bloat, and the instruction cache behavior degradation. They report the `switch` dispatch to 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 [19] propose to optimize code at the assembly level to eliminate interferences between targets of the indirect branch. Two techniques are developed: forcing alignment, and reordering 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

Table 6. (IT)TAGE misprediction results for “hard to predict” branch, TAGE 1, TAGE 2 and TAGE 3 (all numbers in %)

Python		indirect		ARG	Javascript				CLI			
	IT 1	IT 2	IT3	TAGE		IT 1	IT 2	IT3		IT 1	IT 2	IT 3
call-meth	0.827	0.827	0.827	0.000	ai-astar	0.012	0.012	0.012	164.gzip	0.612	2.698	0.569
call-meth-slots	0.827	0.827	0.827	0.000	a-beat-detec.	0.064	0.130	0.063	175.vpr	12.527	27.812	0.905
call-meth-unk	0.626	0.626	0.626	0.000	audio-dft	0.003	0.003	0.003	177.mesa	0.050	1.026	0.019
call-simple	0.991	0.991	0.990	0.000	audio-fft	0.064	0.064	0.064	179.art	0.077	0.083	0.075
chaos	0.165	21.362	0.163	3.456	a-oscillator	0.001	0.001	0.001	181.mcf	1.020	1.994	0.681
django-v2	0.308	22.421	0.016	0.372	i-darkroom	0.023	0.083	0.023	183.equake	0.185	0.541	0.113
fannkuch	0.652	0.718	0.630	0.001	i-desaturate	0.000	0.000	0.000	186.crafty	32.405	45.311	9.688
fastpickle	0.478	0.634	0.397	0.042	i-gaussian-blur	0.062	0.062	0.062	188.ammp	0.382	1.752	0.222
fastunpickle	0.723	3.730	0.547	0.060	j-parse-financial	0.317	0.572	0.163	197.parser	1.881	7.939	0.786
float	0.008	0.010	0.005	0.821	j-s-tinderbox	2.460	2.753	2.043	256.bzip2	2.190	3.102	0.470
formatted-log	0.136	48.212	0.021	1.216	c-aes	0.323	7.725	0.288				
go	4.407	13.521	1.728	0.980	c-ccm	3.429	5.666	0.252				
hexiom2	1.749	4.510	0.571	1.042	c-pbkdf2	0.100	2.039	0.094				
json-dump-v2	0.015	0.200	0.001	0.841	c-sha256-it.	0.305	0.827	0.085				
json-load	2.580	3.676	0.057	1.630	box2d	7.362	12.618	0.936				
meteor-contest	0.460	0.558	0.281	0.583	code-load	31.662	36.134	24.257				
nbody	0.002	0.003	0.002	0.758	crypto	0.166	0.494	0.108				
nqueens	0.518	0.526	0.515	0.357	deltablue	0.380	3.042	0.032				
pathlib	0.104	3.635	0.027	0.965	earley-boyer	0.902	5.264	0.748				
pidigits	1.719	3.169	0.719	2.958	gbemu	1.690	2.688	0.569				
raytrace	0.873	6.691	0.228	3.861	mandreel	2.356	4.742	0.366				
regex-compile	9.852	16.284	0.567	0.589	navier-stokes	0.024	0.029	0.022				
regex-effbot	1.311	1.608	0.848	0.177	pdf	0.506	1.424	0.351				
regex-v8	1.678	2.374	0.918	0.112	raytrace	6.100	20.337	0.439				
richards	0.420	6.775	0.337	1.602	regexp	0.427	0.813	0.324				
silent-logging	0.316	0.321	0.308	0.004	richards	0.544	2.769	0.474				
simple-logging	0.030	53.552	0.021	1.197	splay	1.016	4.509	0.895				
telco	0.264	0.342	0.172	0.044	typescript	18.291	29.630	4.100				
unpack-sequence	0.027	0.029	0.025	0.001	zlib	2.079	6.771	0.228				

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 [7] 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 they are more efficient when sophisticated translation and optimizations are applied. This is orthogonal to the dispatch loop.

6.2 Superinstructions and Replication

Sequences of bycodes are not random. Some pairs are more frequent than others (e.g. a compare is often followed by a branch). Superinstructions [24] consist in such sequences of frequently occurring tuples of bytecode. New bytecodes are defined, whose payloads are the combination of the payloads of the tuples. 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 discuss 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 [24], 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.

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. But many studies go back when branch predictors were not very aggressive, folklore has retained that a highly mispredicted 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 accuracy of branch prediction on interpreters has been dramatically improved over the three last Intel processor generations. This accuracy on Haswell, the most recent Intel processor generation, has reached a level

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

where it can not be considered as an obstacle for performance anymore. We have also shown that this accuracy is on par with the one of the literature state-of-the-art ITTAGE. While the structure of the Haswell indirect jump predictor is undisclosed, we were able to confirm with simulations of ITTAGE that the few cases where the prediction accuracy is relatively poor are due to footprint issues on the predictor and not inherent to the prediction scheme.

Acknowledgment

This work was partially supported by the European Research Council Advanced Grant DAL No 267175.

References

[1] J. R. Bell. Threaded code. *CACM*, 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*, 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*, 1997.

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

[6] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *Micro, IEEE*, 17(3), 1997.

[7] J. W. Davidson and J. V. Gresh. Cint: a RISC interpreter for the C programming language. In *Symposium on Interpreters and interpretive techniques*, 1987.

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

[9] D. Ehringer. The Dalvik virtual machine architecture, 2010.

[10] M. A. Ertl. Stack caching for interpreters. In *PLDI*, 1995.

[11] M. A. Ertl and D. Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *EuroPar*. 2001.

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

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

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

[15] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

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

[17] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[18] J. Marusarz, S. Cepeda, and A. Yasin. How to tune applications using a top-down characterization of microarchitectural issues. Technical report, Intel.

[19] J. McCandless and D. Gregg. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM TACO*, 8(4), 2012.

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

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

[22] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[23] A. Naumann and P. Canal. The role of interpreters in high performance computing. In *ACAT in Physics Research*, 2008.

[24] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *SIGPLAN Not.*, 33(5), 1998.

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

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

[27] G. Savrun-Yeniçeri et al. Efficient interpreter optimizations for the JVM. In *PPPJ*, 2013.

[28] A. Seznec. Analysis of the O-GEometric History Length Branch Predictor. In *ISCA*, 2005.

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

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

[31] A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *JILP*, 8, 2006.

[32] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM TACO*, 4(4), 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*, 1997.

[34] B. Vitale and T. S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *Workshop on Interpreters, virtual machines and emulators*, 2004.

[35] V. M. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *FHPM*, 2010.

[36] E. Wieprecht et al. The HERSCHEL/PACS Common Software System as Data Reduction System. *ADASS XIII*, 2004.

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