Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 435–441

DOI: 10.15439/2015F139
ACSIS, Vol. 5

Performance Research and Optimization on CPython's Interpreter

Huaxiong Cao, Naijie Gu[1], Kaixin Ren, and Yi Li
1) Department of Computer Science and Technology, University of Science and Technology of China
2) Anhui Province Key Laboratory of Computing and Communication Software
3) Institute of Advanced Technology, University of Science and Technology of China
Hefei, China, 230027
Email: chx319@mail.ustc.edu.cn, gunj@ustc.edu.cn

***Abstract*—In this paper, the performance research on CPython's latest interpreter is presented, concluding that bytecode dispatching takes about 25 percent of total execution time on average. Based on this observation, a novel bytecode dispatching mechanism is proposed to reduce the time spent on this phase to a minimum. With this mechanism, the blocks associated with each kind of bytecodes are rewritten in hand-tuned assembly, their opcodes are renumbered, and their memory spaces are rescheduled. With these preparations, this new bytecode dispatching mechanism replaces the time-consuming memory reading operations with rapid operations on registers.**

This mechanism is implemented in CPython-3.3.0. Experiments on lots of benchmarks demonstrate its correctness and efficiency. The comparison between original CPython and optimized CPython shows that this new mechanism achieves about 8.5 percent performance improvement on average. For some particular benchmarks, the maximum improvement is up to 18 percentages.

I. Introduction

The past decade has witnessed the widespread use of Python, which is a typical dynamic language designed to execute on virtual machines. Several software engineering advantages over statically compiled binaries, including portable program representations, thread manage -ment, some safety guarantees, built-in automatic memory, and dynamic program composition through dynamic class loading, are provided by Python. These advanced features enhance the user programming model and drive the success of Python language. However, these features also require the dynamic compilers to do quite a lot of extra operations, such as type checking, wrapping/unwrapping of boxed values, virtual method dispatching, bytecode dispatching, etc. These extra operations usually make Python programs several or dozens of times slower than static programs achieving the same functionality. Moreover, traditional static program optimiza -tion technologies are frustrated, introducing new challenges for achieving high performance.

In response, more and more researchers have paid their attention to optimize dynamic language compilers. The technologies proposed aim to improve performance by monitoring programs' behavior and using this information to drive optimization decisions [1]. The dominant concepts that have influenced effective optimization technologies in today's virtual machines include JIT compilers, interpreters, and their integrations.

JIT (just-in-time) techniques exploit the well-known fact that large scale programs usually spend the majority of time on a small fraction of the code [2]. During the execution of interpreters, they record the bytecode blocks which have been executed more than a specified number of times, and cache the binary code associated to these blocks. The next time these bytecode blocks are executed, it has no need to interpreter these bytecodes once again, just jumps to the cached binary code, and continues the execution. By this way, much interpreting work is omitted, and performance improvement can be achieved. However, since the threshold is quite large in general, the interpreting stage still accounts for a large proportion of total execution time. JIT strategies work well for *for/while* blocks which likely exist in science compute field. For the programs, whose purposes are remote configuration, warning, tracing, statistics, communication, etc, it is very hard to find hot blocks. Though not large, these programs are usually executed quite frequently.

Interpreters have a series of advantages which make them attractive [3]. Firstly, optimizing interpreters reduces the uptime of all Python programs with or without hot blocks. Secondly, they are quite simpler to construct than JIT compilers, making them quicker, more reliable to construct and easier to maintain. Thirdly, interpreters require less memory than JIT compilers, for both the interpreted virtual machine code and the interpreter itself. Interpreters for dynamic language have two main structures [1]: switch-based structure and threaded-based structure, whose details are given in section 2 part B. The latter is the latest and most efficient mechanism. Recently, many researchers have applied dynamic techniques to improve the performance of threaded-based structure. Piumarta and Riccardi [4] describe their techniques to dynamically generate threaded codes for purpose of eliminating a central dispatch site and inlining common bytecode sequences. Ertl and Gregg [5] extend Piumarta and Riccardi's work by duplicating bytecode sequences, researching various interpreter generation heuristics,

[1] Corresponding author. Email: gunj@ustc.edu.cn

and concentrating on improving branch prediction accuracy. Gagnon and Hendren [6] adapt Piumarta and Riccardi's research to work in the context of multithreading and dynamic class loading. Sullivan et al. [7] describe a combination between an interpreter implementation and a dynamic binary optimizer, which enhances the efficacy of the underlying dynamic binary optimizer during the execution of interpreter.

Despite the enhancements above, interpreters are still worse than static compilers from the runtime perspective. The purpose of our research is to explore new ways to achieve further performance improvement. In this paper, the performance research on CPython's latest interpreter is presented, finding that bytecode dispatching takes about 25 percent of total execution time on average. Then, a novel bytecode dispatching mechanism, which aims at reducing memory reading operations during bytecode dispatching and reducing the time spent on this phase to a minimum, is proposed. This mechanism is implemented inside CPython's interpreter. Experiments on lots of benchmarks demonstrate the correctness and efficiency of our new mechanism. The comparisons between original CPython and optimized CPython show that our new mechanism can achieve about 8.5 percent performance improvement on average. For some particular benchmarks, the maximum improvement is up to 18 percentages.

The remainder of this paper is structured as follows: In the next section, we make comparisons among different compilers of Python and present both switch-based and threaded-based mechanisms. Section 3 reports our performance research on CPython's interpreter. Section 4 discusses the main techniques we adopted to construct a new bytecode dispatching mechanism. Benchmarks and Experiments are given in section 5. Section 6 discusses the related work. We conclude this paper in the last section.

II. Background

A. CPython VS Other Python compilers

A series of dynamic compilers are designed to run Python programs, such as CPython, Jython [8], IronPython [9], Pyston [10], PyPy [11], etc. The comparisons among them are shown on Table I, and these benchmarks are provided officially by Pyston/minibenchmarks and Pyston/microbenchmarks. Among these compilers, CPython is the official and standard compiler for Python language, it can support all the grammars and extensions. Others are developed for special applications. They focus on particular scenarios and take in-depth optimization passes. In this context, these compilers show their excellent performances for some benchmarks, but for other benchmarks, they may be several times slower than CPython. As shown on Table I, taking fid.py for example, it takes CPython 3.696 seconds to execute this script, while IronPython, Pyston and PyPy spend less than 2 seconds on the same script. However, it takes CPython 1.082 seconds to execute emwomding.py, while JPython, IronPython, Pyston and PyPy spend more than 3.700 seconds on the execution of emwomding.py.

What's more, some benchmarks, like pydigits.py, empth_lo -op.py, vecf_add.py, nq.py, raytrace.py, cannot be executed successfully by some of these compilers. Based on these comparisons, our research focuses on CPython, and has not only practical application value, but also instruction meaning for the improvement of other interpreters.

B. Switch-based Mechanism VS Thread-based Mechanism

The performance of interpreters depends heavily on their bytecode dispatching mechanisms. CPython's interpreter provides two main bytecode dispatching mechanisms: switch-based mechanism and thread-based mechanism.

The inner loop of switch-based interpreters is quite simple: jump to the dispatch point, fetch the next bytecode and dispatch to its implementation through a switch statement. Its typical framework is shown in Fig. 1.

As shown in Fig. 1, the interpreter is an infinite loop with a big switch block to dispatch bytecodes successively. Each bytecode are implemented by a particular case in the body of this switch block. At the end of each case, control are passed back to the beginning of the infinite loop by breaking out of this switch block. Traditional C compilers like GCC translate this switch block into a series of comparison statements. Considering a particular bytecode whose opcode is BINARY_SUBTRACT, five indispensable comparisons should be executed before reaching its associated block. Assuming that all bytecodes have the same probability of occurrence, it takes N/2 (N is the total kinds of bytecodes) comparisons on average to find the corresponding block. As to

TABLE I.
COMPARISONS AMONG CPYTHON, JYTHON, IRONPYTHON, PYSTON AND PYPY

Benchmarks	CPython-3.3.0	Jython	IronPython	Pyston	PyPy
empty_loop.py	3.532s	5.491s	**Failed**	19.248s	0.248s
pydigits.py	0.034s	2.126s	1.431s	**Failed**	0.039s
fid.py	3.696s	4.776s	1.527s	0.636s	0.864s
vecf_add.py	9.890s	12.970s	25.150s	**Failed**	0.059s
allgroup.py	0.836s	4.428s	3.052s	**Failed**	**18.804s**
chaso.py	26.268s	33.091s	51.366s	**Failed**	1.392s
go.py	53.787s	56.746s	123.404s	**Failed**	33.638s
nbody.py	12.677s	19.535s	17.057s	25.540s	1.470s
nq.py	29.879s	**Failed**	35.673s	**Failed**	**44.418s**
raytrace.py	11.608s	16.418s	26.724s	**Failed**	1.228s
polymorphism.py	4.358s	7.228s	7.959s	4.390s	**14.260s**
unwinding.py	1.082s	3.757s	2.802s	93.180s	4.481s

CPython, N is 101. So, performing bytecode dispatching wastes the majority of execution time and it is very inefficient.

Threaded-based mechanism is the latest and most efficient mechanism for interpreters, popularized by the Forth programming language [12]. There are many kinds of threaded-based interpreters, and direct threading is regarded as the most efficient one. Direct threading mechanism improves performance by eliminating redundant comparisons. In addition, rather than returning to a central dispatch point, the implementation of each direct threading opcode ends with the particular code required to dispatch the next opcode. This optimization eliminates the centralized dispatch, removing lots of jump instructions. The framework of direct threading mechanism is shown in Fig. 2.

As shown in Fig. 2, execution starts with fetching the address of the very first bytecode's implementation and then jumping to that address. For each bytecode, it performs its own work at first, then increases the instruction pointer, thirdly fetches the address of next bytecode's implementation from memory, and jumps to the target address to handle successive bytecode. The native instructions associated with bytecode dispatching is shown in Fig. 3, with one stack reading, one memory reading and one jump. It can be seen from Fig. 3 that the bytecode dispatching overheads associated with direct threading mechanism are quite lower than those associated with switch-based mechanism.

III. Motivation: Performance Research

So far, it can be seen that bytecode dispatching plays a very important role in the performance of CPython. To make this concept intuitional, a series of meticulous experiments are conducted and the experimental results are shown in table II.

To make the results credible and accurate, this work utilizes hardware performance counters provided by Intel, and calculates the number of ticks which are consumed by the procedure to find next bytecode. The results are shown in Table II column 2. The ticks, which are consumed by the total program execution are also recorded and listed in Table II column 3. In Table II column 4, the ratios between them are calculated and listed. According to this table, it can be sure that the time consumed by bytecode dispatching is 25 percent of the total execution time on average. So if this stage could be further optimized, the total performance of CPython will be improved obviously.

As shown in Fig. 3, the bytecode dispatching in direct threading mechanism is composed of one stack reading, one jump and one memory reading. Inside the memory reading, there are an addition and a multiplication. Since operations on stack are quite frequent, the first stack reading instruction will also hit the L1 cache. Table III [13] lists the time needed to read data from register, L1 cache, L2 cache, memory and disc. According to this table, the first instruction takes about 2ns. The second instruction contains an addition operation, a multiplication operation and a load operation. The first two operations can be finished within several ticks. However, since L1 cache can only contain eight pages (32k L1 cache, page size 4k), reading the address of successive bytecode's implementation leads to lots of L1 cache misses. In this context, it has to get the right value from L2 cache, or even memory, which may take dozens of nanoseconds. Assuming that reading from these three memory structures has the same probability of occurrence, it takes about 30ns to finish this memory reading. So, it can be seen that the second instruction takes the majority of the time spent on bytecode dispatching, and optimizing this instruction will contribute a lot to bytecode dispatching.

```
Compiled code:
    Unsigned char code[] = { …
        LOAD_FAST,
        LOAD_CONST,
        BINARY_ADD,
        BINARY_SUBTRACT, … };
Bytecode implementations:
For( ; ; ) {
    insn = get_next_insn(insn);
    opcode = get_opcode(insn);
    switch(opcode)  {
        case NOP: …; break;
        case LOAD_FAST: …; break;
        case LOAD_CONST: …; break;
        case BINARY_ADD: …; break;
        case BINARY_SUBTRACT: …;
break;
        ….
    }
}
```

Fig. 1 The framework of switch structure.

```
compiled code:
void * table[] = { …
    ##LOAD_FAST,
    ##LOAD_CONST,
    ##BINARY_ADD,
    ##BINARY_SUBTRACT, … };
bytecode implementations:
….;
LOAD_FAST:
    …;
    cpcode = get_opcode(insn_next);
    goto *table[opcode];
LOAD_CONST:
    …;
    opcode = get_opcode(insn_next);
    goto *table[opcode];
BINARY_ADD:
    …;
    opcode = get_opcode(insn_next);
    goto *table[opcode];
BINARY_SUBTRACT:
    …;
    opcode = get_opcode(insn_next);
    goto *table[opcode];
….
```

Fig. 2 The framework of direct threading mechanism.

```
//read stack and get the opcode of next
//bytecode
mov     -0x1c4(%ebp),%ebx
//read memory and get the address
//related to next bytecode
mov     0x8220000(,%ebx,4),%eax
//jump to deal with next bytecode
jmp     *%eax
```

Fig. 3 The native instructions associated with opcode dispatch.

TABLE II
TIME SPENT ON BYTECODE DISPATCHING

benchmarks	Dispatching ticks	Total ticks	Ratio
Queen.py	124356740795	560761937239	22.176388%
test_pow.py	66430602	605265570	10.975447%
diff.py	11349007012	42445898103	26.737583%
test_sqrt.py	209325165518	823381752732	25.422613%
test_image.py	121776483	633664878	19.217801%
iterator.py	218799781767	847798974988	25.807980%
generator.py	987262878658	3744968182830	26.362383%
range.py	1238431745418	4568662416948	27.107098%
while.py	1810252155978	6454065479106	28.048246%
average	511107298026	1.8937026e+12	26.989839%

TABLE III
THE TIME TOKEN TO READ DATA FROM DIFFERENT STORAGES

	Reg -ister	L1 cache	L2 cache	Mem -ory	disc
Time (ns)	0.5	2	10~20	50~100	25~50

IV. THE NEW DISPATCHING MECHANISM

According to the above section, bytecode dispatching spends most of the time on memory reading. To optimize bytecode dispatching procedure, a new bytecode dispatching mechanism is proposed and the framework of CPython's interpreter is reconstructed, drawing the inspiration from [14]. Our new techniques proceed in three phases and their functionalities are described below.

Phase 1: Rewriting and statistics. This phase rewrites the statements associated with each kind of bytecodes in hand-tuned assembly, and calculates the length of the final binary code of each case. The results are shown at Table IV. According to this table, there are 19 kinds of bytecodes which own less than 64 byte binary code, 51 kinds of bytecodes which own less than 128 byte but more than 64 byte binary code, 22 kinds of bytecodes which own more than 128 byte but less than 256 byte binary code, and 9 kinds of bytecodes which own more than 256 byte binary code.

TABLE IV
STATISTICS ON LENGTH OF BYTECODES' FINAL BINARY CODE

length	[0,64]	(65,128]	(128,256]	(256,512]
number	19	51	22	9

Phase 2: calculation. This phase calculates the proper size of each memory unit, named BSIZE. Inside the optimized interpreters, the memory allocated to each kind of bytecodes is an integral multiple of BSIZE bytes. Mark *binary[i]* as the length of binary code associated with *i*th kind of bytecodes, the BSIZE should be the minimum positive number which conforms to condition (1) and condition (2).

$$\exists i : 2^i = BSIZE \tag{1}$$

$$\sum_{i=0}^{100} \left\lceil \frac{binary[i]}{BSIZE} \right\rceil \le 256 \tag{2}$$

The first constraint assures the calculation of next bytecode's address is simplified to a quick left shift. The second constraint assures that the maximum opcode of bytecodes isn't bigger than the threshold value (256) defined by CPython. If the BSIZE is too big, there will be a lot of NOP instructions and the executable file will be quite big, causing damage to the performance. That's why BSIZE should be set as small as possible. According to Table IV, the proper value of BSIZE is 128.

Phase 3: opcode redefinition. This phase redefines the opcodes of bytecodes, with their order stay the same. Let *opcode[i]* be the new opcode of *i*th kind of bytecode, and the algorithm used here is shown as follows:

$$opcode[0] = 0$$
$$opcode[i] = opcode[i-1] + \left\lceil \frac{binary[i-1]}{BSIZE} \right\rceil, 1 \le i \le 100 \tag{3}$$

Taking the first three bytecode (POP_TOP, ROT_TWO and ROT_THREE) as an example, their original opcodes are 1, 2 and 3, respectively. Assuming the first bytecode has 200 byte binary code, the second bytecode has 350 byte binary code, and the third bytecode has 100 byte binary code, their final opcodes will be 1, 3 and 6. The memory allocation of the interpreter with new dispatching mechanism is shown in Fig. 4.

As shown in Fig. 4, POP_TOP's binary code is stored in the first grid region (from top to bottom), ROT_TWO's binary code is stored in the second grid region, while the ROT_THREE's binary code is stored in the third grid region. There are gaps between neighboring kinds of bytecodes. In addition, the framework of the new interpreter of CPython is shown in Fig. 5. Inside this new interpreter, the binary code of all kinds of bytecodes is arranged in numerical order and each of them occupies several BSIZE memory spaces. So, every time CPython jumps to the implementation associated to next bytecode, it just need to execute this simple statement: "goto (base + BSIZE*opcode)". Fig. 6 shows the native instructions associated with this statement, including one stack reading, one left shift, one addition and one jump. The middle two instructions are operations on registers and can be finish within

2 ticks (0.8ns). Hence, it just takes 2.8ns to get the address of next bytecode's implementation, instead of 30ns. Since NEXTOP operation is executed so many times, this new structure will bring a lot of performance promotion.

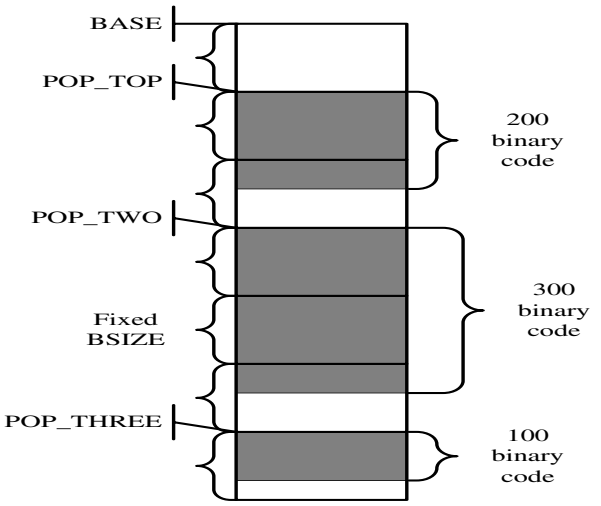

Fig. 4 The memory allocation of post-construction interpreter.

```
bytecode implementations:
BASE:
….;
asm(".balign BSIZE \n\t");
LOAD_FAST:
    …;
    opcode = get_opcode(insn_next);
    goto (BASE+opcode* BSIZE);
asm(".balign BSIZE \n\t");
LOAD_CONST:
    …;
    opcode = get_opcode(insn_next);
    goto (BASE+opcode* BSIZE);
asm(".balign BSIZE \n\t");
BINARY_ADD:
    …;
    opcode = get_opcode(insn_next);
    goto (BASE+opcode* BSIZE);
asm(".balign BSIZE \n\t");
BINARY_SUBTRACT:
    …;
    opcode = get_opcode(insn_next);
    goto (BASE+opcode* BSIZE);
….
```

Fig. 5 The framework of interpreter with new bytecode dispatching.

```
//read stack and get the opcode of next
//bytecode
mov    -0x1c4(%ebp),%eax
shl      $8, %eax  //left shift
//BASE is an immediate value
addl    $BASE, %eax
//jump to deal with next bytecode
jmp    *%eax
```

Fig. 6 The native instructions associated with new bytecode dispatching.

V. Experimental Evaluation

This new bytecode dispatching mechanism has been implemented under Ubuntu-12.04 on Intel(R) Core(TM) 2 CPU E6550 2.33GHz with 2 processors, 2G memory, 32k L1 cache and 4M L2 cache. When assess the optimized CPython, about 35 benchmarks are gathered from CPython-3.3.0 and Pyston-0.2. Two criterions are considered here: (1) correctness (optimized CPython can provide the same functionality as original one.), and (2) efficiency (optimized CPython can execute benchmarks faster than original one).

A. Correctness

Taking benchmark *diff.py* as an example, file sysmodule.c and file _testcapimodule.c are chosen from CPython's source files randomly and are used as two parameters of *diff.py*. Then, *diff.py* is executed by optimized CPython and original CPython separately, and two result files are produced. Later, Linux command *diff* are used to compare these two result files, concluding that there is no difference between them.

In addition, the benchmarks, which are listed in Table I and cannot be executed successfully by Jython, IronPython, Pyston or PyPy, can be executed successfully by optimized CPython. Actually, optimized CPython can execute all these 35 benchmarks and the time spent on these benchmarks is shown in Fig. 7.

B. Efficiency

Optimized CPython and original CPython are compiled with the same parameters, and both of them are used to execute these benchmarks separately. Each benchmark is executed one thousand times to reduce volatility, and the final results are shown in Fig. 7. As to some benchmarks taking too little time to run, we recode the time spent on their several times running. Taking *image.py* for example, it just takes original CPython 0.046 seconds to run this benchmark. So, running *image.py*100* takes 4.6 seconds. Similarly, running *queen.py/10* takes 10.23 seconds, for the reason that it takes CPython 102.3 seconds to finish the queen.py. It can be seen from Fig. 7 that all of these benchmarks achieve performance improvement with our new bytecode dispatching mechanism. The average performance improvement is about 8.5%. In particularly, benchmark *image.py* achieves up to 18% performance improvement.

The performance improvement happens for two main reasons. Firstly, this new interpreter replaces slow memory reading operations with quick operations on registers, and reduces the time spent on bytecode dispatching. Secondly, less memory reading operations cut down on the cache misses, especially for low specification machines. Perf [15] is used to measure L1 data cache misses for part of these benchmarks and the results are shown at Table V. Table V column 2 lists the cache misses reported by original CPython when it is used to execute these benchmarks, while Table V column 3 lists the cache misses reported by optimized CPython when it is used to do the same jobs. The last column in Table V shows that about 14.64 percent of cache misses are left out on average. The more memory reading operations it reduces, the greater chance that

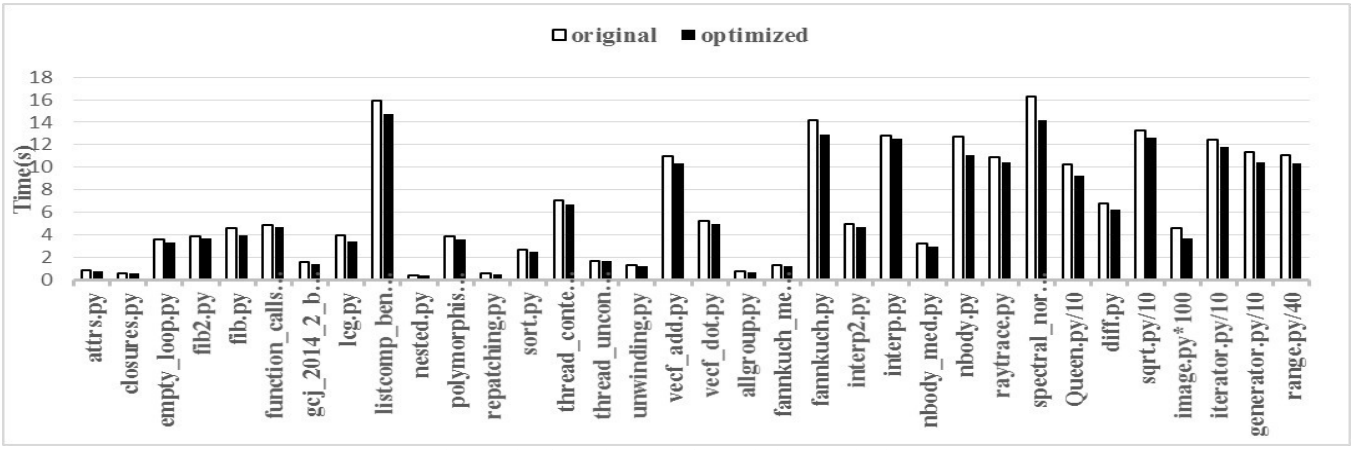

Fig. 7 Ratio of benchmarks before and after optimization.

TABLE V
STATISTICS ON L1 DATA CACHE MISSES

Benchmarks	ori-CPython	opti-CPython	1-(opti/ori)
queen.py	64M	43M	32.8%
pow.py	3.93M	3.54M	9.9%
diff.py	36.9M	33.1M	10.3%
sqrt.py	3.46M	2.56M	26.0%
image.py	1.37M	1.14M	16.8%
iterator.py	83.5M	80.7M	3.35%
generator.py	170M	148M	12.9%
range.py	292M	270M	7.53%
while.py	295M	259M	12.2%
average	-	-	14.64%

Note: "ori-CPython" stands for "original CPython", "opti-CPython" stands for "optimized CPython", and "1-(opti/ori)" stands for "1 - (optimized CPython / original CPython)".

remaining memory reading operations hit the cache. In another word, the remaining memory reading operations can be finished in less time.

VI. RELATED WORK

There are a large quantity of recent papers researching interpreter performance. Romer et al. [16] have reported the performance characteristics of some interpreters. Later, Ertl and Gregg [3] investigated the performance of recent efficient interpreters. Both of these two studies have found that almost every interpreters perform an exceptionally high number of indirect branches. Since most of indirect branches are caused by bytecode dispatching, their conclusion is consistent with this performance research reporting in section 3. Our research aims at reducing the time spent on the second instruction in Fig. 3 to a minimum, while their work aims at optimizing the third instruction in Fig. 3 and reducing indirect branches mispredictions. This is the main difference between us.

Several techniques are used to reduce indirect branches mispredictions. J Hoogerbrugge and L Augusteijn [17], [18] have proposed that software pipelining interpreters is a way to reduce dispatch branch cost on architectures with split indirect branches. In addition, Subroutine threading [19] has also been proposed to avoid the overheads of indirect branches in intrepreter implementations. Each bytecode is implemented with a particular C function. Instead of dispatching or interpret-ing bytecode, a simple JIT compiler generates executable code for a sequence of calls to these functions. This method can eliminate indirect branches at the cost of sacrificing both simplicity and portability.

Cache misses have a significant impact on the program performance [20]. Brunthaler [21] have proposed a formalization of interpreter opcode ordering (bytecode scheduling) for an interpreter with an extended opcode set, and concluded that high cache miss ratio is another bottleneck of interpreters. A lot of techniques, like feedback-guided technique [22], profile-guided technique [23], etc, are conducted to achieve better orderings, improving code locality and reducing cache misses. Jason McCandless and his co-worker [24] implement a metaheuristic (Monte Carlo) to generate better orderings, achieving considerable performance improvement. As shown in section 5, cache misses can also be reduced by our new mechanism, making the new interpreter perform better.

Another method which is widely used is combining sequences of VM instructions into super-instructions [25]. This technique focuses on reducing the number of bytecode dispatches, and has two variants: static super-instructions and dynamic super-instructions. The comparison between these two variants are shown in [5].

As far as we know, the nearest research to our work is [14]. The main difference is that their research plans for each bytecode a fixed-size block of instructions. In such a case, bytecodes with short instruction implementation will incur a lot of NOP instructions, which increases the code size of the interpreter dispatch loop and reduce cache hit ratio. In addition, bytecodes with long instruction implementation will lead to lots of extra jump instructions. Relatively speaking, our mechanism is more flexible and efficient than that, with much less NOP instructions and no extra jump.

VII. CONCLUSIONS

In this paper, the performance research on CPython's interpreter is carried out, figuring out that bytecode dispatching has a big influence on interpreters. Then, a novel bytecode dispatching mechanism is designed, aiming at removing memory reading operations during bytecode dispatching and

reducing the time spent on this phase to a minimum. The final binary code of each kind of bytecodes is arranged in numerical order and each of them occupies several BSIZE memory spaces.

This novel mechanism is implemented, and its correctness and efficiency are demonstrated by a large number of benchmarks. Comparisons are made between original CPython and optimized CPython to show that the new mechanism achieves about 8.5 percent performance improvement on average. For some particular benchmarks, the maximum improvement is up to 18 percentages. This performance improvement happens for two main reasons: lesser memory reading operations and lesser cache misses. The novel mechanism proposed here can also be adopted by other interpreters, and will contribute to their performance improvement.

REFERENCES

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proc. of IEEE,* vol. *93*, no. *2*, pp. *449-466*, Feb., 2005. http://dx.doi.org/10.1109/jproc.2004.840305

[2] D. E. Knuth, "An empirical study of FORTRAN programs," *Softw.: Practice and experience,* vol. *1*, no. *2*, pp. *105-133*, Jun., 1971. http://dx.doi.org/10.1002/spe.4380010203

[3] M. A. Ertl, and D. Gregg, "The structure and performance of efficient interpreters," *JILP,* vol. *5*, pp. *1-25*, Mar., 2003.

[4] I. Piumarta, and F. Riccardi, "Optimizing direct threaded code by selective inlining," *ACM Sigplan Not.,* vol. *33*, no. *5*, pp. *291-300*, May, 1998. http://dx.doi.org/10.1145/277652.277743

[5] M. A. Ertl, and D. Gregg, "Optimizing indirect branch prediction accuracy in virtual machine interpreters," *ACM Sigplan Not.,* vol. *38*, no. *5*, pp. *278-288*, May, 2003. http://dx.doi.org/10.1145/780822.781162

[6] E. Gagnon, and L. Hendren, "Effective inline-threaded interpretation of Java bytecode using preparation sequences," in *Compiler Construction*, G. Goos, J. Hartmanis and J. v. Leeuwen, Ed., Heidelberg, DE: Springer, 2003, pp.170-184. http://dx.doi.org/10.1007/3-540-36579-6_13

[7] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe, "Dynamic native optimization of interpreters," in *Proc. 2003 workshop on Interpreters, virtual machines and emulators*, New York, 2003, pp. *50-57*. http://dx.doi.org/10.1145/858570.858576

[8] R. W. Bill, *Jython for Java programmers*, 1st. ed., USA: SAMS, 2001. ISBN 978-0735711112. http://file182.cordpdf.org/1juv4l_jython-for-java-programmers.pdf

[9] J. Hugunin, "IronPython: A fast Python implementation for .NET and Mono," in *PyCon.*, Washington, USA, 2004.

[10] M. L. Hetland, "Pedal to the Metal: Accelerating Python," in *Python Algorithms*, Trondheim, NO: Springer, 2014, pp.255-258. http://dx.doi.org/10.1007/978-1-4842-0055-1_12

[11] Pypy. (2006). PyPy is a fast, compliant alternative implementation of the Python language. [Online]. Available: http://pypy.org/.

[12] Forth-language for interactive computing. (1970). an technical report on Forth. [Online]. Available: http://mx1.1strecon.org/downloads/Forth_Resources/CM_ForthLanguageInteractiveComputing_1970.pdf.

[13] J. L. Hennessy, and D. A. Patterson, *Computer architecture: a quantitative approach*, Waltham, GB: Elsevier, 2012. ISBN 978-0-12-383872-8. http://www.cpp.edu/~kding/materials/Computer%20Architecture%20A%20Quantitative%20Approach%20(5th%20edition).pdf

[14] Y. Ye, C.-Q. Li, and J.-S. Hu, "Transplantation and Optimization of Dalvik Virtual Machine Based on CK610," *Comput. Eng.,* vol. *16*, pp. *100*, 2011. doi:10.3969/j.issn.1000-3428.2011.16.098

[15] A. Melo, "The new linux' perf' tools," in *17th Int. Linux Sys. Tech. Conf.*, Nuremberg, GE, 2010.

[16] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The structure and performance of interpreters," *ACM Sigplan Not.,* vol. *31*, no. *9*, pp. *150-159*, 1996. http://dx.doi.org/10.1145/248209.237175

[17] J. Hoogerbrugge, and L. Augusteijn, "Pipelined Java Virtual Machine Interpreters," in *Compiler Construction*, Vol. 1781, D. A. Watt, Ed., Heidelberg, GE: Springer, 2000, pp.35-49. http://dx.doi.org/10.1007/3-540-46423-9_3

[18] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. v. d. Wiel, "A code compression system based on pipelined interpreters," *Softw.: Practice and Experience,* vol. *29*, no. *11*, pp. *1005-23*, 1999. http://dx.doi.org/10.1002/(sici)1097-024x(199909)29:11<1005::aid-spe270>3.0.co;2-f

[19] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown, "Context threading: A flexible and efficient dispatch technique for virtual machine interpreters," in *Proc. Int. Symp. Code Gen. Optim.*, Washington, USA, 2005, pp. *15-26*. http://dx.doi.org/10.1109/cgo.2005.14

[20] Ristov, and Sasko, "Performance impact of reconfigurable L1 cache on GPU devices," *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on,* Kraków, Poland, IEEE, 2013, pp. 507-510.

[21] S. Brunthaler, "Interpreter instruction scheduling," in *Compiler Construction*, J. Knoop, Ed., Heidelberg, GE: Springer, 2011, pp.164-178. http://dx.doi.org/10.1007/978-3-642-19861-8_10

[22] P. Zhao, and J. e. N. Amaral, "Feedback-directed switch-case statement optimization," in *Proc. 2005 Int. Conf. Parallel Process. Workshops*, Oslo, NO, 2005, pp. *295-302*. http://dx.doi.org/10.1109/icppw.2005.32

[23] K. Pettis, and R. C. Hansen, "Profile guided code positioning," *ACM Sigplan Not.,* vol. *25*, no. *6*, pp. *16-27*, 1990. http://dx.doi.org/10.1145/93548.93550

[24] D. Gregg, and J. Mccandless, "Optimizing interpreters by tuning opcode orderings on virtual machines for modern architectures," in *Conf. Princip. Prac. Program. Java*, Kongens Lyngby, DK, 2011, pp. *161-170*. http://dx.doi.org/10.1145/2093157.2093183

[25] Optimizations for a java interpreter using instruction set enhancement. (2005). Optimizations for a java interpreter using instruction set enhancement. [Online]. Available: https://www.scss.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-61.pdf.