Efficient Inline Caching without Dynamic Translation

Stefan Brunthaler
Institute of Computer Languages
Vienna University of Technology
Argentinierstr. 8
1040 Vienna, Austria
brunthaler@complang.tuwien.ac.at

ABSTRACT

Inline caching is a very important optimization technique for interpreters, effectively eliminating the overhead in dynamic typing. Unfortunately, inline caches are mostly used together with dynamic translation, which is expensive in terms of implementation costs. We present *efficient* inline-caching techniques that do not require dynamic translation.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Interpreters, Optimization*

General Terms

Languages, Performance, Interpreters, Inline Caching

Keywords

Interpreter, Python, bytecode, inline caching, type feedback

1. MOTIVATION

Usually, inline caches are applied in conjunction with dynamic translation, i.e., in a piece of dynamically translated code, inline caches serve the purpose of eliminating method lookups and/or dynamic typing by caching a resolved target address directly in the translated native code, replacing the call to the system default lookup routine.

Inline caches per se—without their dynamic translation counterpart—are, however, not present in many popular virtual machines, e.g. Lua, Perl, Python, and Ruby. A possible explanation for this is mentioned in Hölzle's PhD thesis on page 31: "*A straightforward interpreter could not use inline caching and would have to use a lookup cache instead.*" [2]. Building on our previous results on the varying potential of optimization techniques [1] for those interpreters, we present techniques for adding efficient inline caches to them, and report on preliminary experimental results. Compared to the basic technique (cf. Section 2), the advanced technique (cf. Section 3) offers improved data locality and instruction decoding efficiency.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

2. BASIC INLINE CACHING

We demonstrate our technique for the Python interpreter, because it offers a lot of potential for type feedback and inline caching in general. As an example of how the programming language is implemented we present the implementation of the `BINARY_ADD` instruction:

```
case BINARY_ADD:
    w= POP();
    v= TOP();
    if (PyUnicode_Check(v) && PyUnicode_Check(w))
        x = unicode_concat(v, w, f, ip);
    else
        x = PyNumber_Add(v, w);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    break;
```

We see in the `BINARY_ADD` example how the types are dynamically resolved in operation implementation. If operands are Unicode-strings they are concatenated, the numerical addition is delegated to the `PyNumber_Add` function.

Our inline caching scheme requires the following steps:

1. creation of an array of pointers with the same number of elements as there are instructions in a given code sequence.
2. change of the operation implementation to use the inline cache pointer.
3. initialization of the array of inline cache pointers.
4. implementation of type feedback primitives.

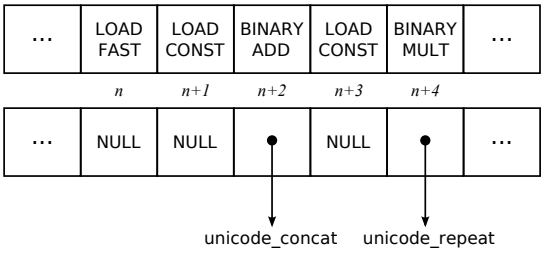

Figure 1: After the first execution, the inline-cache pointers at offsets $n+2$ and $n+4$ were updated.

5. change in the instruction pointer manipulation functions to change the inline cache pointer, too.

For our inline cache to work, we need to change operation implementation to use a pointer instead of a direct call. Continuing with our previous `BINARY_ADD` example, this results in:

```
case BINARY_ADD:
    w= POP();
    v= TOP();
    x= ((binaddfun) *ic_ptr)(v, w, f, next_instr);
    Py_DECREF(v);
    Py_DECREF(w);
    SET_TOP(x);
    break;
```

The array of inline cache pointers is dynamically allocated; we require the same amount of pointers as there are bytecode instructions in a given function. In order to work as expected, the array of inline cache pointers needs to be initialized. Since the initialization functions are known at compile-time, this requires only a simple iteration over the sequence of bytecodes, where we assign a function pointer for each bytecode we implement, e.g. if the i-th instruction of a function is a `BINARY_ADD` instruction, then the corresponding inline-cache pointer is initialized to point to the address of the `ic_add` function, which realizes the operation implementation of the `BINARY_ADD` instruction.

To implement type feedback, we need to instrument specific places which update the inline cache pointer of the current instruction. Say that we want to inline cache whether a `BINARY_ADD` instruction ended up being a string concatenation, then we would have to update the inline cache accordingly (cf. Figure 1, where `unicode_concat` and `unicode_repeat` were inline cached). Since the inline cache can be invalid at times, our technique changes the implementation of the inline cached functions, which check whether the actual parameters match their expected types.

Finally, we need to take care of changing the inline cache pointer whenever we change the instruction pointer of the virtual machine, i.e. when dispatching to the next instruction, jumping as a result of a conditional statement or an unconditional jump, etc.

3. BYTECODE + INLINE CACHE

Figure 2 illustrates the result of combining the separate arrays of bytecodes and inline-cache pointers. Note that for every bytecode instruction $2n$ the corresponding inline-cache-pointer is $2n + 1$.

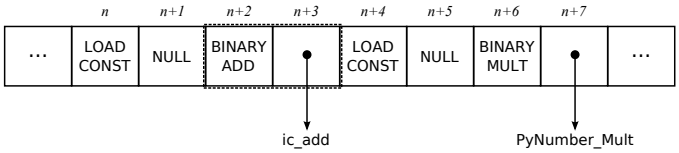

Figure 2: Merged representation.

In order to combine the separate arrays, we have to change the instruction encoding of the Python interpreter. Instead of the irregular encoding which uses dedicated bytes for encoding arguments to instructions, our technique requires the switch to a regular instruction format. This regular instruction format encodes instructions and arguments into just one native machine word. Consequently, the new instruction format ensures that instructions are adjacent at all times, which makes updating the inline cache pointer considerably easier. Next, we need to interleave the instructions with native machine words that we use to store the inline cache pointers.

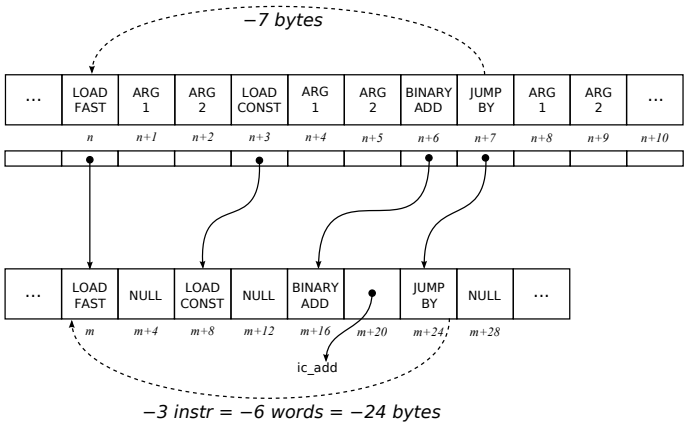

Figure 3: Rewrite pointers (*middle-row*) build the correspondence from the old instruction encoding scheme (*top-row*) to the new one (*bottom-row*).

Since all jumps encoded in the original format include the argument-bytes in their absolute/relative destination positions, we have to relocate the jumps to match the new instruction format. The relocation can be done during initialization of our inline cache (cf. Figure 3).

The new combined data-structure requires significantly more space—two native machine words for each instruction byte. To compensate for the additional space requirements, we use a profiling infrastructure to decide when to switch to this new instruction encoding at run time.

4. EVALUATION AND CONCLUSIONS

First experiments demonstrate that our technique from Section 3 reduces the number of executed native machine instructions thereby improving performance, too. In particular, the reduction in call and jump instructions of up to 30% together with a speedup factor of up to 1.3 is promising, and we are currently investigating this in more detail.

5. ACKNOWLEDGMENTS

I am very grateful to Anton Ertl and Jens Knoop for discussions and invaluable feedback on the topic of this paper.

6. REFERENCES

[1] S. Brunthaler. Virtual-machine abstraction and optimization techniques. In *Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09)*, pages 19–30, York, UK, March 2009. Elsevier.

[2] U. Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, Stanford, CA, USA, 1995.