Interpreter Instruction Scheduling

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8,
A-1040 Wien
brunthaler@complang.tuwien.ac.at

Abstract. Whenever we extend the instruction set of an interpreter, we risk increased instruction cache miss penalties. We can alleviate this problem by selecting instructions from the instruction set and re-arranging them such that frequent instruction sequences are co-located in memory. We take these frequent instruction sequences from hot program traces of external programs and we report a maximum speedup by a factor of 1.142. Thus, interpreter instruction scheduling complements the improved efficiency of an extended instruction set by optimizing its instruction arrangement.

1 Motivation

For compilers instruction scheduling is an important optimization that re-arranges assembler instructions of a program to optimize its execution on a native machine without changing the semantics of the original program, i.e., the native machine is constant, but we can change the order of assembler instructions of the program. Interestingly, the situation is actually the other way around in interpreters. Usually, the bytecode instructions of an interpreter cannot be re-arranged without changing the semantics of the corresponding programs. However, we can re-arrange the instructions of an interpreter, such that frequently executed sequences of instructions become co-located in memory, which allows for better instruction cache utilization. So, for interpreters, the program is constant, but we can change the virtual machine to optimize the execution of a program.

Interpreter instruction scheduling becomes increasingly important when an interpreter has a large instruction set, because in such an interpreter not all instructions can be held in caches at all times. Consequently, there is a trade-off between the optimizations and their benefit being influenced by possible cache miss penalties. Our own previous work [4,3] on improving the efficiency of interpreters using purely interpretative optimization techniques, relies heavily on instruction set extension. Fortunately, these optimization techniques are efficient enough to offset increased cache-miss penalties, however, we feel that by using interpreter instruction selection, the gains of these optimization techniques can be noticeably improved.

Other optimization techniques like superinstructions and replication [5] focus on improving branch prediction and instruction cache utilization by copying instruction implementations together into one instruction or distributing copies of the same instruction over the interpreter dispatch routine to improve locality. Contrary to these approaches, interpreter instruction scheduling does not increase the size of the interpreter's dispatch loop, but focuses on improving instruction cache utilization instead of improving branch prediction.

Our contributions are:

- We formalize the general concept of interpreter instruction scheduling.
- We present detailed examples of how interpreter instruction scheduling works, along with an implementation of our algorithm; the implementation is complemented by a detailed description and results of running the algorithm on an actual benchmark program.
- We report a maximum speedup of 1.142 and an average speedup of 1.061 when using interpreter instruction scheduling on the Intel Nehalem architecture, and provide results of our detailed evaluation.

2 Background

We present a formal description of the problem of interpreter instruction scheduling.

$$\begin{aligned}
I &:= i_0, i_1, \ldots, i_n \\
A &:= a_0, a_1, \ldots, a_n \\
P &:= p_0, p_1, \ldots, p_m \\
&\forall p \in P : \exists j : i_j \in I \wedge a_j \in A \Leftrightarrow p = (i_j, a_j) \\
\\
T &:= \{(p, f) \mid p \in P \wedge f \in \mathbb{N}\} \\
K &:= \{p \mid (p, f) \in T \wedge f \geq L\} \\
K &\subset P
\end{aligned} \tag{1}$$

We define an interpreter I as a set of n instructions i. To each instruction i corresponds a native machine address a of the set of n addresses A, i.e., the address for some interpreter instruction i_j is a_j. Next, we define a program P consisting of m instruction occurrences, which are tuples of an instruction i and the corresponding address a. This concludes the definition of the static view on an interpreter. However, our optimization requires profiling information obtained at run-time. Thus, we define the trace T of a program P as the set of all tuples of an instruction occurrence p and its execution frequency f. Since a trace contains much more information than we need, we define a kernel K, that contains all instruction occurrences p of our program P that have execution frequencies above some definable threshold L.

Given these definitions, the following functions allow us to precisely capture the concept of the distance between instructions.

$$\begin{aligned} s(p_i) &:= |a_{i+1} - a_i| \\ d(p_i, p_j) &:= \begin{cases} |a_j - a_i| - s(p_i) & \text{if } i \leq j, \\ |a_i - a_j| - s(p_j) & \text{if } i > j. \end{cases} \\ d_{overall}(P) &:= \sum_{j=1}^{m} d(p_{j-1}, p_j) \end{aligned} \tag{2}$$

First, we define a function s that computes the size of an instruction i. Next, we define a function d that computes the distance of two arbitrary instructions. Here, the important thing is to note, that if two instruction occurrences p_i and p_j refer to two adjacent instructions, i.e., $p_i = (i_i, a_i)$ and $p_j = (i_{i+1}, a_{i+1})$, then the distance between them is zero. ($d(p_i, p_j) = |a_{i+1} - a_i| - |a_{i+1} - a_i|$) Finally, the overall distance of a program is the sum of all of its sequential distances. Using static program data, this makes no sense, because we do not a priori know which parts of program P are hot. Here, we use our kernel K, which contains only relevant parts of the program, with respect to the overall computational effort. Thus, we define interpreter instruction scheduling as finding a configuration of interpreter instructions that results in a minimal overall distance over some kernel K.

For further illustration, we introduce a working example here. We are going to take a close look on how interpreter instruction scheduling works, using the `fasta` benchmark of the computer language benchmarks game [6]. Running the `fasta` program on the Python interpreter, for example with an argument of 50,000, results in the execution of 10,573,205 interpreter instructions. If we instrument a Python interpreter to trace every instruction executed, with additional location information, such as the instruction offset and the function it belongs to, we can easily extract the computationally relevant kernels from a running program. If we restrict ourselves to only consider kernels for interpreter instruction scheduling, we can significantly reduce the amount of information to consider. For example, an aggregated trace of the `fasta` program shows that the interpreter executes 5,976,237 instructions while interpreting the `genRandom` function, i.e., more than half of the totally executed instructions can be attributed to just one function (cf. Table 1.) Another function—an anonymous list comprehension—requires 4,379,824 interpreted instructions (cf. Table 2.) Together, the `genRandom` function and the list comprehension represent 97.95% of all executed instructions.

Though Tables 1, and 2 indicate that our trace gathering tool is imprecise, since it seems to lose some instruction traces, it is precise enough to indicate which parts of the instructions are kernels. For example, the kernel of function `genRandom` includes all 15 instructions between the offsets 64 and 184, whereas the kernel of the anonymous list comprehension includes all 12 instructions between the offsets 24 and 104. In consequence, our interpreter instruction

Table 1. Dynamic bytecode frequency for `genRandom` function of benchmark program `fasta`

Frequency	Offset	Instruction Identifier
1	16	STORE_FAST_A
1	24	LOAD_GLOBAL_NORC
1	32	LOAD_FAST_B_NORC
1	40	CALL_FUNCTION_NORC
1	48	STORE_FAST_C
1	56	SETUP_LOOP
396,036	64	LOAD_FAST_A_NORC
400,000	72	LOAD_FAST_NORC
400,000	80	INCA_LONG_MULTIPLY_NORC
396,037	88	LOAD_FAST_NORC
400,000	96	INCA_LONG_ADD_NORC_TOS
396,041	104	LOAD_FAST_B_NORC
400,000	112	INCA_LONG_REMAINDER_NORC_TOS
396,040	120	STORE_FAST_A
400,000	128	LOAD_FAST_D_NORC
400,000	136	LOAD_FAST_A_NORC
400,000	144	INCA_FLOAT_MULTIPLY_NORC
396,039	152	LOAD_FAST_C_NORC
400,000	160	INCA_FLOAT_TRUE_DIVIDE_NORC_TOS
396,039	168	YIELD_VALUE
399,999	184	JUMP_ABSOLUTE

Table 2. Dynamic bytecode frequency for an anonymous list comprehension of benchmark program `fasta`

Frequency	Offset	Instruction Identifier
6,600	16	LOAD_FAST_A
402,667	24	FOR_ITER_RANGEITER
396,002	32	STORE_FAST_B
399,960	40	LOAD_DEREF
396,001	48	LOAD_DEREF_NORC
396,000	56	LOAD_DEREF_NORC
396,000	64	LOAD_DEREF_NORC
396,000	72	FAST_PYFUN_DOCALL_ZERO_NORC
395,999	80	FAST_C_VARARGS_TWO_RC_TOS_ONLY
395,999	88	INCA_LIST_SUBSCRIPT
395,998	96	LIST_APPEND
395,998	104	JUMP_ABSOLUTE
6,600	112	RETURN_VALUE

scheduling algorithm only has to consider the arrangement of 27 instructions which constitute almost the complete computation of the `fasta` benchmark. If all 27 instructions are distinct, the optimal interpreter instruction scheduling consists of these 27 instructions being arranged sequentially and compiled adjacently, according to the order given by the corresponding kernel. However, because of the repetitive nature of load and store instructions for a stack-based architecture, having a large sequence of non-repetitive instructions is highly unlikely. Therefore, our interpreter instruction scheduling algorithm should be able to deal with repeating sub-sequences occurring in a kernel. In fact, our `fasta` example contains repetitions, too. The `genRandom` function:

- `LOAD_FAST_A_NORC`, at offsets: 64, 136.
- `LOAD_FAST_NORC`, at offsets: 72, 88.

The anonymous list comprehension contains the following repetition:

- `LOAD_DEREF_NORC`, at offsets: 48, 56, 64.

Fortunately, however, only single instructions instead of longer sub-sequences repeat. Therefore, for the `fasta` case, an optimal interpreter instruction scheduling can easily be computed. We generate a new optimized instruction set from the existing instruction set and move instructions to the head of the dispatch loop according to the instruction order in the kernels. We maintain a list of all instructions that have already been moved, and whenever we take a new instruction from the kernel sequence, we check whether it is already a member of that remembered list. Thus, we ensure that we do not re-reorder already moved instructions. For our `fasta` example, this means that for all of the repeated instructions, we only generate them when we process them for the first time, i.e., only at the first offset position for all occurrences. Consequently, interpreter instruction scheduling generates long chains of subsequently processed instruction sequences that correspond extremely well to the major instruction sequences occurring in the `fasta` benchmark. In fact, we report our highest speedup by a factor of 1.142 for this benchmark.

Thus, if we have an interpreter with many instructions—such as interpreters doing extensive quickening based purely interpretative optimizations, such as inline caching via quickening [4], or eliminating reference counts with quickening [3]—we can reduce potential instruction cache misses using interpreter instruction scheduling.

3 Implementation

The implementation includes all details necessary to implement interpreter instruction scheduling. First, we present an in-depth discussion of how to deal with sub-sequences and why we are interested in them (Section 3.1). Next, we are going to explain how to compile an optimized instruction arrangement with `gcc` (Section 3.2).

3.1 Scheduling in the Presence of Repeating Sub-sequences

As we have seen in the previous section (cf. Section 2), not all kernels contain repeating sub-sequences. However, all larger program traces are likely to contain sub-sequences, or at least similar sequences of instructions. Thus the overall distance of a kernel K depends substantially on the distance of its sub-sequences. We encode the whole trace into a graph data structure and will create an interpreter instruction schedule that contains the most frequent sequences.

In order to demonstrate this approach, we introduce another example from the computer language benchmarks game [6], viz. the `nbody` benchmark. Running the `nbody` benchmark with an argument of 50,000 on top of our instrumented Python interpreter for dynamic bytecode frequency analysis results in the execution of 68,695,970 instructions, of which 99.9% or 68,619,819 instructions are executed in the `advance` function. Its kernel K consists of a trace of 167 instructions, distributed among just 29 instructions.

Creating an instruction schedule using the simple algorithm of the previous section (cf. Section 2) is going to be sub-optimal, since it does not account for representation of repeating sub-sequences. To properly account for these sub-sequences, we create a weighted directed graph data-structure of all 29 instructions as nodes. Since the kernel is a sequence of instructions, we create an edge in this digraph for each pair of adjacent instructions. Whenever we add an edge between two nodes that already exists, we increment the weight of the already existing edge, instead of adding another one. (cf. Figure 1)

Once we have such a digraph, we obtain an instruction schedule with a minimum distance the following way. Given we have some node, our algorithm always chooses the next node by following along the edge with the highest weight. First, we create a list named `open` that contains tuples of nodes and the collective weight of edges leading to that node. We sort the `open` list in descending order of the collective weight component. Because we actually only need the collective weight for choosing the first node and ensuring that we process all nodes, we can now safely zero out all weights of the tuples in the `open` list. Then, we start the actual algorithm by fetching and removing the first tuple element from the `open` list; we assign the node part to n and ignore the weight. Next, we check whether n has already been scheduled by checking whether the `schedule` list contains n. If it has not been scheduled yet, we append it to the `schedule` list. Then, we start looking for a successor node m. We process the successor nodes by having them sorted in descending order of the edge-weight associated between nodes n and m. We repeatedly fetch nodes m from the list of successors until we find a node that has not already been scheduled or the list is finally empty. If we do not find a node m, then we have to restart by fetching the next node from the `open` list. If we find a node m, then we add the reachable nodes from n to m to the `open` list and sort it, such that the successors with the highest weight will be chosen as early as possible. Next we assign m to n and restart looking for m's successors.

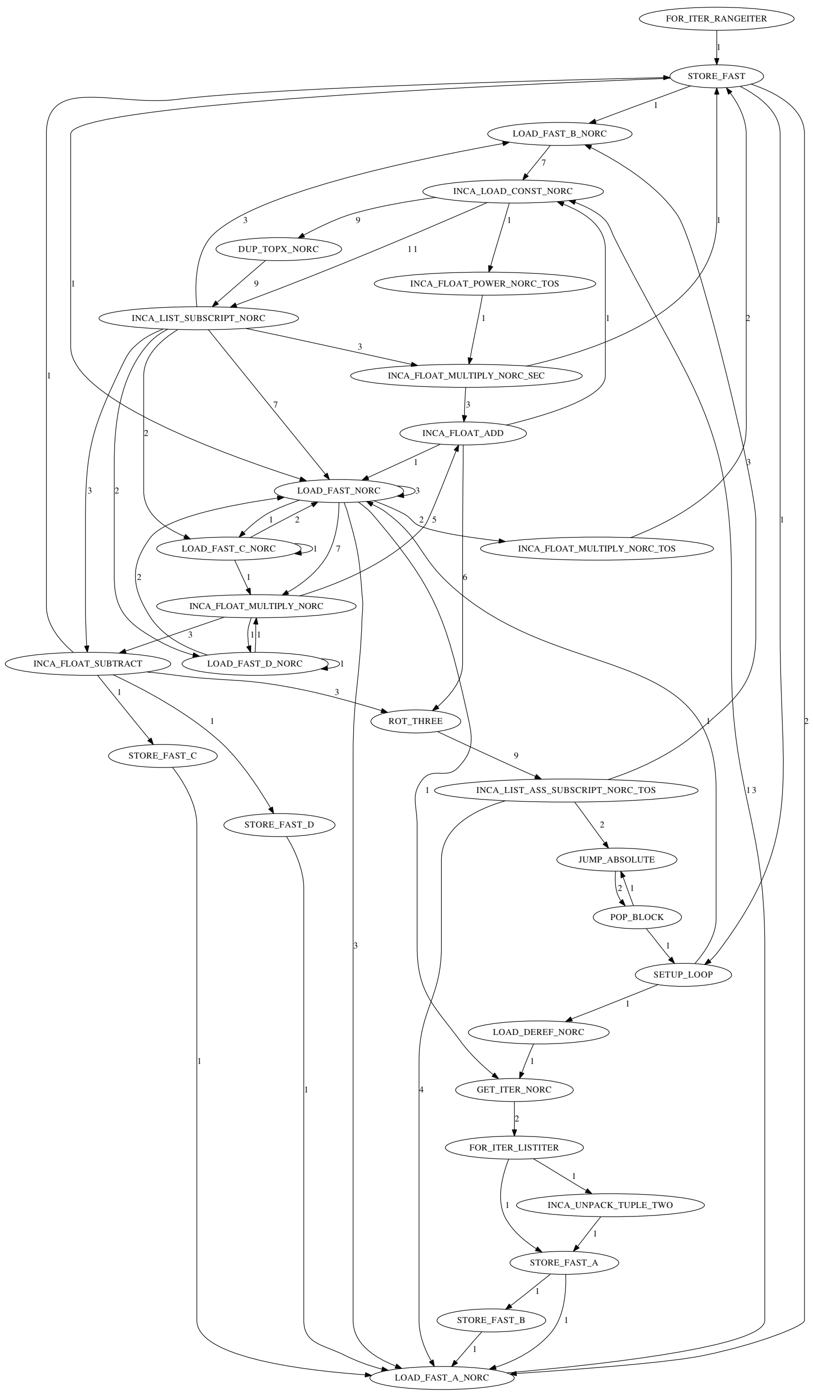

Fig. 1. Instructions of kernel for `nbody` benchmark

The following listing shows our implementation in Python, followed by a detailed description of how it works:

```
def rsorted(dict):
    """Sorts dictionary entries by their numeric values in
       descending order.
    """
    return sorted(dict.items(), key=lambda (key, value): -value)

def schedule_instr(graph):
    schedule= []

    open= rsorted(graph.most_frequent_vertices())
    ## open is a list of tuples (node, number of edges)
    open= [ (node, 0) for (node, edge_count) in open ]
    ## now, we erased the number of edges, such that when we add
    ## the reachable destination nodes for the current source
    ## node, and we sort the <open> list, the node that can be
    ## reached with the edge having the highest weight will be
    ## the first element on the <open> list

    while open:
        ## fetch the tuple, ignore the number of edges
        (n, _)= open.pop(0)
        while n:
            if n not in schedule:
                schedule.append( n )

            reachable= rsorted(n.get_destinations())
            if not reachable:
                break

            ## find reachable nodes that have not been scheduled yet
            (m, _)= reachable.pop(0)
            while m in schedule:
                if len(reachable) > 0:
                    (m, _)= reachable.pop(0)
                else:
                    m= None

            if m:
                n= m                                  ## assign successor node
                open= rsorted( reachable + open )     ## keep reachable nodes sorted
            else:
                break

    return schedule
```

Running this algorithm on our kernel for the nbody benchmark computes the schedule presented in Table 3.

3.2 Compilation of the Interpreter

Once we have computed a schedule of interpreter instructions, we need to compile the interpreter with that schedule. We have extended our interpreter generator from our previous work ([3]) to generate all instructions, not just the optimized derivatives. Since we already have a schedule, it is straightforward to generate an optimized instruction set from the standard instruction set. We just process the schedule in order, move instructions from the old instruction set, and add these instructions to the optimized instruction set. Once, we have processed the plan, we just add the remaining instructions to the new optimized instruction set.

Table 3. Interpreter Instruction Schedule for the `nbody` benchmark

No.	Instruction	No.	Instruction
1	INCA_LOAD_CONST_NORC	16	LOAD_DEREF_NORC
2	INCA_LIST_SUBSCRIPT_NORC	17	GET_ITER_NORC
3	LOAD_FAST_NORC	18	FOR_ITER_LISTITER
4	INCA_FLOAT_MULTIPLY_NORC	19	STORE_FAST_A
5	INCA_FLOAT_ADD	20	STORE_FAST_B
6	ROT_THREE	21	STORE_FAST_D
7	INCA_LIST_ASS_SUBSCRIPT_NORC_TOS	22	INCA_FLOAT_MULTIPLY_NORC_SEC
8	LOAD_FAST_A_NORC	23	JUMP_ABSOLUTE
9	DUP_TOPX_NORC	24	POP_BLOCK
10	LOAD_FAST_B_NORC	25	LOAD_FAST_C_NORC
11	INCA_FLOAT_SUBTRACT	26	LOAD_FAST_D_NORC
12	STORE_FAST_C	27	INCA_UNPACK_TUPLE_TWO
13	INCA_FLOAT_MULTIPLY_NORC_TOS	28	INCA_FLOAT_POWER_NORC_TOS
14	STORE_FAST	29	FOR_ITER_RANGEITER
15	SETUP_LOOP		

There are compiler optimizations that can change the instruction order as computed by our interpreter instruction scheduling. First of all, basic block reordering as done by `gcc` 4.4.3 will eliminate our efforts by reordering basic blocks after a strategy called "software trace-cache" [10]. Fortunately, we can switch this optimization off, by compiling the source file that contains the interpreter dispatch routine with the additional flag `-fno-reorder-blocks`. However, the instructions are still entangled in a switch-case statement. Since it is possible for a compiler to re-arrange case statements, we decided to remove the switch-case statement from the interpreter dispatch routine as well. Because our interpreter is already using the optimized threaded code dispatch technique [2], removing the switch-case statement is simple. However, we stumbled upon a minor mishap: `gcc` 4.4.3 now decides to generate two jumps for every instruction dispatch. Because the actual instruction-dispatch indirect-branch instruction is shared by all interpreter instruction implementations, available expression analysis indicates that it is probably best to generate a direct jump instruction back to the top of the dispatch loop, directly followed by an indirect branch to the next instruction. On an Intel Nehalem (i7-920), `gcc` 4.4.3 generates the following code at the top of the dispatch loop:

```
.L1026:
    xorl %eax, %eax
.L1023:
    jmp  *%rdx
```

And a branch back to the label `.L1026` at the end of every instruction:

```
    movq opcode_targets.14198(,%rax,8), %rdx
    jmp  .L1026
```

Of course, this has detrimental effects on the performance of our interpreter. Therefore, we use `gcc`'s `-save-temps` switch while compiling the interpreter routine with `-fno-reorder-blocks` to retrieve the final assembler code emitted by the compiler. We implemented a small fix-up program that rebuilds the basic blocks and indexes their labels from the interpreter's dispatch routine (`PyEval_EvalFrameEx`), determines if jumps are transitive, i.e., to some basic-block that itself contains only a jump instruction, and copies the intermediate block over the initial jump instruction. Thus, by using this fix-up program, we obtain the original threaded-code jump sequence:

```
movq opcode_targets.14198(,%rax,8), %rdx
xorl %eax, %eax
jmp  *%rdx
```

Finally, we need to assemble the fixed-up file into the corresponding object file and link it into the interpreter executable.

4 Evaluation

We use several benchmarks from the computer language benchmarks game [6]. We ran the benchmarks on the following system configurations:

– Intel i7 920 with 2.6 GHz, running Linux 2.6.32-25 and `gcc` version 4.4.3. (Please note that we have turned off Intel's Turbo Boost Technology to have a common hardware baseline performance without the additional variances immanently introduced by it [7].)
– Intel Atom N270 with 1.6 GHz, running Linux 2.6.28-18 and `gcc` version 4.3.3.

We used a modified version of the `nanobench` program of the computer language benchmark game [6] to measure the running times of each benchmark program. The `nanobench` program uses the UNIX `getrusage` system call to collect usage data, for example the elapsed user and system times as well as memory usage of a process. We use the sum of both timing results, i.e., elapsed user and system time as the basis for our benchmarks. Because of cache effects, and unstable timing results for benchmarks with only little running time, we ran each program 50 successive times and use arithmetic averages over these repetitions for our evaluation. Furthermore, we compare our improvements to the performance of our most recent interpreter without interpreter instruction selection [3].

Figure 2 contains our results of running comparative benchmarks on the Intel Nehalem architecture. For each of our benchmarks, we generate a dedicated interpreter that has an optimized interpreter instruction schedule based on the profiling information obtained by running this benchmark program. We normalized our results by those of our previous work [3], such that we can tell whether interpreter instruction scheduling improves performance of an interpreter with an extended instruction set (our interpreter has 394 instructions). Similarly, Figure 3 contains our results of running this comparative setup on our Intel Atom CPU based system. First, lets discuss the results we obtained on the Intel Atom system (cf. Figure 3). We obtained the maximum speedup by a factor of 1.1344 when running the `spectralnorm` benchmark, the minimum speedup by a factor

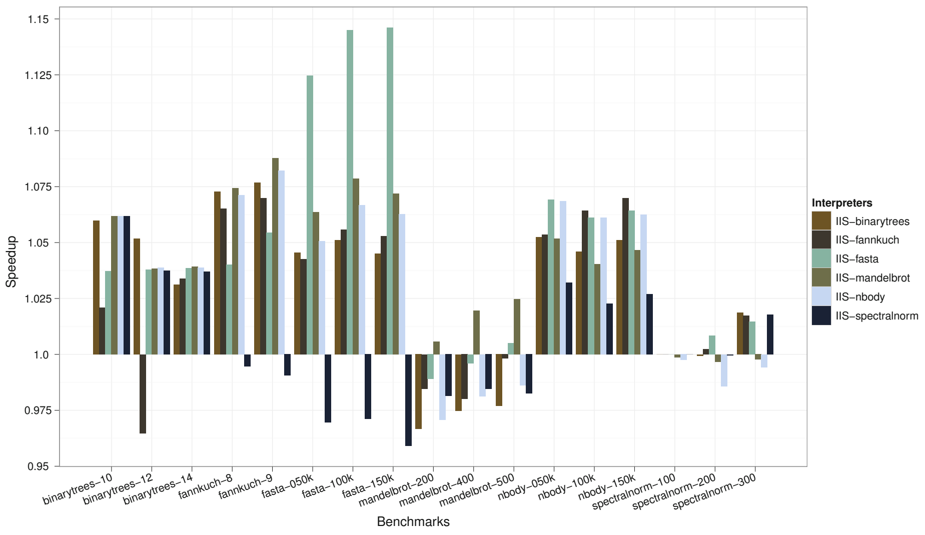

Fig. 2. Comparative benchmark results on the Intel Nehalem CPU

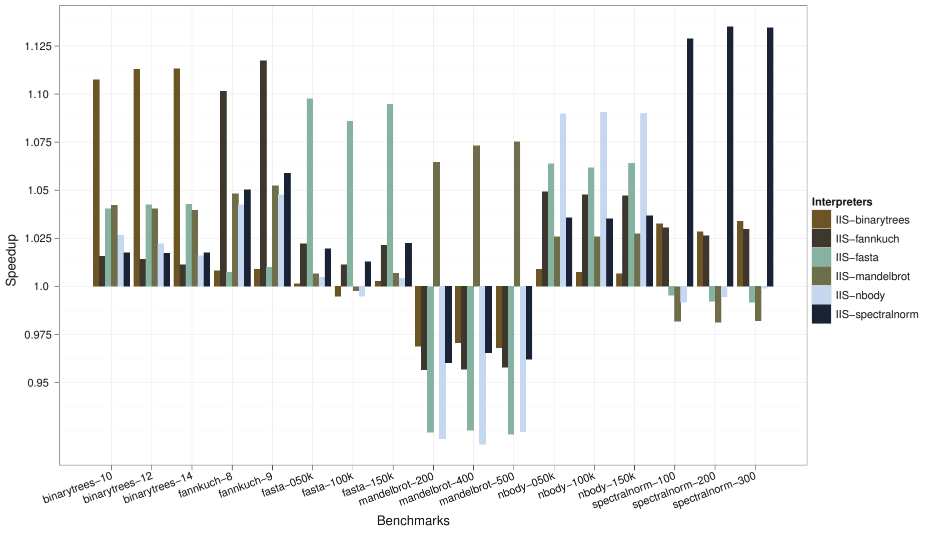

Fig. 3. Comparative benchmark results on the Intel Atom CPU

of 1.0736 when running the `mandelbrot` benchmark, and an average speedup by a factor of 1.1032. The figure clearly indicates that for every benchmark, the interpreter with the instruction scheduling corresponding to that benchmark achieves the highest speedup. Interestingly, most instruction schedules perform better on most benchmarks, with the notable exception being the `mandelbrot` benchmark—a finding that holds true for our results on the Intel Nehalem, too.

Our results on the Intel Nehalem architecture paint a different picture. While the maximum speedup by a factor of 1.142 is higher than the one we report for the Atom CPU, its average speedup by a factor of 1.061 is lower. It is reasonable to assume that this is due to the Nehalem architecture having bigger caches, which affects the performance potential of interpreter instruction scheduling. In addition, there are only two benchmarks, viz. `fasta` and `mandelbrot`, where the interpreter having an optimized instruction schedule for the corresponding benchmark actually perform better than the others. For all other benchmarks, the computed instruction schedule given the profiling information is not optimal, i.e., the schedules computed for some other benchmark allow some of the other interpreters to perform noticeably better. Further investigation is necessary, to identify the cause of this rather surprising finding—particularly in presence of the actually expected findings confirmed on the Atom CPU.

5 Related Work

We group the discussion of related work into two groups, viz., related work on compilers and related work on interpreters. First, we will discuss the related work on compilers.

Pettis and Hansen [9] present their work on optimizing compilers for the Hewlett Packard's PA-RISC architecture. They optimize the arrangement of procedures and basic blocks based on previously obtained profiling information. Interestingly, our reordering algorithm is almost identical to their "algo1" algorithm; they may even be identical, but because no implementation is given, this remains unclear. Another interesting fact is that both our maximum achievable speedups are identical, i.e., both our work achieves a maximum speedup by a factor of 1.14.

More recently, Zhao and Amaral [11] demonstrate algorithms to optimize switch-case computation as well as case-statement ordering in the Open Research Compiler [1]. While both our approaches employ information gathered at run-time, the application scenario is quite different. For instance, their approach focuses on optimizing switch-case statements, and they calculate the order in which they should be generated by their rank according to frequency. In contrast, our work focuses on optimization of interpreters, particularly those without using the switch-case dispatch technique. Because of better instruction cache utilization, we choose to use another algorithm that recognizes the importance of properly covering instruction sequences. So in a sense, the major difference is that their optimization approach focuses on larger compiled programs that use switch-case statements, whereas we recognize the nature of an interpreter, where execution remains within its instruction set *at all times*. Another direct consequence of this fundamental difference is that in an interpreter we are usually not interested in the default case, since this indicates an error, i.e., an unknown opcode, which in practice happens never—the exception being malicious intent of a third party.

As for related work on interpreters, the most important work is by Lin and Chen [8]. Their work is similar to ours, since they show how to partition interpreter instructions to optimally fit into NAND flash pages. Furthermore, they describe that they too use profiling information to decide which combination of interpreter instructions to co-locate on one specific flash page. Their partitioning algorithm pays attention to the additional constraint of NAND flash page size, i.e., their algorithm computes a configuration of interpreter instructions that fits optimally within the flash pages and keeps dependencies between the pages at a minimum. For the context of our work it is unnecessary to superimpose such a constraint to our algorithm. Though, if one were to set the parameter N determining the NAND flash page size of their algorithm to the maximum representable value, all instructions would be packed into just one partition. Then, our algorithms should produce similar interpreter instruction arrangements. Another difference between our respective approaches is that ours operates on a higher level. While they post-process the assembly output generated by `gcc` to enable their optimizations, our approach is based on re-arranging the instruction at the source code level. Though we admittedly have to fix-up the generated assembly file as well, due to the detrimental effects of a misguided optimization. Because of their ties to embedded applications of the technique and its presentation in that context, we think that our presentation is more general in nature. In addition, we complement our work with extensive performance measurements on contemporary non-embedded architectures.

Ertl and Gregg [5] present an in-depth discussion of two interpreter optimization techniques—superinstructions and replication—to improve the branch prediction accuracy and instruction cache utilization of virtual machines. While the optimization technique of replication is not directly related to interpreter instruction scheduling, it improves the instruction cache behavior of an interpreter at the expense of additional memory. The idea of superinstructions is to combine several interpreter instructions into one superinstruction, thus eliminating the instruction dispatch overhead between the single constituents. While this improves branch prediction accuracy, it improves the instruction cache utilization, too: Since all instruction implementations must be copied into one superinstruction, their implementations must be adjacent, i.e., co-located in memory, which is optimal with respect to instruction cache utilization and therefore results in extremely good speedups of up to 2.45 over a threaded-code interpreter without superinstructions. However, superinstructions can only be used at the expense of additional memory, too. Since interpreter instruction scheduling happens at precompile, and compile time respectively, of the interpreter, there are no additional memory requirements—with the notable exception of minor changes because of alignment issues. Because the techniques are not mutually exclusive, using interpreter instruction scheduling in combination with static superinstructions will further improve the performance of the resulting interpreter.

Summing up, the major difference between the related work on compilers and our work is that the former focuses on optimizing elements visible to the compiler, such as procedures, basic blocks, and switch-case statements, whereas our work focuses on re-arranging interpreter instructions—which are transparent to compilers. Related work on interpreters achieves a significantly higher speedup, however, at the expense of additional memory. Our work demonstrates that is possible to improve interpretation speed without sacrificing memory.

6 Conclusion

We present a technique to schedule interpreter instructions at pre-compile time in order to improve instruction cache utilization at run-time. To compute the schedule, we rely on profiling information obtained by running the program to be optimized on an interpreter. From this information, we extract a kernel, i.e., an instruction trace that consumes most of the computational resources, and construct a directed graph of that kernel. We use a simple algorithm that recognizes the importance of repeating sub-sequences occurring in that kernel when scheduling the interpreter instructions, and report a maximum speedup by a factor of 1.142 using this technique.

Future work includes investigation on the effectiveness of other scheduling algorithms—such as implementation of a dynamic programming variant, or comparing the effectiveness of algorithms mentioned in the related work section—, as well as addressing the pending question regarding the optimality of computed schedules. In addition, we are interested in devising a dynamic variant complementing our static interpreter instruction scheduling technique.

Acknowledgments

We express our thanks to Jens Knoop for reviewing early drafts of this paper, as well as to Anton Ertl for providing extensive information concerning related work. Furthermore, we are grateful to the anonymous reviewers for providing helpful remarks to improve the presentation of this paper, and making valuable suggestions for addressing future work. Finally, we thank the Austrian Science Fund (FWF) for partially funding this research under contract number P23303-N23.

References

1. Open Research Compiler (October 2010), `http://ipf-orc.sourceforge.net/`
2. Bell, J.R.: Threaded code. Communications of the ACM 16(6), 370–372 (1973)
3. Brunthaler, S.: Efficient interpretation using quickening. In: Proceedings of the 6th Symposium on Dynamic Languages (DLS 2010), Reno, Nevada, US, October 18, ACM Press, New York (2010)
4. Brunthaler, S.: Inline caching meets quickening. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 429–451. Springer, Heidelberg (2010)
5. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), pp. 278–288. ACM, New York (2003)

6. Fulgham, B.: The computer language benchmarks game, `http://shootout.alioth.debian.org/`
7. Intel: Intel Turbo Boost Technology in Intel Core microarchitecture (Nehalem) based processors. Online (November 2008), `http://download.intel.com/design/processor/applnots/320354.pdf?iid=tech%_tb+paper`
8. Lin, C.-C., Chen, C.-L.: Code arrangement of embedded java virtual machine for NAND flash memory. In: Stenström, P., Dubois, M., Katevenis, M., Gupta, R., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4917, pp. 369–383. Springer, Heidelberg (2008)
9. Pettis, K., Hansen, R.C.: Profile guided code positioning. SIGPLAN Notices 25(6), 16–27 (1990)
10. Ramírez, A., Larriba-Pey, J.L., Navarro, C., Torrellas, J., Valero, M.: Software trace cache. In: Proceedings of the 13th International Conference on Supercomputing (ICS 1999), Rhodes, Greece, June 20-25, pp. 119–126. ACM, New York (1999)
11. Zhao, P., Amaral, J.N.: Feedback-directed switch-case statement optimization. In: Proceedings of the International Conference on Parallel Programming Workshops (ICPP 2005 Workshops), Oslo, Norway, June 14-17, pp. 295–302. IEEE, Los Alamitos (2005)