Fast Query Evaluation with (Lazy) Control Flow Compilation

Remko Tronçon, Gerda Janssens, and Henk Vandecasteele

K.U.Leuven – Department of Computer Science
{remko,gerda,henkv}@cs.kuleuven.ac.be

Abstract. Learning algorithms such as decision tree learners dynamically generate a huge amount of large queries. Because these queries are executed often, the trade-off between meta-calling and compiling & running them has been in favor of the latter, as compiled code is faster. This paper presents a technique named *control flow compilation*, which improves the compilation time of the queries by an order of magnitude without reducing the performance of executing the queries. We exploit the technique further by using it in a just-in-time manner. This improves performance in two ways: it opens the way to incremental compilation of the generated queries, and also gives potentially large gains by never compiling dynamically unreachable code. Both the implementation of (lazy) control flow compilation and its experimental evaluation in a real world application are reported on.

1 Introduction

In previous work, *query packs* [5] were introduced as an efficient method for executing a set of similar queries. A query pack is basically the body of a rule with no arguments, with a huge number of literals and disjunctions. The query pack execution mechanism deals with the disjunctions in a special way, namely by avoiding a branch which already succeeded before. Query packs are interesting in the context of several types of learners, including first order decision trees [4, 8], first order pattern discovery [7], and rule-based learners [9–11]. These query packs can be executed in ilProlog [1], a WAM [17] based Prolog algorithm with special support for Inductive Logic Programming (ILP). They are generated dynamically by the ILP algorithm, compiled by the underlying Prolog system, after which the compiled code is executed on a dataset (which is actually a large collection of different logic programs). This is not an unreasonable approach, and we have indeed measured large speedups in ILP systems based on this approach [5].

However, measurements indicate that the compilation of a complete query pack is a very costly operation, and sometimes causes more overhead than what is gained later when executing the compiled version (instead of meta-calling the query pack). Also, some goals of a generated query pack fail on each example, meaning that the part of the query following that goal was compiled in vain.

Therefore, we started investigating *control flow compilation* [14] as a more flexible and faster alternative to classical compilation. This is basically a hybrid between compilation and meta-call. While classical WAM code for a compiled query contains both instructions encoding the calls to predicates and instructions dealing with the control flow (e.g. selection of clauses/branches), control flow compilation only generates the control flow instructions and uses meta-call to deal with the calls. The resulting compilation is much less expensive, and the generated code is as fast as classical compiled code. Moreover, this technique allows a *lazy* compilation scheme, which only compiles a part of a query when it is actually executed. Not only does this avoid redundant compilation, lazy compilation is also a first step towards supporting the incremental generation of queries and query packs. Introducing laziness in the full WAM compiler is not straightforward, because its variable classification and allocation scheme is optimized towards the situation where all the code is known. Moreover, because query packs are very large, specialized techniques for dealing with its variables are needed [16], which complicates matters even further. On the other hand, the control flow compiler does not have to deal with the variables in the query packs, and can therefore compile an increment to a query almost independently of the previous query.

In this paper, we present control flow compilation and its lazy variant as an innovative way to deal with compilation overhead and to achieve faster execution of queries. We illustrate its advantages with real life examples. Lazy control flow compilation is also an enabling technology for incrementality in the ILP process of query (pack) generation and execution. In principle, any application depending on an efficient meta-call could benefit from this technique. Nevertheless, the focus of this work is on ILP.

Control flow compilation is described and evaluated in Section 3. Based on control flow compilation, we develop a lazy compilation scheme for queries containing conjunctions and disjunctions in Section 4. (Lazy) control flow compilation is extended to query packs in Section 5. We evaluate our approaches using both artificial and real world experiments. Finally, Section 6 concludes and discusses future work.

We assume that the reader is familiar with the WAM [2].

2 Background: Queries in ILP

We start by sketching a particular setting in which our work is relevant, namely the execution of queries in Inductive Logic Programming. The goal of Inductive Logic Programming is to find a theory that best explains a large set of data (or examples). In the ILP setting at hand, each example is a logic program, and the logical theory is represented as a set of logical queries. The ILP algorithm searches for these queries using generate-and-test: generated queries are run on sets of examples; based on the failure or success of these queries, only the ones

with the 'best' results[1] are kept and are extended (e.g. by adding literals). These extended queries are in turn tested on each example, and this process continues until a satisfactory query (or set of queries) describing the examples has been found.

At each iteration of the algorithm, a set of queries is executed on a large set of logic programs (the examples). Since these queries are the result of adding different literals to the end of another query, the queries in this set have a lot of common prefixes. To avoid repeating the common parts by executing each query separately, the set of queries is transformed into a special kind of disjunction: a *query pack* [5]. For example, the set of queries

```
?- a, b, c, d.
?- a, b, c, e.
?- a, b, f, g.
```

is transformed into the query

```
?- a, b, ( (c,(d;e)) ; f,g ).
```

by applying left factoring on the initial set of queries. However, because only the success of a query on an example is measured, the normal Prolog disjunction might still cause too much backtracking. So, for efficiency reasons the ';'/2 is given a slightly different semantics in query packs: it cuts away branches from the disjunction as soon as they succeed. Since each query pack is run on a large set of examples, a query pack is first compiled, and the compiled code is executed on the examples. This compiled code makes use of dedicated WAM instructions for the query pack execution mechanism. More details can be found in [5].

3 Control Flow Compilation

3.1 Technology

Executing compiled queries instead of meta-calling them results in considerable speedups. However, compilation of a query can take as much time as the execution of the query on all examples. Moreover, classical compilation makes it very difficult to exploit the incremental nature of query generation in the ILP setting. It would require a tight coupling between the generation of the queries and their compilation. Also, assignment of variables to environment slots uses a classification of variables which assumes that all the code is known at compile time. This motivated the preliminary study of alternatives for compile & run in [14]. The most interesting alternative is *control flow compilation*, which is a hybrid between meta-calling and compiling a query. In this section, we introduce control flow compilation for queries whose bodies consist of conjunctions and

[1] which queries are best depends on the ILP algorithm. In the case of classification, the information gain can be used as a criterium, whereas in the case of regression, the reduction of variance is often used.

disjunctions. Control flow compilation for query packs is discussed in Section 5.

The essential difference between classical compilation and control flow compilation is the sequence of instructions generated for setting up and calling a goal. Instead of generating the usual WAM `put` and `call` instructions, the latter generates one new `cf_call` instruction, whose argument points to a heap data structure (the goal) that is meta-called. Hence, control flow code only contains the control flow instructions (`try`, `retry`, ...) and `cf_call` (and `cf_deallex`) instructions.

For example, control flow compiling the query

```
q :- a(X,Y), ( b(Y,Z) ; c(Y,Z), d(Z,U); e(a,Y) ).
```

results in the code in the left part of Figure 1. Note that the query itself is a term on the heap, and that we use &a(X,Y) to represent the pointer to its subterm a(X,Y). On the right of Figure 1 is the classical compiled code for the same query. Before calling each goal, the compiled code first sets up the arguments to

```
q :- a(X,Y), ( b(Y,Z) ; c(Y,Z), d(Z,U); e(a,Y) ).
```

Control flow code	Compiled code
`allocate 2`	`allocate 4`
	`bldtvar A1`
	`putpvar Y2 A2`
`cf_call &a(X,Y)`	`call a/2`
`trymeorelse L1`	`trymeorelse L1`
	`putpval Y2 A1`
	`bldtvar A2`
`cf_deallex &b(Y,Z)`	`deallex b/2`
`L1: retrymeorelse L2`	`retrymeorelse L2`
	`putpval Y2 A1`
	`putpvar Y3 A2`
`cf_call &c(Y,Z)`	`call c/2`
	`putpval Y3 A1`
	`bldtvar A2`
`cf_deallex &d(Z,U)`	`deallex d/2`
`L2: trustmeorelsefail`	`trustmeorelsefail`
	`putpval Y2 A2`
	`put_atom A1 a`
`cf_deallex &e(a,Y)`	`deallex e/2`

Fig. 1. Control flow compiled code vs. classical compiled code.

the goal, whereas the control flow compiled code uses a reference to the subterm of the query to indicate the goal that is called. One important aspect is that the control flow code saves emulator cycles, because it contains no instructions

related to the arguments of the goals that are called. Moreover, the absence of this kind of instructions is very interesting for the lazy compilation we have in mind. Suppose that we want to extend a query by adding a disjunction after its last call (e.g. refining e(a,Y) into e(a,Y),(f(Y,Z);g(Y,U),h(U,V))); within the control flow compilation scheme, it is possible to extend the existing code just by adding more control flow instructions at the end, without the usual compilation issues concerning the variables.

Contrary to compiled code, control flow code cannot exist on its own, since it contains external references to terms on the heap. This introduces some memory management issues: (1) these terms have to be kept alive as long as the control flow compiled code exists; (2) when these terms are moved to another place in memory (e.g. by the garbage collector), the references in the code must be adapted as well.

3.2 Evaluation

For evaluating our approach, we added control flow compilation to the ilProlog system [1]. During the experiments, the heap garbage collector was deactivated, as it does not yet take into account the control flow code. The experiments were run on a Pentium III 1.1 GHz with 2 GB main memory running Linux under a normal load.

Two kinds of experiments are discussed: the benchmarks in Table 1 show the potential gain in an artificial setting, whereas the results in Table 2 are obtained from a real world application.

	(5,5,4)		(10,5,4)		(5,10,4)		(10,10,4)		(5,5,6)	
	comp	exec	comp	exec	comp	exec	comp	exec	comp	exec
control flow	25	0.13	52	0.25	390	4.07	735	7.73	682	7.03
compile & run	322	0.28	663	0.48	4676	5.49	11856	9.18	11099	9.32
meta-call	-	2.1	-	3.79	-	31.73	-	58.83	-	58.43

Table 1. Experiments for artificial disjunctions. (timings in milliseconds)

The artificially generated queries in Table 1 have the following parameters:

- *g*: the number of goals in a branch,
- *b*: the branching factor in a disjunction,
- *d*: the nesting depth of disjunctions.

For example, for the values (2,3,1) for (*g*,*b*,*d*) we generate the query *a(A,B,C), a(C,D,E), (a(E,F,G), a(G,H,I); a(E,J,K), a(K,L,M); a(E,N,O), a(O,P,Q))*. For (1,2,2), the generated query has nested disjunctions: *a(A,B,C), (a(C,D,E), (a(E,F,G) ; a(E,H,I)) ; a(C,J,K), (a(K,L,M) ; a(K,N,O)))*. The definition for

a/3 is simply *a(_,_,_)*. These queries have the same structure as query packs: disjunctions obtained from left factoring a set of conjunctions. The different values of (g,b,d) can be found in the upper row of the table. We report on the following three alternatives:

- *control flow*: the query is compiled using the control flow approach before it is executed.
- *compile & run*: the query is compiled using the classical WAM before it is executed.
- *meta-call*: the query is meta-called (no compilation at all).

The *comp* column gives the compilation time, while the *exec* column gives the execution time of a single execution of a query.

The control flow compilation is definitely better than compile & run: the compilation times are improved by one order of magnitude, while the execution times are also better. The compilation in the control flow approach is much faster because it does not need to perform expensive tasks such as assigning variables to environment slots. The better execution times are explained by the fact that only one emulation cycle per call is needed as no arguments have to be put in registers. Doubling the g parameter more or less doubles the timings. For larger queries, namely for (10,10,4) and (5,5,6), control flow compilation becomes a factor 15 faster than compile & run. If the query is executed a number of times, meta-call is outperformed by control flow compilation (e.g. for (5,5,4), this number is 13). Since in ILP, each query is run on a significant number of examples, these results are very promising.

	ACE:muta		ACE:bongard		ACE:carcino	
	Timings (seconds)					
	comp	exec	comp	exec	comp	exec
control flow	0.11	0.17	0.7	19.88	2.91	46.52
compile & run	0.24	0.24	3.13	19.46	16.81	44.45
meta-call	-	0.26	-	22.41	-	83.74
	Benchmark Characteristics					
number of queries	2021		9335		48399	
average runs/query	69.51		244.77		103.07	

Table 2. Experiments for conjunctions from a real world application.

The real world experiment consists in running the TILDE algorithm [4] from the ILP system ACE [1] on three well-known datasets from the ILP community: Mutagenesis [13], Bongard [6] and Carcinogenesis [12]. During the execution of TILDE, queries are subsequently generated, and every query needs to be run on a subset of the examples. These queries contain only conjunctions; disjunctions

are dealt with as query packs in Section 5. Table 2 compares again control flow compilation with compile & run and meta-call. Times are given in seconds. For each data set, *comp* gives the total compilation time (namely the time for compiling all the queries generated by TILDE) and *exec* the total execution time (namely the time to execute all the (compiled) queries). For each dataset, the lower part of Table 2 also gives the number of queries generated and the average number of runs per query.

In the TILDE runs, control flow compilation gains a factor 2 to 6 with respect to usual compilation. Control flow compiled code outperforms classical compiled code for the Mutagenesis dataset, but is about 5% slower for Carcinogenesis (which is still acceptable). When we consider the total time (namely *comp* + *exec*), control flow compilation is clearly the best alternative out of the three for Carcinogenesis. For Bongard, control flow compilation is slightly faster than the other two, which are comparable. Because Mutagenesis has relatively small queries which are run infrequently, meta-call performs best for this dataset.

The results are more pronounced for the artificial benchmarks than for the TILDE ones for several reasons. The artificial queries are longer than the typical TILDE queries; making the artificial queries shorter makes the timings unreliable. During the artificial benchmarks, the time spent in the called goals is very small (only `proceed`), whereas in the TILDE experiments much more time is spent in the predicates, and as such the effect of control flow on the *exec* timing decreases. Another observation is that control flow code uses pointers to the heap, and as the heap garbage collection is currently deactivated, the heap contains all the queries ever generated. This is bad for locality: we have indeed observed that locality can have a large impact on the execution time in the case of control flow compilation. We expect that as soon as the heap garbage collector is adapted and is activated again, the execution times will improve. This line of reasoning is compatible with the fact that the number of queries in Mutagenesis is relatively small, such that locality is better and thus the control flow *exec* timing is better than for normal compilation. Finally, it is important to note that, while meta-call outperforms the other approaches for one of the datasets, its speedup will have to be sacrificed when we want to benefit from removing branches that already succeeded in the query packs approach.

The main goal of control flow compilation was to have a flexible scheme for introducing lazy compilation for query packs, without slowing down execution itself. Our experiments prove that control flow compilation achieves this goal: if the execution times are slower, it is within an acceptable range of 5%, and in all our benchmarks the loss is compensated by the order of magnitude that can be gained for the compilation.

4 Lazy Control Flow Compilation

4.1 Technology

In [3], *lazy compilation* is identified as a kind of *just-in-time* (JIT) compilation or *dynamic compilation*, which is characterized as translation which occurs after a program begins execution. In this paper, we present lazy variants of control flow compilation. The requirement in [3] that the compiler used for JIT compilation should be fast enough is satisfied by our control flow compiler. Our lazy variant implicitly calls the control flow compiler when execution reaches a part of the query that is not yet compiled. As before, we restrict the discussion in this section to queries with conjunctions and disjunctions; the extension to query packs is presented in Section 5.

As with normal control flow compilation, the query is represented by a term on the heap. We introduce a new WAM instruction `lazy_compile`, whose argument is a pointer to the term on the heap that needs compiling when execution reaches this instruction.

Consider the query q :- a(X,Y), b(Y,Z). The initial lazy compiled version of q is

```
allocate 2
lazy_compile &(a(X,Y),b(Y,Z))
```

The `lazy_compile` instruction points to a conjunction: its execution replaces itself by the compiled code for the first conjunct, namely a `cf_call`, and adds for the second conjunct another `lazy_compile` instruction, resulting in:

```
allocate 2
cf_call &a(X,Y)
lazy_compile &b(Y,Z)
```

The execution continues with the newly generated `cf_call` instruction as is expected. After the next execution of `lazy_compile`, the compiled code is equal to code generated without laziness:

```
allocate 2
cf_call &a(X,Y)
cf_deallex &b(Y,Z)
```

Note that lazy compilation overwrites the `lazy_compile` instruction with a `cf_` instruction, and that once we have executed the query for the first time completely, the resulting code is the same as the code produced by non-lazy control flow compilation.

Now, consider the lazy compilation of the query from Figure 1:

```
q :- a(X,Y), ( b(Y,Z) ; c(Y,Z), d(Z,U); e(a,Y) ).
```

Initially, the code is

```
allocate 2
lazy_compile &(a(X,Y),(b(Y,Z);c(Y,Z),d(Z,U);e(a,Y)))
```

The `lazy_compile` changes the code to:

```
allocate 2
cf_call &a(X,Y)
lazy_compile &(b(Y,Z);c(Y,Z),d(Z,U);e(a,Y))
```

Now, `lazy_compile` will compile a disjunction. Where normal (control flow) compilation would generate a `trymeorelse` instruction, we generate a lazy variant of this. The `lazy_trymeorelse` instruction has as its argument the second part of the disjunction, which will be compiled upon failure of the first branch. The instruction is immediately followed by the code of the first branch, which is initially again a `lazy_compile`:

```
allocate 2
cf_call &a(X,Y)
lazy_trymeorelse &(c(Y,Z),d(Z,U);e(a,Y))
lazy_compile &b(Y,Z)
```

Execution continues with the `lazy_trymeorelse`: a special choice point is created such that on backtracking the remaining branches of the disjunction will be compiled in a lazy way. To achieve this, the failure continuation of the choice point is set to a new `lazy_disj_compile` instruction, which behaves similarly to `lazy_compile`. Then, execution continues with the first branch:

```
allocate 2
cf_call &a(X,Y)
lazy_trymeorelse &(c(Y,Z),d(Z,U);e(a,Y))
cf_deallex &b(Y,Z)
```

Upon backtracking to the special choice point created in `lazy_trymeorelse`, the `lazy_disj_compile` instruction continues compilation, and replaces the corresponding `lazy_trymeorelse` by a `trymeorelse` instruction with as argument the address of the code to be generated:

```
    allocate 2
    cf_call &a(X,Y)
    trymeorelse L1
    cf_deallex &b(Y,Z)
L1: lazy_retrymeorelse &(e(a,Y))
    lazy_compile &(c(Y,Z),d(Z,U))
```

Here, `lazy_retrymeorelse` – the lazy variant of `retrymeorelse` – behaves similar to `lazy_trymeorelse`, but instead of creating a special choice point, it alters the existing choice point. It is immediately followed by the code of the next part of the disjunction, which after execution looks as follows:

```
        allocate 2
        cf_call &a(X,Y)
        trymeorelse L1
        cf_deallex &b(Y,Z)
    L1: lazy_retrymeorelse  &(e(a,Y))
        cf_call &c(Y,Z)
        cf_deallex &d(Z,U)
```

Upon backtracking, `lazy_retrymorelse` is overwritten, and a `trustmeorelse` is generated for the last branch of the disjunction, followed by a `lazy_compile` for this branch:

```
        allocate 2
        cf_call &a(X,Y)
        trymeorelse L1
        cf_deallex &b(Y,Z)
    L1: retrymeorelse L2
        cf_call &c(Y,Z)
        cf_deallex &d(Z,U)
    L2: trustmeorelsefail
        lazy_compile &e(a,Y)
```

After the execution of the last branch, we end up with the full control flow code.

The lazy compilation as we described it proceeds from goal to goal. Other granularities have been implemented and evaluated as well (see Table 3):

- *Per conjunction:* All the goals in a conjunction are compiled at once. This avoids constant switching between the compiler and the execution by compiling bigger chunks.
- *Per disjunction:* All the branches of a disjunction are compiled at once up to the point where a new disjunction occurs. This approach is reasonable, because all branches of a disjunction will by tried (and thus compiled) eventually.

Besides the overhead of switching between compiler and execution, these approaches might also generate different code depending on the execution itself. When a goal inside a disjunction fails, the next branch of the conjunction is executed, and newly compiled code is inserted at the end of the existing code. When in a later stage the same goal succeeds, the rest of the branch is compiled and added to the end of the code, and a jump to the new code is generated. These jumps cost extra emulator cycles and decrease locality of the code. Lazy compilation per goal can in the worst case have as many jumps as there are goals in the disjunctions. Compiling per conjunction can have as many jumps as there are disjunctions. If a disjunction is completely compiled in one step, each branch of the disjunction ends in a jump to the next disjunction.

4.2 Evaluation

The experiments of Table 3 use some of the artificial benchmarks from Table 1. Timings (in milliseconds) are given for the different settings of the lazy compilation. The timings report the time needed for one execution of the query, thus including the time of its lazy compilation. The last line gives the times for the non-lazy control flow compilation[2]. Lazy compilation per goal clearly has a substantial overhead, whereas the other settings have a small overhead. We also measured the execution times for the three lazy alternatives once they are compiled: they were all equal, and are therefore not included in the table.

	(5,5,4) cexec	(10,5,4) cexec
per goal	55	111
per conj	34	60
per disj	32	59
control flow	28	59

Table 3. Lazy compilation for several kinds of disjunctions. (timings in milliseconds)

The main message here is that the introduction of laziness in the control flow compilation does not degrade performance much, and that it opens perspectives for query packs compilation: (1) lazy compilation is fast; (2) in non-artificial benchmarks, some branches will never have to be compiled due to failure of goals, whereas in our artificial setting all goals in the queries succeed; (3) in the long run, it allows incremental compilation: if we would allow open ended queries (queries that end with an uninstantiated call), the ILP system can refine the query later by further instantiating the open end, and lazy compilation will automatically compile the new part of the query when it is reached.

5 Lazy Control Flow Compilation for Query Packs

5.1 Technology

So far, we restricted our (lazy) control flow compilation approach to queries containing conjunctions and 'ordinary' disjunctions. However, the main motivation for this work was optimizing the execution of *query packs* [5]. These query packs represent a set of (similar) queries which are to be executed, laid out in a disjunction. The semantics of this disjunction is implemented by dedicated

[2] Note that these timings are slightly higher than the sum of *comp* and *exec* in Table 1. This is probably due to the fact that both experiments are run in different circumstances with different locality.

WAM instructions [5], as explained in Section 2. These instructions replace the instructions generated for encoding ordinary disjunctions.

Extending control flow compilation to handle these query packs is rather straightforward. The difference between the compilation of disjunctions handled so far and the disjunctions of a query pack is that the dedicated WAM instructions have to be generated as control flow instructions for the disjunctions. Introducing laziness in control flow compilation for query packs requires more changes. Originally, query packs used static data structures which were allocated once, since all the information on the size and contents of these data structures was known at compile time. However, when laziness is introduced, only parts of the query pack are analyzed, and so the data structures need to be dynamic and expandable.

To facilitate the implementation of lazy control flow compilation for query packs, we chose to implement only one of the lazy variants described in Section 4. Since the experiments showed little difference between all the variants (except for lazy compilation per goal), this seems like a reasonable decision. We chose to compile one complete disjunction at a time, because this makes integration with the existing query pack data structures easier.

5.2 Experiments

	ACE:muta	ACE:bongard	ACE:carcino
	Timings (seconds) (comp + exec = cexec)		
control flow	0.13+0.08 = 0.21	1.02 +23.75 = 24.77	2.92+7.07 = 9.99
lazy control flow	0.24	24.0	8.15
compile & run	0.69+0.11 = 0.80	12.27+22.48 = 34.75	47.97+5.24 = 53.21
	Query Pack Characteristics		
Nb. of packs	50	4	28
Nb. of queries	6010	63668	204527
Avg. runs/pack	61.52	723.50	134.67
	Code size reduction with lazy compilation		
Reduction	17.0 %	57.2 %	61.4 %

Table 4. Experiments for query packs from a real world application.

The experiments are again performed with the real world applications from Table 2. Instead of a set of queries (the conjunctions of Table 2), TILDE now generates query packs. These query packs are then compiled and finally executed for a subset of the examples. The use of query packs allows us to set up a larger experiment (in ILP terms: we now use a lookahead of 3 instead of 2), which results in more and longer queries.

The timings in Table 4 are in seconds: for compile & run and control flow, we give the sum of the total compilation time and the total execution time; for lazy control flow compilation, no distinction can be made, and so the total time for compilation and execution is given.

First, we compare control flow compilation with compile & run. For query packs, control flow compilation is also up to an order of magnitude faster than classical compilation, even though the ilProlog system already has a compiler that is optimized for dealing with large disjunctions [16] (in particular for the classification of variables in query packs). The execution times show the same characteristics as in the experiments with the conjunctions in Table 2: control flow has a faster execution in the case of Mutagenesis, whereas in the other two cases it is a bit slower. For the ILP application, the total time must be considered: the total time of control flow is up to a factor 4 faster than compile & run.

Next, Table 3 shows that lazy compilation has some overhead, but we hoped that it would be compensated by avoiding the compilation of failing parts in the query packs. For Bongard and Carcinogenesis, lazy control flow timings are indeed better than for the plain control flow. The information about the code size reduction in the case of the lazy variant confirms the idea that we gain by avoiding the compilation of parts of the query packs that fail for all the examples. Also, the locality is better when less code is generated. Mutagenesis is a smaller benchmark with less code reduction, and so the compilation/execution ratio is large. This explains why the overhead of interleaved compilation with execution is not compensated for.

The resulting timings confirm that lazy control flow compilation is the best approach for query packs.

6 Conclusion and Future Work

This paper presents a new method for faster compilation and execution of dynamically generated queries: control flow compilation is up to an order of magnitude faster than classical compilation, while the execution times are similar. To our knowledge, this is also the first time that lazy compilation (as an instance of just-in-time compilation [3]) is used in the context of logic programming, in particular for queries.

The benefits of control flow compilation versus classical compilation are clear and are confirmed in the context of real world applications from the ILP community. For larger benchmarks, the lazy variant gives the best results in combination with query packs.

For control flow compilation itself, the main future work will consist in extending the garbage collector of the ilProlog system to support control flow compiled code. This extension can be realized within the current garbage collector, and mainly requires a coding effort. We expect that garbage collection

improves the locality and the execution times of queries. It also has to be investigated whether it is interesting to put the control flow code on the heap, thus making code garbage collection of queries a part of the heap garbage collection process.

We also plan to adapt (lazy) control flow compilation to extensions of query packs reported in [15]. We expect control flow compilation to yield the same speedups for these execution mechanisms as for query packs. However, the impact of laziness needs to be investigated.

Finally, this work allows us to investigate how incremental generation and compilation of queries can be supported in an ILP system.

Acknowledgments

Remko Tronçon is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.). This work is partially supported by the GOA 'Inductive Knowledge Bases'. We are indebted to Bart Demoen for his significant contributions to the achievements presented in this paper.

References

1. The ACE data mining system. http://www.cs.kuleuven.ac.be/~dtai/ACE/.
2. H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.
3. J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
4. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
5. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence*, 16:135–166, 2002.
6. L. De Raedt and W. Van Laer. Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
7. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
8. S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 812–819, Cambridge/Menlo Park, 1996. AAAI Press/MIT Press.
9. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
10. J. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
11. A. Srinivasan. The Aleph manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

12. A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.
13. A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
14. R. Tronçon, G. Janssens, and B. Demoen. Alternatives for compile & run in the WAM. In *Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 45–58. University of Porto, 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto, December 2003. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=41065.
15. R. Tronçon, H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens. Query optimization: Combining query packs and the once-tranformation. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations*, pages 105–115, 2003. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40938.
16. H. Vandecasteele, B. Demoen, and G. Janssens. Compiling large disjunctions. In I. de Castro Dutra, E. Pontelli, and V. S. Costa, editors, *First International Conference on Computational Logic : Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 103–121. Imperial College, 2000. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=32065.
17. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.