TLP **7** (4): 481–498, 2007. © 2007 Cambridge University Press
doi:10.1017/S1471068406002894 Printed in the United Kingdom

Fast Frequent Querying with Lazy Control Flow Compilation

REMKO TRONÇON*, GERDA JANSSENS, BART DEMOEN
and HENK VANDECASTEELE

Department of Computer Science, K. U. Leuven, Leuven, Belgium
(*e-mail:* {remko,gerda,bmd,henkv}@cs.kuleuven.be)

submitted 15 April 2005; revised 13 October 2005; accepted 13 January 2006

Abstract

Control flow compilation is a hybrid between classical WAM compilation and meta-call, limited to the compilation of non-recursive clause bodies. This approach is used successfully for the execution of dynamically generated queries in an inductive logic programming setting (ILP). Control flow compilation reduces compilation times up to an order of magnitude, without slowing down execution. A lazy variant of control flow compilation is also presented. By compiling code by need, it removes the overhead of compiling unreached code (a frequent phenomenon in practical ILP settings), and thus reduces the size of the compiled code. Both dynamic compilation approaches have been implemented and were combined with query packs, an efficient ILP execution mechanism. It turns out that locality of data and code is important for performance. The experiments reported in the paper show that lazy control flow compilation is superior in both artificial and real life settings.

KEYWORDS: logic programming, inductive logic programming, Warren Abstract Machine, compilation

1 Introduction

In the context of inductive logic programming (ILP), a large number of queries is generated dynamically, and then run on a large set of examples. From the data mining point of view, such a query is a hypothesis, and by running the query on the examples one can check how well the hypothesis covers these examples. From an implementor's point of view, every example is a Prolog program, and the queries are just Prolog queries that have to be executed against this large number of programs. Previously, only 2 options were considered: either the queries are meta-called, or the queries are first compiled to more efficient WAM bytecode after which this code is executed. The second option was identified to be the best in the ILP setting (Blockeel *et al.* 2002). Moreover, query packs (Blockeel *et al.* 2002)

* Remko Tronçon is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.).

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

were developed as a specialized execution mechanism for executing large sets of queries, improving execution time up to a factor 100. Adapting meta-call to handle these query packs is difficult and inefficient, and therefore compilation is needed (Blockeel *et al.* 2002; Demoen *et al.* 1999).

However, experiments indicate that even though compilation improves the total execution time of queries, the compilation time often dominates the total time of an ILP run. This raises the question of what can be done to decrease the compilation time, or, in other words, how we can simplify the compilation step. *Control flow compilation* realizes this by only compiling the control flow of a query (which deals for example with the selection of branches in a disjunction), and by using special meta-call like WAM instructions for calling predicates, thus omitting the need to do the complex step of setting up arguments to calls.

In this paper we propose control flow compilation as a novel fast compilation scheme for queries. This compilation scheme does not affect query execution performance, and moreover can be performed in a lazy (or 'just-in-time') fashion. The contributions of this paper are:

- Control Flow Compilation, which is a hybrid approach between meta-calling and classical compilation. This scheme incorporates the best of both worlds: it has the fast execution times of compiled code, without needing the expensive compilation step (which is a dominating factor in practical ILP settings).
- Lazy control flow compilation, which is a Just-In-Time (JIT) version of the control flow compilation scheme. Unreachable parts of the code are not compiled; this reduces both the compilation time and the code size.
- An evaluation of integrating (lazy) control flow compilation in a practical ILP system. A fast light-weight compiler for (lazy) control flow compilation was implemented in the hipP system, a Prolog system with specific support for ILP (such as query packs).

The topic of this paper has already been introduced by the authors (Tronçon *et al.* 2003; Tronçon *et al.* 2004), discussing an experimental implementation and some preliminary results on this and related techniques.

The organization of the paper is as follows: in Section 2, we briefly sketch the ILP context, the setting that motivated our work. In Section 3, control flow compilation is introduced and evaluated on both artificial and real life benchmarks. A lazy variant of this scheme is introduced in Section 4. Both approaches are then adapted to a practical ILP setting, by extending them to the query packs execution mechanism in Section 5. This extension is again evaluated on real life ILP benchmarks. Section 6 discusses memory management issues of the approaches described in this paper. Finally, conclusions and future work are given in Section 7.

We assume the reader is familiar with the WAM (Warren 1983; Ait-Kaci 1990).

2 Context: Dynamically generated queries in ILP

We start by sketching the role of dynamically generated queries and their execution in the ILP setting. The goal of ILP is to find a theory that best explains a large set

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

of data (or examples). In the ILP setting at hand, each example is a logic program, and the theory is represented as a set of logical queries. The ILP algorithm finds good queries by using generate-and-test. In the first step, it uses its own specific approach to generate queries, which are then evaluated in the second step. Based on the failure or success of these queries, only the ones with the 'best' results are kept and are extended (by adding literals). Which queries are best depends on the ILP algorithm: for example, in the case of classification, the information gain can be used as a criterion, whereas in the case of regression, the reduction of variance is often used. The extended queries are in turn tested on a set of examples, and this process continues until a satisfactory query (or set of queries) describing the examples has been found. We will focus on the efficiency of the second step.

At each iteration of the algorithm, a set of queries is executed against a large set of logic programs (the examples). Since these queries are the result of adding different literals at 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 can be transformed into a special kind of disjunction: a *query pack* (Blockeel *et al.* 2002). 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 relevant to the ILP algorithm, the normal Prolog disjunction might still cause too much backtracking. So, for efficiency reasons the `';'/2` is given a different procedural behavior in query packs: it cuts away branches from the disjunction as soon as they succeed. For this paper, it is sufficient to know that the semantics of query packs is very close to the one of normal disjunctions. 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 special WAM instructions for the query pack execution mechanism. More details can be found in (Blockeel *et al.* 2002).

Our paper proposes a new compilation approach for dynamically generated queries. It is independent of the actual query optimization schemes used. Moreover, it is useful for other Prolog engines as well, and is independent of the engine-specific support for ILP. After discussing how to use the approach to compile queries in general, we will apply it to the specific case of query pack compilation.

3 Control flow compilation

3.1 Technology

To execute a dynamically generated query, we can either meta-call it, or we can transform it into a non-recursive clause (by taking the query as the body and adding

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

a head, e.g. `query/0`), and run the compiled clause. Executing this compiled version of the query instead of meta-calling it results in considerable speedups. Moreover, in order to benefit from an efficient execution mechanism such as query packs, queries have to be compiled into the special WAM instructions. However, compilation of a query can be a costly task; in the ILP setting, compilation of a query can take as much time as its execution on all examples. This motivated the preliminary study of alternatives for compile & run (Tronçon *et al.* 2003). The most interesting alternative turned out to be *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 disjunctions, and explain this approach in terms of the familiar WAM instructions. The experiments confirm the potential of control flow compilation. This scheme will be extended to control flow compilation for ILP query packs 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, control flow compilation 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`[1]) instructions.

For example, control flow compiling the query

```
query :- 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, because queries are dynamically generated by the ILP system, the query itself is a term on the heap, and 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[2]. Before calling each goal, the compiled code first sets up the arguments to the goal, whereas the control flow compiled code uses a reference to the subterm of the query to indicate the goal that is called. The most 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 these kinds of instructions has other positive consequences: (1) it makes the expensive (non-linear) argument register allocation step unnecessary, saving compilation time, and (2) it makes it easy to incrementally add new code to existing parts of code. The latter is very interesting because it makes introducing laziness in the compilation process possible, as explained in Section 4.

Contrary to compiled code, control flow code cannot exist on its own, since it contains external references to terms on the heap. Therefore, an implementation must take the following garbage collection issues into consideration: (1) the terms of the query have to be kept alive as long as the control flow compiled code can be executed; (2) when these terms are moved to another place in memory, the

[1] `cf_deallex` is the instruction obtained by merging `deallocate` and `cf_execute`.

[2] The instruction set used in this example is based on the XSB implementation: the suffixes *tvar* and *pval* correspond to the WAM temporary and permanent variables respectively.

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

```
query :- 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.

Compiled code	**Control Flow code**		
	No Inlining	Built-ins	Special Built-ins
`...`	`...`	`...`	`...`
`call a/2`	`cf_call &a(X,Y)`	`cf_call &a(X,Y)`	`cf_call &a(X,Y)`
`putpval Y2 A1`		`putarg &X A1`	
`putpval Y3 A2`		`putarg &Y A2`	
`b_smaller A1 A2`	`cf_call &(X<Y)`	`b_smaller A1 A2`	`cf_smaller &X &Y`
`...`	`...`	`...`	`...`

Fig. 2. Built-in inlining for (a(X,Y), X < Y).

references in the code must be adapted as well. Moreover, executing general clauses requires that new variables be created prior to executing the body, whereas this is not necessary for queries: because queries will never be called recursively, the variables already existing on the heap can be used.

To speed up execution, the classical compilation scheme typically inlines smaller predicates (such as tests) using dedicated instructions implemented in the system. This is illustrated by the first column of Figure 2: the WAM compiler initializes the argument registers, and instead of calling a (WAM-compiled) `'<'/2` predicate, it emits a built-in instruction to do the test. Since control flow compilation also emits WAM bytecode, the same built-ins can be used for control flow compiled code

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

as for classical compiled code. These built-in instructions typically use argument registers for their arguments, so the compiler just needs to emit extra instructions to move data structures on the heap into the correct argument registers. These are illustrated in the third column of Figure 2, where the `putarg` instructions move references to data structures on the heap into the relevant argument registers for the built-in instruction. Alternatively, the extra emulator cycles needed for filling the argument registers can be skipped by defining special versions of each built-in that, instead of argument registers, have references to the heap as their parameters, such as `cf_smaller` in Figure 2.

3.2 Evaluation

For evaluating our approach, we added support for control flow code to the hipP system (hipP 2004), an efficient WAM based Prolog system written in C, and descendant of ilProlog. A separate control flow compiler for queries was implemented in this system. This compiler was written in Prolog, as is the case for the existing classical compiler. For the built-in predicates that are frequently used in ILP applications (e.g. `'<'/2`, `'>'/2`, `'='/2`, `'\='/2`, ...), we implemented special control flow instructions (such as `cf_smaller` from Figure 2), and these built-ins are inlined by the control flow compiler. The heap garbage collector of hipP was modified to support control flow compiled code. More details about the memory management will be discussed in Section 6.

All experiments were run on a Pentium III 1.1 GHz with 2 GB main memory running Linux, with a minimum of applications running.

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.

For each artificial experiment, a query was generated with 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 $G = 2$, $B = 3$ and $D = 1$, the following query is generated:

```
?- 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 $G = 1$, $B = 2$ and $D = 2$, 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 of `a/3` was taken to be `a(_,_,_)` to minimize the time spent outside of the query execution. For each generated query, the average compile and run time of the query was measured over a significant number of runs. We report on the

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

Table 1. *Experiments for artificial disjunctions*

Query				Experiment	Time	
G[a]	B[b]	D[c]	T[d]		Comp.[e]	Exec.[f]
5	5	4	3905	Meta-call	-	2.14
				Compile & Run	288.0	0.384
				Control Flow	30.8	0.198
10	5	4	7810	Meta-call	-	4.196
				Compile & Run	668.0	0.734
				Control Flow	62.6	0.36
5	10	4	55555	Meta-call	-	33.676
				Compile & Run	6368.0	5.64
				Control Flow	457.4	3.15
10	10	4	111110	Meta-call	-	64.94
				Compile & Run	13876.0	10.812
				Control Flow	847.8	5.718
5	5	6	19531	Meta-call	-	59.198
				Compile & Run	11596.0	9.93
				Control Flow	758.0	5.402

[a] Number of goals in a branch
[b] Branching factor of each disjunction
[c] Nesting depth of disjunctions
[d] Total number of goals ($= G \sum_{n=0}^{n=D} B^n$)
[e] Compilation time of the query (in ms.)
[f] Execution time of the query (in ms.)

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 compilation before it is executed.
- *Meta-call*: the query is meta-called (no compilation at all).

The control flow compilation is clearly better than compile & run: the compilation times are improved by one order of magnitude, while the execution times are also improved. 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 $G = 10$, $B = 10$, $D = 4$, and for $G = 5$, $B = 5$, $D = 6$, control flow compilation becomes up to a factor 16 faster than compile & run. If the query is executed a sufficient number of times, meta-call is outperformed by control flow compilation (e.g. for $G = 5$, $B = 5$, $D = 4$, this number is 15). Since in ILP, each query is run on thousands of examples, these results are very promising.

The real world experiment consists in running the TILDE algorithm (Blockeel and De Raedt 1998) from the ILP system ACE/hipP (ACE 2000) on three well-known

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

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

Dataset	Queries[a]	Runs[b]	Experiment	Comp.[c]	Exec.[d]	Total
Mutagenesis	2021	69.51	Meta-call	-	1.43	1.43
			Compile & Run	1.30	1.06	2.36
			Control Flow	0.21	1.02	1.23
Bongard	9335	244.77	Meta-call	-	24.70	24.70
			Compile & Run	4.98	21.34	26.32
			Control Flow	0.91	21.18	22.09
Carcinogenesis	48399	103.07	Meta-call	-	108.81	108.81
			Compile & Run	17.50	65.30	82.80
			Control Flow	2.24	59.51	61.75

[a] Total number of different queries executed
[b] Average number of examples on which a query is run
[c] Total compilation time of all queries (in seconds)
[d] Total execution time of all queries (in seconds)

datasets from the ILP community: Mutagenesis (Srinivasan *et al.* 1996), Bongard (De Raedt and Van Laer 1995) and Carcinogenesis (Srinivasan *et al.* 1999). During the execution of TILDE, queries are subsequently generated, and every query needs to be run on a subset of the examples. These queries consist only of conjunctions, and every query is executed separately on the examples. Table 2 compares the compilation time and execution time for all queries in the control flow compilation approach with the corresponding times of the compile & run and the meta-call approach. For each dataset, the total number of queries and the average number of runs per query is also given.

In the TILDE runs, control flow compilation gains a factor 5 to 8 over usual compilation. For all datasets, control flow compiled code also outperforms both the classical compiled code and the meta-called queries. Meta-call is slower than control flow compiled code because of the extra emulator cycles spent in testing the incoming goal upon each call. Additionally, specialized variants of `cf_call` are used for calling goals with arities smaller than 4 (which are the most frequent in practical ILP applications).

We conclude that control flow compilation is the fastest approach for executing the queries on these datasets. The main reason for this is that the share of query compilation in the total execution time of the ILP algorithm is reduced significantly. Moreover, control flow compiled code contains less instructions, and as such saves emulator cycles as well.

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.

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

3.3 Conclusion

The main goal of control flow compilation was to reduce high compilation times, without slowing down execution itself. Our experiments prove that control flow compilation achieves this goal: compilation times are reduced by an order of magnitude, while the execution becomes even slightly faster. Moreover, the new compilation scheme is flexible, and allows for extensions such as lazy compilation, as will be discussed in Section 4.

4 Lazy control flow compilation

4.1 Technology

In practical ILP applications, it is observed that large parts of the queries generated by the query generation process are never executed. Hence, unnecessary time is spent in compiling this unreachable code. With a lazy compilation scheme which only compiles code when it is actually reached, this redundancy can be removed. Control flow compilation is particularly suited for this dynamic kind of code, since existing compiled code can be extended without needing to alter the latter because of e.g. argument register allocation (as is the case with classical compilation). In this section, we will extend the control flow compilation scheme to yield a lazy variant.

In Aycock (2003), *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. Our lazy variant implicitly calls the control flow compiler when execution reaches a part of the query that is not yet compiled. 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)
```

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

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 for it. 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 resumes compilation, and replaces the corresponding `lazy_trymeorelse` by a `trymeorelse` instruction with the address of the code to be generated as argument:

```
    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))
```

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

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 proceeds from goal to goal: when the JIT compiler is called, it compiles exactly 1 goal, and then resumes execution. 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 frequent switching between the compiler and the execution by compiling bigger chunks.
- *Per disjunction:* All the branches of a disjunction are compiled up to the point where a new disjunction occurs. This approach is reasonable from an ILP viewpoint: the branches of a disjunction represent different queries, and since the success of each query is recorded, all branches will be tried (and thus compiled) eventually.

Besides the overhead of switching between compilation 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

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

disjunction is completely compiled in one step, each branch of the disjunction ends in a jump to the next disjunction.

Just as for control flow compilation, special control flow instructions for built-in predicates can be used in the lazy variant as well. Care must be taken though: typically, specialized built-ins are emitted depending on the type of arguments (e.g. specialized built-ins for unifying arguments with integers); however, as compilation is now interleaved with execution, arguments of a goal might have been bound after starting the execution of the query, which could make the emitted built-in overly specialized, thus generating code that becomes erroneous after backtracking or when run on another example. The compiler therefore shouldn't emit specialized built-ins depending on the instantiation and/or type of the arguments, or it should keep track of the state of the goal arguments in the original query. In our implementation, we chose for the former approach.

Finally, note that this lazy control flow compilation can be used to exploit the incremental nature of a query generation process such as the one from ILP. Suppose that queries are constructed with an open end, and that the compiler generates a `lazy_compile` instruction for such open ends; these open ends can be instantiated by a later query generation phase, such that when execution reaches the `lazy_compile` instruction, the new part of the query will be compiled and added to the existing code. This avoids the need to recompile the complete query, when only a small part of it changed. However, as experiments will show, control flow compilation times are relatively very low, such that the incremental compilation approach would not yield any significant speedups in total query evaluation time with respect to the use of (lazy) control flow compilation.

4.2 Evaluation

In the first experiment, we will measure the overhead of the new lazy compilation scheme. The artificial queries from Table 3 have no unreachable parts, and as such provide a worst case for lazy compilation overhead. In practical applications, we expect the queries to have unreachable parts, and so the total overhead of the lazy compilation scheme will be compensated by the smaller compilation time. The experiments of Table 3 use only the first two benchmarks from Table 1. The other benchmarks of Table 1 yield similar results. Timings (in milliseconds) are given for the different settings of lazy compilation. The timings report the time needed for one execution of the query, thus including the time of its lazy compilation. These timings are then compared with the time of performing non-lazy control flow compilation of the query and executing it once[3]. 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.

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

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

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

Table 3. *Lazy compilation for several kinds of disjunctions*

Query			Experiment	Time
G[a]	B[b]	D[c]		Total[d]
5	5	4	Per Goal	55
			Per Conjunction	34
			Per Disjunction	32
			No Laziness	28
10	5	4	Per Goal	111
			Per Conjunction	60
			Per Disjunction	59
			No Laziness	59

[a] Number of goals in a branch
[b] Branching factor of each disjunction
[c] Nesting depth of disjunctions
[d] Total Compilation + Execution time of the query (in ms.)

query packs compilation: (1) lazy compilation is fast; (2) in real life benchmarks, some branches will never be compiled due to failure of goals, whereas in our artificial setting all goals in the queries succeed.

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* (Blockeel *et al.* 2002). These query packs represent a set of (similar) queries which are to be executed, laid out in a disjunction. The semantics of this query pack disjunction is implemented by dedicated WAM instructions (Blockeel *et al.* 2002), as explained in Section 2. These instructions replace the instructions generated for encoding ordinary disjunctions.

As experiments in Section 4 pointed out, the choice of the actual lazy compilation variant does not matter with respect to the overhead introduced (except for lazy compilation per goal). We therefore chose to implement only the variant which compiles one complete disjunction at a time, as this made integration with the existing query pack data structures easier. As explained in Section 4, this means that each branch of a disjunction ends in a jump. In the implementation of the query packs execution, this was already the case, so there are no extra emulator cycles in JIT compiled code compared to the other compilation schemes. The memory management aspects of the implementation will be discussed in Section 6.

5.2 Evaluation

We evaluate (lazy) control flow compilation for query packs by running TILDE, but by letting it generate query packs instead of conjunctions (as was the case for Table 2).

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

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

Dataset	Unused[a]	Experiment	Comp.[b]	Exec.[c]	Total[d]
Mutagenesis	17%	Compile & Run	0.52	0.11	0.63
		Control Flow	0.07	0.10	0.17
		Lazy Control Flow	-	-	0.14
Bongard	51%	Compile & Run	1.91	1.17	3.08
		Control Flow	0.28	1.15	1.43
		Lazy Control Flow	-	-	1.37
Carcinogenesis	32%	Compile & Run	7.39	4.63	12.02
		Control Flow	0.81	3.81	4.62
		Lazy Control Flow	-	-	4.34
HIV	74%	Compile & Run	209.47	191.68	401.15
		Control Flow	27.13	178.53	205.66
		Lazy Control Flow	-	-	186.22

[a] Total % of the query code that is never executed
[b] Total compilation time of all queries (in seconds)
[c] Total execution time of all queries (in seconds)
[d] Total query evaluation time (= Comp. + Exec.)

The experiments are performed on the ILP datasets from Table 2. Additionally, the query pack execution mechanism allows us to do experiments on larger datasets, such as the HIV dataset (DTP).

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. Additionally, we give for each dataset the share of query goals that are never reached by the query execution. Comparing the timings for the query packs with the timings for the sets of queries in Table 2 we see that the query packs are considerably faster.

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 traditional compilation, even though the hipP system already has a compiler that is optimized for dealing with large disjunctions (Vandecasteele *et al.* 2000) (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 than classical compilation. For the ILP application, the total time must be considered: the total time of control flow is up to a factor 3 faster than compile & run. Note that this factor is higher for the query packs than for the conjunctions. The timings show that, for our benchmarks, the compilation time in compile & run is systematically larger than the execution time for all the examples such that the impact of improving the compilation has a larger effect on the total times.

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

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

Table 5. *Impact of locality on execution times*

Dataset	Experiment	Comp.[a]	Exec.[b]	Total[c]
Mutagenesis	Compile & Run	0.52	0.11	0.63
	Control Flow (No Copy)	0.08	0.10	0.18
	Control Flow	0.07	0.10	0.17
Bongard	Compile & Run	1.91	1.17	3.08
	Control Flow (No Copy)	0.33	1.19	1.52
	Control Flow	0.28	1.15	1.43
Carcinogenesis	Compile & Run	7.39	4.63	12.02
	Control Flow (No Copy)	0.70	4.11	4.81
	Control Flow	0.81	3.81	4.62
HIV	Compile & Run	209.47	191.68	401.15
	Control Flow (No Copy)	27.81	193.90	221.71
	Control Flow	27.13	178.53	205.66

[a] Total compilation time of all queries, including the time to copy the query (in seconds)
[b] Total execution time of all queries (in seconds)
[c] Total query evaluation time (= Comp. + Exec.)

packs. This is indeed the case for all datasets. As expected, the time gained by not compiling unused parts of queries corresponds roughly with the measured amount of unreached goals.

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

6 Memory management considerations

Experiments pointed out that the layout of the code and data in memory can have a big impact on the execution time of the queries.

Because the execution of the control flow compiled code needs to fetch the data for its calls from the heap, the compiled code should be as close as possible to the data it consumes to have good locality of data. We achieve this by allocating control flow compiled code on the heap, and extending the heap garbage collector to support this new data structure. Because of the dynamic nature of lazy compiled code, control flow blocks can be scattered across the heap during execution; the heap garbage collector moves these blocks closer to each other during collection, which improves locality. Since queries have a volatile nature, the heap collector will also often remove dead code blocks, which typically belong to old queries. Finally, the code itself contains pointers to terms on the heap, which are handled by the heap garbage collector as well.

Finally, the locality of the query goals themselves also has an impact on execution time. During the query generation phase, other data is allocated on the heap, which can lead to a situation where the goal terms of the query (which will be used during execution of the control flow compiled code) are scattered across the heap. In all our experiments, the (possibly scattered) term representing the complete query was

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

copied before compiling it. This ensured that all the terms used in the compiled code are allocated together on the heap. The impact of leaving out the copying step is illustrated in Table 5. These are the results of running TILDE on the same datasets as in Table 4. Without copying the goals, the execution time of control flow compiled code becomes slower than code executed using the classical approach. Although copying the query before compilation costs some time, it improves the locality during the compilation step itself. The effect on the benchmarks is that it sometimes introduces a slight overhead in some of the smaller benchmarks, but this is compensated by the gain in execution time. Control flow compilation with copy turns out to be the best approach.

7 Conclusions

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, without affecting the execution time. 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. Moreover, the lazy variant provides additional speedup in the total time by not compiling unreached parts of the query.

Traditionally, Prolog implementations have implemented a form of JIT, where compilation to WAM code or machine code happens at consult time. Yap (Damas and Santos Costa 2003) goes one step further and compiles a predicate to abstract machine code at the first call to that predicate. BinProlog (Tarau 1992) and hipP switch back and forth between a compiled and an interpreted form of dynamic predicates, based on the relative frequency of modification and execution of the predicate. The granularity of these JIT compilation forms is always a predicate, while control flow compiled code can have a finer grained granularity up to a literal. However, control flow compilation cannot be used for compiling recursive predicates.

Yap (Damas and Santos Costa 2003), which is used by the Aleph ILP system (Srinivasan 2001), has recently introduced other implementation techniques for speeding up the evaluation of many queries against many examples. In particular tabling and dynamic indexing can speed up the query execution phase considerably. Our control flow compilation schema is orthogonal to these techniques and can be combined with them. Especially when tabling is used, it is important to spend little time on compiling the queries, as tabling avoids repeated execution of the same goal (or prefix of a query). So, we expect that control flow compilation is beneficial in combination with tabling.

Within the ILP setting, the applications of (lazy) control flow compilation can be extended further. Firstly, we plan to adapt it to extensions of query packs reported in (Tronçon *et al.* 2003). 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. In (Ramon and Struyf 2004), a technique for efficient theta-subsumption is proposed which uses query pack execution. It has to

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

be investigated whether lazy control flow compilation reduces the compilation time enough in the particular setting that executes the query pack only once, or that a pure meta-call based approach for the query packs performs better.

Acknowledgements

We would like to thank the anonymous referees for their comments and suggestions.

References

ACE. 2000. The ACE data mining system. http://www.cs.kuleuven.be/~dtai/ACE/.

Ait-Kaci, H. 1990. The WAM: a (real) tutorial. Tech. Rep. 5, DEC Paris Research Report. http://www.isg.sfu.ca/~hak/documents/wam.html.

Aycock, J. 2003. A brief history of just-in-time. *ACM Computing Surveys 35,* 2, 97–113.

Blockeel, H. and De Raedt, L. 1998. Top-down induction of first order logical decision trees. *Artificial Intelligence 101,* 1-2 (June), 285–297.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J. and Vandecasteele, H. 2002. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research 16*, 135–166. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=36467.

Damas, L. and Santos Costa, V. 2003. YAP User's manual. http://yap.sourceforge.net.

De Raedt, L. and Van Laer, W. 1995. Inductive constraint logic. In *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, K. P. Jantke, T. Shinohara, and T. Zeugmann, Eds. Lecture Notes in Artificial Intelligence, vol. 997. Springer-Verlag, 80–94.

Demoen, B., Janssens, G. and Vandecasteele, H. 1999. Executing query flocks for ILP. In *Proceedings of the Eleventh Benelux Workshop on Logic Programming*, S. Etalle, Ed. Maastricht, The Netherlands, 1–14. 14 pages.

DTP. The developmental therapeutics program. U.S. Departement of Health and Human Services NIH, National Cancer Institute NCI. http://dtp.nci.nih.gov.

hipP. 2004. A high performance prolog system. http://www.cs.kuleuven.be/~dtai/hipp/.

Ramon, J. and Struyf, J. 2004. Efficient theta-subsumption of sets of patterns. In *Proceedings of the 13th Belgian-Dutch Conference on Machine Learning*. 95–102.

Srinivasan, A. 2001. The Aleph manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

Srinivasan, A., King, R. and Bristol, D. 1999. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*. Lecture Notes in Artificial Intelligence, vol. 1634. Springer-Verlag, 291–302.

Srinivasan, A., Muggleton, S., Sternberg, M. and King, R. 1996. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence 85,* 1,2, 277–299.

Tarau, P. 1992. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In *Russian Conference on Logic Programming*, A. Voronkov, Ed. Number 592 in Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, Heidelberg, 462–473.

Tronçon, R., Janssens, G. and Demoen, B. 2003. Alternatives for compile & run in the WAM. In *Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic*

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press

Programming Systems. University of Porto, 45–58. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto, December 2003. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41065.

Tronçon, R., Janssens, G. and Vandecasteele, H. 2004. Fast query evaluation with (lazy) control flow compilation. In *Logic Programming, 20th International Conference, ICLP 2004, Proceedings*. Lecture Notes in Artificial Intelligence, vol. 3132. Springer Verlag, 240–253. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=41198.

Tronçon, R., Vandecasteele, H., Struyf, J., Demoen, B. and Janssens, G. 2003. Query optimization: Combining query packs and the once-tranformation. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations*. 105–115. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=40938.

Vandecasteele, H., Demoen, B. and Janssens, G. 2000. Compiling large disjunctions. In *First International Conference on Computational Logic : Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, I. de Castro Dutra, E. Pontelli, and V. S. Costa, Eds. Imperial College, 103–121. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=32065.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Tech. Rep. 309, SRI.

https://doi.org/10.1017/S1471068406002894 Published online by Cambridge University Press