AMaχoS—Abstract Machine for Xcerpt: Architecture and Principles

François Bry, Tim Furche, and Benedikt Linse

Institute for Informatics,University of Munich,
Oettingenstraße 67, 80538 München, Germany
http://pms.ifi.lmu.de/

Abstract. Web query languages promise convenient and efficient access to Web data such as XML, RDF, or Topic Maps. Xcerpt is one such Web query language with strong emphasis on novel high-level constructs for effective and convenient query authoring, particularly tailored to versatile access to data in different Web formats such as XML or RDF. However, so far it lacks an efficient implementation to supplement the convenient language features. AMaχoS is an abstract machine implementation for Xcerpt that aims at efficiency and ease of deployment. It strictly separates compilation and execution of queries: Queries are compiled once to abstract machine code that consists in **(1)** a code segment with instructions for evaluating each rule and **(2)** a hint segment that provides the abstract machine with optimization hints derived by the query compilation. This article summarizes the motivation and principles behind AMaχoS and discusses how its current architecture realizes these principles.

1 Introduction

Efficient evaluation of Web query languages such as XQuery, XSLT, or SPARQL has received considerable attention from both academia and industry over recent years. Xcerpt is a novel breed of Web query language that aims to overcome the split between traditional Web formats such as XML and Semantic Web data formats such as RDF and Topic Maps. Thus it avoids the impedance mismatch of using different languages to develop applications that enrich conventional Web applications with semantics and reasoning based on RDF, Topic Maps, or similar emerging formats.

However, so far Xcerpt lacks a scalable, efficient and easily deployable implementation. In this article, we propose principles and architecture of such an implementation. The proposed implementation deviates quite notably from conventional wisdom on the implementation of query languages: it is based on an abstract (or virtual[1]) machine that executes (interprets) low-level code generated from high-level query programs specified in Xcerpt.

[1] Little substantial difference is made in the literature between "abstract" and "virtual" machines. Some authors define virtual machines as abstract machines with *interpreters* in contrast to abstract machines such as Turing machines that are purely

The choice of an abstract machine for implementing a query language might at the first glance seem puzzling. And indeed proper abstract machines that separate execution and compilation have only very seldom been considered in the past for the implementation of query languages (the most notable exception being [19]). This is partially due to the perceived performance overhead introduced by the abstraction/virtualization layer. However, traditional query processors already separate between query compilation, where a high-level query is translated into a low-level physical query plan, and query execution, where the query is evaluated according to that query plan. From this the leap to an abstract machine that fully separates compilation and execution seems small and could even be considered merely a change in name. In traditional DBMS settings it has, however, never occurred due to the way query compilation is linked with query execution: cost-based optimizers consider extensively (statistical) information about the data instances, e.g., for selectivity estimates, and about actual access paths to these data instances. This information is available as the DBMS has full, central control over the data including its storage.

When implementing a Web query language such as Xcerpt, one is however faced with a quite different setting: In memory processing of queries against XML, RDF, or other Web data that may be local and persistent (e.g., an XML database or local XML documents), but just as well may have to be accessed remotely (e.g., a remote XML document) or may be volatile (e.g., in case of SOAP messages or Web Service access). In other words, it is assumed that most of the queried data is *not* under (central) control of a query execution environment like in a traditional DBMS setting, but rather that the queried data is often distributed or volatile. This, naturally, hinders the application of conventional indexing and predictive optimization techniques, that rely on local management of data and statistic knowledge about that managed data. But, it also makes separate compilation and execution possible as the query compilation is already mostly independent of data storage and instances. This is due to the fact that information about these is not available at compilation and execution time but only becomes available at query execution.

To some extent, this setting is comparable to data stream processing where also little is known about the actual data instances that are to be encountered during query evaluation. The efficient data stream systems (such as [3, 1, 6]) compile therefore queries into some form of (finite state or push-down) automata that is used to continuously evaluate the query against the incoming data.

AMAχOS, the abstract machine for Xcerpt on semi-structured data, can be seen as an amalgamation of techniques from these three areas: query optimization and execution from traditional databases and data stream systems, and compilation and execution of general programs based on abstract or virtual machines.

AMAχOS is designed around a small number of core principles:

---

theoretical thought models. However this distinction is not widely adopted. In recent years, the term "virtual" machine seems to dominate outside of logic programming literature.

1. "Compile once"—compilation and execution is separated in AMAχOS thus allowing **(a)** different levels of optimization for different purposes and settings and **(b)** the distribution of compiled query programs among query nodes making light-weight query nodes possible. For details see Section 4.2.
2. "Execute anywhere"—once compiled, AMAχOS code can be evaluated by any AMAχOS query node. It is not fixed to the compiling node. In particular, parts of a compiled program can be distributed to different query nodes. For details see Section 4.1.
3. "Optimize all the time"—not only are queries optimized predictively during query compilation, but also adaptively during execution. For details see Section 4.4.

As a corollary of these three principles AMAχOS employs a novel query evaluation framework for the unified execution of path, tree, and graph queries against both tree- and graph-shaped semi-structured data (details of this framework are discussed in Section 4.3 and [8]).

Following a brief look at the history of abstract and virtual machines for program and query execution (Section 2) and an introduction into Xcerpt (Section 3), the versatile Web query language that is implemented by the AMAχOS abstract machine, we focus in the course of this article first (Section 4) on a discussion of the *principles* of this abstract machine that also serves as a further motivation of the setting. The second part (Section 5) of the paper discusses the proposed *architecture* of AMAχOS and how this architecture realizes the principles discussed in the first part.

2 A Brief History of Abstract Machines

Abstract and virtual machines have been employed over the last few decades, aside from theoretical abstract machines as thought models for computing, in mostly three areas:

Hardware virtualization. Abstract machines in this class provide a layer of virtual hardware on top of the actual hardware of a computer. This provides the programs directly operating on the virtual hardware (mostly operating systems, device drivers, and performance intensive applications) with a seemingly uniform view of the provided computing resources. Though this has been a focus of considerable research as early as 1970, cf. [12] only recent years have seen commercially viable implementations of virtual machines as hardware virtualization layers, most recently Apple's Rosetta[2] technology that provides an adaptive, just-in-time compiled virtualization layer for PowerPC applications on Intel processors. Currently, research in this area focuses on providing scalability, fault tolerance [9] and trusted computing [11] by employing virtual machines, as well as on on-chip support for virtualization.

[2] http://www.apple.com/rosetta/

Operating system-level virtualization A slightly higher level of abstraction or virtualization is provided by operating system-level virtual machines that virtualize operating system functions. Again, this technology has just recently become viable in the form of, e.g., Wine[3], a Windows virtualization layer for Unix operating systems.

High-level language virtual machines From the perspective of AMAχOS the most relevant research has been on virtual machines for the implementation of high-level languages. First research dates back to the 1960s [24] and 1970s [22], but wider interest in abstract machines for high-level languages has been focused on two waves: First, in the 1980s a number of abstract machines for Pascal (p-Machine, [23]), Ada [14], Prolog [30], and functional programming languages (G-machine, [16]) have been proposed that focused on providing *platform neutrality and portability* as well as precise specifications of the *operational semantics* of the languages. Early abstract machines for imperative and object-oriented programming languages have not been highly successful, mostly due to the perceived performance penalty. However, research on abstract machines for logic and functional programming languages has continued mostly uninterrupted up to recent developments such as the tabling abstract machine [26] for XSB Prolog.

Recently, the field has seen a reinvigoration, cf. [25], triggered both by advances in hardware virtualization and a second wave of abstract machines for high-level programming languages focused this time on imperative, object-oriented programming languages like Java and C$\sharp$. Here, *isolation and security* are added to the core arguments for the use of an abstract machine: Each instance of an abstract machine is isolated from others and from other programs on the host system. Furthermore analysis of the abstract machine byte code to ensure, e.g., safety or security properties proves easier than analysis of native machine code.

The most prominent examples of this latest wave are, of course, Sun's Java virtual machine [17] and Microsoft's common language infrastructure [15] (CLI). The latter is adding the claim of "language independence" to the arguments for the deployment of an abstract machine. And indeed quite a number of object-oriented and functional languages have been compiled to CLI code. With this second wave, design and principles of abstract machines are starting to be investigated more rigorously, e.g., in [10] and [29] that compare stack- with register-based virtual machines.

Closest in spirit and aim to the work presented in this paper and to the best knowledge of the authors' the only other work on abstract machines for Web query languages is [19] that presents a virtual machine for XSLT part of recent versions of the Oracle database. However, this virtual machine is focused on a centralized query processing scenario: a single query engine has control over all data and thus can employ knowledge about data instances and access paths for optimization and execution.

[3] `http://www.winehq.com/`

3 Xcerpt: A Versatile Web Query Language

Xcerpt is a query language designed after principles given in [7] for querying both data on the standard Web and data on the Semantic Web. More information, including a prototype implementation, is available at `http://xcerpt.org`.

3.1 Data as Terms

Xcerpt uses **terms** to represent semi-structured data. *Data terms* represent XML documents, RDF graphs, and other semi-structured data items. Notice that subterms (corresponding to, e.g., child elements) may either be "ordered" (as in an XHTML document or in RDF sequence containers), i.e., the order of occurrence is relevant, or "unordered", i.e., the order of occurrence is irrelevant and may be ignored (as in the case of RDF statements).

3.2 Queries as Enriched Terms

Following the "Query-by-Example" paradigm, queries are merely examples or *patterns* of the queried data and thus also terms, annotated with additional language constructs. Xcerpt separates querying and construction strictly.

Query terms are (possibly incomplete) patterns matched against Web resources represented by data terms. In many ways, they are like forms or examples for the queried data, but also may be *incomplete in breadth*, i.e., contain 'partial' as well as 'total' term specifications. Query terms may further be augmented by *variables* for selecting data items.

Construct terms serve to reassemble variables (the bindings of which are gained from the evaluation of query terms) so as to construct new data terms. Again, they are similar to the latter, but augmented by variables (acting as place holders for data selected in a query) and grouping constructs (which serve to collect all or some instances that result from different variable bindings).

3.3 Programs as Sets of Rules

Query and construct terms are related in **rules** which themselves are part of Xcerpt **programs**. Rules have the form:

```
CONSTRUCT construct-term
FROM and { query-term or { query-term ... } ... } END
```

Rules can be seen as "views" specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database).

Xcerpt rules may be *chained* like active or deductive database rules to form complex query programs, i.e., rules may query the results of other rules. More details on the Xcerpt language and its syntax can be found in [27, 28].

4 Architecture: Principles

The abstract machine for Xcerpt, in the following always referred to as AMAχOS, and its architecture are organized around five guiding principles:

4.1 "Execute Anywhere"—Unified Query Execution Environment

As discussed above, possibly the strongest reason to develop virtual machines for high-level languages is the provision of a unified execution environment for programs in that high-level language. In the case of Xcerpt, AMAχOS aims to provide such a unified execution environment. In our case, a unified execution environment brings a number of unique advantages: **(1)** The *distributed execution of queries and query programs* requires that the language implementations are highly interoperable down to the level of answer representation and execution strategies. A high degree of interoperability allows, e.g., the distribution of partial queries among query nodes (see below). An abstract machine is an exceptionally well suited mechanism to ensure implementation interoperability as its operations are fairly fine granular and well-specified allowing the controlling query node fine granular control over the query execution at other ("slave") nodes. **(2)** A rigid definition of the operational semantics as provided by an abstract machine allows not only a better understanding and communication of the evaluation algorithms, it also makes *query execution more predictable*, i.e., once compiled a query should behave in a predictable behavior on all implementations. This is an increasingly important property as it eases query authoring and allows better error handling for distributed query evaluation. **(3)** Finally, a unified query execution environment makes the *transmission and distribution of compiled queries and even parts* of compiled queries among query nodes feasible, enabling easy adaptation to changes in the network of available query nodes, cf. Section 4.5.

4.2 "Compile Once"—Separation of Compilation and Execution

In the introduction, the setting for the AMAχOS abstract machine has been illustrated and motivated: In memory processing of queries against XML, RDF, or other Web data that may be local and persistent (e.g., an XML database or local XML documents), but just as well may have to be accessed remotely (e.g., a remote XML document) or may be volatile (e.g., in case of SOAP messages or Web Service access). In other words, it is assumed that most of the queried data is not under (central) control of a query execution environment like in a traditional database setting, but rather that the queried data is often distributed or volatile. This, naturally, limits the application of traditional indexing and predictive optimization techniques, that rely on local management of data and statistic knowledge about that managed data.

Nevertheless *algebraic optimization techniques* (that rely solely on knowledge about the query and possible the schema of the data, but not on knowledge about

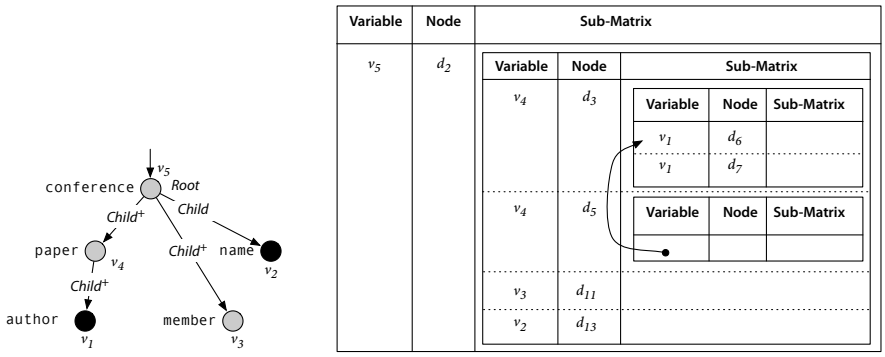

Fig. 1. Sample Query and Memoization Matrix

the actual instance of data to be queried) and *ad-hoc indices* that are created during execution time still have their place under this circumstances.

In particular, such a setting allows for a clean *separation of compilation and execution*: The high-level Xcerpt program is translated into AMAχOS code separately from its execution. The translation may be separated by time (at another time) and space (at another query node) from the actual execution of the query. This is essential to enable the distribution of pre-compiled, globally optimized AMAχOS programs evaluating (parts of) queries over distributed query nodes.

Extensive static optimization. This separation also makes more extensive static optimization feasible than traditionally applied in an in-memory setting (e.g., in XSLT processors such as Saxon[4] or Xalan[5]). Section 5.2 and Figure 5 present a more detailed view of the query compiler and optimizer employed in the AMAχOS virtual machine. To be applicable to different scenarios, a control API for the query compilation stage allows the configuration of strategy and extent used for optimizing a query during the compilation from high-level Xcerpt programs to low-level AMAχOS code.

Aside from traditional tasks such as dead (or tautological) branch elimination, detection of unsatisfiable queries, operator order optimization and selection between different realizations for the same high-level query constructs, the AMAχOS query compiler has another essential task: the *classification of each query* in the query program by its features, e.g., whether a query is a path, tree, or graph query (cf. [20, 8]) or which parts of the data are relevant for the query evaluation. This information is encoded either directly in the AMAχOS code of the corresponding construct-query rule or in a special *hint section* in the AMAχOS program. That hint section is later used by the query engine (the AMAχOS core) to tune the evaluation algorithm.

[4] `http://www.saxonica.com/`
[5] `http://xml.apache.org/xalan-j/`

4.3 "Compile, Classify, Execute"—Unified Evaluation Algorithm

A *single evaluation algorithm* is used in AMAχOS for evaluating a large set of diverse queries and data. At the core of this algorithm stands the "memoization matrix," a data structure first proposed in [27] and refined to guarantee polynomial size in [8]), that allows an efficient representation of intermediary results during the evaluation of an Xcerpt query (or more generally an n-ary conjunctive query over graph data). A sample query and corresponding memoization matrix are shown in Figure 1: The query selects the names of conferences with PC members together with their authors (i.e., it is a binary query). The right hand of Figure 1 shows a possible configuration of the memoization matrix for evaluating that query: d_2 is some conference for which we have found multiple bindings for v_4, i.e., the query node matching papers of the selected conference. The matrix also shows that sub-matrices are shared if the same query node matches the same data node under different constellations of the remaining query nodes. This sharing is possible both in tree and graph queries, but in the case of graph queries the memoization matrix represents only a potential match in which only a spanning tree over the relations in the query is enforced. The remaining relations must be checked on an unfolding of the matrix. This last step induces exponential worst-case complexity (unsurprisingly as graph queries are NP-complete already if evaluated against tree data as shown in [13]), but is in many practical cases of little influence.

How to use the memoization matrix to obtain an evaluation algorithm for arbitrary n-ary conjunctive queries over graphs (that form the core of Xcerpt query evaluation), is shown in [8]. It is shown that the resulting algorithms are competitive with the best known approaches that can handle only tree data and that the introduction of graph data has little effect on complexity and practical performance.

The memoization matrix forms the core of the query evaluation in AMAχOS. As briefly outlined in [8], the method can be *parameterized with different algorithms* for populating and consuming the matrix. Thereby it is possible to adopt the algorithm both to different conditions for the query evaluation (e.g., is an efficient label or keyword index for the data available or not) and to different requirements (e.g., are just variable bindings needed or full transformation queries). The first aspect is automatically adapted by the query engine (cf. Section 5.1), the second must be controlled by the execution control API, cf. Section 5.

4.4 "Optimize All the Time"—Adaptive Code Optimization

As argued above in Section 4.2 a separation of compilation/optimization from execution is an essential property of the AMAχOS virtual machine that allows it to be used for distributed query evaluation and Web querying where control over the queried data is not centralized.

This separation can be achieved partially by providing a unified evaluation algorithm (Section 4.3) that tunes itself, with the help of hints from the static optimization, to the available access methods and answer requirements.

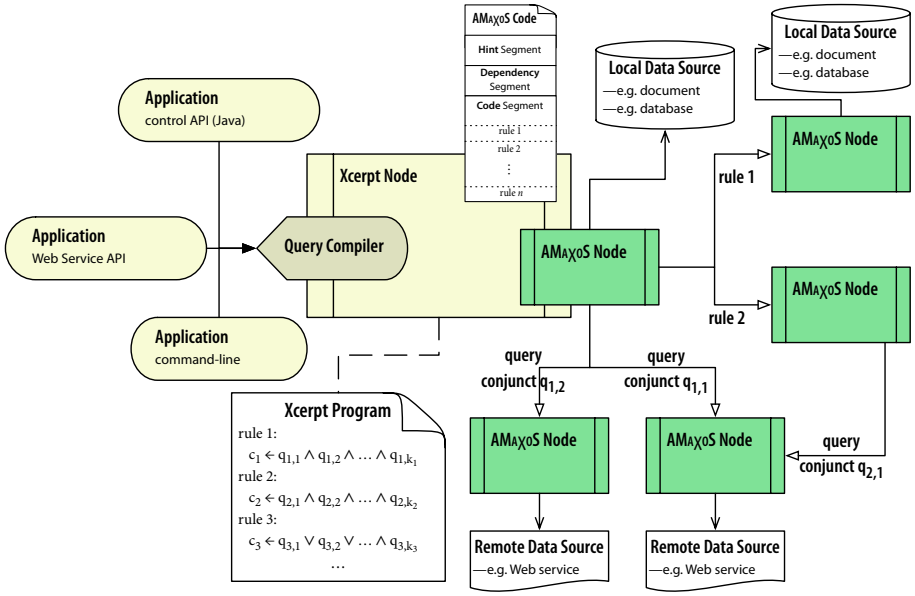

Fig. 2. Query Node Network

However, separate compilation precludes optimizations based on intricate knowledge about the actual instances of the data to be queried (e.g., statistical information about selectivity, precise access paths, data clustering, etc.). This can, to some extent, be offset by *adaptive code optimization*. Adaptive query optimization is a technique sometimes employed in continuous query systems, where also the characteristic of the data instances to be queried is not known a priori, cf. [2].

In the AMAχoS virtual machine we go a step further: Not only can the physical query plan expressed in the AMAχoS code continuously be adapted, but the result of the adaptation can be stored (and transmitted to other query nodes) as an AMAχoS program for further executions of the same query. Obviously, such adaptive code optimization is not for free and will most likely be useful in cases where the query is expected to be evaluated many times (e.g., when querying SOAP messages) or the amount of data is large enough that some slow-down for observation and adaption in the first part of the evaluation is offset by performance gains in later parts.

4.5 "Distribute Any Part"—Partial Query Evaluation

Once compilation and execution are separate, the possibility exists that one query node compiles the high-level Xcerpt program to AMAχoS code using knowledge about the query and possibly the schema of the data to optimize (globally) the query plan expressed in the AMAχoS code. The result of this

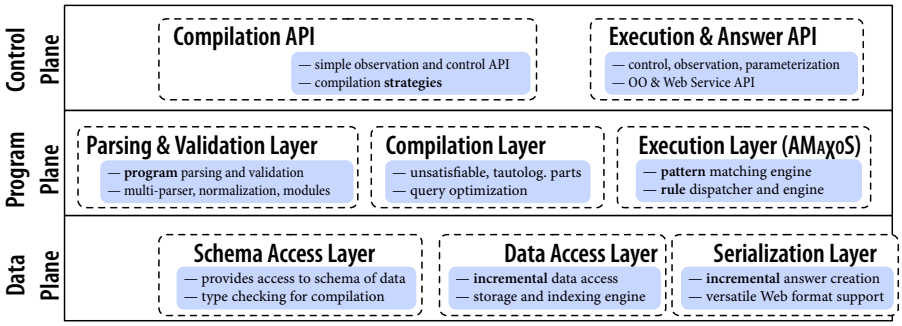

Fig. 3. Overview of AMAχoS Components

translation can than be distributed among several query nodes, e.g., if these nodes have more efficient means to access the resources involved in the query.

Indeed, once at the level of AMAχoS code it is not only possible to distribute say entire rules or sets of rules, but even parts of rules (e.g., query conjuncts) or even smaller units. Figure 2 illustrates such a distributed query processing scenario with AMAχoS: Applications use one of the control APIs (obtaining, e.g., entire XML documents or separate variable bindings) to execute a query at a given Xcerpt node. This implementation of Xcerpt transforms the query into AMAχoS code and hands this code over to its own AMAχoS engine. Depending on additional information about the data accessed in the query, this AMAχoS node might decide to evaluate only some parts of the query locally, e.g., those operating exclusively on local data and those joining data from different sources. All the remaining query may be send parts to other AMAχoS nodes that are likely to have more efficient access to the relevant data.

In contrast to distribution on the level of a high-level query language such as Xcerpt, distribution on the level of AMAχoS has two main advantages: the distributed query parts can be of finer granularity and the "controlling" node can have, by means of code transformation and hint sections, better control of the "slave" nodes.

Notice, that AMAχoS enables such query distribution, but does not by itself provide the necessary infrastructure (e.g., for registration and management of query nodes). It is assumed that this infrastructure is provided by outside means.

5 Architecture: Overview

The previous section illustrates the guiding principles in the development of AMAχoS. The remainder of this article focuses on how these principles are realized in its architecture and discusses several design choices regarding the architecture.

Notice, that only a small part of the full AMAχoS architecture as described here has been implemented so far. We have concentrated on the implementation

on the execution and optimization layer, that are also described in more detail in Sections 5.1 and 5.2. Of the execution layer the core evaluation algorithm (pattern matching engine) is implemented as described in [8].

Figure 3 shows a high-level overview of AMAχOS and its components. The architecture separates the components in three planes:

Control Plane. The control plane enables outside control of the compilation, execution, and answer construction. Furthermore, it is responsible for observation and adaptive feedback during execution.

Program Plane. The program plane contains the core components of the architecture: the compilation and execution layer. It combines all processing that an Xcerpt program partakes when evaluated by an AMAχOS virtual machine. The first step is, naturally, parsing, validation, normalization, module expansion etc. These are realized as transformations on the layer of the Xcerpt language and the resulting normalized, validated, and expanded Xcerpt program can be accessed via the compilation API. However, usually the result becomes input for the compilation layer where the actual transformation into AMAχOS code takes place. The details of this layer are discussed below in Section 5.2. In the architecture overview, we chose to draw the compilation and execution layer as directly connected. However, it is also possible to access the resulting program (again via the compilation API) and execute it at a later time and even at a different place. Indeed, compilation and execution are properly separated with only one interface between them: the AMAχOS program that contains besides the expressions realizing individual rules in the Xcerpt program also supporting code segments that provide hints for the program execution and dependency information used in the rule dispatcher, cf. Section 5.1.

Data Plane. The architecture is completed by the data plane, wherein all access to data and schema of the data is encapsulated. During compilation, if at all, only the schema of the data is assumed to be available.

5.1 AMaχoS Core

The core of the AMAχOS virtual machine is formed by the query execution layer, or AMAχOS proper. Here, an AMAχOS program (generated separately in the compilation layer, cf. Section 5.2) is evaluated against data provided by the runtime data access layer resulting in answers that are serialized by the serialization API.

As shown in Figure 4, the query execution layer is divided in four main components: the rule engine, the construction engine, the static function library, and the storage manager. Once a program containing AMAχOS code is parsed information from the *hint segment* is used to parameterize storage manager and rule engine. These parameters address, e.g., the classification of the contained queries (tree vs. graph queries), the selection of access paths, filter expressions for document projection, the choice of in-memory representation (e.g., fast traversal

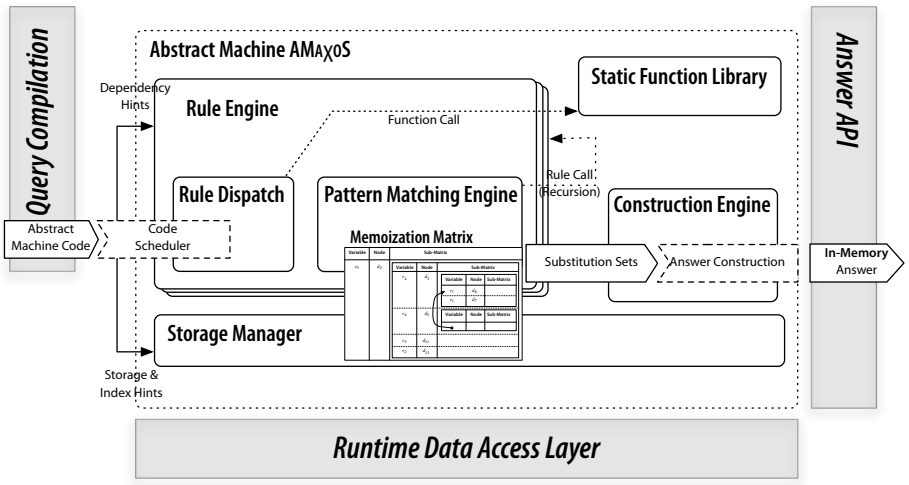

Fig. 4. Architecture of Core Query Engine AMAχOS

vs. small memory footprint), etc. The rule dependency information is provided to the *rule dispatcher* who is responsible for combining the results of different rules and matching query conjuncts with rule heads. Each rule has a separate segment in the AMAχOS program containing code for pattern matching and for result construction. Intermediary result construction is avoided as much as possible, partially by rule unfolding, partially by propagating constraints on variables from rule heads into rule bodies. Only when aggregation or complex grouping expressions are involved, full intermediary construction is performed by the *construction engine*. The rule dispatcher uses the *pattern matching engine* for the actual evaluation of Xcerpt queries compiled into AMAχOS code. The pattern matching engine uses variants of the algorithms described in [8] that are based on the *memoization matrix* for storage and access to intermediary results. The rule engine also detects calls to external functions or Web services and routes such calls to the *static function library*, that provides a similar set of functions as [18] which are implemented directly in the host machine and not as AMAχOS code.

For each goal rule in the AMAχOS programs the resulting substitution sets are handed over to the *construction engine* (possibly incremental) which applies any construction expressions that apply for that goal and itself hands the result over to the serialization layer or to the answer API.

The most notable feature of the AMAχOS query engine is the separation in three core engines: the construction, the pattern matching, and the rule engine. Where the rule engine essentially glues the pattern matching and the construction engine together, these two are both very much separate. Indeed, at least on the level of AMAχOS code even programs containing only queries (i.e., expressions handled by the pattern matching engine) are allowed and can be executed

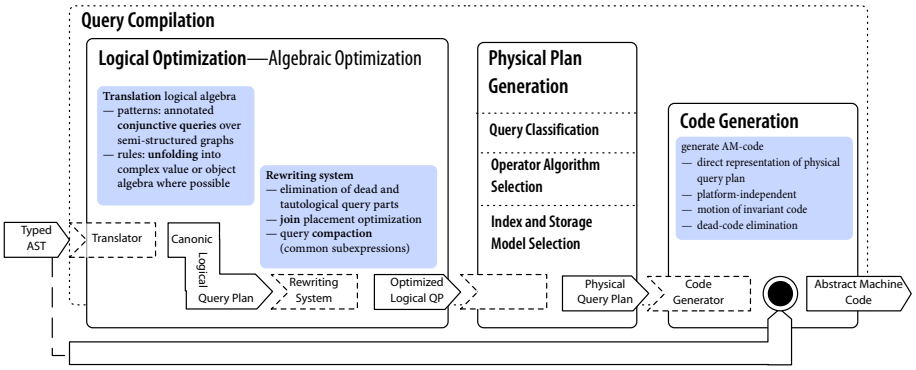

Fig. 5. Architecture of Query Compiler for AMAχOS

by this architecture (the rule dispatcher and construction engine, in this case, merely forwarding their input).

5.2 Query Compiler

Aside from the execution engine, the query compilation layer deserves a closer look. Here, an Xcerpt program—represented by an abstract-syntax tree annotated with type information—is transformed into AMAχOS code. It is assumed that the Xcerpt program is already validated, normalized, modules are expanded, and type information is added in the prior parsing layer. The query compilation is essentially divided in three steps: logical optimization, physical plan generation, and code generation.

Logical optimization is similar as in traditional database systems but additionally has to consider rules and rule dependencies: Xcerpt programs get translated into a logical algebra based on n-ary conjunctive queries over semi-structured graphs [8]. Expressions in this algebra are then optimized using various rewriting rules, including dead and tautological query part elimination, join placement optimization, and query compaction. Furthermore, where reasonable, rules are unfolded to avoid the construction of intermediary results during execution.

In contrast, *physical plan generation* differs notably, as the role of indices and storage model is inverted: In traditional databases these are given, whereas in the case of AMAχOS the query compiler generates code in the hint section indicating to execution engine and storage manager which storage model and indices (if any) to use. Essential for execution is also the classification of queries based on shape of the query and (static) selectivity estimates. E.g., a query with highly selective leaves but low selectivity in inner nodes is better evaluated in a bottom-up fashion, whereas a query with high selectivity in inner nodes profits most likely from a top-down evaluation strategy. Operator selection is rather basic, except that it is intended to implement also holistic operators for structural relations

where entire paths or even sub-trees in the query are considered as parameter for a single holistic operator, cf., e.g., [5, 21].

To conclude, the query compilation layer employs a mixture of traditional database and program compilation techniques to obtain an AMAχOS program from the Xcerpt input that implements the Xcerpt program and is, given the limited knowledge about the actual data instances, likely to perform well during execution. The compilation process is rather involved and expected to be time expensive if all stages are considered. A control API is provided to control the extent of the optimization and guide it, where possible. We believe that in many cases an extensive optimization is called for, as the query program can be reused and, in particular if remote data is accessed, query execution dominates by far query compilation.

6 Conclusion and Outlook

We present a brief overview over the principles and architecture of a novel kind of abstract or virtual machine, the AMAχOS virtual machine, designed for the efficient, distributed evaluation of Xcerpt query programs against Web data.

In particular, we show how the Web setting affects traditional assumptions about query compilation and execution and forces a rethinking of the conclusions drawn from these assumptions. The proposed principles and architecture reflect these changing assumptions

1. by emphasizing the *importance of a coherent and clearly specified execution environment* in form of an abstract machine for distributed query evaluation,
2. by *separating query compilation from query execution* (as in general programming language execution),
3. by employing a *unified query evaluation algorithm* for path, tree, and graph queries against tree and graph data, and
4. by emphasizing *adaptive optimization* as a means to ameliorate the loss of quality in predictive optimization due to lack of knowledge about remote or volatile data instances.

Implementation of the proposed architecture is still underway, first results on the implementation of the query engine have been reported in [8] and in [4], demonstrating the promise of the discussed method and architecture.

Acknowledgments This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

1. D. J. Abadi, D. Carney, U. Çetintemel, *et al.* Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

2. R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM SIGMOD*, pages 261–272, 2000.
3. S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.
4. S. Berger, F. Bry, T. Furche, *et al.* Beyond XML and RDF: The Versatile Web Query Language Xcerpt. In *Proc. Int'l. Conf. on World Wide Web*, 2006.
5. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD*, pages 310–321, 2002.
6. F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML Stream Query Processor SPEX. In *Proc. ICDE*. 2005.
7. F. Bry, T. Furche, *et al.* Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *J. of Semantic Web and Inf. Sys.*, 1(2), 2005.
8. F. Bry, A. Schroeder, T. Furche, and B. Linse. Efficient Evaluation of n-ary Queries over Trees and Graphs. Submitted for publication, 2006.
9. E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM TOCS*, 15(4):412–447, 1997.
10. B. Davis, A. Beatty, K. Casey, *et al.* The Case for Virtual Register Machines. In *Proc. Workshop on Interpreters, Virtual Machines and Emulators*, p. 41–49, 2003.
11. T. Garfinkel, B. Pfaff, J. Chow, *et al.* Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proc. of ACM S. on Op. Sys. Princ.*, p. 193–206, 2003.
12. R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, 7(6):34–45, 1974.
13. G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive Queries over Trees. *J. of the ACM*, 53(2), 2006.
14. L. J. Groves and W. J. Rogers. The Design of a Virtual Machine for Ada. In *Proc. ACM Symposium on Ada Programming Language*, p. 223–234, 1980.
15. ISO/IEC. 23271, Common Language Infrastructure (CLI). Int'l. Standard, 2003.
16. T. Johnsson. Efficient Compilation of Lazy Evaluation. *SIGPLAN N.*, 19(6), 1984.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 2nd edition, 1999.
18. A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2005.
19. A. Novoselsky. The Oracle XSLT Virtual Machine. In *XTech 2005*, 2005.
20. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Man.*, LNCS 2490, 2002.
21. P. O'Neil, E. O'Neil, S. Pal, *et al.* ORDPATHs: Insert-friendly XML Node Labels. In *Proc. ACM SIGMOD*, p. 903–908. 2004.
22. D. L. Overheu. An Abstract Machine for Symbolic Computation. *J. of the ACM*, 13(3):444–468, 1966.
23. S. Pemberton and M. Daniels. *Pascal Implementation: The P4 Compiler and Interpreter*. Ellis Horwood, 1982.
24. B. Randell and L. J. Russell. *ALGOL 60 Implementation*. Academic Press, 1964.
25. M. Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2(5):34–40, 2004.
26. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-order Stratified Logic Programs. *ACM TOPLAS*, 20(3), 1998.
27. S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.
28. S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.
29. Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual Machine Showdown: Stack versus Registers. In *Proc. Int. Conf. on Virtual Execution Env.s*, p. 153–163, 2005.
30. D. H. D. Warren. An Abstract Prolog Instruction Set. Note 309, SRI Int'l., 1983.