Comparing Machine Learning Approaches for Context-Aware Composition

Antonina Danylenko[1], Christoph Kessler[2], and Welf Löwe[1]

[1] Linnaeus University, Software Technology Group,
351 95 Växjö, Sweden
{antonina.danylenko,welf.lowe}@lnu.se

[2] Linköping University, Department for Computer and Information Science,
581 83 Linköping, Sweden
{christoph.kessler}@liu.se

Abstract. Context-Aware Composition allows to automatically select optimal variants of algorithms, data-structures, and schedules at runtime using generalized dynamic Dispatch Tables. These tables grow exponentially with the number of significant context attributes. To make Context-Aware Composition scale, we suggest four alternative implementations to Dispatch Tables, all well-known in the field of machine learning: Decision Trees, Decision Diagrams, Naive Bayes and Support Vector Machines classifiers. We assess their decision overhead and memory consumption theoretically and practically in a number of experiments on different hardware platforms. Decision Diagrams turn out to be more compact compared to Dispatch Tables, almost as accurate, and faster in decision making. Using Decision Diagrams in Context-Aware Composition leads to a better scalability, i.e., Context-Aware Composition can be applied at more program points and regard more context attributes than before.

Key words: Context-Aware Composition, Autotuning, Machine Learning

1 Introduction

Context-dependent computation is an essential part of a wide range of application domains, where an application should behave according to the conditions arising during the execution. A technique that enables this type of computation is called Context-Oriented Programming—a programming approach that treats context explicitly and makes it accessible and manipulatable by software [1, 2]. In order to get the desirable behavior of the application, a specific composition, called *Context-Aware Composition*, has to be made regarding the current context of the execution. Context-aware composition separates the concerns of defining component variants and the decision in favor of any of these variants in a composition context. The former is done by component designers who develop variants that might be advantageous in certain composition contexts possibly supported by variant generators, generating for instance different schedules. The latter is fully automated. In a learning phase, variants are tested in different contexts, and the champion variant of each context is captured in a generalized Dispatch Table that is later used to compose with this variant in the actual composition context. The learning and composition phases can be separated (offline learning) or interleaved (online learning) leading to self-adaptive systems.

Context-aware composition can improve performance of software systems considerably as it dynamically composes with the prospected best variant of alternative algorithms, data structures, etc. Composition context can range from domain-specific to technology-dependent attributes, and even include properties that may be based on hardware or software (e.g. number of available processors, problem size, etc.). Hence the number of attributes in Dispatch Table can be very large. Scalability of context-aware composition generally depends on scalability of variants-modeling technique that, in turn, depends on the actual number of contexts. However, Dispatch Tables are not very scalable. Capturing the learned knowledge in Dispatch Tables mapping context attributes to variants can be memory consumptive: the tables grow exponentially with the number of context attributes. Hence, capturing these tables for every composition point, i.e., all calls to a function f of a component with several implementation variants, might lead to problems with the overhead in memory consumption, implying that the amount of required memory for execution of a certain application might be dramatically increased.

To remedy this problem, we enhance the scalability of the variants-modeling technique by introducing technology from machine learning. In fact, we learn decision functions $d_f : X_1, ..., X_n \rightarrow Y$, with X_i a table dimension (i.e., a context attribute) and Y the table entry (i.e., the presumably best algorithm variant of f). The actual composition (dynamic dispatch) then evaluates the function d_f for the actual context attributes and invokes the best variant. Therefore, as always in machine learning, we need to bias:

– Accuracy of the decision: does d_f always decide for the optimal variant, and what is the impact of suboptimal decisions on the overall performance,
– Decision time and its impact on the overall performance,
– Memory size for capturing d_f.

For this work, we assume an offline learning and hence ignore the learning time.

The remainder of this paper is structured as follows: Section 2 introduces the different technologies for learning decision functions and assesses their theoretical memory consumption and decision time. Section 3 assesses their memory consumption and decision overhead practically in a number of experiments, compares their accuracy with Dispatch Tables as the baseline, and shows the speedups of context-aware composition using different decision function implementations. We base our experiments on a sorting problem. Sorting is an extreme problem for context-aware composition in the sense that the ratio of dynamic decisions and payload operations is rather high. However, the insight on this extreme case can deliver a guidance for other cases, since it presents a general overview and interpretation of each learning technology. Section 4 discusses related work, and Section 5 concludes the paper and points out directions of future work.

2 Assessment of Learning Strategies

Learning is the process of constructing, from training data, a fast and/or compact surrogate function that heuristically solves a decision, prediction or classification problem for which only expensive or no algorithmic solutions are known. It automatically abstracts from sample data to a total decision function.

Different learning strategies might be employed; depending on the strategy and the problem to solve, resulting classifiers differ in speed of learning and deciding, their memory consumption, and their decision accuracy. Machine learning strategies are usually chosen based on the problem domain, sometimes even on the sample data [3, 4]. However, bounds on memory and decision overheads can be theoretically estimated for every learning strategy regardless of the specific problem domain. In this section, we present such bounds and compare memory consumption and decision overhead for four classifiers representing three most common used generalization models [5]: Naive Bayes Classifier representing the Probability model; Decision Trees and Decision Diagrams representing the Tree-based model; Support Vector Machines representing the Maximum Margin model. Additionally, we take Dispatch Tables as our baseline decision function implementation.

Dispatch Tables are implemented as n-dimensional arrays, n the number of context attributes. Each dimension i contains entries corresponding to the sample values of the context attribute X_i. Thus, the memory consumption M of the Dispatch Table can be approximated from below by

$$M = size \times m^n$$

where *size* bytes are necessary to encode all variants in Y, and m is the minimum number of samples of any of the context attributes.

Before making a decision, the closest sample point for each actual context attribute value needs to be computed. Assume we sample discrete attributes completely and continuous attributes with a logarithmic distance. Then we find the index corresponding to a discrete attribute value by a mapping table lookup, and the index of a continuous attribute value by computing the logarithm of the actual attribute value. Additionally, each access to an n-dimension array is basically an access to a 1-dimensional array requiring some offset calculations:

$$offset = base\_address + (((d_1 \times |X_1| + d_2) \times \ldots |X_{n-1}| + d_{n-1}) \times |X_n| + d_n) \times size,$$

where d_i is the index and $|X_i|$ the sample size of the context attribute X_i. Therefore, a decision time for an n-dimensional Dispatch Table can be estimated as

$$T(n) = (log \times k + n) \times T_{flop} + (n - k + 1) \times T_{aa} + c,$$

where k is the number of continuous attributes, log is the number of floating point operations for calculating the logarithm[3], T_{flop} is the time for a flop, T_{aa} is

[3] Many processors provide the integer $log2$ in a single instruction in hardware; in our Java implementation we need 21 flops.

the array access time, and c is a constant time used for small operations. Dispatch Tables capture variants at sample points of continuous attributes. Finding the sample point closest to an actual context attribute value must be done prior to each table lookup. In contrast, classification models learned by different learning strategies can decide on continuous attributes directly as discussed below.

Decision Trees are trees encoding context attributes in the inner nodes. Each outgoing edge of such a node corresponds to a value (or value range) of the context attribute. Each path from the root node to a leaf in the Decision Tree represents actual context values leading to a classification result. Using a Decision Tree as a dispatch repository for a given context $(X_1, \ldots, X_n)$ is straight forward: in each inner node corresponding to attribute X_i, starting at the root, we use the attribute value x_i^j to decide which child to visit next. Decision Trees could capture Dispatch Tables without any loss of accuracy. However, learning algorithms compute approximations in order to avoid Decision Trees to over-fit the sample data. Most of these algorithms employ a recursive top-down approach that changes the order of the attributes for each path in order to select the most relevant attributes first and possibly ignore the less relevant attributes [6]. Assume an initial data set R of tuples $(x_1, \ldots, x_n) \to y$ and root node r:

– *Base Case:* If all instances in R belong to the same class label y, then r is a leaf with label y.
– *Recursive:* Otherwise, select one attribute X_i and create a child node $node^j$ for each possible attribute value x_i^j. Partition R into smaller subsets R^j based on their particular value for the attribute X_i. Apply the algorithm recursively on each child node $node^j$ using data sets R^j.

The number of tests necessary to reach a leaf is equal to the *depth* of the Decision Tree. This *depth* varies around the number n of context attributes: for *discrete* context attributes it is at most n; *continuous* attributes can even occur several times on the path due to data partitioning [7]. So, generally, the prediction time is $depth \times T_{aa}$ and we approximate

$$T(n) \approx n \times T_{aa} + c.$$

In the worst case, the memory required for capturing the Decision Tree is even larger than for the corresponding table: k leaves if the table has k entries, and (almost) k inner nodes. This size reduces when the learning strategy approximates the decisions. It can also be reduced if all paths from an inner node lead to the same decision (making this whole subtree redundant). Hence, the memory consumption is

$$M = size \times edges,$$

where *edges* is number of edges in the tree (assuming that *size* bytes are even sufficient to encode all different nodes).

Decision Diagrams represent Decision Trees in a compact way by eliminating redundant subtrees. In particular, diagrams are a generalization of Ordered Binary Decision Diagrams (OBDDs) [8, 9], known as a compact representation of Boolean functions. In practice, they reduce the exponential memory consumption of table representations of these functions to acceptable sizes.

We do not have binary but general decisions, since a context attribute could have any domain, and the function (variant) values are not binary either, since there are, in general, more than two variants to choose from. These more general Decision Diagrams are known as *Multi-Terminal Multi-Valued Decision Diagrams* (MMDDs) [10]. They correspond to rooted directed acyclic graphs with inner nodes representing context attributes and leaves representing decisions. Generally, decision time and worst case memory size of diagrams are the same as for Decision Trees. Thus, the worst case memory and look-up overhead for Decision Trees and Decision Diagrams are equal, but, due to the elimination of redundancies, the size is expected to be considerably smaller in practice.

A Naive Bayes classifier is a simple probabilistic classifier based on Bayesian statistics. It naively assumes conditional independence of the context attributes from each other using a simple classification method which classifies an item by determining a probability of its belonging to a certain class $y \in Y$ [11, 12].

For example, we wish to approximate a target function: $\boldsymbol{X} \to Y$, or equivalently $P(Y|\boldsymbol{X})$. Let us assume that Y is a binary class with values y_1 and y_2 and $\boldsymbol{X}$ is a vector space of n context attributes. Applying Bayes' rule, $P(Y = y_1|\boldsymbol{X})$ can be calculated as:

$$P(Y = y_1|\boldsymbol{X}) = \prod_{i=1}^{n} \frac{P(\boldsymbol{X} = \boldsymbol{x}_i|Y = y_1) \times P(Y = y_1)}{P(\boldsymbol{X} = \boldsymbol{x}_i|Y = y_1) \times P(Y = y_1) + P(\boldsymbol{X} = \boldsymbol{x}_i|Y = y_2) \times P(Y = y_2)},$$

where $\boldsymbol{x}_i$ denotes the ith possible vector of $\boldsymbol{X}$, and the summation in the denominator is done over all values of class Y.

Naive Bayes classifiers use the sample data and compute all possible probabilities for each attribute vector $\boldsymbol{x}_i$ over all class values y_k. As a result, a Naive Bayes classifier is just a set of probabilities that are accessed during classification for computing the most probable variant. For a discrete attribute X_i, the probability is stored in an array with $|Y| \times |X_i|$ elements; for a continuous attribute, a *mean* and a *variance* are computed and stored in two arrays of size $|Y|$. So the memory consumption with k continuous attributes is

$$M = |Y| \times size \times (2k + 1 + (n - k) \sum_{i=1}^{(n-k)} |X_i|).$$

A decision using a Naive Bayes classifier takes quite some time: it requires 4 flops for each discrete attribute and 88 flops for each continuous attribute (including mathematical operations computing Gaussian) for each possible class. Thus, the decision time is estimated as

$$T(n) = (4n + 84k) \times T_{flops} + (2n + k) \times T_{aa} + c.$$

Support Vector Machines (SVM) construct a linear separating hyperplane in the vector space defined by the table dimensions with the maximal margin in a higher dimensional space. If the vectors cannot be separated by planes, artificial dimensions can be introduced using so-called kernel functions [13].

We refer to LIBSVM [14], a library for Support Vector Machines, as a black box with a radial basis function for the kernel function and follow the normalization and parameter finding suggestions detailed in [15].

SVMs can be trained to be very accurate classifiers. However, the training time of SVMs can be very long, especially on large data sets. Training an SVM requires solving a constrained quadratic programming (QP) problem which usually takes $O(s^3)$ computations, where s is the number of sample points [16].

More importantly, deciding requires an evaluation of the kernel function that takes for the selected kernel approximately $20 + (n + |Y|)$ flops for each support vector and a number of additional flops for finding the actual variant. The number l of support vectors is usually proportional to s, so we can approximate

$$T(s, n) \approx T(l, n) \approx (20 + n + 2|Y| - 1) \times l \times T_{flops} + l \times (2|Y| - 1) \times T_{aa} + c.$$

The encoded SVM classifier has a quite high memory consumption for capturing the support vectors, approximated with

$$M = 12l + 24|Y| + 4|Y|^2 + 100s$$

double precision values.

Based on these theoretical estimations, we cannot decide which classifier to prefer. It depends on the bias between acceptable decision time and memory overhead, and on the concrete problem, i.e., the number of context attributes, sample points etc. However, once the number of attributes and the sample data points are decided, the above approximations can be used to (pre-)select a preferred classifier. We will discuss this in the next section by instantiating the approximations for a concrete Context-Aware Composition example comparing theoretical assessments with results of practical measurements.

3 Experiments

Algorithm Variants We implemented the well-known sorting algorithms Selection sort, Quicksort, and Merge sort along with two parallel versions of Quicksort and Merge sort, which spawn a new thread for one of the sub-problems in each divide step. We used textbook implementations [17] and did not optimize the variants, especially, we greedily create new threads regardless of the number of cores available. All algorithms are implemented in Java/JDK 1.6.
Platforms All experiments are executed on three different multi-core machines:

M1 a 2 core Dell Latitude PC running Windows XP (2002, SP 3) on an Intel Dual Core T2300 at 1.66GHz and 2GB RAM,
M2 an 8 core IBM Blade Server running Linux (CentOS) on an Intel 8 Core 5450 at 3GHz and 16GB RAM, and

	Table	Tree	Diagram	Bayes	SVM
Classifier size in bytes	136	24/40	24/40	80	544

Table 1. Memory overhead of different classification models.

M3 a 2 core MacBook Pro running Mac OS X (10.6.5) on an Intel Core i5 at 2.4 Ghz and 8GB RAM.

All tests are run on the respective native JVMs with setting -Xms128m -Xmx1g.

3.1 Memory Overhead of the Approaches

To compare different classification approaches, we constructed three Dispatch (decision) Tables for our sorting problem for different multi-core machines (M1, M2, M3). The dispatch technique attempts to speed up sorting by selecting the best algorithm (class) Y for the current context (problem size N, processor availability P), where N is a continuous integer sampled at powers of two between $2^0 \ldots 2^{16}$, P is boolean with 0 and 1 as possible values (encoding whether or not processors are available). Y is a discrete integer in the range $1 \ldots 5$ each representing one of the algorithm variants. The memory needed for storing $2 \times 17 = 34$ entries is rather small, i.e. $M = 4 \times 34 = 136$ bytes, cf. Table 1 for the size of the Dispatch Table and the alternative classifiers constructed as discussed in Section 2. However, for a higher number of context attributes the memory need will grow exponentially.

To encode a Dispatch Table in a Decision Tree or a Decision Graph, we used the FC4.5 learning algorithm, a fast and efficient implementation of the well-known and widely used classification algorithm C4.5 [18]. Decision Trees have a moderate learning effort. The memory compression rate is quite high: $\approx 82\%$ for M1, M2 (6 *edges*, requiring 24 bytes) and $\approx 71\%$ for M3 (10*edges*, 40 bytes).

The Decision Diagram is a redundancy-free encoding of the Decision Trees and, hence, the depth does not change. However, Dispatch Table compression does not improve compared to Decision Trees (the number of edges is the same). In our example, the diagram uses up to 27% fewer nodes than the tree.

The Naive Bayes classifier was the fastest to construct and the encoded Dispatch Tables take only 80 bytes giving a 41% of reduction immediately without any additional optimizations, since the size of the classifier only depends on the context attributes, not their values.

In order to construct an accurate prediction model based on Support Vector Machines, a learning phase requires an optimization of kernel function parameters specific to each problem domain and input sample data. Although done automatically, finding parameters giving an acceptable decision accuracy requires a cross-validation pre-learning over the large range of parameter values. In our example, (pre-)learning time was still acceptable in range of a few seconds. The memory required to encode the SVM classifier is 544 bytes (based on the LIBSVM library implementation). This is the highest memory overhead and even 75% larger than Dispatch Table memory consumption.

Platform	Problem	Tree [%]	Diagram [%]	Bayes [%]	SVM [%]
PC 2 cores (M1)	Sample points	0	0	15	0
	10.000	21	21	0	10
	100.000	21	21	4	10
	1.000.000	21	21	100	10
Server 8 cores (M2)	Sample points	0	0	9	0
	10.000	25	25	0	0
	100.000	25	25	1	0
	1.000.000	25	5	100	25
MAC 2 cores (M3)	Sample points	0	0	6	0
	10.000	21	21	0	9
	100.000	21	21	3	9
	1.000.000	21	21	100	9

Table 2. Errors of different decision approaches.

3.2 Decision Accuracy of the Approaches

In this section, we compare the accuracy of the different decision approaches. As we know the right decision for each actual context (processors available and problem size) only for the sample points measured in the training phase, we can assess accuracy only approximatively: (1) by comparing the decisions of the different approaches at these sample points, and (2) by comparing their decisions with the decisions of the table approach as the baseline. We define a decision error as (1) the ratio of decisions not suggesting the best algorithm variant in the sample points over all decisions, and (2) the ratio of decisions diverging from the suggestion of the decision table over all decisions. The Dispatch Table captures the best implementation for the sample points. Hence, its error is 0 in the measure (1). However, we do not know if the Dispatch Table suggests the best variant between sampled problem sizes. Hence, (2) is an accuracy measure relative to the baseline implementation of Context-Aware Composition using Dispatch Tables.

For the different platforms, Table 2 shows the error (1) at the sample points and the error (2) for selected sizes of arrays to sort. All ratios are given in %.

Dispatch Tables (trivially), Decision Trees and Diagrams, and Support Vector Machines (cf. rows "Sample points") are to 100% accurate according to measure (1) and, generally, all decision approaches perform very accurate at the sample points with an error of at most 15%.

The error is somewhat higher for the accuracy measure (2). Independent of the problem size, Decision Trees and Diagrams suggest in 21% to 25% of the cases different algorithm variants than the Dispatch Table (cf. columns "Tree" and "Diagram"). For the Bayesian classifier and the classifier based on Support Vector Machines, the error (2) increases with the problem size. However, it is in most cases smaller than for Decision Trees and Diagrams. Note that the error of 100% for the Bayesian classifier for the very large problems is due to its first (wrong) decision in favor of Selection sort. As Selection sort does not contain

	Table	Tree	Diagram	Bayes	SVM
Flops	22	0	0	46	644
Array access	1	2.5	2.5	5	140

Table 3. Decision overhead (# operations) of different classification models.

any recursive calls, there are no other decision points and this wrong decision is the only one.

As a conclusion, none of the approaches ought to be dropped from further evaluations since their accuracy is comparable with that of Dispatch Tables.

3.3 Decision Overhead of the Approaches

We compare the overhead of the different approaches for decision making first based on our theoretical assumptions and then experimentally.

For the 2-dimensional Dispatch Table based on one continuous and one discrete attribute, the prediction time includes 22 floating point operations and one array access, that is $T_{flops} \times 22 + T_{aa}$. It is worth mentioning that in our cost model the time for a look-up to any 2-dimension Dispatch Table is constant and does not depend on the number of entries stored, which is a simplification ignoring caching effects. Table 3 shows the floating point operations and the number of array accesses for the classifiers.

The decision time required by Decision Trees and Diagrams depends on the tree's and diagram's depth, respectively. For each of the three Decision Trees (Diagrams) constructed, the decision requires at most three indirect array accesses, since the maximum depth is three for all tables (the average depth is 2.5). Thus, the expected overhead can be estimated as $T_{aa} \times 2.5$.

As discussed, a Bayes classifier is quite effective in memory consumption but has quite a high runtime overhead for making decisions. In our setting, it requires 46 floating point operations and 5 array accesses, $T_{flops} \times 46 + T_{aa} \times 5$. This is twice as many flops and five times as many array accesses compared to Dispatch Table, making it a slow classifier for our problem.

Decisions in SVM includes 21 flops that correspond to a kernel function computation calculated for each of the $l = 28$ support vectors and then additional 56 flops for classification. Altogether, the decision phase takes 644 flops and the whole decision time is $T_{flops} \times 644 + T_{aa} \times 140$. This is the highest look-up overhead among all classification models.

Experiments confirm these overhead predictions. For these experiments, we use the fastest homogeneous solution – Quicksort – as the baseline. The third column of Table 4 shows the time for Quicksort for the different platforms and three selected array sizes to sort.

On each recursive invocation of Quicksort (with a certain number of processors still available and a certain sub-problem size), we look up the best algorithm variant in the different decision repositories. However, in order to compare the overhead in a fair way, we always invoke Quicksort even in Context-Aware

Platform	Problem size	Quicksort in msec.	Table in %	Tree in %	Diagram in %	Bayes in %	SVM in %
PC 2 cores (M1)	10.000	3.62	43	25	22	1161	4099
	100.000	51.91	28	14	10	816	2871
	1.000.000	998.36	15	8	6	423	1449
Server 8 cores (M2)	10.000	1.87	52	34	34	680	1595
	100.000	22.93	55	29	30	553	1321
	1.000.000	552.51	24	12	12	243	576
MAC 2 cores (M3)	10.000	1.99	37	36	33	328	1008
	100.000	28.08	29	19	18	224	707
	1.000.000	478.55	25	16	15	139	426

Table 4. Time overhead (in %) of different decision approaches.

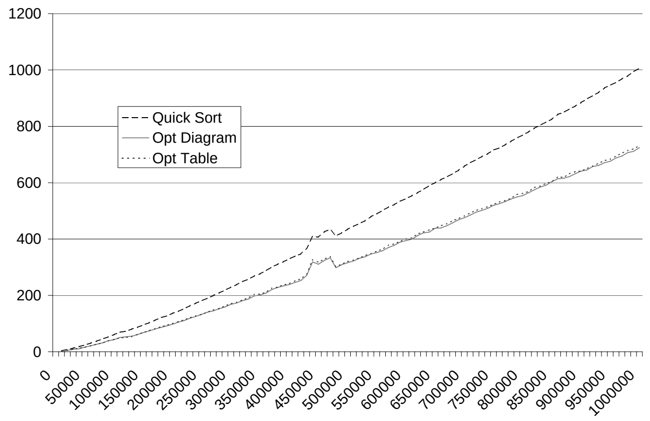
Composition regardless of the best algorithm variant suggested. Hence, all compared solutions essentially implement Quicksort.

Table 4 shows the execution times of the Context-Aware Composition based on the different decision functions relative to the execution time of Quicksort on the same platform and architecture. For instance, the Dispatch Table (cf. column "Table") introduces an overhead of 43% on the PC with 2 cores and the problem size of 10,000 array elements as it requires 1.43×3.62msec = 5.18msec of the corresponding Quicksort execution times without table lookup (3.62msec).

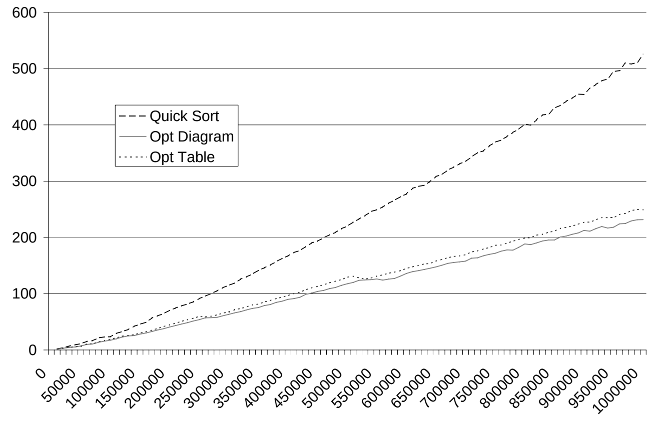
There are $O(N)$ expected lookups for problems of size N in Quicksort; the expected work of Quicksort is $O(N \log N)$. The lookup time is $O(1)$; it only depends on the number of algorithm variants, the number of context attributes and the decision repository variant, but it does not grow with the problem size. Hence, for all decision repository variants, the overhead gets smaller with increasing problem size. This is confirmed by the measurements.

The Decision Diagram (column "Diagram") introduces the lowest overhead in almost all cases with the Decision Tree (column "Tree") not far behind. These overheads are between 6% and 36% depending on problem size and platform. The Dispatch Table comes with an overhead between 15% and 55%. Algorithm variant selection using Bayesian classifiers and Support Vector Machines (columns "Bayes" and "SVM", resp.) slows down the execution by factors between 2.4 and almost 42. Quicksort (and other recursive sorting algorithms) are extreme in the sense that ratio between decision points (recursive calls) and workload is rather high. Therefore, we observed a rather high overhead for Context-Aware Composition compared to the homogeneous Quicksort variant. In conclusion, Dispatch Tables, Trees, and Diagrams introduce an overhead that is still acceptable as it can be expected that a clever algorithm variant selection compensates for the overheads. At least for recursive sorting problems, this is not the case for Bayesian classifiers and Support Vector Machines.

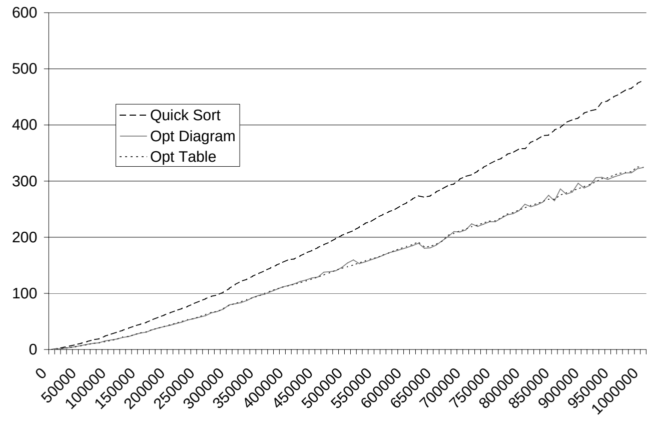
As the conclusion, we discard Bayesian classifiers and Support Vector Machines as decision repositories for improving sorting using Context-Aware Composition due to their (too) high decision overhead. As Decision Trees and Dia-



(a) PC 2 cores (M1).



(b) [illegible] (M2).



(c) MAC 2 cores (M3).

Fig. 1. Homogeneous Quicksort and Context-Aware Sorting using Decision Diagrams ("Opt Diagram") and Dispatch Tables ("Opt Table"). The *x*-axis displays the array size, the *y*-axis the time in *msec*.

grams by construction always suggest the same variant and Decision Diagrams are smaller in size and have the slightly smaller overhead, we discard Decision Trees in the final overall assessment.

3.4 Overall Performance

Now we are ready to assess the overall performance of Context-Aware Composition using Decision Diagrams vs. using Dispatch Table. Figure 1 shows the experimental results on the different platforms. As a reference, it also shows how the fastest homogeneous implementation variant (sequential Quicksort) performs on the three platforms.

On the PC with two cores, the optimized version using Decision Diagrams gains a speed-up of 1.47 over sequential Quicksort, on average over all array sizes from $10,000 - 1,000,000$ (step $10,000$) while the optimized version using Dispatch Tables gains a speed-up of 1.46. On the server with 8 cores, the difference between the two implementations is even more pronounced: an average speed-up of 1.92 for the diagram-based solution vs. 1.79 for the table-based solution. On the MAC with 2 cores, the speed-up results are 1.39 vs. 1.37, again in favor of the diagram-based solution.

Obviously, the decisions of the Dispatch Table are (slightly) more accurate, but this is more than compensated with the lower runtime overhead of the Decision Diagrams. Altogether, the experiments showed that the Decision Diagram is not only smaller by a factor of five than the Dispatch Table, but also (slightly) faster when used as a decision repository in Context-Aware Composition.

4 Related work

Context-Aware Composition, i.e., the run-time context dependent binding of a call to a matching callee defined by another component, is gaining importance as a knob for performance optimization, in particular since the stagnation of CPU clock rates puts an urgent need to exploit new sources for performance improvements. Andersson *et al.* compose and optimize special implementations of data structures and algorithms, considering matrix multiplication as a case study [19]. This work may be considered as a generalization of the dispatch mechanism in object-oriented languages, for instance in the sense of Context-Oriented Programming (COP) [20, 1, 2].

The optimization of libraries for specific functionality such as linear algebra or signal processing is a natural target for optimized composition because the domain and code base is limited and statically known, computations can often be described in a restricted domain-specific language from which variants can be generated and tuned automatically, and because even high off-line tuning times are acceptable due to the high reuse of libraries. Well-known examples include the library generators ATLAS for basic linear algebra computations and FFTW and SPIRAL [21, 22] for transforms in signal processing.

More recently, optimized composition has been proposed as an optimization technique also in the more general context of component based systems, where the programmer is responsible for annotating components so that their composition can be optimized for performance. However, only few approaches consider recursive components with *deep composition*, and only few consider the co-optimization of the selection of implementation variants with other variation possibilities, such as the layout and data structure of operands or scheduling.

Li *et al.* [23] implement a library generator for sorting that uses dynamic tuning to adapt the library to the target machine at installation time. They use a number of machine parameters (such as cache size, the size of the input and the distribution of the input) as input to a machine learning algorithm. The machine learning algorithm, a problem-specific combination of two different classifiers, is trained to pick the best algorithm for any considered scenario.

Large-scale distributed memory systems as used in high-performance computing are usually programmed in SPMD (single program, multiple data) style using MPI or partitioned global address space languages to enable tight resource control. Brewer [24] investigated dynamic algorithm selection for such a system for sorting and PDE solving, and also considered limited support for the dynamic selection of array distributions on distributed shared memory systems. The run-time predictor is constructed from measured samples by calibrating the parameters of a generic prediction function that is semi-automatically generated from user-provided information about relevant properties and ranges, using linear regression. Our learning approach differs in that we require no assumptions or user hints about the terms that occur in the generic run-time prediction function.

STAPL [25] non-recursively applies dynamic algorithm selection for sorting and matrix computations. Three different learning algorithms are used: a Decision Tree learner based on Quinlan's ID3 algorithm with different pruning strategies, found to perform best in the experimental evaluation, a standard feed-forward neural network with back-propagation learning, and a Naive Bayes classifier found to be inferior in its classification accuracy.

Yu and Rauchwerger [26] take a similar approach for reductions. From measurements of the individual implementation variants they construct predictor functions in two steps. First, they select a small number of polynomial terms in the context and machine model parameters from a term pool to build a generic prediction function. Second, they calibrate the coefficients from the training data using general linear regression and Decision Tree learning. For each call, the calibrated run-time prediction functions are then evaluated and the decision is memorized so it can be reused if the same parameter configuration should occur again. This way, the overhead of the dispatch at each call is reduced.

Kessler and Löwe [27]consider optimized composition at the level of annotated user-defined components (i.e., not limited to closed libraries) together with scheduling, resource allocation and other optimizations, which allows for simultaneous optimization. For the off-line search and optimization phase, the approach uses an interleaved dynamic programming algorithm to construct a component variant Dispatch Table (V-table) for each component-provided functionality f and a resource allocation and schedule Dispatch Table (S-table) for independent calls. The tables are constructed simultaneously bottom-up for increasing problem sizes and resource assignments.

Olszewski and Voss [28] proposed a dynamic adaptive algorithm selection framework for divide-and-conquer sorting algorithms in a fork-join parallel setup. Their approach is divided into two phases. First, they use a dynamic programming algorithm to select the best sequential algorithm for different problem sizes. Then, they determine the threshold problem sizes for when to submit the subproblems to a shared work queue and execute in parallel rather than to execute sequentially. For the construction of a classifier, they use the C4.5 algorithm to generate a Decision Tree. While apparently sufficient prediction accuracy was achieved in the considered examples, the resulting sizes and overheads are not discussed. PetaBricks [29] applies a similar approach where, in the offline search phase, the variant choice functions for recursive components are not computed by dynamic programming but by a genetic algorithm, essentially applying heuristic cuts to the optimization space. There appears to be no further compression of the variant selection function. Schedules and resource allocation are not co-optimized with variant selection but delegated to a run-time system with a work-stealing dynamic scheduler. Numerical accuracy is considered as an additional algorithmic property in composition. The recent paper by Wernsing and Stitt [30] presents an approach that builds upon PetaBricks but targets multicore CPU systems with FPGA-based accelerators. Like PetaBricks, this approach relies on dynamic scheduling and the learning is done offline, and like the approach by Kessler and Löwe [27],the learned execution plan is stored in table form.

5 Conclusions and Future Work

This paper contributed with:

1. A framework for plugging classifiers into Context-Aware Composition. It was instantiated with 5 classifier variants: Dispatch Tables (baseline), Decision Trees, Decision Diagrams, Naive Bayesian, and SVM-based classifiers.
2. A theoretical assessment of memory and decision overhead of the four alternative classification technologies.
3. A practical assessment on three different hardware platforms using Sorting as a running Context-Aware Composition example.

It turns out that Decision Diagrams are the preferred learning technology: they reduce memory consumption (in our example by a factor of five) and increase the overall performance (in our example by a few percent). Future work needs to validate this observation in extended experiments.

Sorting is an extreme problem for Context-Aware Composition in the sense that the ratio of dynamic decisions and payload operations is rather high. We need to extend our experimental basis to problems with a lower rate of decisions and payload, which possibly allows for a higher overhead in decision making. In these scenarios, speed-ups could be achieved using Bayesian and SVM-based classifiers benefiting from their higher decision accuracy despite their high decision overhead.

Also, composition based on more context attributes (e.g., including data-structure implementations of arguments and results) ought to be considered. This leads to larger Dispatch Tables and allows to validate the memory reduction of Decision Diagrams and their alternatives. Moreover, we need to reevaluate our findings in online learning scenarios as required in self-adaptive systems. Here, we would change the system environment, e.g., force energy saving mode or generate extra workload outside the program's control, which invalidates the learned best-fit variants. Execution of suboptimal variants detects such changed system environments. In these scenarios, learning time becomes crucial as it adds to the overhead of Context-Aware Composition.

Finally, we used the different learning technology (almost) as black boxes. Adapting them to our specific application context could improve the overall speed-up (and the performance of the different technologies relative to each other) and further reduce the memory consumption. For instance, we could capture the Decision Diagram in a very compact array containing indices (inner nodes) and variants (leafs). Also, we could split the SVM classifier in several classification steps each possibly requiring a linear kernel function making them both faster and more compact and, hence, competitive again.

Acknowledgments C. Kessler: EU FP7 project PEPPHER (#248481), www.peppher.eu.

References

1. R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented programming," *Journal of Obj. Tech., March-April 2008, ETH Zurich*, vol. 7, no. 3, pp. 125–151, 2008.
2. P. Costanza and R. Hirschfeld, "Language constructs for context-oriented programming: an overview of contextl," in *Proc. of the 2005 symposium on Dynamic lang.*, ser. DLS '05. New York, NY, USA: ACM, 2005, pp. 1–10.
3. M. S. P.-N. Tan and V. Kumar, *Introduction to Data Mining*. Addison Wesley, 2005.
4. N. J. Nilsson. (1996) Introduction to machine learning: An early draft of proposed text book. Stanford University. Stanford.
5. J. Han and M. Kamber, *Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, 2nd ed. Morgan Kaufmann, Sep. 2000.
6. M. Moshkov, "Algorithms for constructing of decision trees," in *PKDD '97: Proc. of the First European Symposium on Principles of Data Mining and Knowledge Discovery*. London, UK: Springer-Verlag, 1997, pp. 335–342.
7. L. Rokach and O. Maimon, *Data Mining with Decision Trees: Theory and Applications*. Singapore: World Scientific, 2008.
8. R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, 1992.
9. ——, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, 1986.

10. S. Johnson, *Branching programs and binary decision diagrams: theory and applications by Ingo Wegener society for industrial and applied mathematics*. New York, NY, USA: ACM, September 2010, vol. 41.
11. T. M. Mitchell, *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.
12. E. J. Keogh and M. J. Pazzani, "Learning augmented bayesian classifiers: A comparison of distribution-based and classification-based approaches," 1999.
13. C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, September 1995.
14. C.-C. Chang and C.-J. Lin. (2001) Libsvm – a library for support vector machines. National Taiwan University, Dep. of Comp. Science and Inf. Eng.
15. C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A practical guide to support vector classification," National Taiwan University, Dep. of Comp. Science, Tech. Rep., 2003.
16. K. A. Kramer, L. O. Hall, D. B. Goldgof, A. Remsen, and T. Luo, "Fast support vector machines for continuous data," *Trans. Sys. Man Cyber.*, vol. 39, pp. 989–1001, 2009.
17. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. New York: The MIT Press, 2001.
18. J. R. Quinlan, *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.
19. J. Andersson, M. Ericsson, C. Kessler, and W. Löwe, "Profile-guided composition," in *Proc. 7th Int. Symp. on Software Composition (SC 2008) at ETAPS, Budapest, Hungary, March 2008. Springer LNCS 4954*, 2008, pp. 157–164.
20. M. von Löwis, M. Denker, and O. Nierstrasz, "Context-oriented programming: beyond layers," in *Proc. of the 2007 Int. Conf. on Dynamic lang.: in conjunction with the 15th Int. Smalltalk Joint Conf. 2007*. New York, USA: ACM, 2007, pp. 143–156.
21. J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, B. Singer, M. Veloso, and J. Xiong, "Generating platform-adapted DSP libraries using SPIRAL," in *High Performance Embedded Computing (HPEC)*, 2001.
22. J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, and M. Veloso, "SPIRAL: Automatic implementation of signal processing algorithms," in *High Performance Embedded Computing (HPEC)*, 2000.
23. X. Li, M. J. Garzarán, and D. Padua, "A dynamically tuned sorting library," in *Proc. CGO'04*, 2004, pp. 111–124.
24. E. A. Brewer, "High-level optimization via automated statistical modeling," in *PPoPP'95*, 1995.
25. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *Proc. ACM SIGPLAN Symp. on Princ. and Pract. of Parallel Programming*. ACM, jun 2005, pp. 277–288.
26. H. Yu and L. Rauchwerger, "An adaptive algorithm selection framework for reduction parallelization," *IEEE Trans. Par. Distr. Syst.*, vol. 17, pp. 1084–1096, 2006.
27. C. Kessler and W. Löwe, "A framework for performance-aware composition of explicitly parallel components," in *ParCo-2007, Jülich/Aachen, Germany, Sep. 2007. In C. Bischof et al. (eds.): Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing Series, Volume 15, IOS Press*, 2008, pp. 227–234.
28. M. Olszewski and M. Voss, "An install-time system for the automatic generation of optimized parallel sorting algorithms," in *Proc. PDPTA'04, Vol. 1*, jun 2004.
29. J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe, "PetaBricks: a language and compiler for algorithmic choice," in *Proc. ACM SIGPLAN Conf. on Progr. Language Design and Implem.* ACM, 2009, pp. 38–49.
30. J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *Proc. ACM conf. on Lang., compilers, and tools for embedded systems (LCTES'10)*. ACM, 2010, pp. 115–124.