Towards Adjusting Mobile Devices to User's Behaviour

Fricke, P. [*], **Jungermann, F.**[*], **Morik, K.**[*], **Piatkowski, N.**[*], **Spinczyk, O.**[†], and **Stolpe, M.**[*]
Technical University of Dortmund

Abstract

Mobile devices are a special class of resource-constrained embedded devices. Computing power, memory, the available energy, and network bandwidth are often severely limited. These constrained resources require extensive optimization of a mobile system compared to larger systems. Any needless operation has to be avoided. Time-consuming operations have to be started early on. For instance, loading files ideally starts before the user wants to access the file. So-called prefetching strategies optimize system's operation. Our goal is to adjust such strategies on the basis of logged system data. Optimization is then achieved by predicting an application's behavior based on facts learned from earlier runs on the same system. In this paper, we analyze system-calls on operating system level. The learned model predicts if a system-call is going to open a file fully, partially, or just for changing its rights.

1 Introduction

Users demand mobile devices to have long battery life, short application startup time, and low latencies. Mobile devices are constrained in computing power, memory, energy, and network connectivity. This conflict between user expectations and resource constraints can be reduced, if we tailor a mobile device such that it uses its capacities carefully for exactly the user's needs, i.e., the services, that the user wants to use. Predicting the user's behavior given previous behavior is a machine learning task. For example, based on the learning of most often used file path components, a system may avoid unnecessary probing of files and could intelligently prefetch files. Prefetching those files, which soon will be read by the user, leads to decreased startup latencies for applications and, accordingly, conservation of energy.

The resource restrictions of mobile devices motivate the application of machine learning for predicting user behavior. At the same time, machine learning dissipates resources. There are three critical resource constraints:

- Data gathering: logging user actions uses processing capacity.
- Data storage: the training and test data as well as the learned model use memory.
- Communication: if training and testing is performed on a central server, sending data and the resulting model uses the communication network.
- Response time: the prediction of usage, i.e., the model application, has to happen in short real-time.

The dilemma of saving resources at the device through learning which, in turn, uses up resources, can be solved in several ways. Here, we set aside the problem of data gathering and its prerequisites on behalf of operation systems for embedded systems [Lohmann *et al.*, 2009] [Tartler *et al.*,] [Cantrill *et al.*, 2004].This is an important issue in its own right. Regarding the other restrictions, especially the restriction of memory, leads us to two alternatives.

Server-based learning: The learning of usage profiles from data is performed on a server and only the resulting model is communicated back to the device. Learning is less restricted in runtime and memory consumption. Just the learned model must obey the runtime and communication restrictions. Hence, a complex learning method is applicable. Figure 1 shows this alternative.

Device-based learning: The learning of usage profiles on the device is severely restricted in complexity. It does not need any communication but requires training data to be stored. Data streaming algorithms come into play in two alternative ways. First, descriptive algorithms incrementally build-up a compact way to store data. They do not classify or predict anything. Hence, in addition, simple methods are needed that learn from the aggregated compact data. Second, simple online algorithms predict usage behavior in realtime. The latter option might only be possible if specialized hardware is used, e.g., General Purpose GPUs. Figure 2 shows this alternative.

In this paper, we want to investigate the two alternatives using logged system calls. Server-based learning is exemplified by predicting file-access types in order to enhance prefetching. It is an open question whether structural models are demanded for the prediction of user behavior on the basis of system calls, or simpler models such as Naive Bayes suffice. Should the sequential nature of system calls be taken into account by the algorithm? Or is it sufficient to encode the sequences into the features? Or should features as well as algorithm be capable of explicitly addressing se-

[*] Artificial Intelligence Group
Baroper Strasse 301, Dortmund, Germany
{fricke,jungermann,morik,piatkowski,stolpe}@ls8.cs.tu-dortmund.de

[†] Embedded System Software Group
Otto-Hahn-Strasse 16, Dortmund, Germany
olaf.spinczyk@tu-dortmund.de

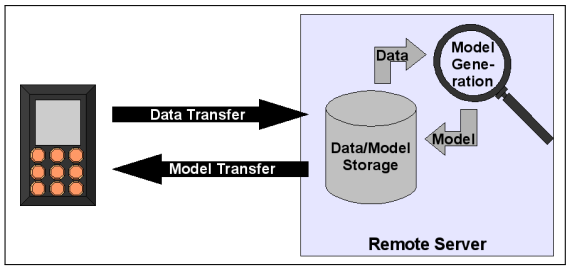

Figure 1: Server-based Architecture

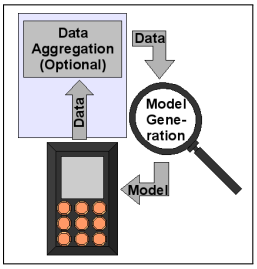

Figure 2: Device-based Architecture

quences? We investigate the use of two extremes, Conditional Random Fields (CRF) and Naive Bayes (NB). In particular, we inspect their memory consumption and runtime, both, for training and applying the learned function. Section 2 presents the study of server-based learning for ubiquitous devices. We derive the learning task from the need of enhancing prefetching strategies, describe the log data used, and present the learning results together with resource consumptions of NB and CRF.

Device-based learning is exemplified by recognizing applications from system calls in order to prevent fraud. We apply the data streaming algorithm Hierarchical Heavy Hitters (HHH) yielding a compact data structure for storage. Using these, the simple kNN method classifies systems calls. In particular, we investigate how much HHH compress data. Section 3 presents the study of device-based learning using a streaming algorithm for storing compact data. We conclude in Section 4 by indicating related and future work.

2 Server-based Learning

In this section we present the first case-study, where log data are stored and analyzed on a server (data are described in Section 2.2). Learning aims at predicting file access in order to prefetch files (see Section 2.1). The learning methods NB and CRF are introduced shortly in Section 2.3 and Section 2.4, respectively. The results are shown in Section 2.5.

2.1 File-access pattern prediction

A prediction of file-access patterns is of major importance for the performance and resource consumption of system software. For example, the Linux operating system uses a large "buffer cache" memory for disk blocks. If a requested disk block is already stored in the cache (*cache hit*), the operating system can deliver it to the application much faster and with less energy consumption than otherwise (*cache miss*). In order to manage the cache the operating system has to implement two strategies, block replacement and prefetching. The *block replacement* strategy is consulted upon a cache miss: a new block has to be inserted into the cache. If the cache is already full, the strategy has to decide which block has to be replaced. The most effective victim is the one with the longest forward distance, i.e. the block with the maximum difference between now and the time of the next access. This requires to know or guess the future sequence of cache access. The *prefetching* strategy proactively loads blocks from disk into the cache, even if they have not been requested by an application, yet. This often pays off, because reading a bigger amount of blocks at once is more efficient than multiple read operations. However, prefetching should only be performed if a block will be needed in the near future. For both strategies, block replacement and prefetching, a good prediction of future application behavior is crucial.

Linux and other operating systems still use simple heuristic implementations of the buffer cache management strategies. For instance, the prefetching code in Linux [Bovet and Cesati, 2005] continuously monitors read operations. As long as a file is accessed sequentially the read ahead is increased. Certain upper and lower bounds restrict the risk of mispredictions. This heuristics has two flaws:

- No prefetching is performed *before* the first read operation on a specific file, e.g., after "open", or even earlier.
- The strategy is based on assumptions on typical disk performance and buffer cache sizes, in general. However, these assumptions might turn out to be wrong in certain application areas or for certain users.

Prefetching based on machine learning avoids both problems. Prefetching can already be performed when a file is opened. It only depends on the prediction that the file will be read. The prediction is based on empirical data and not on mere assumptions. If the usage data change, the model changes, as well.

2.2 System Call Data for Access Prediction

We logged streams of system calls of type FILE, which consist of various typical sub-sequences, each starting with an `open`- and terminating with a `close`-call, like those shown in Figure 3.We collapsed such sub-sequences to one observation and assign the class label

- **full**, if the opened file was read from the first seek (if any) to the end,
- **read**, if the opened file was randomly accessed and
- **zero**, if the opened file was not read after all.

We propose the following generalization of obtained filenames. If a file is regular, we remove anything except the filename extension. Directory names are replaced by "DIR", except for paths starting with "/tmp" – those are replaced by "TEMP". Any other filenames are replaced by "OTHER". This generalization of filenames yields good results in our experiments. Volatile information like thread-id, process-id, parent-id and system-call parameters is dropped, and consecutive observations are compound to one sequence if they belong to the same process. The resulting dataset consists of 673887 observations in 80661 sequences, a snippet[1] is shown in Table 1.

We used two feature sets for the given task. The first encodes information about sequencing as features, resulting in 24 features, namely f_t, f_{t-1}, f_{t-2}, f_{t-2}/f_{t-1}, f_{t-1}/f_t,

[1]The final dataset is available at:
`http://www-ai.cs.tu-dortmund.de/PUBDOWNLOAD/MUSE2010`

```
1,open,1812,179,178,201,200,eclipse,/etc/hosts,524288,438,7 : 361, full
2,read,1812,179,178,201,200,eclipse,/etc/hosts,4096,361
3,read,1812,179,178,201,200,eclipse,/etc/hosts,4096,0
4,close,1812,179,178,201,200,eclipse,/etc/hosts
```

Figure 3: A sequence of system calls to *read* a file. The data layout is: timestamp, syscall, thread-id, process-id, parent, user, group, exec, file, parameters (optional) : read bytes, label (optional)

user	group	exec	file	label
20005	10000	firefox-bin	cookies.sqlite-journal	zero
20005	10000	firefox-bin	default	zero
20005	10000	firefox-bin	hosts	full
20005	10000	firefox-bin	hosts	full
20005	10000	multiload-apple	mtab	full
10028	10000	kmail	png	zero

Table 1: Snippet of the final dataset.

predicted\true	full	zero	read
full	0	2	1
zero	5	0	4
read	4	2	0

Table 2: Cost matrix

$f_{t-2}/f_{t-1}/f_t$, with $f \in \{user, group, exec, file\}$. The second feature set simply uses two features $exec_{t-1}/exec_t$ and $file_{t-2}/file_{t-1}/file_t$ as its only features.

Errors in predicting the types of access result in different degrees of failure. Predicting a partial caching of a file, if just the rights of a file have to be changed, is not as problematic as predicting a partial read if the file is to be read completely. Hence, we define a cost-matrix (see Table 2) for the evaluation of our approach.

2.3 Naive Bayes Classifier

The Naive Bayes classifier [Hastie *et al.*, 2003] assigns labels $y \in Y$ to examples $x \in X$. Each example is a vector of m attributes written here as x_i, where $i = 1...m$. The probability of a label given an example is according to the Bayes Theorem:

$$p(Y|x_1, x_2, ..., x_m) = \frac{p(Y)p\left(x_1, x_2, ..., x_m|Y\right)}{p\left(x_1, x_2, ..., x_m\right)} \quad (1)$$

Domingos and Pazzani [Domingos and Pazzani, 1996] rewrite eq. (1) and define the *Simple Bayes Classifier* (SBC):

$$p(Y|x_1, x_2, ..., x_m) = \frac{p(Y)}{p\left(x_1, x_2, ..., x_m\right)} \prod_{j=1}^{n} p\left(x_j|Y\right) \quad (2)$$

The classifier delivers the most probable class Y for a given example $x = x_1 \dots x_m$:

$$\arg\max_Y p(Y|x_1, x_2, ..., x_m) = \frac{p(Y)}{p\left(x_1, x_2, ..., x_m\right)} \prod_{j=1}^{m} p\left(x_j|Y\right) \quad (3)$$

The term $p\left(x_1, x_2, ..., x_m\right)$ can be neglected in eq. (3) because it is a constant for every class $y \in Y$. The decision for the most probable class y for a given example x just depends on $p(Y)$ and $p\left(x_i|Y\right)$ for $i = 1 \dots m$. These probabilities can be calculated after one run on the training data. So, the training runtime is $\mathcal{O}(n)$, where n is the number of examples in the training set. The number of probabilities to be stored during training are $|\mathcal{Y}| + (\sum_{i=1}^{m} |\mathcal{X}_j| * |\mathcal{Y}|)$, where $|\mathcal{Y}|$ is the number of classes and $|\mathcal{X}_i|$ is the number of different values of the ith attribute. The storage requirements for the trained model are $\mathcal{O}(mn)$.

It has often been shown that SBC or NBC perform quite well for many data mining tasks [Domingos and Pazzani, 1996; Huang *et al.*, 2003; Frank and Asuncion, 2010].

2.4 Linear-chain Conditional Random Fields

Linear-chain Conditional Random Fields, introduced by Lafferty et al. [Lafferty *et al.*, 2001], can be understood as discriminative, sequential version of Naive Bayes Classifiers. The conditional probability for an actual sequence of labels $\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_m}$, given a sequence of observations $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}$ is modeled as an exponential family. The underlying assumption is that a class label at the current timestep t just depends on the label of its direct ancestor, given the observation sequence. Dependency among the observations is not explicitly represented, which allows the use of rich, overlapping features. Equation 4 shows the model formulation of linear-chain CRF

$$p_\lambda\left(Y = \mathbf{y}|X = \mathbf{x}\right) = \frac{1}{Z\left(\mathbf{x}\right)} \prod_{t=1}^{T} \exp\left(\sum_k \lambda_k f_k\left(y_t, y_{t-1}, \mathbf{x}\right)\right) \quad (4)$$

with the observation-sequence dependent normalization factor

$$Z\left(\mathbf{x}\right) = \sum_{\mathbf{y}} \prod_{t=1}^{T} \exp\left(\sum_k \lambda_k f_k\left(y_t, y_{t-1}, \mathbf{x}\right)\right) \quad (5)$$

The sufficient statistics or *feature functions* f_k are most often binary indicator functions which evaluate to 1 only for a single combination of class label(s) and attribute value. The parameters λ_k can be regarded as weights or scores for this feature functions. In linear-chain CRF, each attribute value usually gets $|\mathcal{Y}| + |\mathcal{Y}|^2$ parameters, that is one score per state-attribute pair as well as one score for every transition-attribute triple, which results in a total of $\sum_{i=1}^{m} |\mathcal{X}_i| \left(|\mathcal{Y}| + |\mathcal{Y}|^2\right)$ model parameters, where $|\mathcal{Y}|$ is the number of classes, m is the number of attributes and $|\mathcal{X}_i|$ is the number of different values of the ith attribute. Notice that the feature functions explicitly depend on the whole observation-sequence rather than on the attributes at time t. Hence, it is possible and common to involve attributes of preceding as well as following observations from the current sequence into the computation of the total score

$\exp\left(\sum_k \lambda_k f_k\left(y_t, y_{t-1}, \mathbf{x}\right)\right)$ for the transition from y_{t-1} to y_t given $\mathbf{x}$.

The parameters are usually estimated by the maximum-likelihood method, i.e., maximizing the conditional likelihood (Eq. 6) by quasi-Newton [Malouf, 2002], [Sha and Pereira, 2003], [Nocedal, 1980] or stochastic gradient methods [Vishwanathan *et al.*, 2006], [Schraudolph and Graepel, 2002], [Schraudolph *et al.*, 2007].

$$\mathcal{L}\left(\lambda\right) = \prod_{i=1}^{N} p_\lambda(Y = \mathbf{y}^{(i)} | X = \mathbf{x}^{(i)}) \tag{6}$$

The actual class prediction for an unlabeled observation-sequence is done by the Viterbi algorithm known from Hidden Marcov Models [Sutton and McCallum, 2007], [Rabiner, 1989].

Although CRF in general allow to model arbitrary dependencies between the class labels, efficient exact inference can solely be done for linear-chain CRF. This is no problem here, because they match the sequential structure of our system-call data, presented in section 2.2.

2.5 Results of Server-based Prediction

Comparing the prediction quality of the simple NB models and the more complex CRF models, surprisingly, the CRF are only slightly better when using the two best features (see Tables 3 and 5). CRF outperforms NB when using all features (see Tables 4 and 6). These two findings indicate that the sequence information is not as important as we expected. Neither encoding the sequence into features nor applying an algorithm which is made for sequential information outperforms a simple model. The Tables show that precision, recall, accuracy, and misclassification cost are quite homogeneous for CRF, but vary for NB. In particular, the precision of predicting "read" and the recall of class "zero" differs from the numbers for the other classes, respectively. This makes CRF more reliable.

Inspecting resource consumption, we stored models of the two methods for both feature sets and for various numbers of examples to show the practical storage needs of the methods.Table 9 presents the model sizes of the naive Bayes classifier on both feature sets and for various example set sizes. We used the popular open source data mining tool *RapidMiner*[2] for these experiments. Table 9 also shows the model sizes of CRF on both feature sets and various example set sizes.

We used the open source CRF implementation *CRF++*[3] with L_2-regularization, $\sigma = 1$ and L-BFGS optimizer in all CRF experiments. Obviously, the storage needs for a model produced by a NB classifier are lower than those for a CRF model. This is the price to be paid for more reliable prediction quality. CRF don't scale-up well. Considering training time, the picture becomes worse. Table 10 shows the training time of linear-chain or HMM-like CRF consuming orders of magnitude more time than NB.

predicted\true	full	zero	read	prec.
full	1427467	19409	3427	98.43
zero	12541	2469821	40258	97.91
read	80872	217380	2467695	89.22
recall	93.86	91.25	98.26	

Table 3: Result of Naive Bayes Classifier on best two features, 10x10-fold cross-validated, accuracy: 94.45 $\pm$ 0.00, missclassification costs: 0.152 $\pm$ 0.001

full	zero	read	prec.
1426858	21562	22717	96.99
15392	2371009	97566	95.45
78630	314039	2391097	85.89
93.82	87.60	95.21	

Table 4: Result of Naive Bayes Classifier on all 24 features, 10x10-fold cross-validated, accuracy: 91.84 $\pm$ 0.00, missclassification costs: 0.218 $\pm$ 0.002

predicted\true	full	zero	read	prec.
full	1446242	7123	29051	97.56
zero	19452	2639097	133007	94.54
read	55186	60390	2349322	95.31
recall	95.09	97.51	93.55	

Table 5: Result of HMM-like CRF on the best two features, 10x10-fold cross-validated, accuracy: 95.49 $\pm$ 0.00, missclassification costs: 0.150 $\pm$ 0.000

full	zero	read	prec.
1450147	8335	25629	97.71
14563	2639724	126403	94.93
56170	58551	2359348	95.36
95.35	97.53	93.95	

Table 6: Result of HMM-like CRF on all 24 features, 10x10-fold cross-validated, accuracy: 95.70 $\pm$ 0.00, missclassification costs: 0.143 $\pm$ 0.000

predicted\true	full	zero	read	prec.
full	1467440	4733	7503	99.17
zero	10883	2659294	108340	95.71
read	42557	42583	2395537	96.57
recall	96.49	98.25	95.39	

Table 7: Result of linear-chain CRF on the best two features, 10x10-fold cross-validated, accuracy: 96.79 $\pm$ 0.00, missclassification costs: 0.112 $\pm$ 0.000

full	zero	read	prec.
1468095	4117	5022	99.38
10306	2662966	107859	95.75
42479	39527	2398499	96.69
96.53	98.39	95.51	

Table 8: Result of linear-chain CRF on all 24 features, 10x10-fold cross-validated, accuracy: 96.89 $\pm$ 0.00, missclassification costs: 0.110 $\pm$ 0.000

3 Device-based Learning

In this section, we present the second case-study, where streams of log data are processed in order to store patterns of system use. The goal is to aggregate the streaming system data. A simple learning method might then use the aggregated data. The method of Hierarchical Heavy Hitters (HHH) is defined in Section 3.1. The log data are shown in Section 3.2. For the comparison of different sets of HHH, we present a distance measure that allows for clustering or classifying sets of HHH. In addition to the quality of our HHH application, its resource consumption is presented in Section 3.3.

3.1 Hierarchical Heavy Hitters

The *heavy hitter problem* consists of finding all frequent elements and their frequency values in a data set. According

[2] RapidMiner is available at:
`http://www.rapidminer.com`

[3] CRF++ is available at:
`http://crfpp.sourceforge.net/`

#Att.\#Seq.	0	67k	135k	202k	270k	337k	404k	472k	539k	606k	674k
2 nB	2	78	100	118	132	143	154	161	169	176	181
24 nB	5	247	310	355	392	417	448	469	488	505	517
2 CRF++ (HMM)	5	247	366	458	490	512	569	592	614	634	649
24 CRF++ (HMM)	12	615	878	1102	1170	1216	1367	1420	1463	1521	1551
2 CRF++	6	523	776	978	1043	1089	1213	1260	1299	1345	1378
24 CRF++	19	1339	1914	2415	2559	2652	2988	3095	3184	3303	3365

Table 9: Storage needs (in kB) of the naive Bayes (nB), the HMM-like CRF (CRF++ (HMM)) and the linear-chain CRF (CRF++) classifier model on different numbers of sequences and attributes.

#Att.\#Seq.	0	67k	135k	202k	270k	337k	404k	472k	539k	606k	674k
2 nB	< 1	< 1	< 1	< 1	1	< 1	< 1	< 1	< 1	< 1	< 1
24 nB	< 1	< 1	< 1	1	< 1	1	1	1	1	2	1
2 CRF++ (HMM)	< 1	9.09	28.56	44.08	60.1	75.76	107.28	127.04	149.95	165.94	199.2
24 CRF++ (HMM)	< 1	27.92	55.9	103.24	153.53	160.33	230.7	273.29	232.84	309.19	317.62
2 CRF++	< 1	16.69	50.23	85.18	113.21	145.96	173.56	200.98	234.65	260.56	325.54
24 CRF++	< 1	41.06	105.29	156.67	296.31	300.83	343.28	433.03	440.88	463.84	632.96

Table 10: Training time (in seconds) of the naive Bayes (nB), the HMM-like CRF (CRF++ (HMM)) and the linear-chain CRF (CRF++) classifier model on different numbers of sequences and attributes.

to Cormode [Cormode *et al.*, 2003], given a (multi)set S of size N and a threshold $0 < \phi < 1$, an element e is a *heavy hitter* if its frequency $f(e)$ in S is not smaller than $\lfloor \phi N \rfloor$. The set of heavy hitters is then $HH = \{e | f(e) \geq \lfloor \phi N \rfloor\}$.

If the elements in S originate from a hierarchical domain D, one can state the following problem [Cormode *et al.*, 2003]:

Definition 1 (HHH Problem) *Given a (multi)set S of size N with elements e from a hierarchical domain D of height h, a threshold $\phi \in (0, 1)$ and an error parameter $\epsilon \in (0, \phi)$, the* Hierarchical Heavy Hitter Problem *is that of identifying prefixes $P \in D$, and estimates f_p of their associated frequencies, on the first N consecutive elements S_N of S to satisfy the following conditions:*

- *accuracy: $f_p^* - \varepsilon N \leq f_p \leq f_p^*$, where f_p^* is the true frequency of p in S_N.*
- *coverage: all prefixes $q \notin P$ satisfy $\phi N > \sum f(e) : (e \preceq q) \wedge (\nexists p \in P : e \preceq p)$.*

Here, $e \preceq p$ means that element e is *generalizable* to p (or $e = p$). For the extended multi-dimensional heavy hitter problem introduced in [Cormode *et al.*, 2004], elements can be multi-dimensional d-tuples of hierarchical values that originate from d different hierarchical domains with depth $h_i, i = 1, \ldots, d$. There exist two variants of algorithms for the calculation of multi-dimensional HHHs: Full Ancestry and Partial Ancestry, which we have both implemented. For a detailed description of these algorithms, see [Cormode *et al.*, 2008].

3.2 System Call Data for HHH

The kernel of current Linux operating systems offers about 320 different types of system calls to developers. Having gathered all system calls made by several applications, we observed that about 99% of all calls belonged to one of the 54 different call types shown in Table 11. The functional categorization of system calls into five groups is due to [Silberschatz *et al.*, 2010]. We focus on those calls only, since the remaining 266 call types are contained in only 1% of the data and therefore can't be frequent.

HHHs can handle values that have a hierarchical structure. We have utilized this expressive power by representing system calls as tuples of up to three hierarchical feature values. Each value originates from a taxonomy (*type*, *path* or *sequence*) that either can be derived dynamically from the data itself or has to be defined explicitly by the user. The groups introduced in Table 11 form the top level of the taxonomy for the hierarchical variable *type* (see Fig. 4). The `socket` call is a child of group COMM and FILE is the parent of calls like `open` and `fcntl64`. Subtypes of system calls can be defined by considering the possible values of their parameters. For example, the `fcntl64` call which operates on file descriptors has *fd*, *cmd* and *arg* as its parameters. We have divided the 16 different nominal values of the *cmd* parameter into seven groups — `notify`, `dflags`, `duplicate`, `sig`, `lock`, `fflags` and `lease` — that have become the children of the `fcntl64` system call in our taxonomy (see Fig. 4). One may further divide `fcntl64` calls of subtype `fflags` by the values `F_SETFL` and `F_GETFL` of the *arg* parameter. In the same way, we defined parents and children for each of the 54 call types and their parameters.

FILE	COMM	PROC	INFO	DEV
open	recvmsg	mmap2	access	ioctl
read	recv	munmap	getdents	
write	send	brk	getdents64	
lseek	sendmsg	clone	clock_gettime	
_llseek	sendfile	fork	gettimeofday	
writev	sendto	vfork	time	
fcntl	rt_sigaction	mprotect	uname	
fcntl64	pipe	unshare	poll	
dup	pipe2	execve	fstat	
dup2	socket	futex	fstat64	
dup3	accept	nanosleep	lstat	
close	accept4		lstat64	
			stat	
			stat64	
			inotify_init	
			inotify_init1	
			readlink	
			select	

Table 11: We focus on 54 system call types which are functionally categorized into five groups. FILE: file system operations, COMM: communication, PROC: process and memory management, INFO: informative calls, DEV: operations on devices.

The *path* variable is filled whenever a system call accesses a file system path. Its hierarchy comes naturally along with the given path hierarchy of the file system. The *sequence* variable expresses the temporal order of calls within a process. The directly preceding call is the highest, less recent calls are at deeper levels of the hierarchy.

We collected system call data from eleven applications

		Memory			Run-time			Similarity	
		Min	Max	Avg	Min	Max	Avg	Avg	Dev
FA	T	19	151	111	16	219	79	**0.997**	0.006
	TP	25	9,971	5,988	31	922	472	0.994	0.003
	TPS	736	**73,403**	48,820	78	14,422	6,569	0.987	0.008
PA	T	7	105	70	15	219	74	**0.985**	0.010
	TP	7	4,671	2,837	31	5,109	2,328	0.957	0.017
	TPS	141	18,058	10,547	78	**150,781**	74,342	0.921	0.026

Table 12: Memory consumption (number of stored tupels), run-time (milliseconds) and similarity to exact solution of the Full Ancestry (FA) and Partial Ancestry (PA) algorithms (ε = 0.0005, ϕ = 0.002). Minimum (Min), maximum (Max) and average (Avg) values were calculated over measurements for the first log file of all eleven applications with varying dimensionality of the element tupels (T = *type* hierarchy, P = *path* hierarchy, S = *sequence* hierarchy).

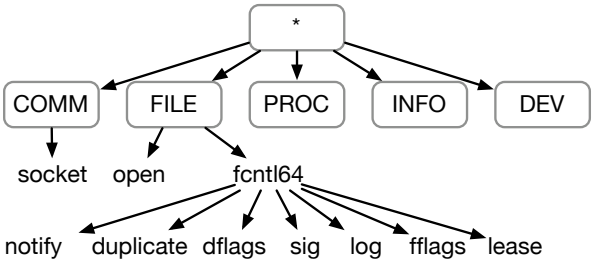

Figure 4: Parts of the taxonomy we defined for the hierarchical variable *type*.

(like Firefox, Epiphany, NEdit, XEmacs) with the *strace* tool (version 4.5.17) under Ubuntu Linux (kernel 2.6.26, 32 bit). All child processes were monitored by using option `-f` of *strace*. For each application, we logged five times five minutes and five times ten minutes of system calls if they belonged to one of the 54 types shown in Table 11, resulting in a whole of 110 log files comprising about 23 million of lines (1.8 GB).

3.3 Resulting Aggregation through Hierarchical Heavy Hitters

We have implemented the Full Ancestry and Partial Ancestry variants of the HHH algorithm mentioned in Section 3.1. The code was integrated into the RapidMiner data mining tool. Regarding run-time, all experiments were done on a machine with Intel Core 2 Duo E6300 processor with 2 GHz and 2 GB main memory.

Since we want to aggregate system call data on devices that are severely limited in processing power and available memory, measuring the resource usage of our algorithms was of paramount importance. Table 12 shows the run-time and memory consumption of the Full Ancestry and Partial Ancestry algorithms using only the *type* hierarchy, the *type* and *path* hierarchy, or the *type*, *path*, and *sequence* hierarchy. Minimum, maximum and averages were calculated over a sample of the ten gathered log files for each of the eleven application by taking only the first log file for each application into account.

Memory consumption and run-time increase with the dimensionality of the elements, while at the same time approximation quality decreases. Quality is measured as similarity to the exact solution. Full Ancestry has a higher approximation quality in general. The results correspond to observations made by Cormode and are probably due to the fact that Partial Ancestry outputs bigger HHH sets, which was the case in our experiments, too. Note that approximation quality can always be increased by changing parameter ε to a smaller value at the expense of a longer run-time.

Even for three-dimensional elements, memory consumption is quite low regarding the number of stored tuples. The largest number of tuples, 73,403, only equates to a few hundred kilobytes in main memory! The longest run-time of 150,781 ms for Partial Ancestry in three dimensions relates to the size of the biggest log file (application Rhythmbox).

Figure 5 shows the behaviour of our algorithms on the biggest log file (application Rhythmbox) for three dimensions with varying ε and constant ϕ. Memory consumption and quality decrease with increasing ε, while the run-time increases. So the most important trade-off involved here is weighting memory consumption against approximation quality — the run-time is only linearly affected by parameter ε. Again, Full Ancestry shows a better approximation quality in general.

Classification results

For the 110 log files of all applications, we determined the HHHs, resulting in sets of frequent tupels of hierarchical values. Interpreting each HHH set as an example of application behaviour, we wanted to answer the question if the profiles could be separated by a classifier. So we estimated the expected classification performance by a leave-one-out validation for kNN.

Therefore, we needed to define a distance measure for the profiles determined by HHH algorithms. The data structures of HHH algorithms contain a small subset of prefixes of stream elements. The estimated frequencies f_p are calculated from such data structure by the output method and compared to ϕ, thereby generating a HHH set. The similarity measure DSM operates not on the HHH sets, but directly on the internal data structures D_1, D_2 of two HHH algorithms:

$$\mathrm{sim}(D_1, D_2) = \frac{\sum_{p \in P_1 \cap P_2} \mathrm{contrib}_{\mathrm{DSM}}(p)}{|P_1 \cup P_2|} .$$

Be f_p^i the estimated frequency of prefix p for data structure D_i as normally calculated by the HHH output method. The contribution of individual prefixes to overall similarity can then be defined as

$$\mathrm{contrib}_{\mathrm{DSM}}(p) = \frac{2 \cdot \min(f_p^1, f_p^2)}{\min(f_p^1, f_p^2) + \max(f_p^1, f_p^2)} .$$

The so defined similarity measure is independent from the choice of ϕ, as no HHH sets need to be calculated.

The classification errors for different values of k, hierarchies and distance measures are shown in Table 13. The new DSM distance measure which is independent of parameter ϕ shows the lowest classification error in all validation experiments. As a baseline, we also determined the relative frequencies (TF, term frequencies) of call types per

	T		TS	
k	DSM	TF	DSM	TF
3	**10.3**	17.0	**7.7**	17.0
5	**12.7**	18.7	**8.7**	18.7
7	**14.0**	21.7	**8.7**	21.7
9	**14.0**	21.0	**9.0**	21.0

Table 13: Results for kNN (k = 3, 5, 7, 9), $\varepsilon = 0.0005$, $\phi = 0.002$ and distance measures DSM and TF, when only the *type* hierarchy or *type* and *sequence* hierarchy together are used.

log file and classified them using kNN (with Euclidean distance). The error for profiling by HHH sets is significantly lower than for the baseline.

4 Conclusion

Server-based and device-based learning has been investigated regarding resource constraints. memory consumption. Aggregation using HHH worked successfully for the classification of applications. Further work will exploit HHH aggregation for other learning tasks and inspect other data streaming algorithms. Concerning server-based learning, we may now answer the questions from the introduction, whether structural models are demanded for the prediction of user behavior on the basis of system calls, or simpler models such as Naive Bayes suffice. Should the sequential nature of system calls be taken into account by the algorithm? Or is it sufficient to encode the sequences into the features? Or should features as well as algorithm be capable of explicitly addressing sequences? We have compared CRF and NB with respect to their model quality, memory consumption, and runtime. Neither encoding the sequence into features nor applying an algorithm which is made for sequential information (i.e., CRF) outperforms a simple model (i.e., NB).

This is in contrast with studies on intrusion detection, where it was shown advantageous to take into account the structure of system calls, utilizing Conditional Random Fields (CRF) [Gupta *et al.*, 2007] and special kernel functions to measure the similarity of sequences [Tian *et al.*, 2007]. Structured models in terms of special tree kernel functions outperformed n-gram representations when detecting malicious SQL queries [Bockermann *et al.*, 2009]. Possibly, for prefetching strategies, the temporal order of system calls is not as important as we expected it to be. In the near future the resulting improvements in terms of cache hit rate and file operation latencies will be evaluated systematically based on a cache simulator and by modifying the Linux kernel.

Given regular processors, CRF are only applicable in server-based learning. Possibly, the integration of special processors into devices and a massively parallel training algorithm could speed up CRF for device-based learning. Further work will implement CRF on a GPGPU (general purpose graphic processing unit). GPGPUs will soon be used by mobile devices. It has been shown that their energy efficiency is advantagous [Timm *et al.*, 2010].

References

[Bockermann *et al.*, 2009] Christian Bockermann, Martin Apel, and Michael Meier. Learning sql for database intrusion detection using context-sensitive modelling. In *Proc. 6th Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 196 – 205. Springer, 2009.

[Bovet and Cesati, 2005] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[Cantrill *et al.*, 2004] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proc. of USENIX ATEC '04*, Berkeley, USA, 2004. USENIX.

[Cormode *et al.*, 2003] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 464–475. VLDB Endowment, 2003.

[Cormode *et al.*, 2004] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 155–166, New York, NY, USA, 2004. ACM.

[Cormode *et al.*, 2008] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in streaming data. *ACM Trans. Knowl. Discov. Data*, 1(4):1–48, 2008.

[Domingos and Pazzani, 1996] Pedro Domingos and Michael Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *Machine Learning*, pages 105–112. Morgan Kaufmann, 1996.

[Frank and Asuncion, 2010] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[Gupta *et al.*, 2007] K. Gupta, B. Nath, and K. Ramamohanarao. Conditional random fields for intrusion detection. In *21st Intl. Conf. on Adv. Information Netw. and Appl.*, pages 203–208, 2007.

[Hastie *et al.*, 2003] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, corrected edition, July 2003.

[Huang *et al.*, 2003] Jin Huang, Jingjing Lu, and Lambda Charles X. Ling. Comparing naive bayes, decision trees, and svm with auc and accuracy. In *in: Third IEEE International Conference on Data Mining, ICDM 2003*, pages 553–556. IEEE Computer Society, 2003.

[Lafferty *et al.*, 2001] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proc. 18th International Conf. on Machine Learning*, pages 282–289, 2001.

[Lohmann *et al.*, 2009] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proc. of USENIX ATEC*, Berkeley, USA, 2009. USENIX.

[Malouf, 2002] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In *COLING-02: proceedings of the 6th conference on Natural language learning*, pages 1–7, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

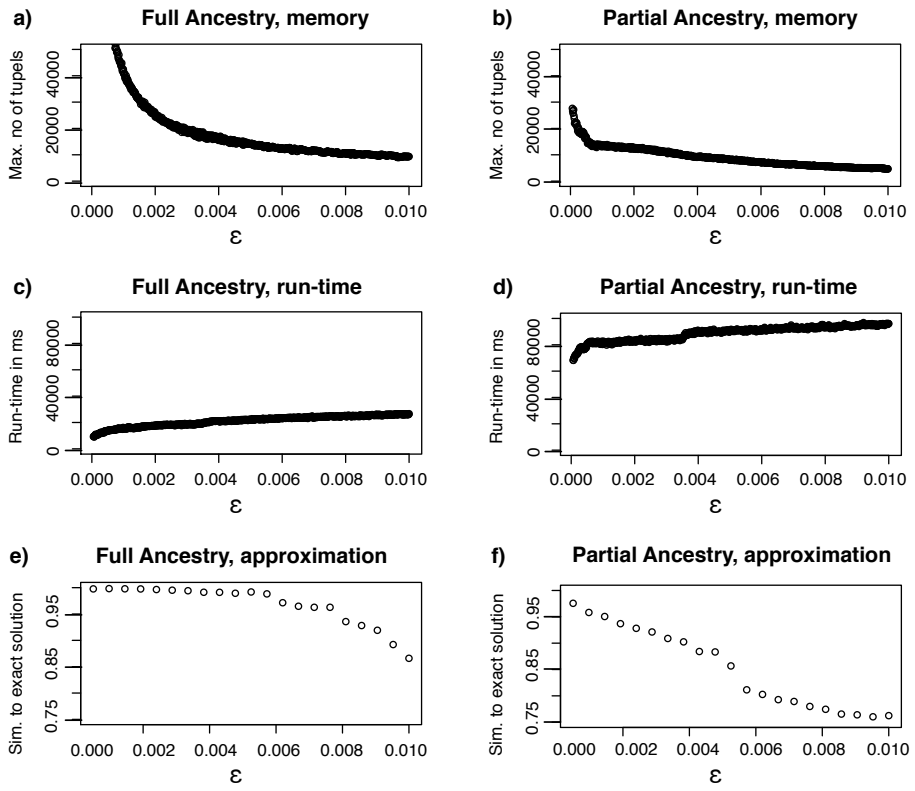

Figure 5: Memory consumption (a, b), run-time (c, d) and similarity to exact solution (e, f) of HHH algorithms (three-dimensional) with varying ε, $\phi = 0.001$ on biggest log file of application Rhythmbox.

[Nocedal, 1980] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[Rabiner, 1989] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[Schraudolph and Graepel, 2002] Nicol N. Schraudolph and Thore Graepel. Conjugate directions for stochastic gradient descent. In *ICANN '02: Proceedings of the International Conference on Artificial Neural Networks*, pages 1351–1358, London, UK, 2002. Springer-Verlag.

[Schraudolph *et al.*, 2007] Nicol N. Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-Newton method for online convex optimization. In Marina Meila and Xiaotong Shen, editors, *Proc. 11th Intl. Conf. Artificial Intelligence and Statistics (AIstats)*, volume 2 of *Workshop and Conference Proceedings*, pages 436–443, San Juan, Puerto Rico, 2007.

[Sha and Pereira, 2003] Fei Sha and Fernando Pereira. Shallow parsing with conditional random fields. In *NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 134–141, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[Silberschatz *et al.*, 2010] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 2010.

[Sutton and McCallum, 2007] C. Sutton and A. McCallum. An Introduction to Conditional Random Fields for Relational Learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[Tartler *et al.*,] Reinhard Tartler, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Dynamic AspectC++: Generic advice at any time. In *The 8th Int. Conf. on Software Methodologies, Tools and Techniques*, Prague. IOS Press. (to appear).

[Tian *et al.*, 2007] S. Tian, S. Mu, and C. Yin. Sequence-similarity kernels for SVMs to detect anomalies in system calls. *Neurocomput.*, 70(4–6):859–866, 2007.

[Timm *et al.*, 2010] C. Timm, A. Gelenberg, F. Weichert, and P. Marwedel. Reducing the Energy Consumption of Embedded Systems by Integrating General Purpose GPUs. Technical Report 829, Technische Universität Dortmund, Fakultät für Informatik, 2010.

[Vishwanathan *et al.*, 2006] S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 969–976, New York, NY, USA, 2006. ACM.