ACE: Exploiting Correlation for Energy-Efficient and Continuous Context Sensing

Suman Nath, *Member, IEEE*
Microsoft Research, Redmond, USA

Abstract—We propose ACE (Acquisitional Context Engine), a middleware that supports continuous context-aware applications while mitigating sensing costs for inferring contexts. ACE provides user's current context to applications running on it. In addition, it dynamically learns relationships among various context attributes (e.g., whenever the user is `Driving`, he is not `AtHome`). ACE exploits these automatically learned relationships for two powerful optimizations. The first is *inference caching* that allows ACE to opportunistically infer one context attribute (`AtHome`) from another already-known attribute (`Driving`), without acquiring any sensor data. The second optimization is *speculative sensing* that enables ACE to occasionally infer the value of an expensive attribute (e.g., `AtHome`) by sensing cheaper attributes (e.g., `Driving`). Our experiments with two real context traces of 105 people and a Windows Phone prototype show that ACE can reduce sensing costs of three context-aware applications by about $4.2\times$, compared to a raw sensor data cache shared across applications, with a very small memory and processing overhead.

Index Terms—Continuous context-aware applications, Sensor-rich mobile environment, Rule-based optimizations

1 Introduction

The increasing availability of sensors integrated in smartphones provides new opportunities for continuous context-aware applications that react based on the operating conditions of users and the surrounding environment [12], [14], [17], [18], [19], [22]. Examples of such applications include a *location-based reminder* that reminds a user of something when he is at a predefined location (e.g., GeoNote for Android phones), a *jogging tracker* that keeps track of how much exercise the user does in a day (e.g., JogALot for iPhones), and a *phone buddy* that mutes the phone when the user is in a meeting (e.g., RingWise for Android phones) or plays a customized greeting message when he is driving. These applications use context information inferred from various sensors such as GPS, accelerometer, and microphone of the mobile phone.

A big overhead of running context-aware applications is their high energy consumption [15], [17]. Many of these applications require monitoring user's context *continuously*. Even some applications that require context information occasionally may need continuous or frequent context monitoring due to relatively high context detection latency. For example, the phone buddy application may need to know if a user is in a meeting to mute an incoming call. However, without continuously (or frequently) monitoring user's context, it may not be able to quickly infer whether the user is in a meeting when a call arrives. If not designed carefully, the cost of continuous context sensing can be prohibitively high, especially when applications require expensive attributes such as location and multiple such applications run concurrently.

Prior work has proposed various optimizations for energy-efficient continuous context sensing [12], [13], [15], [17], [18]. These are orthogonal to, and can be used with, our work. Prior work also proposed exploiting redundancy *across* applications—by sharing sensor data and inferred context attributes among applications through a shared cache [10]. However, as we will show, this alone can still be costly if applications involve expensive sensors such as GPS and microphone.

Our Contributions. We propose ACE (Acquisitional Context Engine), a middleware that supports continuous context-aware applications while mitigating sensing cost for acquisition of necessary context attributes. ACE exposes an extensible collection of context attributes such as `AtHome` and `IsDriving` (more examples later). Applications interact with ACE by requesting the current values of these attributes. In providing context information to applications, ACE can achieve significant energy savings—in our experiments with three applications and 105 real users' context traces, ACE reduces sensing costs by about $4.2\times$ compared to a shared cache.

How does ACE get such a high energy savings? The key observation we exploit is that *human contexts are limited by physical constraints* and hence values of various context attributes can be highly correlated. Thus, each individual user follows various *behavior invariants* that can be discovered by finding relationships among various context attributes (e.g., one-to-one mappings from one set of attribute values to another). We can then use these invariants to infer an unknown context attribute from a known context attribute or from an unknown but cheaper context attribute, reducing the total energy

consumption of context acquisition. We illustrate this with an example.

Suppose Bob is running multiple applications on ACE including the following two: *App1* that uses accelerometer to monitor Bob's transportation mode (`Walking` vs. `Running` vs. `Driving`) and *App2* that uses location sensors (GPS, WiFi, and Bluetooth) to monitor whether he is `AtHome` or `InOffice`. Suppose ACE, by analyzing Bob's context history, discovers various *context invariant rules* such as "When `Driving` is `True`, `AtHome` is `False`."[1] We will give more examples of context rules later; for now we assume rules that naturally make sense. ACE exploits these context rules in two ways.

Inference-based Intelligent Context Caching. Suppose at 10am, App1 asks ACE for the value of `Driving`. ACE determines the value of the attribute and puts it in cache. If, within a short time window, another application asks for the value of `Driving`, ACE can return the value like a traditional cache. In addition, if App2 asks for the value of `AtHome` within a short time window, and if the cached value of `Driving` is `True`, ACE can also return the value `AtHome` = `False`. ACE can do this by exploiting the negative correlation between the attributes `Driving` and `AtHome` and *without acquiring any sensor data*. Thus, ACE can reuse the cached value of `Driving` not just for an exact lookup, but also as a *proxy* for a *semantically related* lookup.

Speculation-based Cheaper Sensing Plan. Context rules allow finding the value of an attribute in many different ways. For example, the value of `InOffice` can be determined by acquiring necessary sensor data (e.g., GPS, WiFi signatures). Moreover, by exploiting context rules, it can also be answered `False` if `Driving` $=$ `True` or `Running` $=$ `True` or `AtHome` $=$ `True`; and can be answered `True` if `InMeeting` = `True`. Each of these proxy attributes can further be determined through additional proxy attributes. This gives ACE the flexibility to choose, and speculatively sense values of, proxies that yield the value of the target attribute with the lowest cost. For example, if `Running` is the cheapest attribute and if there is a high probability that the user is jogging now, ACE will use `Running` as a proxy, and speculatively determine its value, to determine the value of `InOffice` (by using the rule that whenever `Running` is `True`, `InOffice` is `False`). If the value of `Running` is indeed `True`, ACE can infer the value of `InOffice` to be `False`, without using expensive location sensors. If `Running` is found to be `False`, however, ACE cannot make such a conclusion and needs to proceed with the next best proxy attribute.

The benefit of the above *speculative sensing* idea may seem counter-intuitive— evaluating an attribute can become cheaper by evaluating additional attributes. However, as we will show later, if such additional attributes are low-cost and are most likely to uniquely determine the value of the target attribute, such additional attribute can reduce the total *expected* cost of the system.

The above ideas are powerful in practice and can be generally applicable to a broader class of applications. However, realizing them requires addressing several challenges. First, how can the system automatically and efficiently learn context rules that can help finding cheap proxy attributes? Second, how can the system find suitable proxy attributes in the cache that can determine a value of the target attribute? Third, how can it determine the best sequence of proxy attributes to speculatively sense in order to determine the value of a target attribute? All these challenges need to be addressed without much overhead on the phone and with the goal of reducing the overall energy consumption. In the rest of the paper, we describe how ACE achieves this goal.

Intuitively, the energy savings in ACE come from sensing redundancy across applications and context attributes, well-defined relationship among attributes, and the possibility of inferring expensive attributes through cheaper attributes. The savings depend on application mix and user habits. *How common is such redundancy in practice?* To understand this, we have evaluated ACE with three applications and two real datasets containing context traces of total 105 users. Our results show that ACE consumes around $4.2\times$ less energy than consumed by a raw sensor data cache shared across applications. Our ACE prototype on a Samsung Focus phone running Windows Phone 7.5 OS shows that ACE imposes little overhead (< 0.1 ms latency and < 15 MB RAM footprint).

Note that inference caching and speculative sensing based on rules can sometimes produce results that are (a) incorrect, if user's behavior deviates from context rules, or (b) stale, if results are inferred from stale values in cache, and hence incorrect if cached values no longer hold. ACE employs several mechanisms to reduce such inaccuracy. Experiments show that such inaccuracy is $< 4\%$ for our datasets and is small compared to classification errors in typical context inference algorithms [13], [15], [16]. If an application cannot tolerate such small additional inaccuracy, it can request ACE to get true context values by collecting sensor data (and paying high energy cost).

In summary, we make the following contributions.

- We describe design and implementation of ACE, a middleware for efficient and continuous sensing of user's context in a mobile phone (§ 2).
- We describe how ACE efficiently learns rules from user's context history (§ 3), and use the rules for intelligent context caching (§ 4) and speculative sensing (§ 5).
- By using a prototype on a real phone and two datasets containing continuous context traces of 105 users, we show that ACE can significantly reduce sensing cost with a minimal runtime overhead (§ 7).

1. Currently ACE uses simple association rules. However, the core ideas of ACE can be used with more general rules such as temporal or probabilistic rules.

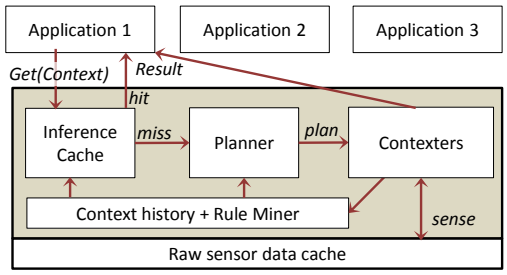


Fig. 1. ACE Architecture

TABLE 1
Context attributes implemented in ACE

Attribute	Short	Sensors used (sample length)	Energy (mJ)
IsWalking	W	Accelerometer (10 sec)	259
IsDriving	D	Accelerometer (10 sec)	259
IsJogging	J	Accelerometer (10 sec)	259
IsSitting	S	Accelerometer (10 sec)	259
AtHome	H	WiFi	605
InOffice	O	WiFi	605
IsIndoor	I	GPS + WiFi	1985
IsAlone	A	Microphone (10 sec)	2895
InMeeting	M	WiFi + Microphone (10 sec)	3505
IsWorking	R	WiFi + Microphone (10 sec)	3505

2 ACE Overview

2.1 ACE Architecture

Figure 1 shows a high level architecture of ACE. It works as a middleware library between context-aware applications and sensors. An application interacts with ACE with a single API $Get(X)$, asking for the current value of a context attribute X. ACE consists of the following components.

(1) Contexters. This is a collection of modules, each of which determines the current value of a context attribute (e.g., `IsWalking`) by acquiring data from necessary sensors and by using necessary inference algorithms. An application can extend it by implementing new contexters for additional attributes. The names of attributes sensed by various contexters are exposed to applications, to be used with the $Get()$ call.[2]

(2) Raw sensor data cache. This is a standard cache.

(3) Rule Miner. This module maintains user's context history and automatically learns *context rules*—relationships among various context attributes—from the history.

(4) Inference Cache. This implements the intelligent caching behavior mentioned before. It provides the same Get/Put interface as a traditional cache. However, on $Get(X)$, it returns the value of X not only if the value of X is in the cache, but also if it can be inferred by using context rules and cached values of other attributes.

(5) Sensing Planner. On a cache miss, this finds the cheapest sequence of proxy attributes to speculatively sense in order to determine the value of the target attribute.

The last three components form the core of ACE. We will describe them in more detail later in the paper.

Importance of Rule Miner. One might argue that context invariants or relationships among various context attributes can be hard-coded within applications. Rule Miner automates this process. Moreover, it can learn additional rules that are not-so-obvious to developers, can change over time, are specific to an individual user, and involve attributes from different applications (and hence developer of one application may not find them). Our experiments show that the energy savings of ACE would have reduced by more than half had it used a set of static rules for all users, instead of dynamically learning personalized rules for each user.

For simplicity, we assume Boolean context attributes. Even though our techniques can be generalized to real-valued attributes, we believe that Boolean attributes are sufficient for a large collection of context-aware applications as they react based on Boolean states of arbitrarily complex context attributes (e.g., mute the phone for `True` values of `InMeeting` or `InMovieTheater`).

2.2 Contexters

We have implemented contexters for the attributes shown in Table 1. The table shows attribute names and the shorthands we use for them for brevity. We denote an attribute and its value such as `IsDriving` = `True` as a *tuple*, and often write it in shorthand as `D` = `T`.

Table 1 also shows the sensors various contexters use. The contexters for attributes `W`, `D`, `J`, and `S` collect accelerometer data over a 10-second window and derive values of the attributes based on a decision tree learned from training data [13]. The values of `AtHome` and `InOffice` are determined by matching the current WiFi signature with a pre-learned dictionary. The contexter for `IsIndoor` uses GPS location and WiFi signature; i.e., the user is assumed to be indoor if GPS signal is not available or current WiFi access points are in a pre-learned dictionary. The value of `IsAlone` is determined by collecting an audio sample of 10 seconds and comparing the average signal strength with a pre-learned threshold [16]. Finally, the values of `InMeeting` and `IsWorking` are determined by using signal strength of surrounding sound and fine-grained location based on WiFi signature and acoustic background signature [25].

We have measured energy consumed by various contexters on an Android HTC desire phone (shown in Table 1). The key observation, which we expect to hold on other platforms and contexter implementations as well, is that energy consumed by various contexters vary by an order of magnitude. This alludes to the potential savings in ACE when an expensive attribute can be inferred from a cheap attribute.

Extensibility. Note that implementation of various contexters is not at the core of this work, and one can

2. ACE also exposes raw sensor data for applications that need additional attributes and more accuracy than is provided by ACE.

think of better implementations. ACE treats contexters as black boxes; the only two pieces of information a contexter needs to expose to ACE are (1) the name of the attribute that it determines, and (2) its energy cost. Therefore, implementation details of contexters do not affect the rest of ACE. One can implement a new contexter or replace an existing contexter with a more accurate and energy-efficient one, and can expose the attribute name and energy cost to ACE through a configuration file; ACE will seamlessly incorporate the new contexter.

This highlights another useful feature of ACE: it does not require applications to agree on semantic meaning of context labels. Applications can simply register new contexters that other applications are unaware of, but they still benefit indirectly through correlations that ACE might detect and exploit.

2.3 Work flow of ACE

Figure 2 shows the next level of details of ACE's architecture to show how ACE works end-to-end. Applications interact with ACE by requesting the current value of a context attribute X, by calling the API $Get(X)$. If an unexpired value of X (computed previously for the same or a different application) is found in Inference Cache, ACE returns the value. Otherwise, ACE uses the current set of context rules (learned by Rule Miner) and currently unexpired tuples in the cache to see if a value of X can be inferred. If so, ACE returns the inferred value of X. Otherwise, ACE needs to invoke necessary contexters to determine the value of X. To explore the possibility of determining the value of X through cheaper proxy attributes, ACE invokes the Sensing Planner module. It repeatedly chooses the next best proxy attribute and speculatively runs its contexter until the value of X is determined. The value of X is finally put in cache (with an expiration time), added to context history (to be analyzed by Rule Miner), and returned to application.

Reducing Inaccuracies. As mentioned in Section 1, ACE uses several mechanisms to reduce possible inaccuracies in results. First, ACE conservatively uses rules that almost always hold for a user. Second, the rules are dynamically updated, to capture any change in user's habit. Third, it uses a small (5 minutes) expiration time of tuples in the Inference Cache so that inference is done only based on fresh tuples. Fourth, it occasionally cross-validates its results with ground truths determined by explicitly running contexters. The default cross-validation probability is 0.05—our experiments show that this gives a good balance between energy and accuracy. Finally, an application can provide feedback (e.g., got from the user) to ACE about inaccuracies in ACE results. ACE conservatively removes any rule inconstant with ground truths.

2.4 Applications

We target context-aware applications that need to monitor user's context continuously or frequently. Different

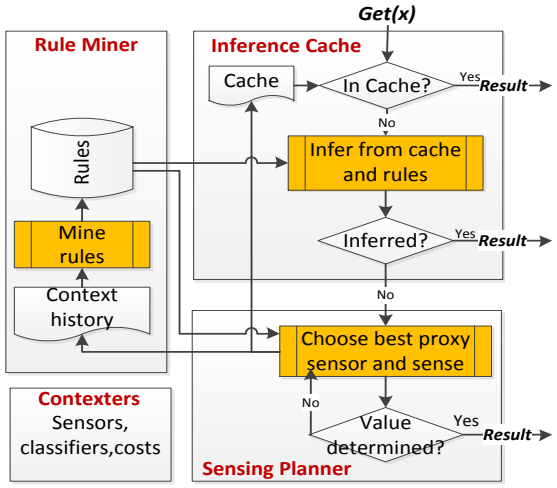


Fig. 2. Workflow in ACE

applications may require different context attributes exposed by ACE such as user's location (home/office), transportation mode (walking/driving), work state (working/meeting), and group state (alone/in group). For concreteness, we consider the following three applications we have built on top of ACE.

GeoReminder. Like GeoNote for Android, it continuously monitors a user's current location and displays custom messages when he is at a pre-defined location. The location is computed based on GPS data and WiFi access point signature (using similar techniques as [15]).

JogTracker. Like JogALot for iPhone, it monitors a user's transportation mode (e.g., `IsWalking`, `IsRunning`, `IsDriving`) and keeps track of how much calories he burns in a day by walking and running.

PhoneBuddy. Like RingWise on Android, it monitors whether a user is in a meeting (`InMeeting`) or is driving (`IsDriving`); it mutes the phone if he is in a meeting and plays a custom greeting message if he is driving.

2.5 Datasets

Apart from a real prototype, we use two real datasets to evaluate various techniques in ACE. Using real traces is important because effectiveness of ACE depends on its ability to infer context rules from a user's context history. Each dataset contains continuous context traces of a number of users, where each trace contains a sequence of lines in the following format: `timestamp`, $\texttt{attrib}_1 = \texttt{value}_1, \texttt{attrib}_2 = \texttt{value}_2, \cdots$. There is a line in the trace every time any attribute changes its value.

►**Reality Mining Dataset.** This dataset,[3] collected part of the Reality Mining project at MIT [7], contains continuous data on daily activities of 100 students and staff at MIT, recorded by Nokia 6600 smartphones over the 2004-2005 academic year. The trace contains various continuous information such as a user's location (at a

3. http://reality.media.mit.edu/download.php

granularity of cell towers), proximity to others (through Bluetooth), activities (e.g., making calls, using phone apps), transportation mode (driving, walking, stationary),[4] etc. over different times of the day. The cell-tower id is mapped back to latitude-longitude, based on the dataset in [8]. We consider 95 users who have at least 2 weeks of data. The total length of all users' traces combined is 266,200 hours. The average, minimum, and maximum trace length of a user is 122 days, 14 days, and 269 days, respectively.

The dataset allows us to infer the following context attributes of a user at any given time: `IsDriving`[5] (`D` in short), `IsBiking` (`B`), `IsWalking` (`W`), `IsAlone` (`A`), `AtHome` (`H`), `InOffice` (`O`), `IsUsingApp` (`P`), and `IsCalling` (`C`).

►ACE **Dataset.** This dataset, similar to the Reality Mining dataset, is collected with our continuous data collection software running on Android phones. The subjects in the dataset, 9 male and 1 female, worked at Microsoft Research Redmond. They all drove to work and had usual 9am-5pm work schedule. The dataset has all the context attributes in Table 1. The maximum, minimum, and average trace length of a user is 30 days, 5 days, and 14 days respectively.

Even though the dataset is smaller than the Reality Mining dataset, its ground truths for context attributes are more accurate. They came from a combination of three sources: (1) labels manually entered by a user during data collection time (e.g., when a user started driving, he manually entered $\texttt{Driving} = \texttt{True}$ through a GUI), (2) labels manually provided by the user during post processing, and (3) outputs of various contexters.

3 Rule Miner

The Rule Miner in ACE maintains a history of timestamped tuples (e.g., $\texttt{7am} : \texttt{AtHome} = \texttt{True}$), sensed for various applications. It then incrementally derives rules regarding relationships among various attributes.

Discovering interesting relationships among variables in large databases has been studied extensively in the data mining research, with many efficient algorithms developed (see [26] for a survey). In ACE, we would like to generate rules that are easy to compute and use, and are compact to represent. We choose the well-known Apriori algorithm for association rule mining [1]. The algorithm can efficiently discover association rules from databases containing transactions (for example, collections of items bought together by customers). In ACE, we are interested in association rules that have the following general form:$\{l_1, l_2, \cdots, l_n\} \Rightarrow r$, which implies that whenever all the tuples $l_1, \cdots, l_n$ hold, r holds as well. Thus, the left side of a rule is basically a conjunction of tuples $l_1, l_2, \cdots, l_n$. For example, the rule $\{\texttt{O} = \texttt{T}, \texttt{W} = \texttt{F}, \texttt{A} = \texttt{F}\} \Rightarrow \texttt{M} = \texttt{T}$ implies that if a user is in the office ($\texttt{O} = \texttt{T}$) and not walking ($\texttt{W} = \texttt{F}$) and is not alone ($\texttt{A} = \texttt{F}$), then he is in a meeting ($\texttt{M} = \texttt{T}$).

4. This is inferred from a user's speed and survey data.
5. Denotes if the user is driving is own car or is taking a bus.

TABLE 2
A few example rules learned for one user

$\{\texttt{IsDriving} = \texttt{True}\} \Rightarrow \{\texttt{Indoor} = \texttt{False}\}$
$\{\texttt{Indoor} = \texttt{T}, \texttt{AtHome} = \texttt{F}, \texttt{IsAlone} = \texttt{T}\} \Rightarrow \{\texttt{InOffice} = \texttt{T}\}$
$\{\texttt{IsWalking} = \texttt{T}\} \Rightarrow \{\texttt{InMeeting} = \texttt{F}\}$
$\{\texttt{IsDriving} = \texttt{F}, \texttt{IsWalking} = \texttt{F}\} \Rightarrow \{\texttt{Indoor} = \texttt{T}\}$
$\{\texttt{AtHome} = \texttt{F}, \texttt{IsDriving} = \texttt{F}, \texttt{IsUsingApp} = \texttt{T}\} \Rightarrow \{\texttt{InOffice} = \texttt{T}\}$
$\{\texttt{IsJogging} = \texttt{T}\} \Rightarrow \{\texttt{AtHome} = \texttt{T}\}$

Apart from simplicity, association rules naturally support ACE's desired inference process: if for any rule $A \Rightarrow b$, *all* the tuples in A exist (and not expired) in ACE's cache, then we can immediately infer the tuple b, without additional sensor data acquisition.

The Apriori algorithm takes as input a collection of *transactions*, where each transaction is a collection of co-occurring tuples. Each association rule has a *support* and a *confidence*. For example, if a context history has 1000 transactions, out of which 200 include both items A and B and 80 of these include item C, the association rule $\{A, B\} \Rightarrow C$ (read as "If A and B are true then C is true") has a support of $8\%(= 80/1000)$ and a confidence of $40\%(= 80/200)$. The algorithm takes two input parameters: $minSup$ and $minConf$. It then produces all the rules with support $\geq minSup$ and confidence $\geq minConf$.

The original Apriori algorithm works in two steps. The first step is to discover frequent itemsets. In this step, the algorithm counts the number of occurrences (called *support*) of each distinct tuple (e.g., $\texttt{W} = \texttt{T}$) in the dataset and discards infrequent tuples with support $< minSup$. The remaining tuples are frequent patterns of length 1, called *frequent 1-itemset* S_1. The algorithm then iteratively takes the combinations of S_{k-1} to generate frequent k-itemset candidates. This is efficiently done by exploiting the anti-monotonicity property: any subset of a frequent k-itemset must be frequent, which can be used to effectively prune a candidate k-itemset if any of its $(k-1)$-itemset is infrequent [1].

The second step of the Apriori algorithm is to derive association rules. In this step, based on the frequent itemsets discovered in the first step, the association rules with confidence $\geq minConf$ are derived.

Parameters. The Apriori algorithm takes two parameters: $minSup$ and $minConf$. Many real-world applications care only about frequent rules, and hence use a large $minSup$ (e.g., $> 75\%$). We, however, are also interested in rules that hold relatively infrequently. For example, if a user drives < 1 hour in a day, Apriori will require a very small support of $< 4\%$ to find rules involving the context attribute `IsDriving`. We are interested in such rare rules because they can sometimes result in big energy savings (e.g., when `InMeeting` is inferred from `IsDriving`). Setting $minSup$ close to zero, however, can produce many infrequent and useless rules. We found that the value $minSup = 4\%$ works well in practice.

Ideally, we would like to use $minConf = 100\%$, so that ACE uses rules that always hold and hence ACE does not introduce any inference errors. However, real contex-

ters use imperfect classification algorithms on noisy data, and hence they can occasionally produce inconsistent context attributes. E.g., two contexters can mistakenly conclude that the user is `atHome` and `isDriving` at the same time. Our experiments show that a $minSup$ value of 99% strikes a good balance between tolerating such occasional inconsistencies and capturing good rules.

Example Rules. Table 2 shows a few example rules automatically learned by ACE for a single user in the ACE Dataset. Some rules are not so obvious; e.g., consider the fifth rule in Table 2: if the user is not at home and not riding a bus but is using phone apps, he is in the office. Apparently, this particular user uses apps only at home, in bus, and in office. Some rules can be specific to a single user and may not generally apply to all users. For example, the last rule in Table 2 applies to a user who jogs in a treadmill at home, and it may not apply to someone who runs outside.

3.1 Challenges

We need to address several challenges to use existing association rule mining algorithms in ACE.

3.1.1 Defining Transactions

Association rule mining algorithms take *transactions* of tuples: each transaction defines which tuples occur together. For a rule $A \Rightarrow B$ to be derived, both A and B need to appear together in the same transaction. In ACE, various contexters may be invoked for different applications asynchronously. Thus, co-occurring context attributes may not be inferred exactly at the same time, rather within a short window of time. Yet, for mining useful rules, we need to batch them as a single transaction of co-occurring tuples.

One straightforward way to address this is to use a time window: all attributes sensed within a window is considered to be co-occurring and hence are batched together in a single transaction. Unfortunately, deciding a good window size is nontrivial. A small window will generate small (or even singleton) transactions that will miss many rules involving multiple tuples. On the other hand, a large window may put conflicting tuples (e.g., $\texttt{AtHome} = \texttt{True}$ and $\texttt{AtHome} = \texttt{False}$) in the same transaction when the user changes his context in the middle of the window. Then, a rule mining algorithm can produce conflicting itemsets such as $\{\texttt{AtHome} = \texttt{True}, \texttt{AtHome} = \texttt{False}\}$ and meaningless rules such as $\{\texttt{AtHome} = \texttt{True}\} \Rightarrow \{\texttt{AtHome} = \texttt{False}\}$.

To understand the impact of window size, we run Rule Miner on ACE Dataset as follows. We assume that a hypothetical application wakes up once every 30 seconds, and determines the value of a randomly chosen attributes in Table 1. (This simulates the combined behavior of a random collection of continuous context-aware applications.) Thus, we get a sequence of tuples, with two consecutive tuples 30 seconds apart from each other. Then, for a given window size W, we batch all tuples within W as a single transaction. Note that a batch may contain the same tuple multiple times, in which case we keep only one. Moreover, a batch may also contain conflicting tuples, in which case we drop all tuples involving the corresponding attribute.

Figure 3 shows the effect of various window sizes. The y-axis shows the fraction of rules learned, normalized to the maximum number of manually-verified "good" rules we could learn for various parameters and averaged over all users in ACE Dataset. As shown, a 5-minute window gives the maximum number of good rules.

Dynamic Window Size. Even with a carefully chosen window size, we do not discover all the good rules. This is because due to removal of conflicting tuples within the same window, many transactions contain a small number of tuples. Since a user can change his context any time within a window, such conflicting tuples occur very often. To avoid this, we use *dynamic windowing*, where we use a default window size (say 5 minutes), but *trim* a window into a shorter one whenever the value of any context attribute changes (e.g., when the user's `AtHome` context is changed from `True` to `False`). After such a trimmed window, we restore to the default window size. Thus, a window never contains conflicting tuples. We also ignore windows containing only a single tuple. Figure 3 shows that dynamic windowing with 5-minute default window size produces 18% more "good" rules than static windowing with 5-minute window.

Increasing default window size can reduce the number of rules produced, as shown in Figure 3. This is because increasing window size can reduce relative frequencies of popular tuples since multiple occurrences of a popular tuple are treated as a single occurrence within a transaction. If the relative frequency of an itemset containing a popular item goes below the $minSup$ threshold, APriori algorithm will ignore the itemset and no rules containing the itemset will be produced.

Based on the discussion above, ACE uses dynamic window with a default length of 5 minutes.

3.1.2 Dealing with Low Support

As mentioned before, ACE uses a small $minSup$ value (e.g., 4%). With such a small support threshold, Apriori needs to enumerate a very large number of itemsets and becomes extremely expensive. This is because with a small support, many itemsets will be considered frequent and each of the frequent itemsets will contain a combinatorial number of shorter sub-itemsets, each of which will be frequent as well.

Figure 4 shows the time required by ACE's rule mining algorithm run with the Reality Mining dataset on a Samsung Focus phone with 1 GHz Scorpion CPU and 512MB RAM. The $minSup$ and $minConf$ are set to be 4% and 99% respectively. Depending on the length of the history, the process can take more than a few hours. Such a high overhead is not acceptable in practice.

Offloading. ACE addresses this problem by offloading

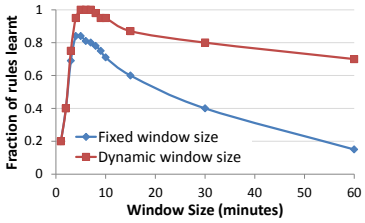


Fig. 3. Effect of window size in batching

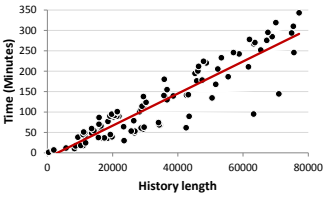


Fig. 4. Time to learn rules on a Samsung Focus phone

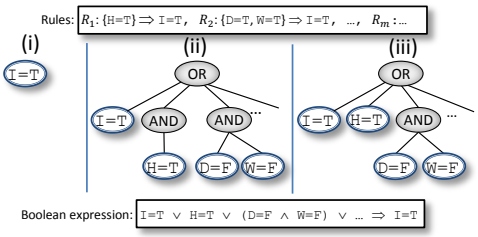


Fig. 5. Expanding a node

the task of rule mining to a remote, powerful server (similar to many systems such as Maui [5]). ACE locally logs all context tuples, periodically uploads them to a server when it has connectivity and time, and gets the rules back. By default, this is done once a day. Computing rules in a server takes a few minutes. With a 2.7GHz dual-core server, the maximum time required for offloading context history, computing rules, and downloading them to the phone is around 15 minutes for a 269-day trace in the Reality Mining Dataset. For traces smaller than 90 days, the time is around 3 minutes.

Incremental Update. Between periodic downloads of rules from the remote server, Rule Miner incrementally updates its rules in the phone as follows. With rules, ACE also downloads various statistics required to compute supports and confidence of various rules. As new context tuples are added to context history, Rule Miner incrementally updates support and confidence statistics of existing rules and deletes the rules whose supports and confidence go below Rule Miner thresholds. Note that this only removes existing rules if they no longer hold, but does not produce new rules. However, this affects ACE's efficiency only (as ACE may miss opportunities to exploit potential rules), not the correctness of its inference. The inefficiency is addressed when new rules are downloaded from remote server.

3.1.3 Dealing with Inaccuracies

Since Inference Cache and Speculative Sensing are based on context rules, it is important than the rules are as accurate as possible. ACE conservatively uses rules that almost always hold (e.g., $minConf = 99\%$ in Apriori). A user, however, may change his habits, which may invalidate some rules. For example, he may buy a treadmill and start running at home instead of outside. Then, the rule $\{\texttt{IsRunning} = \texttt{True}\} \Rightarrow \{\texttt{AtHome} = \texttt{False}\}$ needs to be replaced with a new rule. To achieve this, ACE cross validates its inferred results with ground truths (collected by occasionally running contexters or getting feedback from users/applications, as mentioned in Section 2). ACE conservatively drops a rule if its support and confidence go below $minSup$ and $minConf$ due to the ground truths. New rules emerged due to change in habits are eventually learned by the Rule Miner as their itemsets become frequent.

3.1.4 Suppressing Redundant Rules

Rule mining algorithms often produce a large number of rules, especially with a small $minSup$. The cost of using the rules for Inference Cache and Sensing Planner increases linearly with the number of rules. Fortunately, many rules generated by standard rule mining algorithms are redundant; for example, the standard APriori algorithm can produce rules such as $\{A \Rightarrow B, B \Rightarrow C, A \Rightarrow C, AD \Rightarrow B\}$. However, given the two rules $\{A \Rightarrow B, B \Rightarrow C\}$, other two rules can be generated and hence they are redundant.

In order to reduce the inference overhead, ACE uses an algorithm for mining a compact, non-redundant set of association rules [28]. We have used both this algorithm and the Apriori algorithm on both our datasets. The basic Apriori algorithm generates around 750 rules on average on both datasets; eliminating redundant rules brings the number of rules below 40, on average, for both the datasets.

3.1.5 Bootstrapping

It takes a few days of context history for ACE to learn meaningful rules. Thus, in the beginning, ACE does not have any context history or rules that it can exploit. To deal with that, ACE starts with a set of seed rules that apply to almost all the users. Such *universal* rules are learned by the remote server where users offload their rule mining task. To learn such rules, the server runs the same rule mining algorithm, but on all users' context history. In the Reality Mining Dataset, more than 60% of the rules of each user are universal rules. As new context tuples are added to context history, ACE updates its rule set by learning new personalized rules and dropping universal rules that do not apply to the user.

4 Intelligent Context Caching

The Inference Cache in ACE, like traditional cache, provides a Get/Put interface. $Put(X, v)$ puts the tuple $X = v$ in the cache, with a predefined expiration time (default is 5 minutes). On a $Get(X)$ request for a context attribute X, it returns the value of X not only if it is in the cache, but also *if a value of X can be inferred from unexpired tuples in the cache by using context rules learned by the Rule Miner*.

To illustrate the operation of Inference Cache, assume that an application requires the value of `D` (`IsDriving`).

Suppose, ACE determines its value to be `True` by using the corresponding contexter. The tuple `D = T` is put in the cache with an expiration time of, e.g., 5 minutes. Suppose, before this tuple expires, another application requires the value of `D`. Like traditional cache, Context Cache returns the value `True`. Now, suppose a third application requires the value of `I` (`IsIndoor`). Even though there is no tuple with `I` in the cache, Context Cache infers that the user cannot be indoor, thanks to the rule $\{\texttt{D} = \texttt{T}\} \Rightarrow \texttt{I} = \texttt{F}$, and hence it returns the value `False` as the answer.

Note that Inference Cache can sometimes return results that are (a) incorrect, if user's behavior deviates from context rules, or (b) stale, if results are inferred from unexpired but stale values in cache, and hence incorrect if cached values no longer hold. As mentioned before, the severity of (a) is reduced by the Rule Miner conservatively choosing rules that almost always hold. Similarly, to reduce the effect of (b), we choose a small cache expiry time of 5 minutes. This helps us keep the overall inaccuracy of ACE within $< 4\%$ for our dataset. (Using an expiry time of 10 minutes and 15 minutes make the inaccuracy 6% and 10% respectively.)

A key challenge in enabling Inference Cache is to efficiently exploit rules. One straightforward way to do this is to go over all the rules to find the ones whose right hand sides contain the target attribute and all the tuples in their left hand sides are satisfied by currently unexpired tuples in the cache. Then, the cache can output the tuples on the right hand sides of those rules. However, this simple approach fails to exploit transitive relationship among rules. For example, consider two rules: $\{\texttt{M} = \texttt{T}\} \Rightarrow \texttt{O} = \texttt{T}$ and $\{\texttt{O} = \texttt{T}\} \Rightarrow \texttt{I} = \texttt{T}$. Without using transitivity of the rules, the cache will fail to infer the state of $\texttt{I} = \texttt{T}$ even if $\texttt{M} = \texttt{T}$ is in the cache. Such transitivity is common, especially since Rule Miner produces a succinct set of non-redundant rules.

To address this, ACE constructs *expression trees* over rules and performs inference on the trees.

4.1 Expression Trees

Informally, an expression tree for a tuple $\texttt{a} = \texttt{v}$ is a tree representation of the Boolean expression that implies the value `v` for the attribute `a`. It is generated by combining various rules that imply $\texttt{a} = \texttt{v}$ directly, or indirectly via transitive relationships with other rules. More formally, an expression tree is a Boolean AND-OR tree, where (a) a non-leaf node represents an AND or OR operation on the value of its child nodes, and (b) a leaf node represents a tuple. An AND node (or, OR node) evaluates to `True` if all (or any, respectively) of its child nodes recursively evaluate to `True`. The expression tree for the tuple $\texttt{a} = \texttt{v}$ is the one that evaluates to `True` if and only if the tuple holds according to context rules. Figure 6(b) shows an expression tree for the tuple $\texttt{Indoor} = \texttt{True}$.

The expression tree for a given tuple $\texttt{a} = \texttt{v}$ is constructed as follows. We start from a tree with a single leaf node $\texttt{a} = \texttt{v}$ (Figure 5(i)). Then, we continue *expanding* each leaf node in the current tree by using context rules until no leaf nodes can be further expanded or all the rules have been used for expansion. To expand a leaf node $\texttt{a} = \texttt{v}$, we find all the rules $R_1, R_2, \cdots, R_m$ that have $\texttt{a} = \texttt{v}$ on the right hand side. We then replace the leaf node with a subtree rooted at an OR node. The OR node has child nodes $\texttt{a} = \texttt{v}, X_1, X_2, \cdots, X_m$, where X_i is an AND node with all the tuples on the left hand side of R_i as child nodes. An example of this process is shown in Figure 5(ii), where the node $\texttt{I} = \texttt{T}$ is expanded using the rules shown at the top of the figure.

After a node is expanded, all the leaf nodes in the tree are recursively expanded (to deal with transitive relationship). Note that we do not need to expand the same tuple more than once; for example, in Figure 5, if the $\texttt{I} : \texttt{T}$ node is expanded again, we will unnecessarily replicate a portion of the tree, without changing the semantics of the tree. We thus expand each tuple only once—only if it has not already been expanded and there are rules with it on the right hand side. This ensures termination of the expansion process.

4.2 Inference Cache

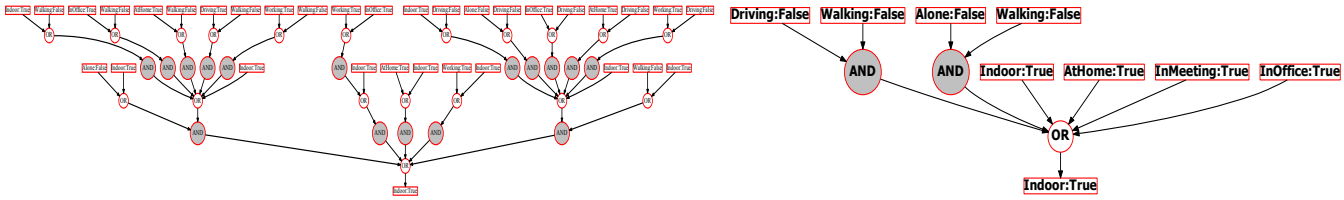
The Inference Cache maintains one expression tree for each tuple; e.g., one for the tuple $\texttt{D} = \texttt{T}$, one for $\texttt{D} = \texttt{F}$, and so on. On a Get request on attribute `X`, the cache evaluates two expression trees—one for $\texttt{X} = \texttt{T}$ and one for $\texttt{X} = \texttt{F}$—and see if any of these trees is satisfied (explained shortly) by unexpired tuples in the cache. If the tree for $\texttt{X} = \texttt{T}$ is satisfied, the Get call returns the value `True`, if the tree for $\texttt{X} = \texttt{F}$ is satisfied, `False` is returned, otherwise, the cache invokes the Sensing Planner module to invoke necessary contexters.

The procedure to check if an expression tree is satisfied with the tuples currently available in the cache is recursive. A leaf node is satisfied if and only if the corresponding tuple is in the cache. A non-leaf OR node is satisfied if *any* of its child nodes are satisfied (recursively). Similarly, an AND node is satisfied if *all* of its child nodes are satisfied. Finally, the whole tree is satisfied if the root node is satisfied.

4.3 Minimal Expression Tree

Expression trees obtained by expanding rules can be large. Since Inference Cache needs to maintain one tree for each tuple and needs to evaluate two trees on every Get request, larger trees impose memory and computation overhead. To avoid this, ACE normalizes the trees to shorter ones, by using standard Boolean logic to eliminate their redundancy.

►**Alternating AND-OR level:** An AND-OR tree can always be converted so that (a) a level has either all AND nodes or all OR nodes, and (b) AND levels and OR levels alternate. This can be done as follows: if an OR node (or an AND node) u has an OR child node (or an AND child



(a) Initial expanded tree (labels can be ignored) (b) Equivalent reduced tree

Fig. 6. An expression tree for `Indoor` = `True`, shown upside down

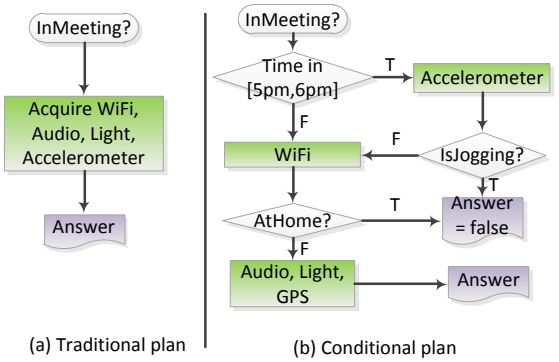


Fig. 7. A conditional acquisition plan that acquires different sensor data depending on various conditions

node, respectively) v, child nodes of v can be assigned to u and v can be deleted. This compacts the tree.

►**Absorption:** If a non-leaf node N has two child nodes A and B such that the set of child nodes of A is a subset of the set of child nodes of B, we can remove B and its subtrees. In a tree with alternating AND-OR levels, if N is an OR node, then A and B will be AND nodes. Suppose $A = a \wedge b$ and $B = a \wedge b \wedge c$. Then, $N = A \vee B = (a \wedge b) \vee (a \wedge b \wedge c) = a \wedge b = A$. Similarly, if N is an AND node, $(a \vee b) \wedge (a \vee b \vee c) = (a \vee b)$, and hence the longer subexpression $B = (a \vee b \vee c)$ can be removed.

►**Collapse:** If a node N has one child, the child can be assigned to N's parent node and N can be removed (Fig. 6(iii)).

Such transformation significantly reduces the size of expression trees without changing their semantics. Figure 6 shows the result of such transformation: (a) shows the tree for the tuple `Indoor` = `True` after initial expansion, and (b) shows the result of compacting the tree. (In (a), we intend to show the large size of the tree; the labels inside nodes are not expected to be read.)

5 Sensing Planner

When Inference Cache fails to determine the value of X on a $Get(X)$ call, ACE needs to call necessary contexters to determine the value of X. The straightforward way to find the value is to invoke the contexter that directly produces value of X (e.g., the `AtHome` contexter that uses GPS and WiFi signature). However, by using context rules, ACE may be able to find the value of X in an indirect and cheaper way, by *speculatively sensing* additional proxy attributes.

5.1 Conditional Plan for Speculative Sensing

We define a *sensing plan* as the order in which various attributes are checked in order to determine the value of a target attribute. An optimal ordering may depend on various conditions such as the cost c_i of checking an attribute i (i.e., the cost of running the corresponding contexter) and the likelihood p_i of the attribute returning a `True` value. A plan that orders attributes to check based on such conditions is called a *conditional plan*. We illustrate this with an example.

Suppose an application wants to find if the user is `InMeeting`. A traditional system will determine the value of the attribute by acquiring necessary sensor data, e.g., audio, GPS, and WiFi signature and by using necessary classifiers. However, ACE's conditional planning exploits context rules to opportunistically infer the value of `InMeeting` by acquiring attributes that are cheaper (in terms of sensing cost) and are likely to conclude a value of `InMeeting`. For example, if the current time is 5.30pm and the user is likely to be jogging between 5pm and 6pm, ACE can speculatively acquire the value of `IsJogging`, and if it is found to be `True`, ACE can infer `InMeeting` = `False` according to the rule `IsJogging` = `True` ⇒ `InMeeting` = `False`. Similarly, if checking `AtHome` is cheaper (by sensing WiFi only) and the user is likely to be at home, ACE can check if the user is actually at home and if so, can conclude `InMeeting` = `False`. If no other cheaper ways provide the value of the target attribute, ACE acquires the necessary sensor data to directly determine the value of the target attribute. Figure 7 shows both traditional and conditional plans for `InMeeting`.

Given a conditional plan, the true sensing cost depends on the exact path ACE takes to generate a result. In the previous example, if the user is indeed jogging, then the cost equals to the cost of acquiring accelerometer data only. The expected sensing cost of a plan can be computed by using the values of c_i and p_i of various attributes in the plan. ACE knows the value of c_i from the configuration file for contexter for attribute i. The value of p_i is computed from context history—ACE incrementally maintains one p_i for every hour of the day.

Figure 7 shows only one conditional sensing plan for the attribute `InMeeting`. However, there are many other plans—another valid plan is to first check whether both `IsDriving` and `IsWalking` are `False`, and if not, to check other attributes. In general, the number of valid plans can be exponential in the number of attributes. ACE

aims to choose the plan with the *minimum expected cost*.

Note that, even when using the best conditional plan, ACE's speculation may occasionally end up spending more energy than directly acquiring the target attribute. In the previous example, `IsJogging` may return `False`, in which case ACE needs to acquire `IsIndoor`, making the cost of acquiring `IsJogging` a waste. However, hopefully such mistakes are not made often (e.g., `IsJogging` is acquired only if it is likely to return `True`), and the expected sensing cost is less than the cost of directly acquiring target attributes.

Hardness of the Problem. The value of an attribute `A` is `True` if the Boolean expression given by the expression tree for the tuple $\texttt{A} = \texttt{True}$ is satisfied. How can we evaluate a Boolean expression with minimum cost? Note that value of a Boolean expression may be determined by evaluating only a subset of variables in the expression. Thus the evaluation cost depends on the order various attributes are checked. Suppose $\texttt{A}_1 = \texttt{False}, \texttt{A}_2 = \texttt{True}, \texttt{A}_3 = \texttt{False}$. Then the value of the Boolean expression $\texttt{A}_1 \vee \texttt{A}_2 \vee \texttt{A}_3$ can be determined by checking $\texttt{A}_2$ alone, without checking the values of $\texttt{A}_1$ and $\texttt{A}_3$. On the other hand, first checking $\texttt{A}_1$ does not yield any value of the expression, and requires checking other variables. An optimal algorithm would evaluate variables with small cost but high likelihood to determine the value of the whole expression before other variables.

The problem outlined above is an instance of the *probabilistic And-Or tree resolution* (PAOTR) problem. The problem is shown to be NP-Hard in general [9]. Efficient algorithms can be found for special cases when (a) each attribute appears at most once in the expression, (b) attributes are independent, and (c) the expression tree has a height of at most 2 [9]. The assumptions do not hold for our expression trees—they may violate any of these conditions.

5.2 Speculative Sensing API

Before describing how ACE chooses the best sensing plan, we describe how it programmatically uses a plan. One option is to enumerate the entire plan as a graph as shown in Figure 7. However, given such a plan, ACE will traverse only part of the graph, based on the outcomes of conditional nodes. Therefore, instead of enumerating the whole plan, ACE incrementally enumerates only the next node to traverse in the plan. More specifically, it implements following four methods, the details of which we will describe later.

- `Init(X)` : Initializes for a target attribute `X`.
- `attribute` ← `Next()` : Returns the attribute whose value needs to be determined next, or `null` if the target attribute value has already been determined.
- `Update(a, v)` : Updates the planner's internal state with the tuple $\texttt{a} = \texttt{v}$. The planner moves to the conditional branch of the plan based on this value of `a`.
- `value` ← `Result()`: Returns the value of attribute `X`.

Algorithm 1 Exhaustive search for optimal sensing plan

```
1: procedure INIT(X)
2:     target ← X
3:     trace, next, result, dpCache ← φ

4: procedure RESULT
5:     return result

6: procedure UPDATE(attrib, value)
7:     trace ← trace ∪ [attrib = value]
8:     if attrib = target then
9:         result ← value

10: procedure NEXT
11:     if result = φ then
12:         return φ
13:     (attrib, cost) ← NextHelper(trace)
14:     return attrib

15: procedure NEXTHELPER(trace)
16:     if trace is in dpCache then
17:         [next, cost] ← dpCache[trace]
18:         return [next, cost]
19:     minCost ← ∞
20:     bestAttrib ← φ
21:     for all State s ∉ trace do
22:         traceT ← trace ∪ {s = true}
23:         if traceT satisfies expressionTree(target = T) then
24:             CostT ← 0
25:         else
26:             [next, CostT] ← NextHelper(traceT)
27:         traceF ← trace ∪ {s = false}
28:         if traceF satisfies expressionTree(target = F) then
29:             CostF ← 0
30:         else
31:             [next, CostF] ← NextHelper(traceF)
32:         ExpctedCost ← Cost(s) + Prob(s = true) · CostT +
    Prob(s = false) · CostF
33:         if ExpectedCost < minCost then
34:             minCost ← ExpectedCost
35:             bestAttrib ← s
36:     dpCache[trace] ← [bestAttrib, minCost]
37:     return [bestAttrib, minCost]
```

When the Inference Cache requests the Sensing Planner to determine the value of the attribute X, it first calls `Init(X)`. Then, it repeatedly calls `a` ← `Next()`, determines the value `v` of `a` by using necessary contexter, and calls `Update(a, v)`. The process continues until the `Next()` call returns a `null`, at which point it returns the output of the `Result()` method to the Inference Cache.

5.3 Optimal Speculative Sensing Algorithm

The hardness result in Section 5.1 indicates that a polynomial time algorithm for obtaining an optimal plan is unlikely to exist. However, when the number of attributes is relatively small (fewer than 10), it may be feasible to exhaustively search for all possible ordering of the attributes and then to choose the one that minimizes the expected cost. With n attributes, there are $n!$ such ordering. Fortunately, we can reduce the computational

overhead with a dynamic programming algorithm that prunes the search space by early evaluation and caching.

The search space of our dynamic programming algorithm is defined over *traces*, where a trace is a sequence of tuples already sensed by various contexters. For example, suppose three invocations of the `GetNext()` method returned attributes `a`, `b`, and `c` and in response, their contexters have sensed their values to be `True`, `False`, and `True` respectively. Then the trace at that point is the sequence ¡`a` = `T`, `b` = `F`, `c` = `T`¿. A trace represents a sensing plan as it defines the attributes to be sensed and the order of sensing. The key observation that allows us to employ dynamic programming is that once an attribute `X` is sensed, the original problem of finding the optimal plan is decomposed into two independent subproblems: one for finding the best trace having `X` = `True` and the other for finding the best trace having `X` = `False`. Each subproblem covers a disjoint subspace of the attribute-value space covered by the original problem, and hence can be solved independently.

Algorithm 1 shows the pseudocode of four methods in Sensing Planner. Given a target attribute, the methods start with an empty trace. Given the current trace T, the `Next()` method finds the optimal next attribute as follows. For each attribute `A` $\notin T$, it considers its `True` and `False` value. If the current trace plus `A` = `T` or `A` = `F` satisfies the expression tree, the search is pruned on additional attributes. Otherwise, expected cost of each possibility is recursively computed and added together, with weights equal to the probability of `A` = `T` and `A` = `F`, to compute the overall expected cost (Line 32). Finally, the attribute with the minimum expected cost is returned. For efficiency, values of various traces are cached in $dpCache$ for memoization.

5.4 Heuristic Algorithm

When the number of attributes is large, the exhaustive algorithm can become prohibitively expensive. In such cases, ACE uses an efficient heuristic algorithm. The algorithm is *sequential* in nature—it does not use any conditioning predicate to dynamically choose the next best attribute, rather it ranks the attributes once based on their c_i and p_i values, and the attributes are sensed in that order regardless of observed attribute values. One ordering of attributes is computed for each target attribute. The nice aspect of this algorithm is that the ordering of attributes can be determined offline once (e.g., by a remote server) and updated periodically when the values of p_i change. `Next()` method simply returns attributes according to the optimal order.

We now describe how the ordering for a target attribute `X` is computed; pseudocode for four methods of Sensing Planner is omitted for brevity. Intuitively, the algorithm traverses an expression tree in a depth-first manner and ranks attributes with small cost but high probability of producing a conclusion before other attributes.

Algorithm 2 Generating attribute sensing order

```
1: procedure AssignCostAndProb(And-OR Tree Node T)
2:     if T is a leaf node then
3:         (C_T, P_T) ← (c_T, p_T)
4:         return (C_T, P_T)
5:     for all child node N_i of T, 1 ≤ i ≤ k do
6:         (C_N, P_N) ← AssignCostAndProb(N)
7:         if T is an AND node then
8:             P_N ← 1 − P_N
9:     Sort the child nodes of T in decreasing order of their
       values of P_{N_i}/C_{N_i}, 1 ≤ i ≤ k
10:    C_T ← C_{N_1} + Σ_{i=2}^{k} C_{N_i} Π_{j=1}^{i−1} (1 − P_{N_j})
11:    if T is an AND node then
12:        P_T ← Π_{i=1}^{k} P_{N_i}
13:    else
14:        P_T ← 1 − Π_{i=1}^{k} (1 − P_{N_i})
15:    return (C_T, P_T)

16: procedure FindOrdering(And-Or Tree Node T)
17:    if T is a leaf node then
18:        Q.Enqueue(T)
19:    for all child N_i of T in ascending order of P_{N_i}/C_{N_i} do
20:        FindOrdering(N_i)
```

1. Convert expression trees for `X` = `T` and `X` = `F` to their disjunctive normal forms. Combine them into one tree T by merging roots of both trees into one OR node.

2. Initialize a queue Q to be empty. At the end of the procedure, Q will contain attributes in the same order they need to be sensed. I.e., `Next()` simply needs to dequeue attributes from Q.

3. For each leaf node i in T, let $\mathcal{C}_i = c_i$ and $\mathcal{P}_i = p_i$.

4. Use the Procedure AssignCostAndProb shown in Algorithm 2 to assign costs and probabilities of all non-leaf nodes of T. The procedure simply traverses the tree in a depth-first manner, first considering nodes that have lower cost and higher probability to conclude a value for its parent node. If node i's parent is an OR node, then the probability $\mathcal{P}_i$ that i can conclude (a `True` value of) its parent is $\mathcal{P}_i$, the same as the probability of i being `True`. If the parent is an AND node, then the probability $\mathcal{P}_i$ that i can conclude (a `False` value of) its parent is $1 - \mathcal{P}_i$.

5. Finally, use FindOrdering in Algorithm 2 to enqueue attributes to Q in the order they should be sensed. The ordering prefers nodes with higher values of $\mathcal{P}_i/\mathcal{C}_i$; i.e., with higher $\mathcal{P}_i$ and lower $\mathcal{C}_i$.

The heuristic above is a generalization of the 4-approximate algorithm proposed by Mungala et al. [20] for finding the optimal sequential plan for conjunctive queries. The generalization is done in order to support arbitrary AND-OR expression. The depth-first nature and sorting child nodes based on $\mathcal{P}/\mathcal{C}$ values are similar to an algorithm in [9], which is optimal for a depth-2 tree with independent and non-repeated attributes. Our evaluation in Section 7 shows that the heuristic algorithm provides only $< 10\%$ worse plans than the

optimal algorithm, but runs significantly faster than the exhaustive algorithm.

6 Discussion

As mentioned before, energy savings in ACE come at the cost of occasional inaccuracy in context inference. This limitation is fundamental to ACE because the ground truth value of a context attribute can be found by sensing the attribute, which ACE tries to avoid as much as possible. There are several important parameters that can affect this inaccuracy: rule mining parameters (support and confidence), cache expiry time, and cross-validation frequency (i.e., how often outputs of ACE are compared with ground truths, in order to drop invalid rules). These parameters need to be carefully set based on target application scenarios.

There are few other limitations in the current version of ACE. These limitations are not fundamental and can be addressed in future work. First, currently ACE supports Boolean attributes only. Supporting non-Boolean attributes (e.g., a user's location) would introduce complexity in various components of ACE. One possible approach would be to make the Rule Miner learn more complex rules in the form of decision trees, the Inference Cache to employ decision tree-based inference algorithm, and the Sensing Planner to use planning techniques such as the ones used in [6].

Second, ACE currently does not exploit temporal correlation across attributes. We believe that capturing and exploiting rules such as "if a user is `inOffice` now, he cannot be `atHome` for the next 10 minutes" can be very useful in practice. Such rules can also be useful in efficient support for context-aware triggers (e.g., a location-based reminder) as the rules would enable applications to check contexts less frequently (similar to SeeMon [12]).

Third, the Inference Cache uses the same expiry time for all context attributes. In practice, expiry time can depend on attribute semantics. ACE can support this by letting each contexter define a suitable expiry time of its attribute—ACE can easily use this expiry time without knowing the semantics of the attribute.

Fourth, the cost function in Sensing Planner considers only the energy cost of acquiring a context attribute. However, ACE could use a more complex cost function (e.g., a weighted linear combination) of various parameters such as energy, latency, and accuracy of relevant contexters. How these various parameters are combined may depend on application scenarios. Once such a cost function is defined, the algorithms described in Section 5 will work seamlessly.

Fifth, the Inference Cache currently returns only the value of an attribute. It can be easily extended to return the confidence of a returned value. Note that the Rule Miner computes confidence of each rule (to keep the ones with confidence $\geq minConf$). Thus, the Inference Cache can compute the confidence of a result from the confidence of all rules used to infer the result.

7 Evaluation

In this section we evaluate various components of ACE and find that (1) the Inference Cache alone can reduce the sensing energy consumption by a factor of $2.2\times$ (for the Reality Mining Dataset), compared to a baseline cache that shares sensor data and context attribute across applications. (2) the Sensing Planner can further reduce the energy consumption by an additional factor of $\approx 2\times$. Overall, applications running on ACE consume $4.2\times$ less sensing energy on average compared to what they consume when running on a baseline cache. The savings would have been less than half had ACE used a set of predefined rules instead of dynamically-learned personalized rules for each user. (3) the latency and memory overhead introduced by ACE is negligible (< 0.1ms and < 15MB respectively on a Samsung Focus phone).

Our performance related experiments are done with a prototype of ACE on a Samsung Focus phone running Windows Phone 7.5 OS. The phone has 1 GHz Scorpion CPU, 512MB RAM, and a Li-Ion 1500 mAh battery. We measure effectiveness of various ACE components with the Reality Mining Dataset and the ACE Dataset mentioned in Section 2.5, and these trace-driven experiments are done with an ACE prototype running on a laptop running Windows 7.

Prototype Implementation. We have implemented ACE in Windows Phone 7.5 (Mango) platform. It is implemented as a C# library shared by context-aware applications. (We use the same prototype on a laptop running Windows 7 for our trace-driven evaluation.)

Windows Phone OS currently imposes several restrictions on its apps. It does not allow cross-application sharing or communication within the phone; all sharing/communication happen through the Cloud. To avoid such expensive communication, we build our context-aware "applications" as separate tabs (i.e., *pivots*) within a single Windows Phone app. This allows ACE to share sensor data and context attributes among multiple "applications." This restriction, however, does not affect the overhead numbers we report here. We are currently porting our prototype to Android that does not pose the above restriction and hence independent applications will be able to use ACE as a library.

Trace-driven Experiments. We use three applications and two datasets described in Section 2 for evaluating ACE's effectiveness in reducing sensing energy. The applications use various context attributes mentioned in Table 1. Specifically, the JogTracker application requires the attributes `IsWalking`, `IsJogging`, and `IsBiking`(Reality Mining dataset only), the GeoReminder application requires `AtHome`, `InOffice`, `Indoor`(ACE dataset only), and PhoneBuddy requires `IsDriving`, `InMeeting` (ACE dataset only), `IsAlone`, and `InOffice`.

To work with the datasets, we port our ACE prototype to run on a laptop and replace its contexters with dummy ones that return a user's current context from

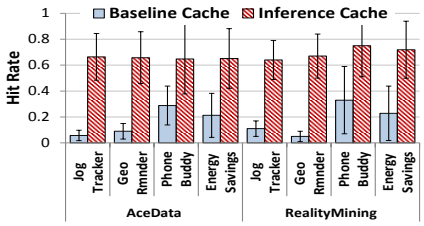


Fig. 8. Hit rates and energy savings of Inference Cache

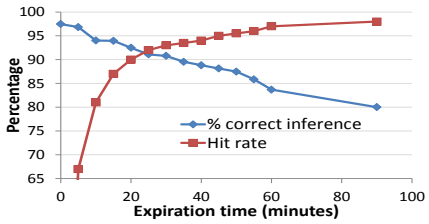


Fig. 9. Effect of cache expiration time on hit rate and accuracy

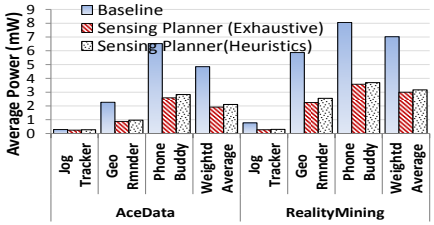


Fig. 10. Energy savings by Sensing Planner

a given trace, instead of by acquiring necessary sensor data. The energy costs of various contexters are modeled according to our energy measurements of real contexters, as shown in Table 1. We process each user's trace in discrete ticks, where each tick represents 5 minutes in wall clock time.[6] At each tick, we run all three applications and a contexter invoked by an application returns the true context of a user at corresponding wall clock time from his trace. To avoid any bias in caching results, at each tick we run the applications in a random order, and each application requests for required attributes in a random order. Sensor data and context attributes have an expiration time of 1 tick; i.e., applications can share them within the same tick, but not across ticks. Increasing the expiration time will further increase the benefit of ACE.

For each user, we start our experiment with rules learned from first 1 week of data (and then the rules are dynamically updated). Unless otherwise stated, we report the sensing energy only; i.e., the energy consumed by contexters executed in order to satisfy requests of various applications. Total energy is reported in Section 7.5.

7.1 Effectiveness of Inference Cache

We first show that Inference Cache of ACE gives a significant energy-savings over a baseline cache that shares sensor data and context attributes across applications.

Figure 8 shows the average and 95% confidence interval of hit rates of various applications under two datasets. The applications have some overlap in terms of required sensor data and context attributes, and hence the baseline cache gives nonzero hit rates ($5-30\%$). The hit rate is the highest for PhoneBuddy because it has the maximum overlap with other applications. Compared to the baseline cache, Inference Cache consistently provides significantly higher hit rates ($65-75\%$), for all applications and datasets. The additional benefit comes from its ability to return the value of an attribute even if it is not in the cache, yet can be inferred from existing attributes in the cache.

Since various attributes have different energy costs, the overall energy savings due to a cache hit differs across attributes (and applications). Therefore, in Figure 8, we also plot the fraction of energy savings compared to a *No Sharing* scheme that does not share sensor data or context attribute among applications. Both baseline cache and Inference Cache provides non-zero energy savings; but the energy savings with Inference Cache is $> 3\times$ more than that with the baseline cache.

7.2 Accuracy

Figure 9 shows the effect of expiry time on cache hit rates and accuracy of results returned by $Get()$. The effect is shown for the Reality Mining dataset only. As expected, increasing expiration times increases cache hit rates. However, with large expiry times, Inference Cache can contain stale tuples, and any inference based on them can be stale or incorrect. Figure 9 shows inaccuracy due to such staleness as well as due to users deviating from automatically learned context rules. As shown, the inaccuracy is negligible with a small expiration time (e.g., $< 4\%$ with ACE's default expiration time of 5 minutes).

7.3 Effectiveness of Sensing Planner

In Figure 10, we show average power consumed by various applications for context sensing. We compare with a baseline planner that determines the value of an attribute by directly executing corresponding contexter. Both schemes are run with the Inference Cache enabled; i.e., the value of an attribute is determined only on a cache miss.

Figure 10 shows that, compared to the baseline planner, the heuristics-based Sensing Planner reduces energy consumption by $5-60\%$ for various applications and datasets. The savings is more significant in applications requiring expensive attributes (e.g., PhoneBuddy), because often those expensive attributes are computed by executing cheaper contexters. Since Jog Tracker already uses cheap attributes, the Planner does not provide significant savings beyond the savings due to Inference Cache. Overall, the savings is around 55% on average for both datasets.

The above savings can be slightly higher if Sensing Planner uses the dynamic programming-based exhaustive algorithm, instead of the heuristic algorithm. However, as shown in Figure 10, the heuristic algorithm performs close to the exhaustive algorithm—the difference is less than 10% for both datasets. On the other hand, the heuristic algorithm runs significantly faster than the exhaustive algorithm, as we will show later. Therefore, ACE uses the heuristic algorithm as default.

6. A shorter tick will increase the absolute energy savings by ACE.

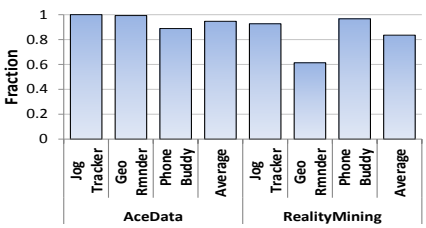


Fig. 11. Fraction of times Sensing Planner performs better than or as good as baseline

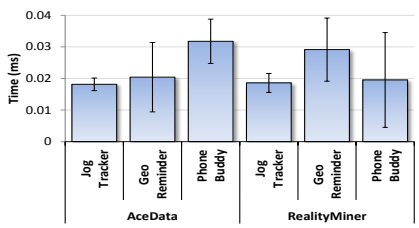


Fig. 12. Latency of a Get() request on Inference Cache hit

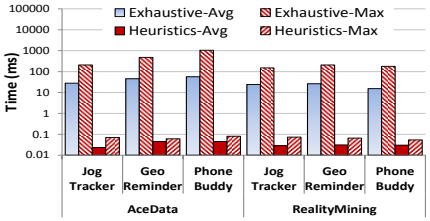


Fig. 13. Time required to generate a plan on a Samsung Focus phone

As mentioned in Section 5, speculative sensing may occasionally be useless and consume more energy than the baseline planner. Figure 11 shows the fraction of times Sensing Planner consumes no more energy than the baseline planner. As shown, Sensing Planner consumes less or equal energy as the baseline planner most of the time. The only case Sensing Planner makes a lot of "mistakes" ($\approx$ 40%) is with the GeoReminder application and Reality Mining dataset. This is due to the fact that probability distributions of various attributes in this dataset have high variances for a few users (because the data is not precise enough; e.g., location is based only at the granularity of a cell tower). Since Sensing Planner chooses an attribute a_i based on its *average* probability p_i of being `True`, it makes more mistakes for p_i with high variance. The possibility of such high variance decreases if the user's behavior is repetitive and the prior probability distribution is tight. This is generally the case, as we found in the ACE Dataset. Note that even though Sensing Planner consumes more energy sometimes, on average, it consumes less energy than the baseline planner in all our applications and datasets.

7.4 Overhead of ACE

Memory. The memory footprint of our ACE prototype running on the Samsung phone is $< 15MB$ for all users in both datasets.

Latency. Figure 12 shows the average latency of a $Get()$ request when it is answered by the Inference Cache (i.e., a cache hit). The latency depends on complexity of expression trees, and we report both average and 95% confidence interval of the latency on the Samsung phone. The latency is negligible.

On an Inference Cache miss, ACE needs to use Sensing Planner to determine the value of the target attribute. Figure 13 shows the time spent by Sensing Planner on such a miss. (The latency does not include the sensing latency.) We show both average and maximum latencies for the exhaustive algorithm and the heuristic algorithm. The exhaustive algorithm can be slow and take up to a second. Planning time of exhaustive algorithm depends on the number of attributes. The ACE Dataset has one more attribute than the Reality Mining Dataset, and this results in an average latency increase of $\approx 2\times$ for the ACE Dataset. This shows that the exhaustive algorithm does not scale well as the number of attributes increases. In contrast, the heuristic algorithm, which is the default in ACE, runs very fast (< 0.1 ms) and remains comparable across various datasets and applications.

Overall, the overhead introduced by ACE on a $Get()$ request (hit or miss) is less than a millisecond. This is negligible compared to the time needed to acquire sensor data and to infer user contexts (can be up to many seconds).

7.5 End-to-end Energy Savings

Figure 14 shows the average end-to-end sensing energy savings of ACE for the Reality Mining dataset. "No sharing" refers to a scheme where applications do not share any sensor data or context attributes with one another (e.g., if two applications require `IsDriving` attribute, they both invoke their contexters, which collect accelerometer data independently). As shown, a baseline cache reduces the energy consumption by 24%. In comparison, Inference Cache and the combination of Inference Cache and Sensing Planner reduce energy consumption by 64% and 82% respectively. Overall, ACE consumes $4.2\times$ less sensing energy than the combination of baseline cache and baseline planner.

Clearly, the overall benefit of ACE depends on individual users as the context rules vary across users. Figure 15 shows the savings of various schemes for all 95 users in the Reality Mining dataset. User IDs are sorted according to energy consumed by ACE. As shown, for most users, both Inference Cache and Sensing Planner provide significant savings.

Total Energy. So far we have reported ACE's savings only with respect to sensing energy. How significant sensing energy is compared to *total energy* consumption of an app due to display, CPU, network, etc. depends on how often the app senses—the more frequently it senses user's context, the higher its sensing energy overhead is, and the more significant the savings provided by ACE are. We empirically show this in Figure 16. It shows ACE's savings compared to the total energy consumed by an app, as a function of its average sensing interval. Without any sensing (i.e., an infinite interval), the app consumes 366mW. As the sensing interval decreases, it consumes more total energy, due to overhead of additional sensing. As show in Figure 16, ACE can save

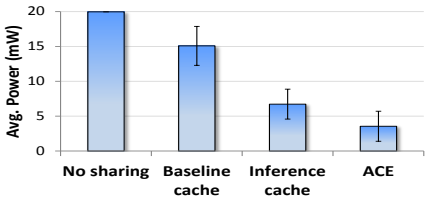


Fig. 14. End-to-end sensing energy

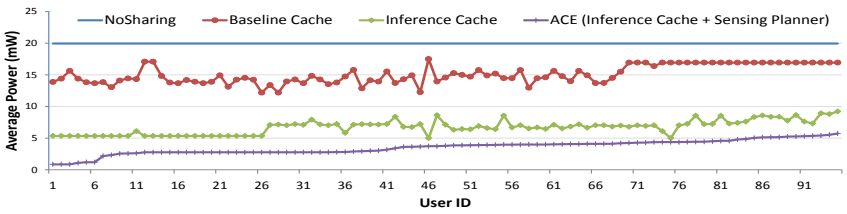


Fig. 15. Per user power consumption (users sorted by ACE power)

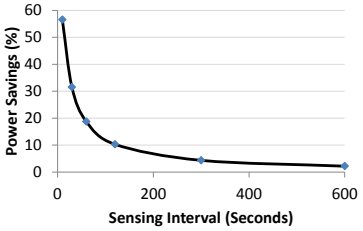


Fig. 16. Savings of ACE compared to total energy (including sensing, display, CPU, network, etc.)

a significant fraction of this additional sensing energy. For example, it can save around 20% and 10% of total energy for apps with sensing intervals of 2 minutes and 1 minute, respectively.

7.6 Importance of Dynamic Rule Learning

Finally we ask *how much value do dynamically learned rules add compared to a set of predefined rules?* To answer this, we compare two versions of ACE: **ACE-D:** ACE with its default Rule Miner and **ACE-S:** ACE with a set of predefined static rules.

The performance of **ACE-S** will depend on the set of rules chosen a priori. For concreteness, we select universal rules from the ones discovered by Rule Miner. Ideally, we would like to select universal rules that hold for *all* the users. However, Rule Miner may miss a few of such rules for some users if their context traces do not have enough evidence of necessary patterns. We therefore choose k most frequent rules among all users. A caveat of this approach is that some of the selected rules may not hold for some users, and hence the value of k needs to be carefully chosen.

We measure the end-to-end energy of **ACE-D** and **ACE-S** for the Reality Mining dataset. **ACE-D** consumes around 3.5mW (shown in Figure 14). In contrast, **ACE-S** consumes around 10mW and 7.5mW with $k = 10$ and $k = 15$, respectively. (Energy consumption increases for larger k as some rules do not hold for some users). This $> 2\times$ savings in **ACE-D** show the significance of dynamically learned personalized rules.

8 Related Work

The idea of context-aware computing is not new [23]; however it has gained a lot of interest recently, mostly due to the increasing availability of rich sensors on today's mobile devices. iOS, Android, and Windows Phone marketplaces contain a large number of context-aware applications. Most of these applications aim to enable new scenarios and to provide new features, with no or very basic optimizations for reducing energy overhead.

Research community has also proposed many techniques to infer various context attributes such as activity [4], [13], location [2], [15], group state [27], proximity to others [3], surrounding sound events [16], etc. All these works are orthogonal to ACE that performs context inference within its black-box contexters. ACE can use any of these techniques in its contexter implementation. Techniques have also been proposed for inferring context attributes in energy-efficient ways (e.g., a-Loc [15] for energy-efficient location and [13] for energy-efficient activity). Unlike ACE, these energy-efficient optimizations are applicable within a single attribute, rather than across multiple attributes. ACE could easily incorporate these optimizations within its contexter implementation. For example, a-Loc [15] uses different modalities of sensors (e.g., GPS, WiFi, cell tower, dead reckoning, etc.) to infer a user's location based on availability of sensors and accuracy requirements of applications. ACE could use an a-Loc location contexter that would dynamically choose the cheapest sensing modality to infer various location attributes (e.g., `AtHome`). For such optimizations where the energy cost of an optimized contexter changes depending on its modality, however, ACE needs to consider energy cost of the current mode in its Inference Cache and Planner.

SeeMon [12], like ACE, focuses on efficient and continuous context monitoring. It uses three optimizations: detecting change in context at earlier stages of the processing pipeline, exploiting temporal continuity of contexts, and selecting a small subset of sensors sufficient to answer a query. These optimizations can be used inside ACE's black-box contexters. The second optimization can also be used in ACE to produce richer rules such as "if a user is `inOffice` now, he cannot be `atHome` in next 10 minutes." Exploring such rich rules is part of our future work.

Our idea of sharing context attributes across applications is similar to using middleware cache in a mobile system [10]. However, unlike ACE's Inference Cache, these systems support exact lookup only. Sensor substitution techniques substitute one sensor with another semantically related sensor for inferring a target attribute. SENST* [24] and a-Loc [15] have shown this for inferring location by substituting one expensive sensor with

another cheap sensor. Inference cache is more powerful in that 1) it supports arbitrary context attributes, 2) it supports attributes related to each other by a set of general association rules, and 3) the rules are learned automatically.

Our idea of speculative sensing is analogous to speculative execution used in pipelined processor [11] and other system architecture [21]. In contrast, we use speculation for sensing and with conditional planning. Prior work has used conditional planning for acquisitional query processing [6], however it 1) exploits correlation of various attributes, instead of association rules, 2) uses plans at sensor-level rather than context attribute-level, and 3) uses conditional plan for query processing rather than general context-aware applications.

9 Conclusion

We have presented ACE, a middleware for efficient and continuous sensing of user's context in a mobile device. ACE automatically learns relationships among various context attributes and uses them for two novel optimizations: inference caching and speculative sensing. Experiments with two real context traces of total 105 people and a Windows Phone prototype show that, compared to a traditional cache, ACE can reduce sensing costs of three context-aware applications by around $4.2\times$, with a very small memory and processing overhead.

Acknowledgements. We thank our MobiSys 2012 shepherd, Alexander Varshavsky, and anonymous reviewers for their feedback. We thank Tarek Abdelzaher, Jie Liu, and Oriana Riva for feedback on earlier drafts; Miguel Palomera for writing an earlier version of the ACE Data collection tool; and various members of the Sensing and Energy Research Group at Microsoft Research for collecting context data.

References

[1] Agrawal, R., Imieliński, T., and Swami, A. Mining association rules between sets of items in large databases. In *ACM SIGMOD* (1993).

[2] Azizyan, M., Constandache, I., and Roy Choudhury, R. Surroundsense: mobile phone localization via ambience fingerprinting. In *ACM MobiCom* (2009).

[3] Banerjee, N., Agarwal, S., Bahl, P., Chandra, R., Wolman, A., and Corner, M. Virtual compass: Relative positioning to sense mobile social interactions. In *Pervasive* (2010).

[4] Choujaa, D., and Dulay, N. TRAcME: Temporal activity recognition using mobile phone data. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (2008).

[5] Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: making smartphones last longer with code offload. In *MobiSys* (2010).

[6] Deshpande, A., Guestrin, C., Hong, W., and Madden, S. Exploiting correlated attributes in acquisitional query processing. In *ICDE* (2005).

[7] Eagle, N., Pentland, A., and Lazer, D. Inferring social network structure using mobile phone data. In *Proceedings of the National Academy of Sciences (PNAS)* (2009), vol. 106, pp. 15274–15278.

[8] Ficek, M., and Kencl, L. Spatial extension of the reality mining dataset. In *International Conference on Mobile Adhoc and Sensor Systems (MASS)* (2010).

[9] Greiner, R., Hayward, R., Jankowska, M., and Molloy, M. Finding optimal satisficing strategies for and-or trees. *Artificial Intelligence 170* (January 2006), 19–58.

[10] H. Höpfner, K. S. Cache-supported processing of queries in mobile dbs. *Database Mechanisms for Mobile Applications* (2003), 106–121.

[11] Hammond, L., Willey, M., and Olukotun, K. Data speculation support for a chip multiprocessor. In *ASPLOS* (1998).

[12] Kang, S., Lee, J., Jang, H., Lee, H., Lee, Y., Park, S., Park, T., and Song, J. SeeMon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *ACM MobiSys* (2008).

[13] Kwapisz, J. R., Weiss, G. M., and Moore, S. A. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl. 12* (March 2011), 74–82.

[14] Lane, N. D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. T. A survey of mobile phone sensing. *Comm. Mag. 48* (September 2010), 140–150.

[15] Lin, K., Kansal, A., Lymberopoulos, D., and Zhao, F. Energy-accuracy trade-off for continuous mobile device location. In *ACM MobiSys* (2010).

[16] Lu, H., Pan, W., Lane, N. D., Choudhury, T., and Campbell, A. T. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys* (2009).

[17] Lu, H., Yang, J., Liu, Z., Lane, N. D., Choudhury, T., and Campbell, A. T. The jigsaw continuous sensing engine for mobile phone applications. In *ACM SenSys* (2010).

[18] Miluzzo, E., Lane, N. D., Fodor, K., Peterson, R., Lu, H., Musolesi, M., Eisenman, S. B., Zheng, X., and Campbell, A. T. Sensing meets mobile social networks: the design, implementation and evaluation of the CenceMe application. In *ACM SenSys* (2008).

[19] Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., and Boda, P. PEIR, the personal environmental impact report, as a platform for participatory sensing systems research. In *ACM MobiSys* (2009).

[20] Munagala, K., Babu, S., Motwani, R., and Widom, J. The pipelined set cover problem. In *ICDT* (2005).

[21] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst. 24* (November 2006), 361–392.

[22] Qin, C., Bao, X., Choudhury, R. R., and Nelakuditi, S. TagSense: A smartphone based approach to automatic image tagging. In *ACM MobiSys* (2011).

[23] Schilit, B., Adams, N., and Want, R. Context-aware computing applications. In *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications* (1994).

[24] Schirmer, M., and Höpfner, H. SENST*: approaches for reducing the energy consumption of smartphone-based context recognition. In *CONTEXT* (2011).

[25] Tarzia, S. P., Dinda, P. A., Dick, R. P., and Memik, G. Indoor localization without infrastructure using the acoustic background spectrum. In *ACM MobiSys* (2011).

[26] Ullman, J. D. A survey of association-rule mining. In *Third International Conference on Discovery Science* (2000).

[27] Wirz, M., Roggen, D., and Troster, G. Decentralized detection of group formations from wearable acceleration sensors. In *Intl. Conf. on Computational Science and Engineering - Volume 04* (2009).

[28] Zaki, M. J. Mining non-redundant association rules. *Data Mining and Knowledge Discovery 9*, 3 (Nov 2004), 223–248.

Suman Nath (Ph. D. and Masters, Carnegie Mellon University, 2005, 2003, B.Sc., Bangladesh University of Engineering and Technology, 1998) is a senior researcher in Microsoft Research Redmond. His research interests lie in the intersection of mobile, sensing, and database systems. His research work has been recognized by best paper awards in ACM MobiSys 2012, SSTD 2011, IEEE ICDE 2008, USENIX NSDI 2006, and IEEE/CreateNet BaseNets 2004.