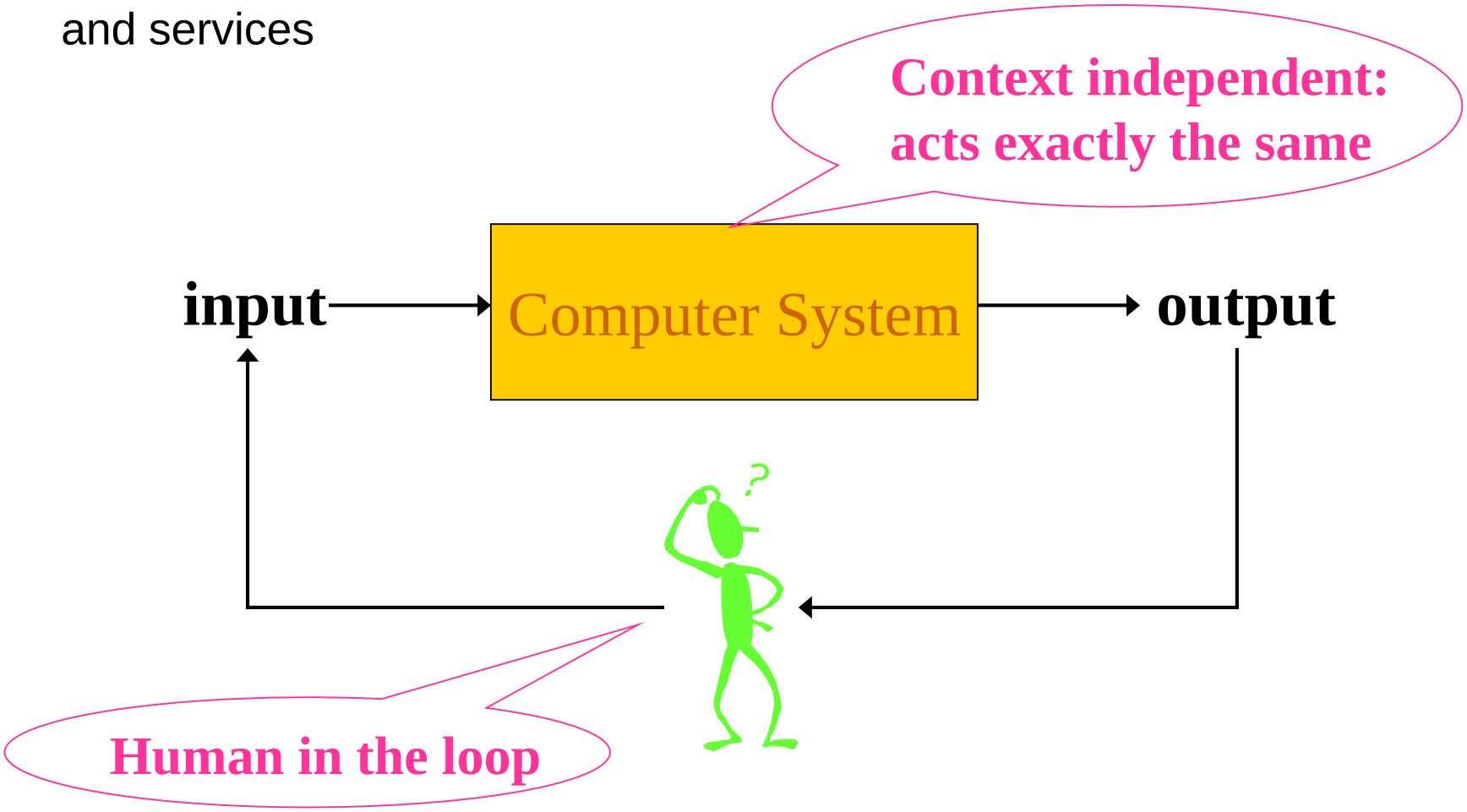
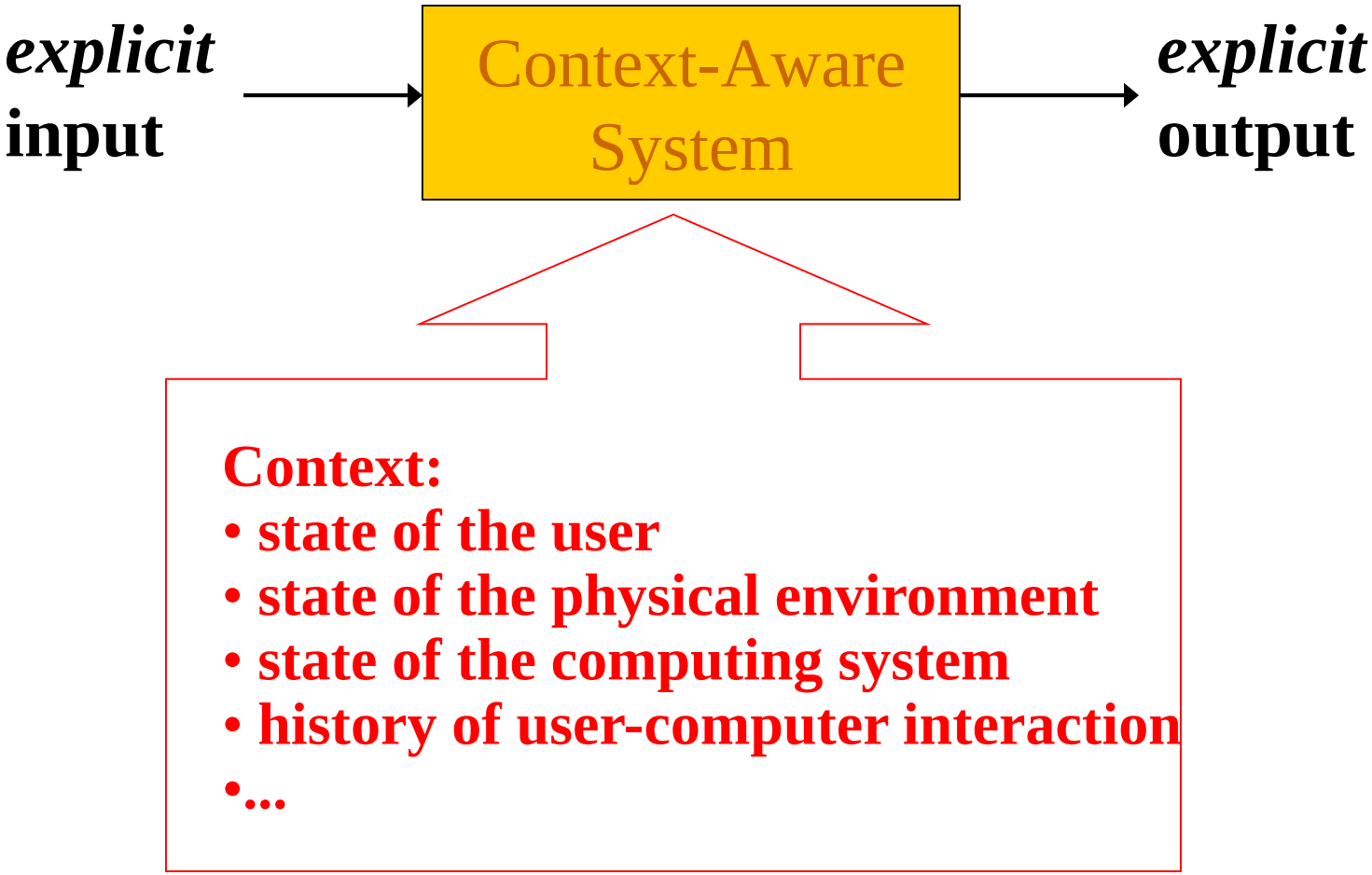
Context Sensing

Context Aware Computing

- *Context aware computing:* the use of sensors and other sources of information about a user's context to provide more relevant information and services



Context Aware Computing



What is context?

- Identity (Who)
- Activity (What)
- Time (When)
- Location (Where)

- Who + What + When + Where → Context

Examples of Context

- Identity
- Spatial: location, orientation, speed
- Temporal: date, time of day, season
- Environmental: temperature, light, noise
- Social: people nearby, activity, calendar
- Resources: nearby, availability
- Physiological: blood pressure, heart rate, tone of voice
- Emotion

Rule based Context Sensing

Background

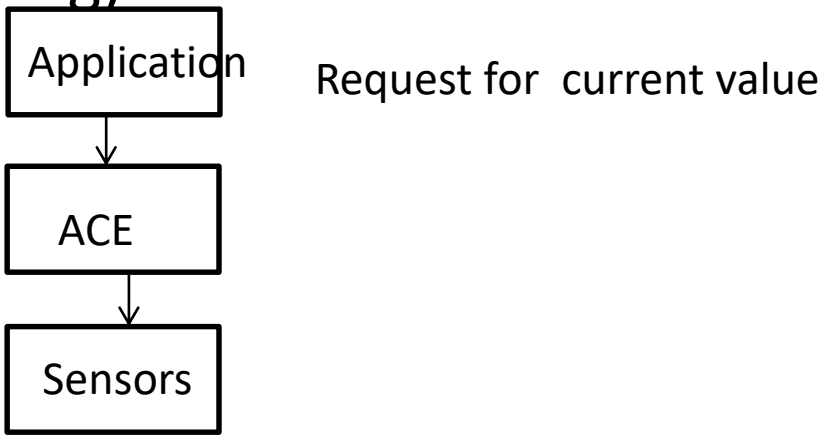
- Context sensing
  - Sensors in smartphone
  - Applications react based on operating condition
- Example
  - Location based reminder, Mute phone while in meeting, Greeting message while driving
- Energy consumption
  - Big overhead
  - Continuous context sensing

Contribution

- Develop ACE
  - Acquisitional Context Engine
  - A middleware that supports continuous context-aware applications
  - While mitigating sensing costs for inferring context

ACE

- Provides user's current context to applications running on it
- Context attributes (AtHome, IsDrving)



- Key Idea: Human contexts are limited by physical constraints => context attributes are correlated
- Dynamically learns relationships among various context attributes
  - e.g., whenever the user is Driving, he is not AtHome

Key idea

- We can infer unknown context attribute values from
  - Known context attributes
  - Unknown but cheaper context attribute
- Bob is running App1 (monitors **walking, driving, running** - accelerometer) and App2 (monitors location AtHome, Isoffice---GPS, WiFi, BT)
- ACE infers rule: When driving is T, AtHome is F

- Context rules

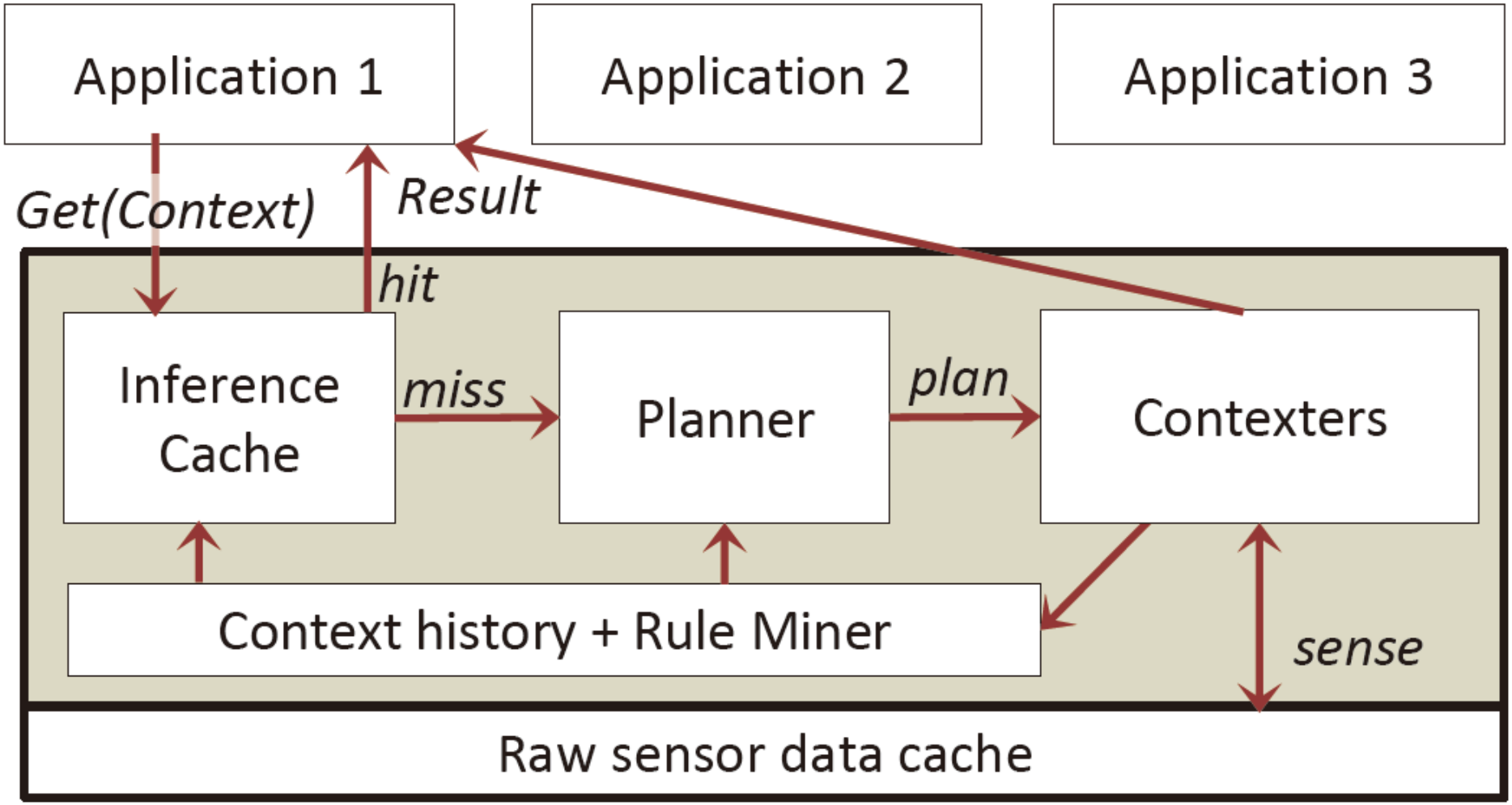
Two Powerful Optimizations

- ACE exploits these automatically learned relationships for two powerful optimizations
- The first is inference caching
  - allows ACE to opportunistically infer one context attribute (AtHome) from another already-known attribute (Driving)
  - without acquiring any sensor data
- Example: App1 requests value of Driving ---- Request, cache, App2---cache
- App3 requests for AtHome =>short window
- Respond without sensor data=>proxy

Two Powerful Optimizations

- The second optimization is speculative sensing IsOffice=>GPS, wifi
- Driving-T, Running-T, AtHome-T
  - IsOffice-F
- InMeeting-T=>IsOffice-T
- Enables ACE to occasionally infer the value of an expensive attribute (e.g., AtHome) by sensing cheaper attributes (e.g., Driving)

ACE Architecture



(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 the necessary inference algorithm. An application can extend it by implementing additional contexters. The set of attributes sensed by various contexters is exposed to applications for using 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 sequence of proxy attributes to speculatively sense to determine the value of the target attribute in the cheapest way.

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

Contexters

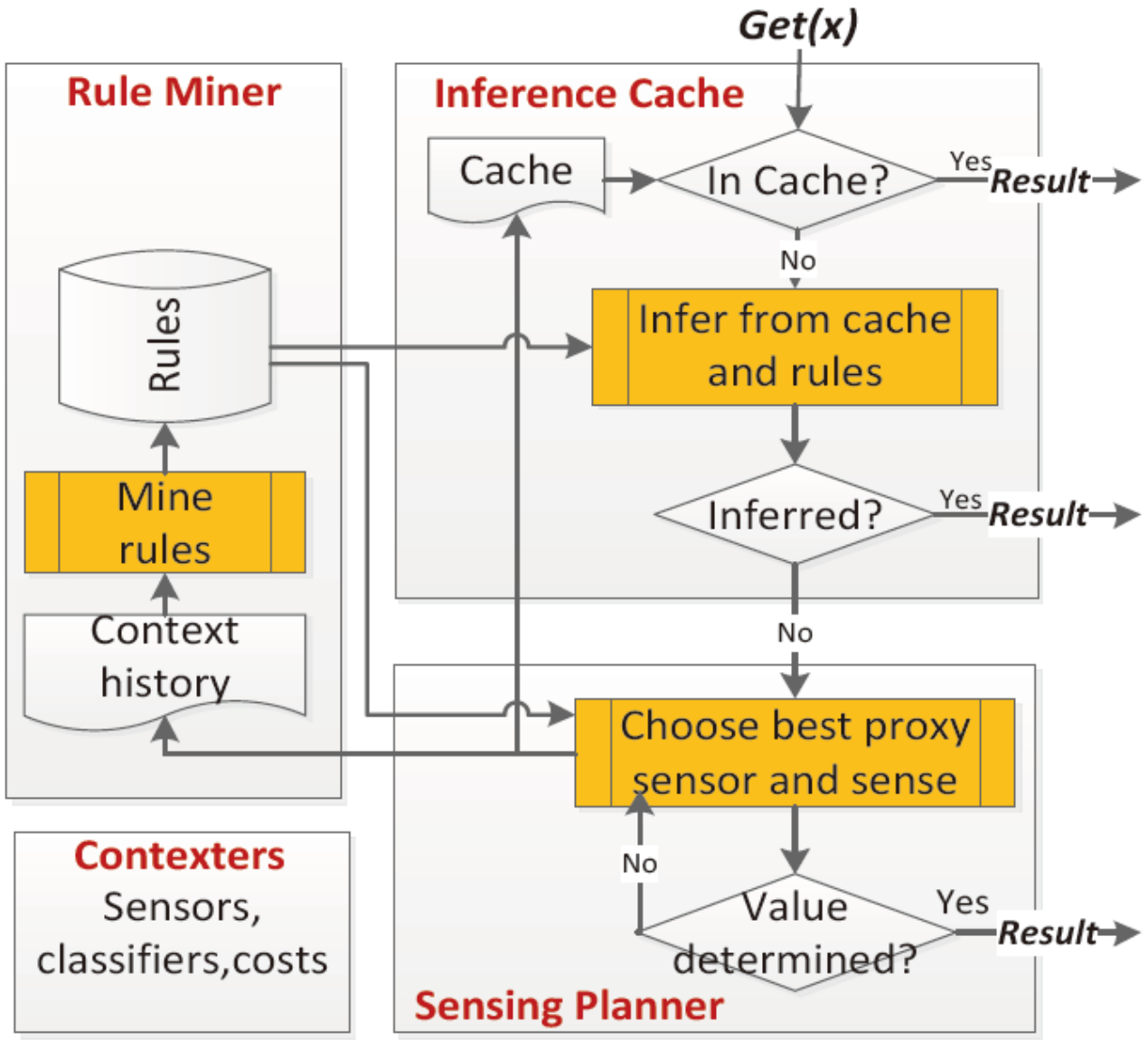
- Collection of modules
  - Determine the current value of context attribute
  - Data from sensors
- Denote each context attribute and value as (tuple) {D=T}
- Derived values based on decision tree
  - Based on training data

Contexters

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

Work Flow of ACE



Rule Miner

- Maintains the history of time stamped tuples {AtHome=true}
- Derives rules regarding relationship among various context attributes
- Association rule mining (Apriori algo)

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 $\{\mathtt{O} = \mathtt{T}, \mathtt{W} = \mathtt{F}, \mathtt{A} = \mathtt{F}\} \Rightarrow \mathtt{M} = \mathtt{T}$ implies that if a user is in the office ($\mathtt{O} = \mathtt{T}$) and not walking ($\mathtt{W} = \mathtt{F}$) and is not alone ($\mathtt{A} = \mathtt{F}$), then he is in a meeting ($\mathtt{M} = \mathtt{T}$).

Algo Outline

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 $\geq minConf$.

Rule Miner

Determine frequent item set

Apriori algo

Itemsets
{1,2,3,4}
{1,2,4}
{1,2}
{2,3,4}
{2,3}
{3,4}
{2,4}

Support threshold=3

Item	Support
{1}	3
{2}	6
{3}	4
{4}	5

Item	Support
{1,2}	3
{1,3}	1
{1,4}	2
{2,3}	3
{2,4}	4
{3,4}	3

Item	Support
{2,3,4}	2

The pairs {1,2}, {2,3}, {2,4}, and {3,4} all meet or exceed the minimum support of 3, so they are frequent. The pairs {1,3} and {1,4} are not.

any larger set which contains {1,3} or {1,4} cannot be frequent.

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., $\mathtt{W} = \mathtt{T}$) 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.

Rule Miner

Table 2: A few example rules learned for one user

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

Intelligent Context Caching

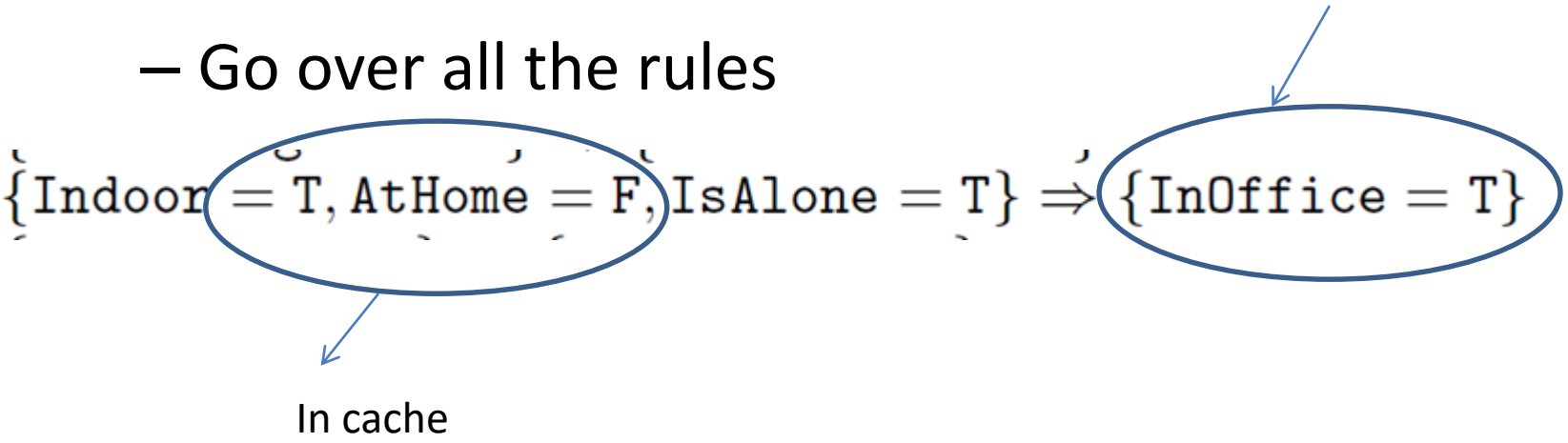
- The Inference Cache in ACE, like traditional cache, provides a Get/Put interface
- Put(X, v) puts the tuple X = v in the cache
  - with a predefined expiration time (default is 5 minutes)
- On a Get(X) request for a context attribute X
  - it returns the value of X
    - not only if it is in the cache
    - 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

Intelligent Context Caching

- Example: App1 requests value of IsDriving ---- Request contexter
- Tuple D=T cached with exp time 5 mints
- App2 requests D---get from cache
- App3 requests for IsIndoor =>short window
- No tuple with I in cache
- {D=T}=>I=F
- Respond I=F

Key challenge

- Efficiently exploit the rules
  - Go over all the rules



- Fails for transitive rules

consider two rules: $\{M = T\} \Rightarrow O = T$ and $\{O = T\} \Rightarrow I = T$ Without using transitivity of the rules, the cache will fail to infer the state of $I = T$ even if $M = T$ is in the cache. Such

Expression tree

- Expression tree for tuple {D=T}
  - Tree representation of boolean expression
  - Implies {D=T}
- Boolean AND-OR tree

(a) Non-leaf represents AND-OR operation with child nodes

(b) Leaf represents a tuple

- Evaluation of AND and OR node
- Expression tree for {D=T} evaluates true iff it satisfies all the context rules

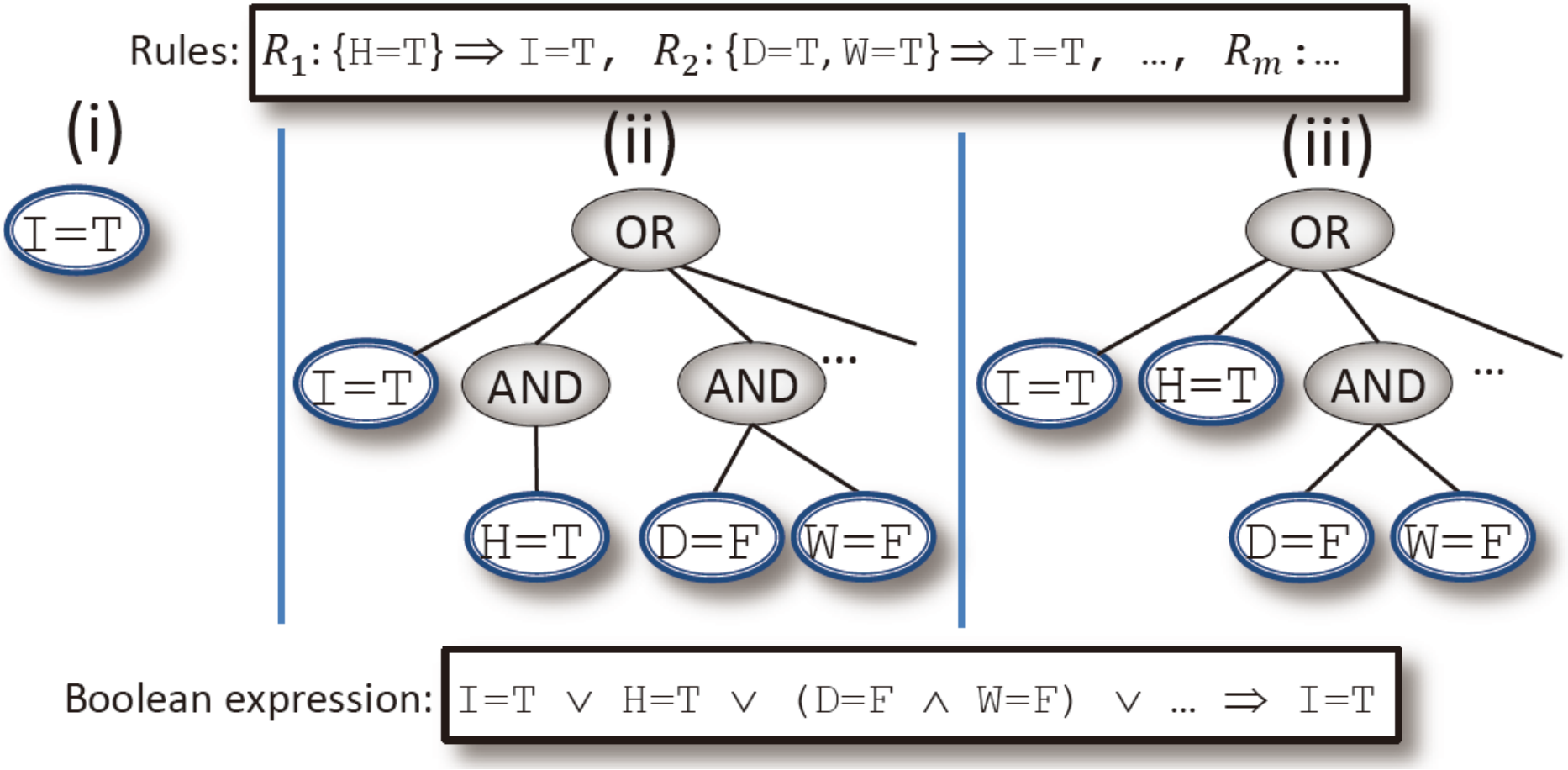
Expression tree

The expression tree for a given tuple $\mathtt{a} = \mathtt{v}$ is constructed as follows. We start from a tree with a single leaf node $\mathtt{a} = \mathtt{v}$ (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 $\mathtt{a} = \mathtt{v}$, we find all the rules $R_1, R_2, \cdots, R_m$ that have $\mathtt{a} = \mathtt{v}$ 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 $\mathtt{a} = \mathtt{v}, X_1, X_2, \cdots, X_m$, 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 $\mathtt{I} = \mathtt{T}$ is expanded using the rules shown at the top of the figure.

Expression tree

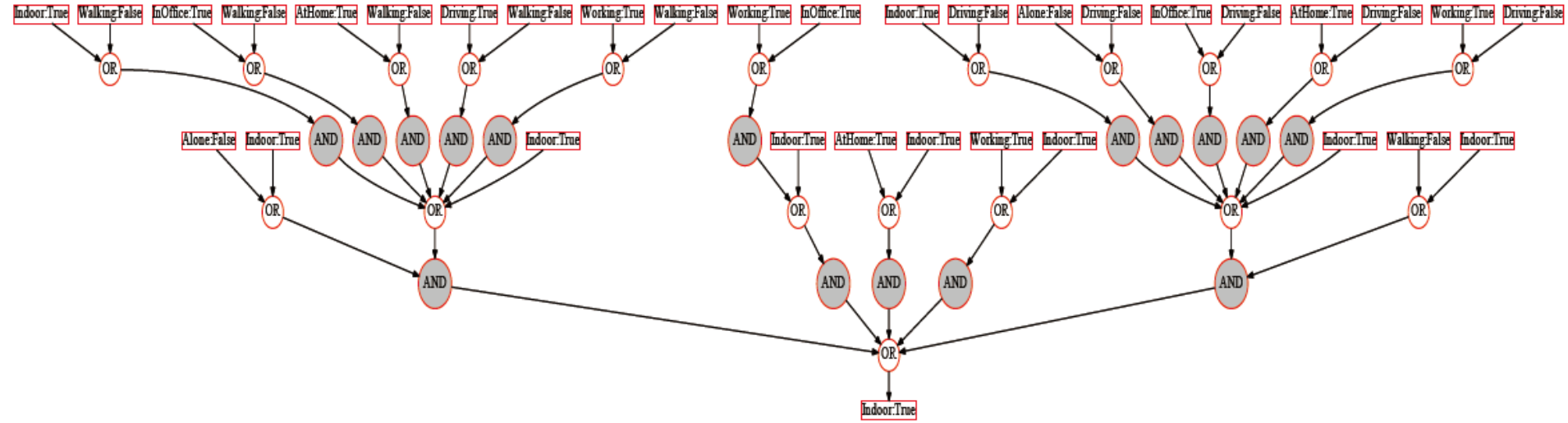
The expression tree for a given tuple $\mathtt{a} = \mathtt{v}$ is constructed as follows. We start from a tree with a single leaf node $\mathtt{a} = \mathtt{v}$ (Figure 5(i)). Then, we continue *expanding* each leaf node in the current tree by using context rules until no leaf nodes can be further expanded or all the rules have been used for expansion. To expand a leaf node $\mathtt{a} = \mathtt{v}$, we find all the rules $R_1, R_2, \cdots, R_m$ that have $\mathtt{a} = \mathtt{v}$ 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 $\mathtt{a} = \mathtt{v}, X_1, X_2, \cdots, X_m$, 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 $\mathtt{I} = \mathtt{T}$ is expanded using the rules shown at the top of the figure.

Expression Trees

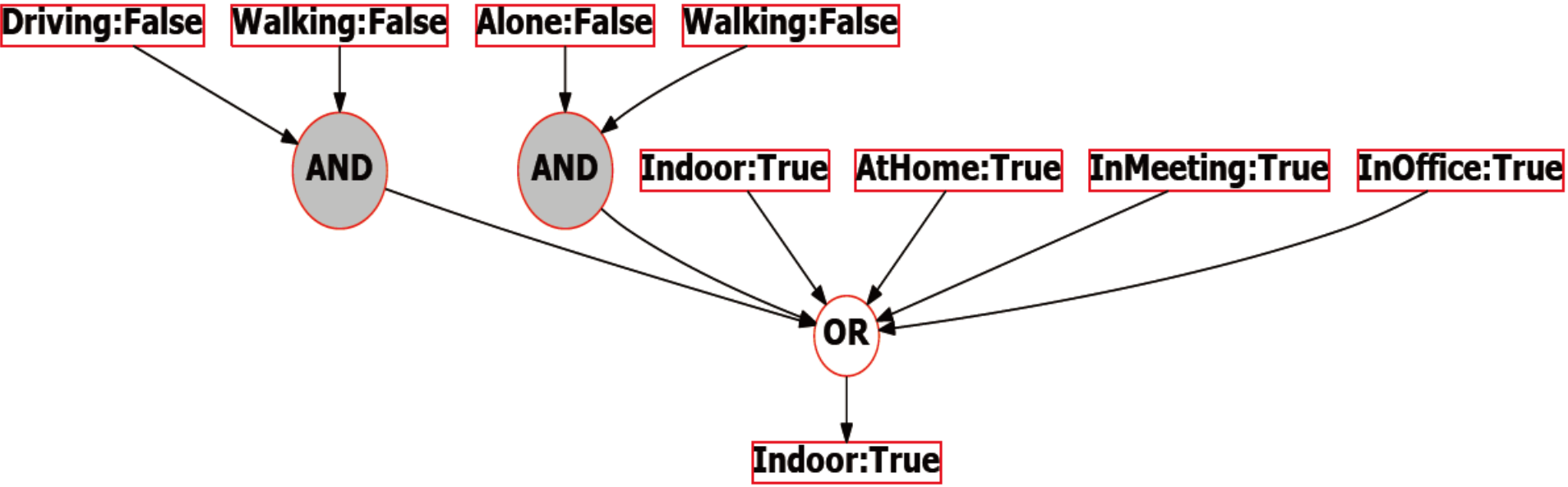


Inference cache (evaluation of tree)

- Maintains one expression tree for each tuple {D=T}, {D=F}
- Get(D) evaluates both the tuple
  - Check satisfiability by tuples in cache
  - Otherwise invoke sensing planner
- How to evaluate?
- Leaf node is satisfied iff=>corresponding tuple in cache
- Non leaf OR is satisfied if any child node is satisfied
- AND is satisfied if all child nodes are satisfied
  - Recursive
- Final=>if root is satisfied

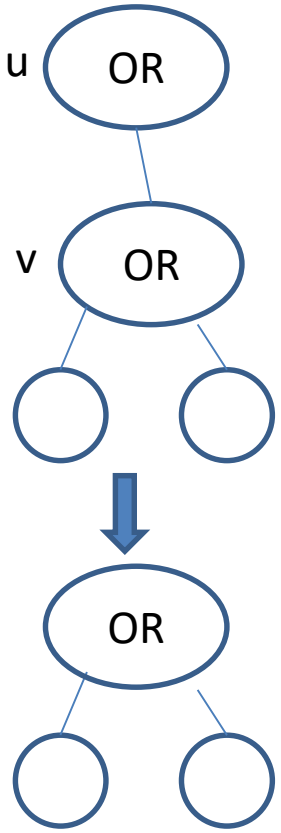


(a) Initial expanded tree (labels can be ignored)



(b) Equivalent reduced tree

Minimal expression tree



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

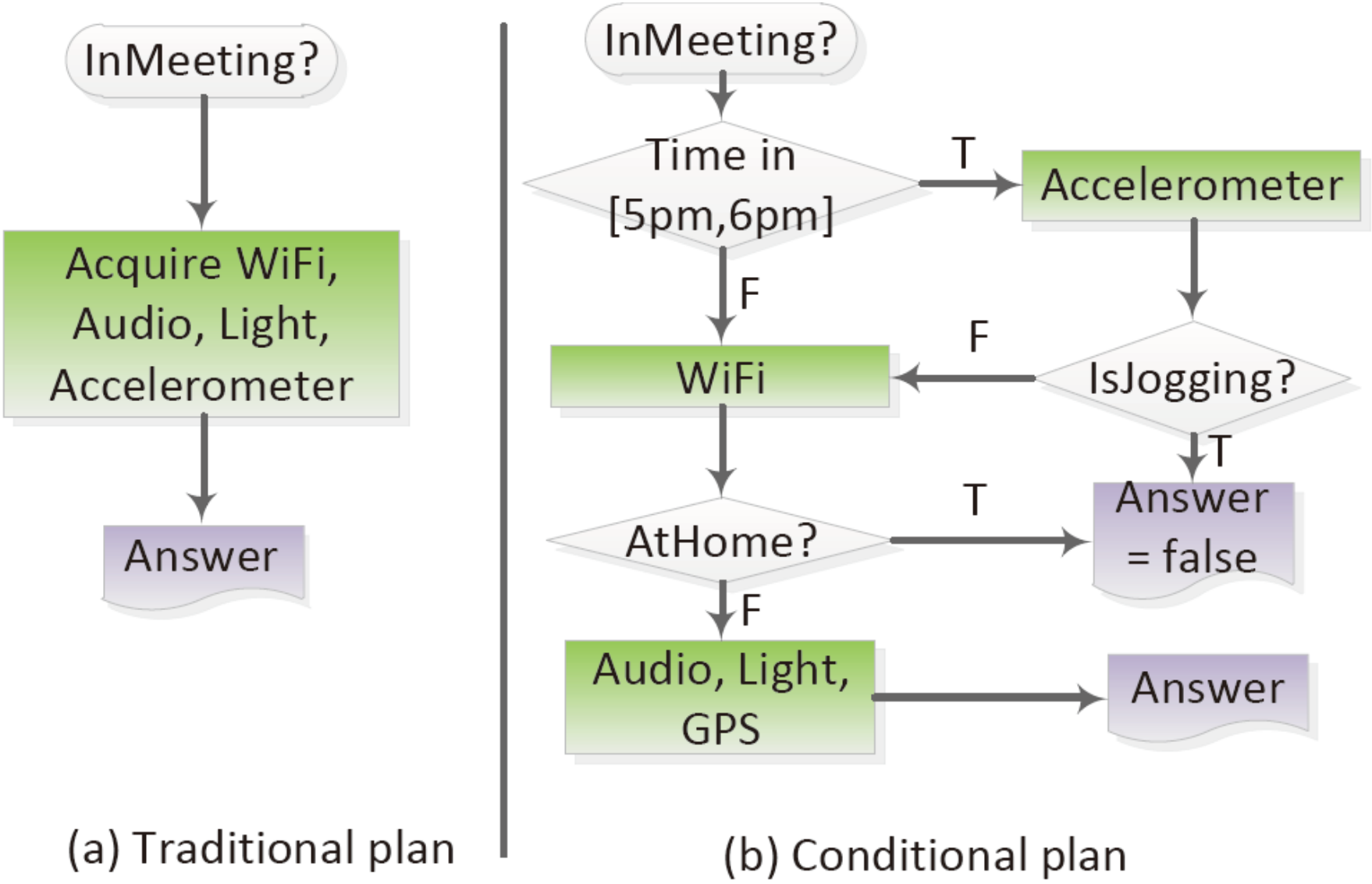
►**Absorption:** If a non-leaf node N has two child nodes A and B such that the set of child nodes of A is a subset of the set of child nodes of B, we can remove B and its subtrees. In a tree with alternating AND-OR levels, if N is an OR node, then A and B will be AND nodes. Suppose $A = a \wedge b$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

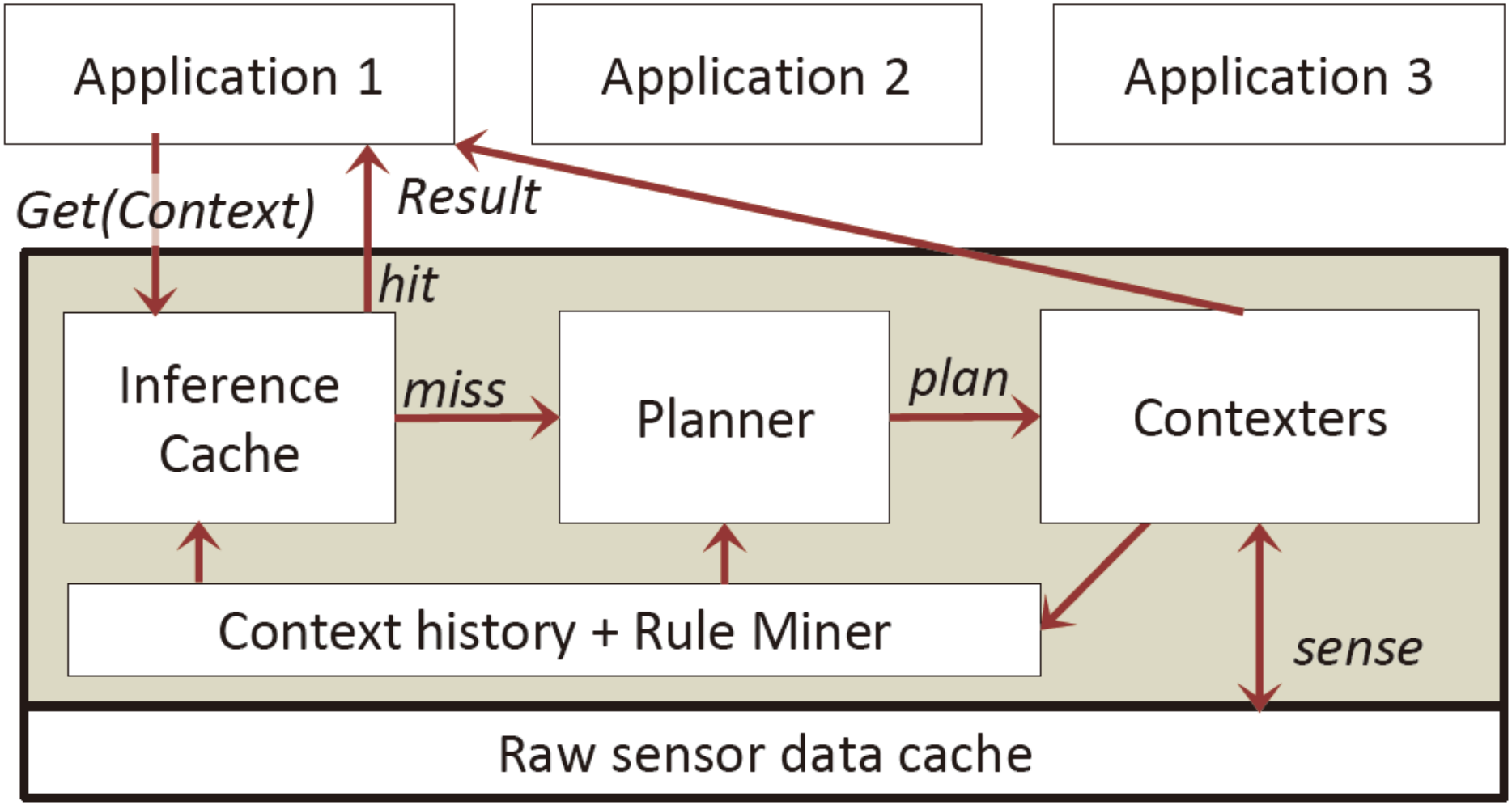
Sensing planner

- Inference cache fails to determine value of X on Get(X)
  - Need to call necessary contexters
- [AtHome] contexter uses GPS, WIFI
- ACE may find X in indirect and cheaper way
  - Speculative sensing

Sensing Planner



ACE Architecture



Sensing plan

- Order in which various attributes needs to be checked
  - Determine the target attribute
- Optimal order may depend on
  - Cost of contexter (c_i)
  - Likelihood the attribute returning True value (p_i)
- Expected sensing cost can be computed using ci (config. of contexter), pi (context history)

Speculative sensing API

- Option 1: Generate the plan graph
  - Traverse only a part
- Option 2: Incrementally enumerate only next node

Init (X): Target attrb X

a <- next(): returns the next attribute OR null

Update(a,v): Update tuple {a=v}

Value <- Result(): returns the value of X

Algorithm 1 Exhaustive search for the 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
```

Contexter returns the value of *attrib*

Update trace

trace = {collection of tuples}
Determines the order of proxy attributes for sensing planer

Invoke the contexter to evaluate *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]
```

For memoization

Try all *unexplored* context attributes s

Get it from inference cache

Get it from inference cache

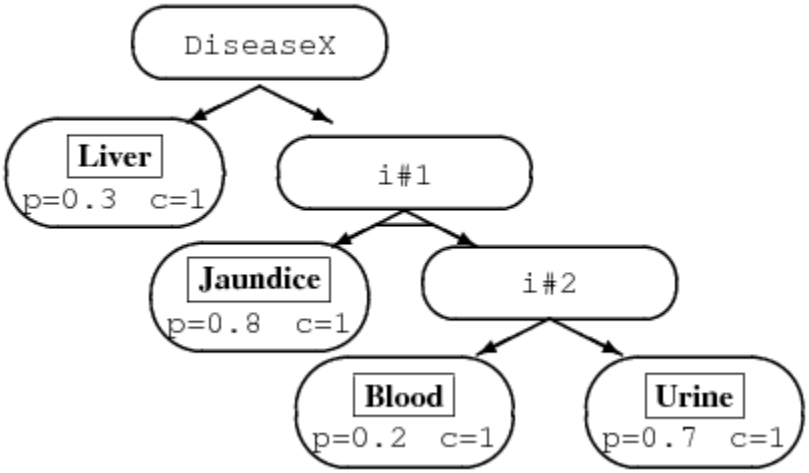
Choose the best proxy attribute

Cache it for memoization

Finding Optimal Satisficing Strategies for And-Or Trees*

Russell Greiner, **Ryan Hayward**, **Magdalena Jankowska**
Dept of Computing Science
University of Alberta
{greiner, hayward}@cs.ualberta.ca
magda@phys.ualberta.ca

Michael Molloy
Dept of Computer Science
University of Toronto
molloy@cs.toronto.edu



Algorithm 2 Generating attribute sensing order

procedure AssignCostAndProb(And-OR Tree Node T)
  if T is a leaf node **then**
    $(\mathcal{C}_T, \mathcal{P}_T) \leftarrow (c_T, p_T)$
    return $(\mathcal{C}_T, \mathcal{P}_T)$
  for all child node N_i of T, $1 \leq i \leq k$ **do**
    $(\mathcal{C}_N, \mathcal{P}_N) \leftarrow$ AssignCostAndProb(N)
    if T is an AND node **then**
      $\mathcal{P}_N \leftarrow 1 - \mathcal{P}_N$
  Sort the child nodes of T in decreasing order of their values of $\mathcal{P}_{N_i}/\mathcal{C}_{N_i}, 1 \leq i \leq k$
  $\mathcal{C}_T \leftarrow \mathcal{C}_{N_1} + \sum_{i=2}^{k} \mathcal{C}_{N_i} \prod_{j=1}^{i-1} (1 - \mathcal{P}_{N_j})$
  if T is an AND node **then**
    $\mathcal{P}_T \leftarrow \prod_{i=1}^{k} \mathcal{P}_{N_i}$
  else
    $\mathcal{P}_T \leftarrow 1 - \prod_{i=1}^{k} (1 - \mathcal{P}_{N_i})$
  return $(\mathcal{C}_T, \mathcal{P}_T)$

procedure FindOrdering(And-Or Tree Node T)
  if T is a leaf node **then**
    $Q.Enqueue(T)$
  for all child N_i of T in ascending order of $\mathcal{P}_{N_i}/\mathcal{C}_{N_i}$ **do**
    FindOrdering(N_i)

Evaluation

- Effectiveness of Inference Cache
- Accuracy
- Effectiveness of Sensing Planner
- Overhead of ACE
- End-to-end Energy Savings

Applications

- GeoReminder
  - Monitors user's current location and displays message
- JogTracker
  - Monitors user's transportation mode and keeps track of calories burn
- PhoneBuddy
  - Mute the phone while in a meeting

Datasets

- Reality Mining Dataset
  - collected part of the Reality Mining project at MIT
  - contains continuous data on daily activities of 100 students and staff at MIT
  - recorded by Nokia 6600 smartphones over the 2004-2005 academic year
  - Trace contains sequence of lines

  Time, attrib1=value1, attrib2=value2

Dataset

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 about 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`).

Datasets

- ACE Dataset
  - collected with continuous data collection software running on Android phones
  - The subjects in the dataset, 9 male and 1 female, worked at Microsoft Research Redmond

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 come from a combination of three sources: (1) labels manually entered by a user during data collection time (e.g., when the user starts driving, he manually inputs `Driving = True` through a GUI), (2) labels manually provided by the user during post processing, and (3) outputs of various contexters.

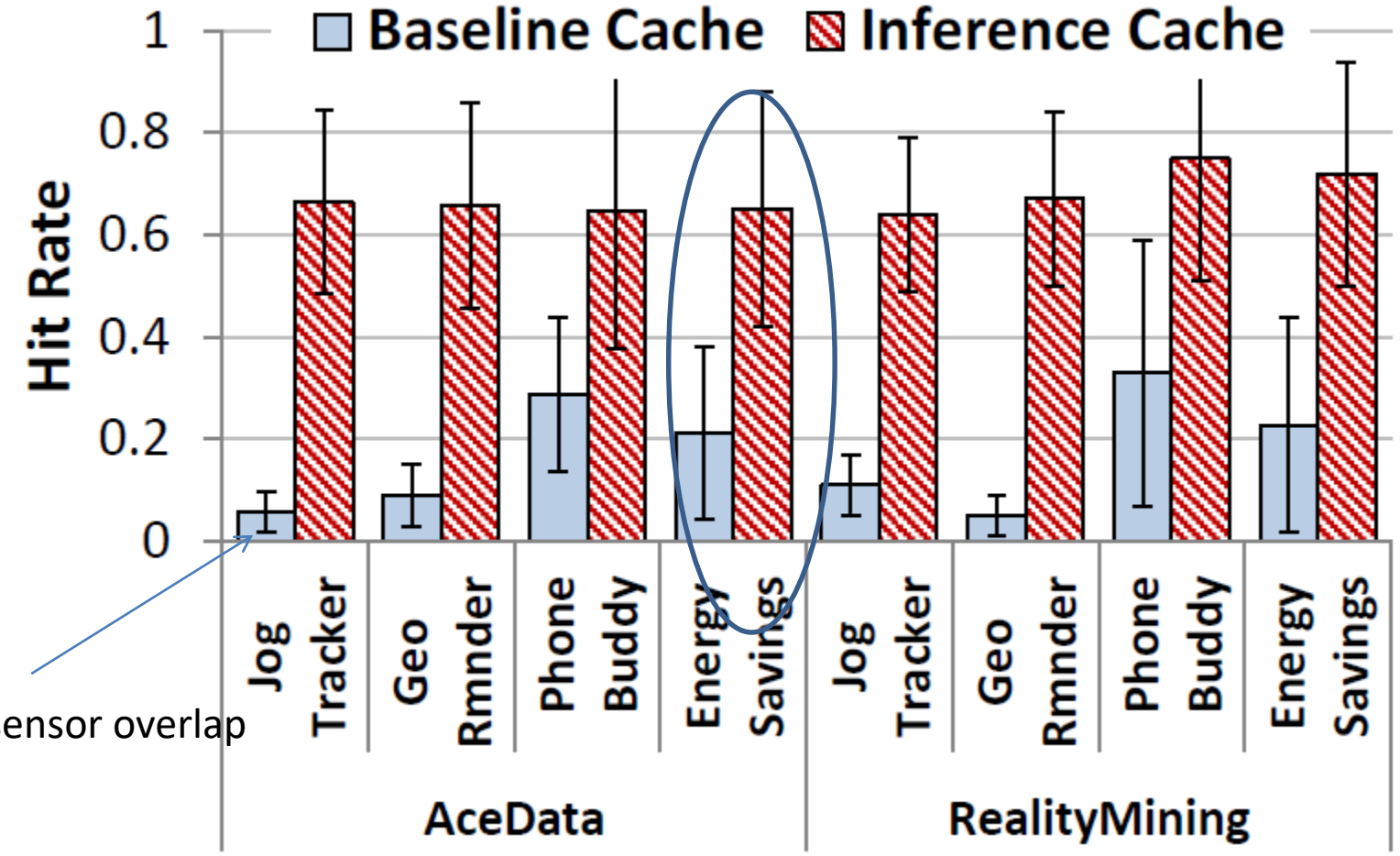
Evaluation

- Port ACE prototype on a laptop
- Replace contexters with the fake one
  - Return's current context from trace
- Energy consumption is modelled from real contexter.
- Run applications at each tick (5 mints)
- Expiration time – 1 tick

Evaluation

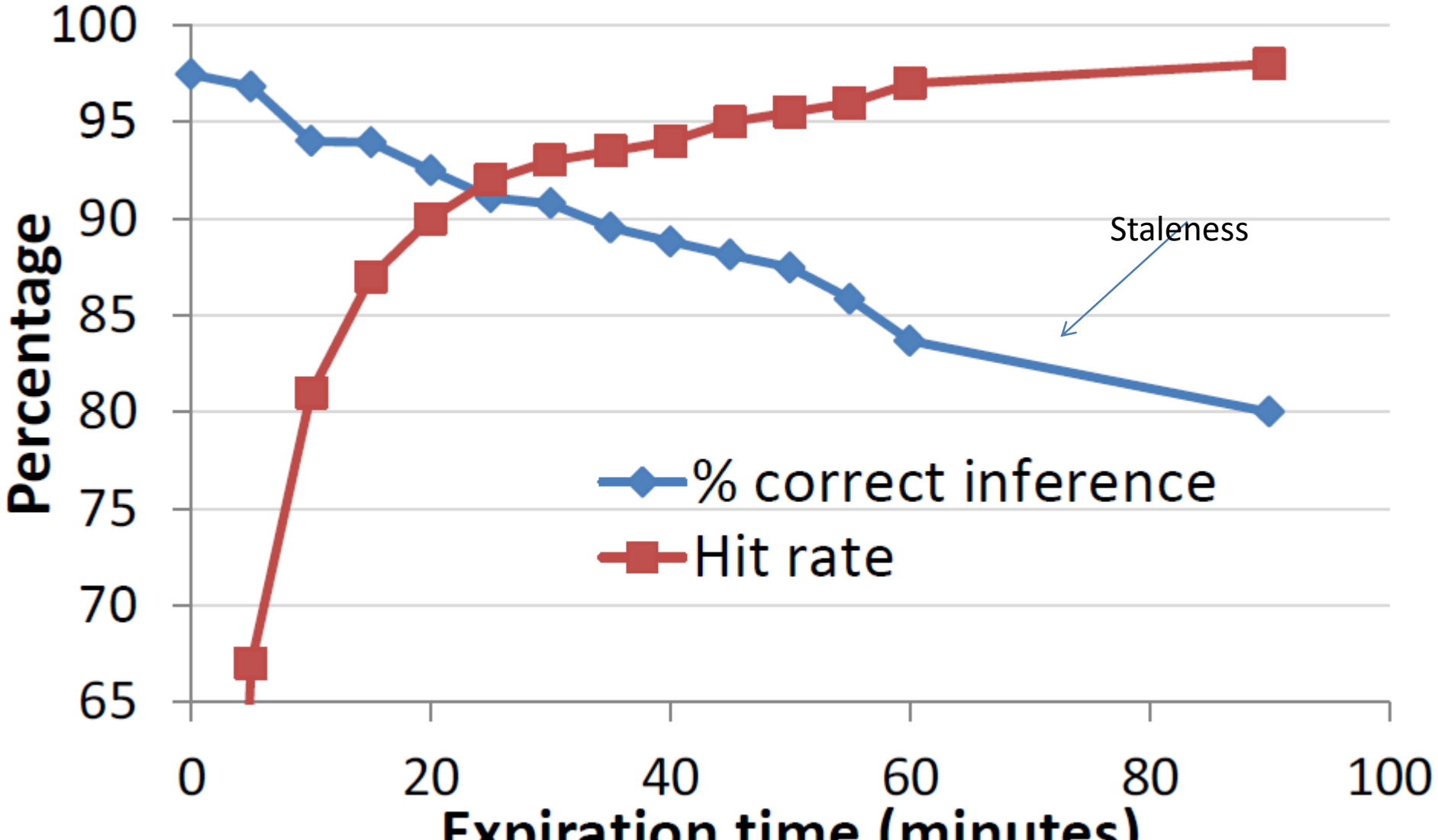
- Effectiveness of Inference Cache
- Accuracy
- Effectiveness of Sensing Planner
- Overhead of ACE
- End-to-end Energy Savings

Hit rates and energy savings of Inference Cache

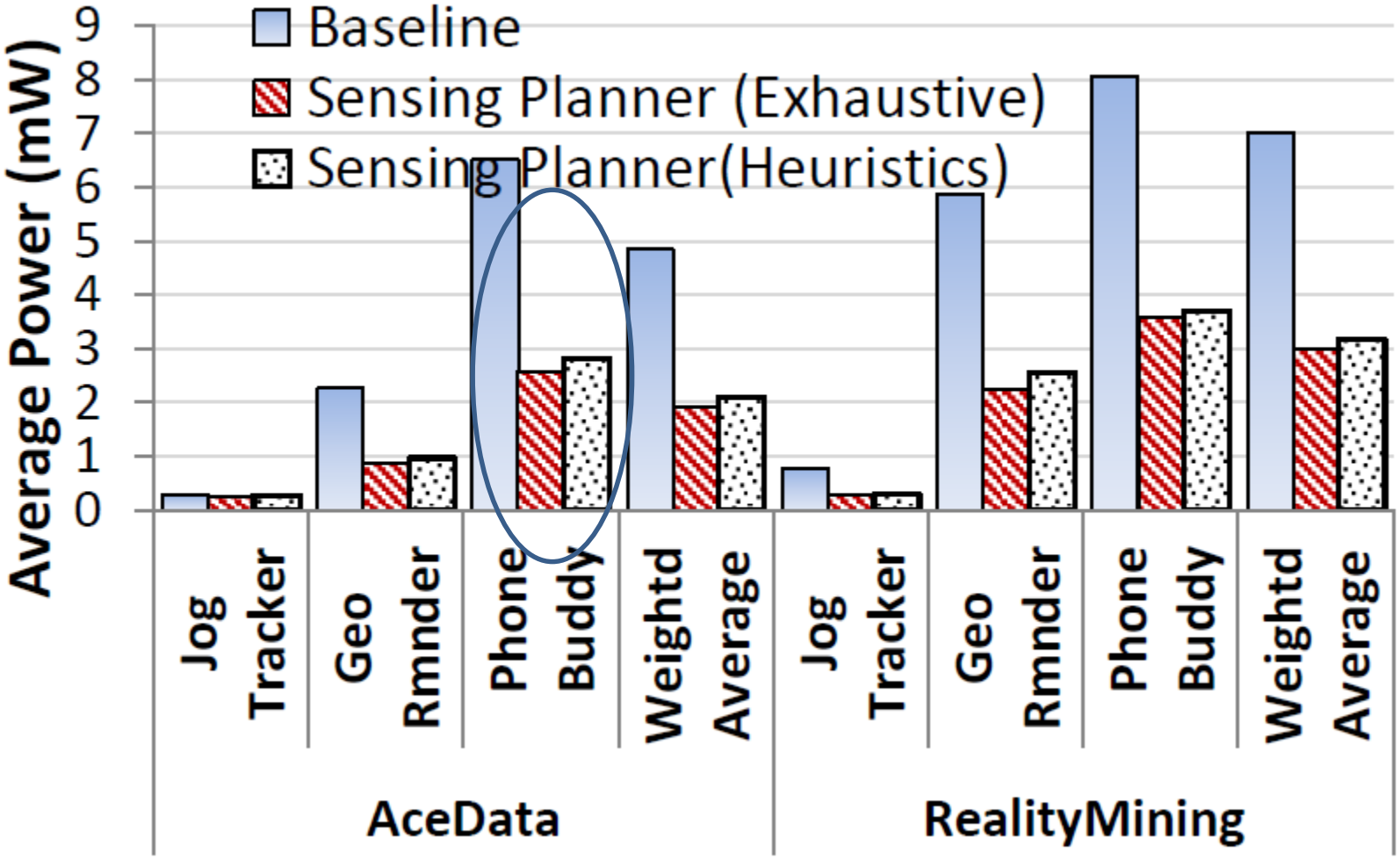


Due to sensor overlap

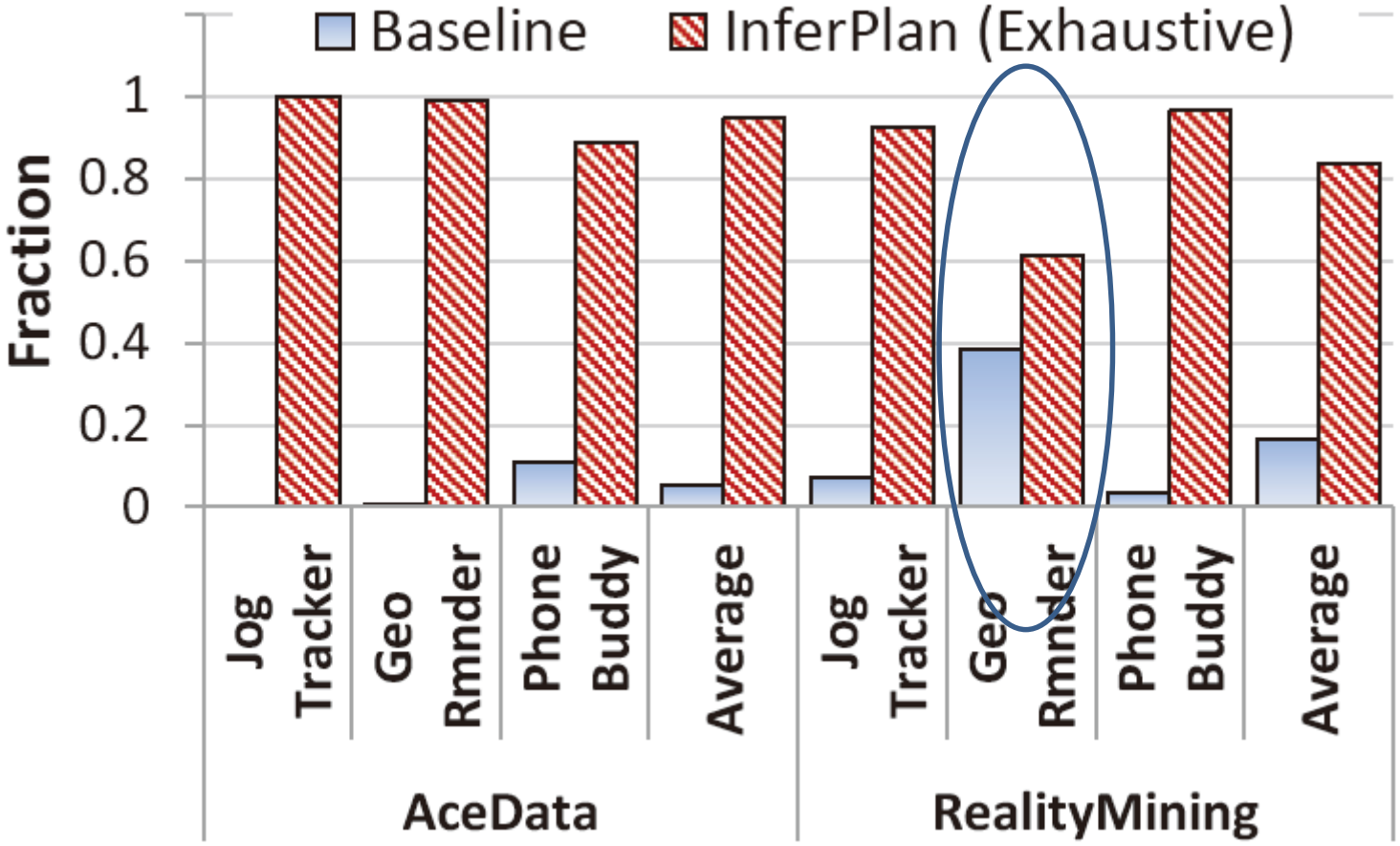
Effect of cache expiration time on hit rate and accuracy



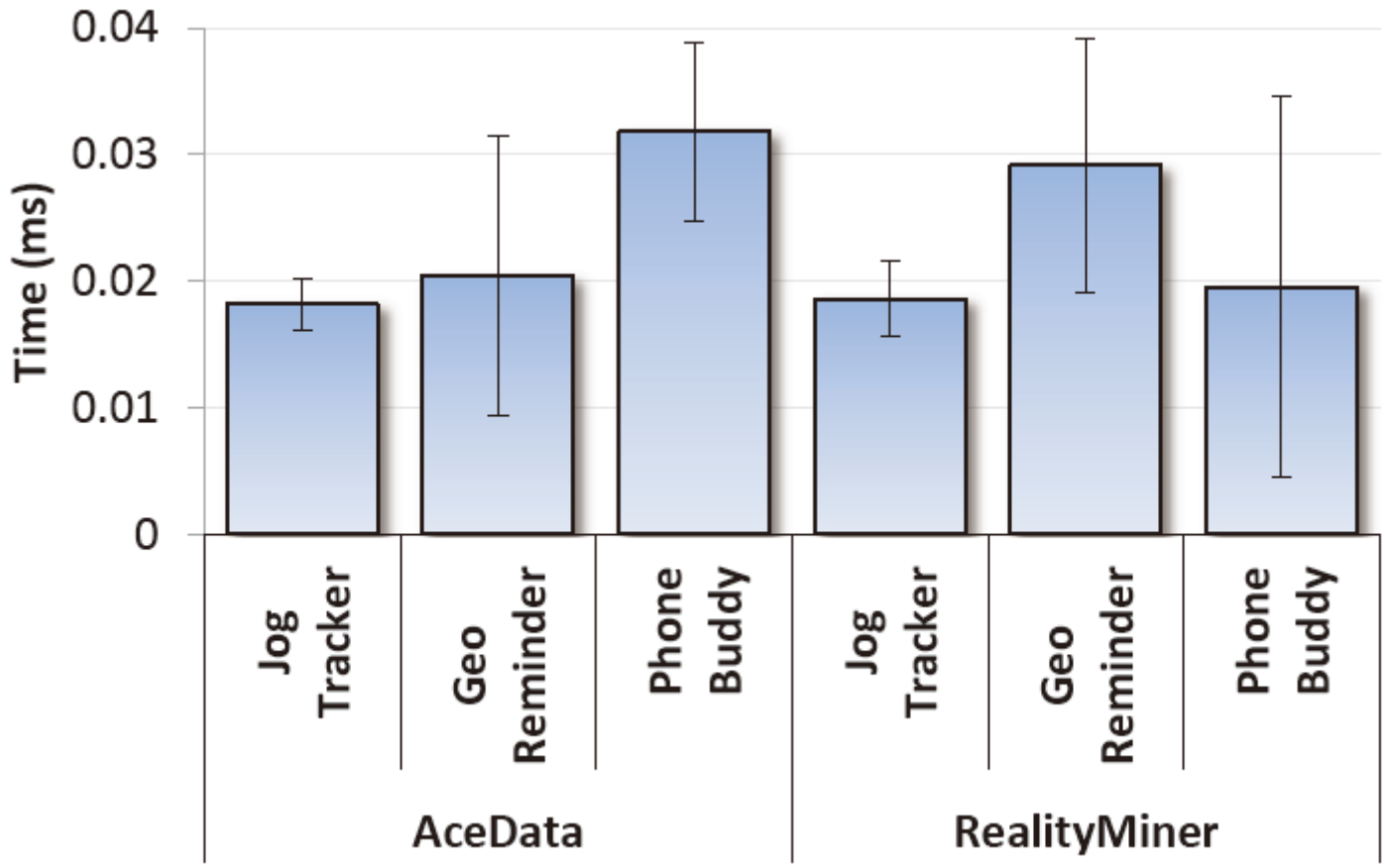
Energy savings by Sensing Planner



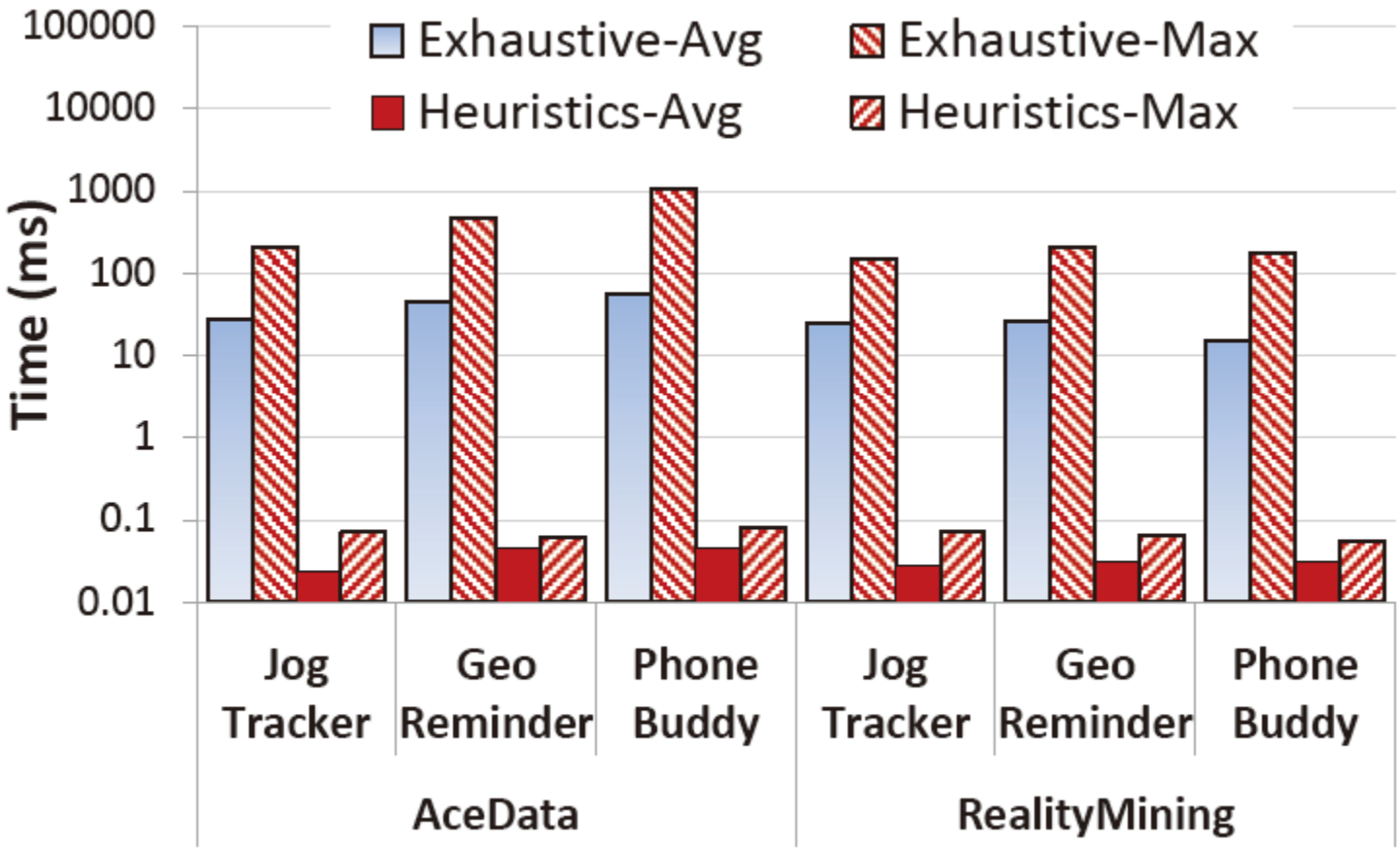
Fraction of times Sensing Planner does better



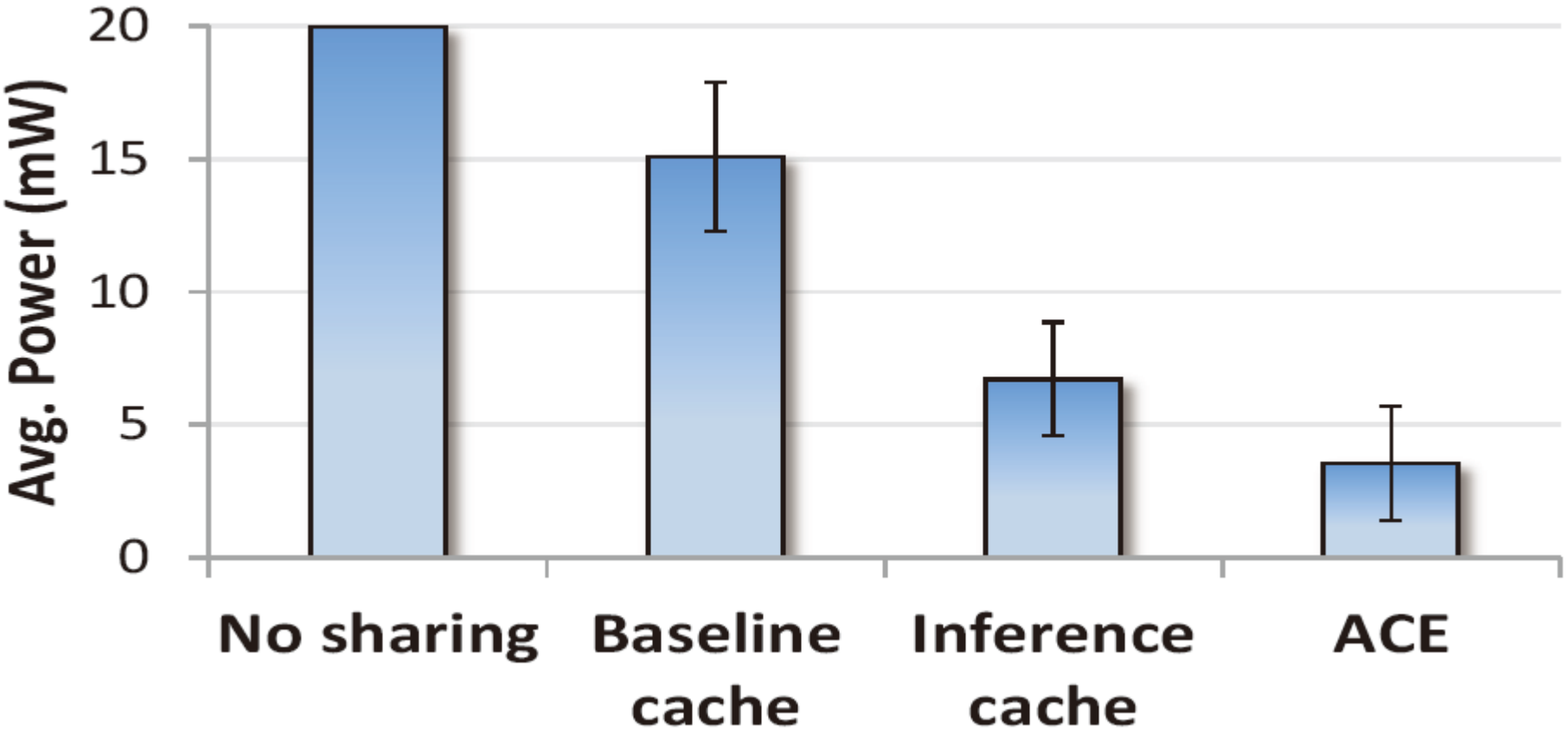
Latency of a Get() request on Inference Cache hit



Time required to generate a plan on a Samsung Focus phone



End-to-end energy savings



Per user power consumption (users sorted by ACE power)