Enabling Energy-Efficient Context Recognition with Configuration Folding

Muhammad Umer Iqbal, Marcus Handte, Stephan Wagner, Wolfgang Apolinarski, Pedro José Marrón
Networked Embedded Systems
Universitaet Duisburg-Essen
Duisburg, Germany
{*umer.iqbal*|*marcus.handte*|*stephan.j.wagner*|*wolfgang.apolinarski*|*pjmarron*}*@uni-due.de*

***Abstract*—Existing context recognition applications for personal mobile devices are usually fine-tuned to recognize the set of characteristics required to support a particular user task. Outside of a laboratory environment, however, users are often involved in multiple tasks at a time which requires the simultaneous execution of several applications. Yet, due to the energy constraints of most personal mobile devices this approach is inherently limited in scale. In this paper, we show how this problem can be avoided by applying a component-based approach to application development and execution. We present a component system that enables developers to build individual applications in isolation without compromising runtime efficiency. When applications are executed simultaneously, the component system applies a novel technique called configuration folding to automatically remove redundancies. Our experimental evaluation of configuration folding shows that it can save up to 48 percent of energy when applied to a music and a speech detection application, thus, amortizing energy costs after a few seconds of execution.**

***Keywords*-Context recognition, energy efficiency, component system**

I. Introduction

Personal mobile devices such as smart phones, tablets and laptops have become major platforms for context recognition applications. They have lead researchers and practitioners to develop a number of recognition applications for different domains such as physical activity recognition [1], traffic monitoring [2], social networking [3], healthcare [4], etc. Such applications are usually fine-tuned to recognize the set of context characteristics required for a particular user task. A road monitoring application such as [2], for example, may be fine-tuned to support the recognition of potholes or honking cars to detect a crowded intersection. A social networking application, on the other hand, such as [3] may be able to classify music to generate status updates.

Outside of a laboratory environment, however, users are often involved in multiple tasks at a time [5] which requires the simultaneous execution of several applications. As a simple example consider that when driving around, a user might want to use both, the travel time as well as the road monitoring application, while using the social networking application to update the status with the music playing on the radio. As a result, the applications have to continuously sample and process acceleration data for road monitoring and speed estimation as well as audio data for music classification and honk detection. Yet, since running multiple applications simultaneously increases the energy requirements additively and due to the fact that mobile devices are usually battery-powered, this approach is inherently limited in scale.

In this paper, we show how this problem can often be avoided when relying on a component-based approach to develop context recognition applications. Thereby, we make the following contributions. First, we present a component system that enables developers to build recognition applications in isolation. Second, we show how the resulting component structure can be exploited at runtime to reduce energy requirements when executing several recognition applications simultaneously. To do this, we introduce a novel technique called configuration folding that detects and removes redundancies. Third, we evaluate the overall approach using two music and speech detection applications that we have implemented based on the description given in [6]. Our experimental evaluation shows that configuration folding can save up to 48 percent of energy when applied to the music and speech detection applications. Thereby, the technique introduces energy costs that amortize after a few seconds of executing the applications.

Our work builds upon existing applications, e.g., [3], [2], in that it aims at minimizing their energy requirements. However, instead of focusing on one concrete application, configuration folding targets the simultaneous execution of multiple. Consequently, it is related to frameworks such as [7] or [8]. Yet, instead of focusing on specific types of context information that are known at system design time, configuration folding is open and generic in the sense that it can be extended by application developers.

The remainder is structured as follows. In Section II, we discuss the basic design rationale. In Section III, we describe our component system. In Section IV we introduce configuration folding to reduce the energy requirements when running multiple context recognition applications. In Section V, we evaluate the approach. We describe related work in Section VI and conclude the paper in Section VII.

II. Design Rationale

When executing multiple recognition applications simultaneously, additive increases in energy requirements can

often be avoided. The reason for this is twofold. First, applications often use overlapping sets of sensors which operate on low duty cycles. Second, the signal processing methods applied by them are often similar. Consequently, it is possible to reduce the energy requirements by reusing samples and avoiding duplicate computations. To do this, it is necessary to identify identical samples and (intermediate) results and to replace them with a single one.

Clearly, it is possible to exploit the overlapping sets of sensors and duplicate computations by manually integrating a targeted set of applications. However, such a manual integration of context recognition applications would require a significant development effort that increases fast with the number of used recognition applications and thus, it should be automated. As a first step towards such an automation, it is necessary to detect duplicate sampling and processing code which requires an analysis of the applications that are supposed to be executed simultaneously. Intuitively, the complexity of such an analysis depends on the degree of structure applied to the application. If an arbitrarily structured application shall be analyzed, the overall problem is at least as complex as the detection of code clones [9]. Due to the associated computational effort, such an analysis can only be done offline on a powerful device. Consequently, to support the online analysis on a personal mobile device, it is necessary to further structure the applications.

For this, we chose a component-based approach which requires application developers to structure their applications by composing a set of components. The reason for this is threefold. First, there already exist rapid prototyping toolkits such as [10] that validate the suitability of component abstractions to develop context recognition applications. Second, given a repository of standard components, a component-based approach can speed up application development through reuse which also ensures the effectiveness of duplicate removal[1]. Last but not least, given a suitable connector model, the components that constitute an application can easily be manipulated at runtime.

To avoid restrictions, we apply the component structure only to the parts of the application that deal with the actual recognition. Other parts such as user interfaces, networking logic, etc. can be structured arbitrarily. Consequently, our approach introduces a clear separation between the recognition of context characteristics and their further use.

III. Component System

The fundamental building blocks of the resulting component system are depicted in Figure 1. Each context recognition application consists of two parts, namely the part containing the recognition logic and the part containing the remaining application logic. The part that contains the recognition logic consists of a set of components that is composed according to a configuration. The part that contains the remaining application logic can be structured arbitrarily. Upon start up, a context recognition application passes the required configuration to the component manager which then takes care of executing the recognition logic in an energy-efficient manner. Upon shut down, the context recognition application removes its configuration from the component manager which will eventually release the components that are no longer required. As long as the recognition logic is running, the recognized context is signalled.

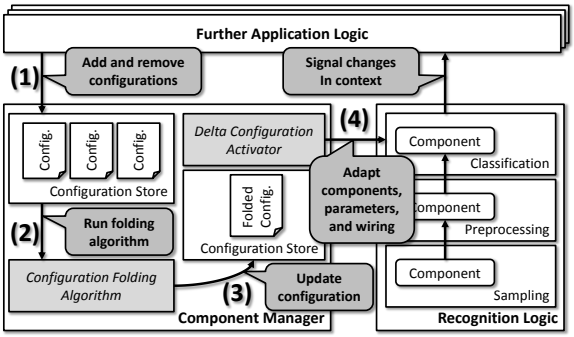

Figure 1. Component System

A. Component Model

To structure the recognition logic, our component system realises a lightweight component model which introduces three abstractions.

Components represent recognition logic at a developer-defined level of granularity, similar to other systems such as J2EE or OSGi. Yet, in contrast, they can be instantiated multiple times and they are parametrizable to support different application requirements. As indicated in Figure 2, at the sampling layer, these parameters might be used, for example, to express different sampling rates, depths, frame sizes or duty cycles. At the preprocessing layer, they might be used to configure different filters or the precision of a transformation. Due to the parameters, the component model is more flexible than other models which simplifies reuse but, at the same time, it also prevents a straightforward multiplexing of components. Besides parameters, all components exhibit a simple life cycle that consists of a started and a stopped state. To interact with other components, a component may declare a set of typed input and output ports that can be connected using connectors.

Connectors are used to represent both, the data- as well as the control-flow between individual components. To avoid frequent instantiations of events that are passed between ports, connectors automate their management. From a high-level perspective, the resulting model is similar to PCOM [11]. Yet since PCOM is targeted at the automatic configuration of distributed applications, it uses stub objects to

[1]Open source projects such as Eclipse or TinyOS demonstrate that this approach can lead to a number of (de-facto) standard components.

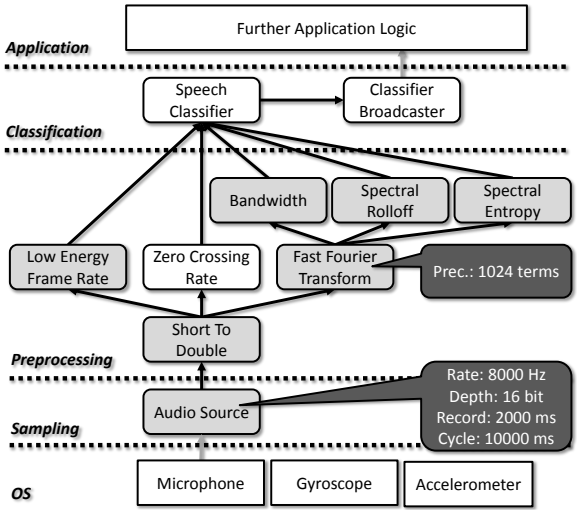

Figure 2. Componentized Speech Recognition Logic

mediate component interaction and it performs search to determine a suitable configurations. To avoid this overhead, we realize connectors through a lightweight observer pattern [12] and we introduce configurations to define a composition that recognizes one or more context characteristics.

Within its *configuration*, each recognition application explicitly lists all required components together with the associated wiring and parametrization. Since ports are used to model both, the data as well as the control flow, a wiring always represents a directed acyclic graph (DAG) in order to guarantee termination. To define configurations, we introduce a simple XML format. Although, XML is a rather verbose format, it is widely used and can be manipulated easily. However, during load time the XML is transferred into a more efficient graph representation so that the overhead for parsing and validation occurs only once.

B. Component Manager

The component manager controls the execution of the componentized recognition logic of all running applications. To manipulate the components executed at any point in time, the component manager provides an API that enables developers to add and remove configurations at runtime. When a new configuration is added (Figure 1 (1)), the component manager first stores the configuration internally. Thereafter, it initiates a reconfiguration of the running recognition logic that reflects the modified set of required configurations. To reduce the energy requirements, however, the component manager does not directly start the components contained in the configuration. Instead, it uses the set of active configurations as an input into our configuration folding algorithm (Figure 1 (2)). Using the set of configurations, the configuration folding algorithm computes a single, folded configuration that produces all results required by all running applications without duplicate sampling or computation.

Once the configuration has been folded, the component manager forwards it to the delta configuration activator (Figure 1 (3)). By comparing the running and the folded configuration, the activator determines and executes the set of life cycle and connection management operations (starting, stopping and rewiring of components) that must be applied to the running configuration in order to transform it into the folded target configuration. When executing the different operations (Figure 1 (4)), the delta activator takes care of ensuring that their ordering adheres to the guarantees provided by the component life cycle. To do this, it stops existing components before their are manipulated.

Since configuration folding cannot be reversed easily, the component manager performs a similar procedure once a configuration shall be removed. First, it removes the configuration from the internal storage (Figure 1 (1)), then it performs configuration folding (Figure 1 (2), (3)) and finally, it activates it (Figure 1 (4)). Although, we could avoid this procedure in some cases by caching several previously folded configurations, we decided to use this simple procedure instead. The reason for this is that caching is only useful in cases where the removals are made in the inverse order of the additions.

C. Implementation

To validate the proposed abstractions, we have implemented them in Java. In addition, we implemented a number of wrappers that simplify the usage of the component system on different platforms including Linux, Windows and Android. On top of the system, we have developed a variety of components which we used as a toolkit to build several applications. This includes sensor components (e.g. for accelerometers, microphones, GPS, wifi and Bluetooth scanning), preprocessors (e.g. average, percentile, variance, entropy, etc.), filters (e.g. finite response filters), transformations (e.g. FFT) among others.

IV. Configuration Folding

To execute multiple applications efficiently, the component manager applies configuration folding. The goal is to remove components that perform redundant sampling or compute redundant results. In the following, we first discuss how such components can be identified in general. Thereafter, we present an algorithm that can fold two configurations that exhibit only identically parametrized components in an optimal manner. Finally, we discuss how we extended the algorithm to fold different parametrizations.

A. Problem Formalization

Informally, we can define redundancy as follows. Given two configurations, a set of components is redundant if this set is contained in both configurations in such a way that all identical components exhibit the same connections and parametrizations and that each set does not depend

on the inputs of components that are not part of the set. Based on this definition, the outputs generated by both sets of components cannot differ. To justify this consider that components at the sampling level do not require inputs from other components. Consequently, if two configurations contain the same sampling component with an identical parametrization, it is possible to remove one of the two components by reconnecting the components that require its output to the other component. If the components require inputs, however, it is also necessary to ensure that the inputs are identical to ensure that their outputs cannot differ.

To model the configuration details, we introduce a function $F(vertex) \rightarrow (impl, params)$ that maps each vertex to its associated implementation and parametrization. Given two configurations $G_1 = \{V_1, E_1\}$ with F_1 and $G_2 = \{V_2, E_2\}$ with F_2, where V and E represent set of vertices (components & parameters) and set of edges (wirings) in directed acyclic graphs, we can define redundancy as a subset $S \subseteq V_1$ and an associated mapping $R : s \rightarrow v$ with $s \in S, v \in V_2$ subject to the following conditions: First, we ensure that the redundant wiring is identical, i.e. has the same edges.

$$\forall s_1, s_2 \in S : (s_1, s_2) \in E_1 \leftrightarrow (R(s_1), R(s_2)) \in E_2 \quad (1)$$

Second, we ensure that redundant vertices represent the same components with the same parametrization.

$$\forall s \in S : F_1(s) = F_2(R(s)) \quad (2)$$

Finally, we ensure that neither components in V_1 nor components in V_2 require non-redundant inputs.

$$\forall s \in S, v \in V_1 : (v, s) \in E_1 \rightarrow v \in S \quad (3)$$

$$\forall s \in S, v \in V_2 : (v, R(s)) \in E_2 \rightarrow R^{-1}(v) \in S \quad (4)$$

Given the condition 1 alone, the problem would correspond to finding a homomorphism in a sub graph of G1 and G2 which is known to be NP complete. However, due to equations (2), (3) and (4) as well as the DAG structure of configurations, it is possible to find an efficient solution.

B. Basic Algorithm

Our configuration folding algorithm is based on the idea that two components can only be redundant if they are on the same level in the topological ordering of the two graphs. If they are not on the same level, the component at the higher level will be connected to at least one component that was not present in the other configuration. Consequently, in each step of the algorithm, we can restrict comparisons between the graphs to the set of vertices at a particular level. By traversing the graph in topological order, we can then constructively ensure the conditions 1 and 2 by matching the component descriptions and we can ensure condition 3 and 4 recursively by ensuring that all incoming edges are originating in a redundant component. To do this efficiently, we incrementally replace the incoming edges in the input graphs with labels to their parent vertices in the folded graph.

```
1  Define Graph As          // configuration
2   Vertices: Set<Vertex> // all components
3  Define Vertex As         // component & wiring
4   Incoming: Set<Vertex> // connected inputs
5   Outgoing: Set<Vertex> // connected outputs
6   Labels  : Set<Vertex> // labels of inputs
7   Comp    : Component   // component impl.
8
9  FoldGraphs(Graph G1, Graph G2): Graph
10  Graph Res = New Graph
11  // start on first level
12  Set<Vertex> Cur1 = GetRoots(G1)
13  Set<Vertex> Cur2 = GetRoots(G2)
14  While (! (Cur1.IsEmpty() & Cur2.IsEmpty()))
15   Set<Vertex> Next1 = New Set
16   Set<Vertex> Next2 = New Set
17   // create hash map for constant lookup
18   Map<Component, Vertex> M = New Map
19   Foreach (Vertex N2 In Cur2)
20    M.Put(N2.Comp, N2)
21   // handle vertices that can be folded
22   Foreach (Vertex N1 In Cur1)
23    Node N2 = M.Get(N1.Comp)
24    If (N2 != NULL && AllowsFolding(N1,N2))
25     Vertex N = AddVertex(Res, N1.Comp)
26     Visit(N, N1, Next1)
27     Visit(N, N2, Next2)
28     Cur1.Remove(N1)
29     Cur2.Remove(N2)
30   // handle vertices that cannot be folded
31   Foreach (Vertex N1 In Cur1)
32    Vertex N = AddVertex(Res, N1.Comp)
33    Visit(N, N1, Next1)
34   Foreach (Vertex N2 In Cur2)
35    Vertex N = AddVertex(Res, N2.Comp)
36    Visit(N, N2, Next2)
37   // continue with next level
38   Cur1 = Next1
39   Cur2 = Next2
40  Return Res
41
42 Visit(Vertex Nu, Vertex Ol, Set<Vertex> S)
43  // connect according to configuration
44  Foreach (Vertex N In Ol.Labels)
45   Nu.Incoming.Add(N)
46   N.Outgoing.Add(Nu)
47  // label edges and compute next level
48  Foreach (Vertex N In Ol.Outgoing)
49   N.Incoming.Remove(Ol) // mark parent done
50   N.Labels.Add(Nu)    // memorize connection
51   If (N.Incoming.isEmpty())
52    S.Add(N)    // no inc. -> parents done
53
54 AllowsFolding(Vertex N1, Vertex N2): Boolean
55  Return N1.Comp.Equals(N2.Comp) &
56     N1.Labels.Equals(N2.Labels)
57
58 AddVertex(Graph G, Component C): Vertex
59  Vertex Res = New Vertex[Comp=C]
60  G.Vertices.Add(Res)
61  Return Res
62
```

```
63 GetRoots(Graph G): Set<Vertex>
64   Set<Vertex> Res = new Set
65   Foreach (Vertex N In G.Vertices)
66     If (N.Incoming.IsEmpty()) Res.Add(N)
67   Return Res
```

Listing 1. Configuration Folding Algorithm

As depicted in Listing 1, the folding algorithm, i.e., *FoldGraphs*, starts by determining the lowest level in each graph using *GetRoots*. Then it checks whether two vertices on that level are redundant. To allow constant time comparisons, the vertices of one of the graphs is hashed by components (for simplicity, we omit collision handling). When two vertices represent identical components, *AllowsFolding* additionally checks whether their incoming edges have been labeled identically, i.e. whether they originate in the same set of components – which enforces condition 2. On the lowest level of the graph, this will always be the case since the roots have no inputs. For consecutive levels, this will only be the case for redundant components – which enforces the conditions 3 and 4 recursively. If redundant components have been identified, a new vertex is created using *AddVertex*. Then, the new vertex is connected according to the configuration using *Visit*. Thereby, the visit method will add labels to all vertices that are connected to the vertex that is about to be folded. Furthermore, to speed up the traversal, *Visit* also determines the vertices on the next level of the graph. This is done by removing incoming edges in the outgoing vertices and testing for emptiness. Note that this simply reflects topological traversal as described in [13]. Once all possible vertices have been folded, the remaining vertices are handled by creating new vertices and adjusting the connections while computing the vertices on the next level in *Visit*. Thereafter, the algorithm switches to the next level in both graphs, which has been gathered already. After handling all levels, the algorithm returns the folded graph.

To analyze the algorithm's complexity, one must consider that the algorithm simply traverses both graphs in topological order. Thereby, each vertex and each edge is visited exactly once. Due to the fact that the comparisons between the vertices on each level are performed in constant time using hashing, the overall complexity of topological sorting is not increased. Consequently, the complexity of the algorithm is $O(n_1 + m_1 + n_2 + m_2)$ where n_i and m_i represent the number of vertices and edges in the graphs which results in $O(n_1^2+n_2^2)$ since there are at most n_i^2 edges.

C. Extended Algorithm

While the above algorithm can be applied to any configuration, in principle, the fact that it only folds components with identical parametrization can be suboptimal, in practice. The reason for this is that parameters are often used to configure sensor resolutions or precisions during computations. Consequently, it is possible to define lightweight transformations that can be used to derive the result computed with a particular parametrization from a result with another parametrization. As an example consider that an 8bit audio sample can easily be derived from a 16bit sample. As a result, it is often beneficial to use transformations instead of performing duplicate sampling or processing. However, without further knowledge such cases cannot be identified automatically. To gather this knowledge, the component system makes use of typed parameters. The parameter types allow the component developer to specify a transformation for each type (if one exists). Besides from providing a component to perform the transformation, the developer also specifies which parametrization shall be set (e.g. higher or lower). Furthermore, if a component supports multiple parameters, the developer also specifies an ordering.

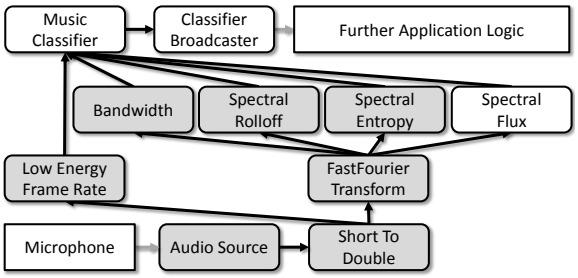

Figure 3. Componentized Music Recognition Logic

Using this, it is possible to extend the basic algorithm shown in Listing 1 as follows. First, we replace the Boolean *AllowsFolding* function with a tristate variant which returns the third state in cases where the components differ only in parametrization and all parameters can be transformed. If this state is returned to *FoldGraphs*, the algorithm first computes the parameter value to set at the component. Next, it identifies the graph that can be connected directly to the component and it runs the *Visit* procedure for it. Thereafter, the algorithm introduces the chain of transformations into the graph and sets the associated parameters. Finally, the algorithm connects the second graph by running *Visit* using the vertex representing the last transformation.

D. Example Applications

To create a set of applications, we implemented the speech and the music recognition logic using the set of audio features described in [6]. We used RapidMiner [14] to train a classifier for speech and music recognition using several hours of recorded audio data. Thereby, we increased the frame size of recordings to two second long periods which enabled us to achieve a 90 percent precision using a decision tree classifier. The resulting set of used features are shown in Figure 2 for speech and in Figure 3 for music.

As depicted in these figures, both classifiers use a varying set of frequency as well as time domain features. As a result, the componentized recognition logic contains a number of identically configured components (marked in grey). The

identical components include the actual sampling component (audio source), a conversion component (short to double), the frequency domain transformation (FFT) as well as features from the time domain (i.e. the low energy frame rate) and the frequency domain (i.e. bandwidth, spectral rolloff, spectral entropy). Besides that, each logic also contains components to compute unique features that are not relevant to the other application (i.e. zero crossing rate, spectral flux). Together, this provides an indication for the optimization potential of configuration folding.

V. Evaluation

To evaluate the effectiveness of configuration folding, we performed a detailed experimental analysis of the speech and music recognition applications introduced earlier. As target platform for our experiments, we used an HTC Tattoo which is a low-end smart phone (Qualcomm MSM7225 processor running at 528MHz, 512MB ROM, 256MB RAM) running the Android operating systems in version 1.6. In order to perform precise power measurements, we rely on a setup that closely follows the one described in [15].

Using this setup, we profiled four different software configurations that are depicted in Figure 4. In order to improve the readability, the figure shows the average power in W aggregated over periods of 100 ms. As a baseline for the power requirements, we measured the power required by the device when the display is running and the screen brightness is on its lowest setting (*idle*) which resulted in an average power drain of 103 mW. In addition, we individually profiled the music and the speech recognition applications when running on the device in isolation and we profiled the folded configuration (*folded*) that combined both, the music and the speech recognition applications. Thereby, we use identical parametrizations for all components to allow the most effective configuration folding that is possible.

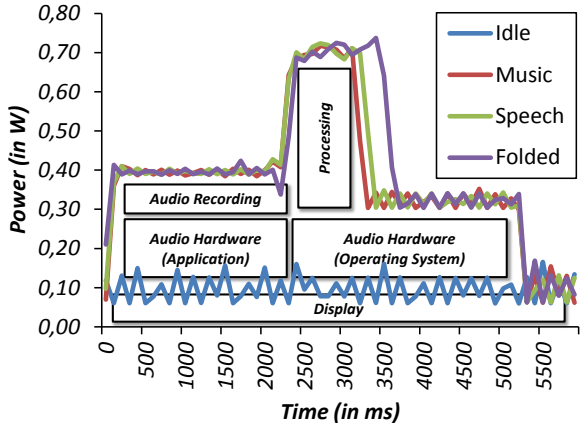

Figure 4. Decomposed Power Usage

Figure 4 shows one processing cycle for each of the tested software configurations over a period of 6 seconds. Using micro benchmarks that we created by stripping out parts of the applications, we decomposed their power profile by averaging over the relevant periods as follows. Since the device's display is continuously configured to the lowest brightness setting during all measurements, there is a constant power drain of 103 mW. When the recognition applications begin with their task, this power drain increases by 285 mW for a period of 2 seconds. This can be attributed to the fact that all recognition applications initially perform audio recording. The 285 mW can be further divided into 219 mW which are required to power the audio hardware and 66 mW which are required for the analogue to digital conversion and the buffering of samples. After the recording is done, all applications begin with the processing which results in an average power drain of 527 mW for the time the computations are performed. However, while decomposing the profiles we found that only 308 mW are actually due to the processor running at full speed. The remaining 219 mW can be attributed to the fact that the HTC Tattoo does not deactivate the audio hardware immediately after recording. Instead, once it is no longer required, it will remain powered on for an additional 3 seconds. Using additional experiments, we found that these 3 seconds are not dependent on the recording time but they appear to be a constant coded into the resource management code of the operating system.

Given these power measurements, we can easily approximate the overall energy requirements for each recognition cycle by determining the computation times required for the individual applications. To do this, we measured the processing times of 100 recognition cycles which resulted in an average processing time of 1002 ms for music recognition, 1023 ms for speech recognition and 1220 ms for the folded configuration. For all measurements, the standard deviation was well below 10 percent but there were systematic deviations which we attribute primarily to garbage collection that introduces around 100 ms of delay per collection. Note that these timings already show that for an identically parametrized music and speech recognition application, configuration folding saves 40 percent of processing.

To compute the actual energy savings between running the unfolded configurations simultaneously as well as running the folded configuration, however, it is important to consider that the power profile also contains components that cannot be added directly. Thus, in order to get generalizable results it is necessary to consider the interleaving of the music and the speech recognition application at runtime. The resulting extremes are depicted in Figure 5. All other possibilities for execution lie between them. In the best case (most energy efficient), both applications are starting their cycles at the exact same instance of time. This means that they share the cost for enabling the audio hardware as well as the analogue digital conversion during recording. We can compute the energy requirements in this case as

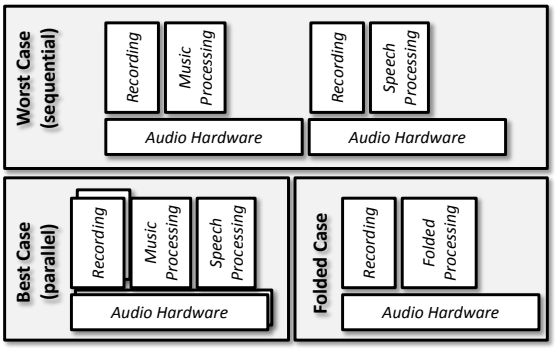

Figure 5. Analytical Energy Model

$5s \cdot 0.219W + 2s \cdot 0.066W + 2.025s \cdot 0.308W = 1.851J$

In the worst case, both applications are running sequentially which results in an energy usage of $10s \cdot 0.219W + 4s \cdot 0.066W + 2.025s \cdot 0.308W = 3.078J$. In contrast to this, a single cycle of running the folded configuration results requires on average $5s \cdot 0.219W + 2s \cdot 0.066W + 1.220s \cdot 0.308W = 1.603J$ of energy. Thus, the actual energy savings will always range between 13 and 48 percent. If we assume that context recognition applications usually have a low duty cycle in order to conserve resources, we can expect the savings to be closer to 48 than 13 percent.

To determine the effect of the configuration folding algorithm, we performed a similar set of measurements on the algorithm itself. This resulted in an average power usage of 448 mW for 1739 ms which results in 0.78 J for one run. From this effort, 865 ms and 724 ms can be attributed to XML loading and parsing as well as validation and graph transformation, respectively. The remaining 150 ms are required to execute the folding algorithm. Thus, the major overheads will only be experienced once. However, even when including these overheads, given the potential energy savings of 0.248 J or 1.475 J per cycle, the 0.78 J required by the algorithm are amortized after 1 to 3 cycles.

To determine the effect of parameters, we retrained the music classifier with a FFT that uses 4096 points instead of the initial 8192. We measured the average computation time of a folded configuration that used a transformation to handle the different parametrizations and we compared it with one that uses two fast Fourier components (one for music and one for speech). When compared to running two components, the introduction of the parameter transformation reduced the processing time by 630 ms which results in a saving of $0.630s \cdot 0.308W = 0.194J$ per cycle. At the same time, the usage of a more complicated folding algorithm, increased its execution time by 63 ms when compared to a simpler version that treats deviating parametrizations as non-matching components. Consequently, the additional $0.063s \cdot 0.448W = 0.028J$ of energy can be amortized within the first cycle. Thus, avoiding duplicate computations using transformations is always beneficial in this example.

VI. Related Work

There exist a number of context recognition applications for mobile devices, e.g. [3], [2], [6], [16]. Due to the energy-constraints of these devices, these applications apply a variety of techniques to save energy. [3], for example, relies on a low duty cycle to reduce the amount of sampling and processing. Furthermore, it applies data compression during network transmissions. [2] uses triggered sensing, in which sensors with high energy requirements are only enabled once an event has been detected by less energy-consuming ones. [6] extends the idea of triggered sensing to processing by introducing an admission control to avoid computations on samples that cannot contain an event due to low entropy. When compared with configuration folding, such techniques exhibit two main differences. First, they are focused on enabling the energy-efficient execution of a single known application. Thus, their usage is a necessary premise for energy efficient recognition but the optimizations are orthogonal to configuration folding which targets at efficiently executing *several unknown applications* (each of which should already be able to efficiently recognize the context). Second, in contrast to configuration folding which is *fully automated*, the usage of such techniques requires a significant amount of application knowledge, e.g. to determine optimal duty cycles or to define a set of triggers.

Consequently, configuration folding is more closely related to approaches like [16] that allow multiple location-based applications to efficiently localize a phone using GPS or WiFi. [16] has presented four broad design principles to perform energy efficient location sensing. From a high-level perspective, configuration folding can be though of as an implementation of the piggybacking principle (which aims at satisfying location requests from different applications with a single computation). However, in comparison with the implementation detailed in [16], configuration folding is more *generic* since is not restricted to localization but can be applied to arbitrary context characteristics.

There are also some generic approaches that specifically focus on energy efficiency. [17], [18] are frameworks that generalize the idea of triggered event detection. The recognition is thereby split into a set of smaller tasks that are ordered hierarchically and executed sequentially. Due to the sequential execution it is often possible to save energy since only parts of the hierarchy must be evaluated. As discussed previously, the applicability of such approaches is application-specific and thus, they require additional application knowledge in order to be applied. In contrast to this, configuration folding is fully automated and in principle, it is orthogonal to such optimizations. Similar to our component system, [8] reduces the energy requirements of several applications by minimizing the amount of sampling and processing. It introduces an application-overarching system

that enables developers to specify conditions over context in which they are interested. Based on the conditions, the system then determines the minimum amount of sensors and computations required and only executes those. A key difference between this and our approach is that [8] is closed – meaning it supports a fixed set of characteristics that can be detected. The component system presented in this paper is *open* as applications are free to implement arbitrary logic by providing components to recognize the desired context.

VII. Conclusion

Existing context recognition applications focus on support for a particular user task. To support parallel tasks, it is necessary to run multiple applications simultaneously. In this paper, we have shown how the simultaneous execution of multiple applications can be supported in an energy-efficient manner by applying a component-based approach to application development. Using configuration folding, our component system is able to automatically detect and remove redundancies in applications that have been developed in isolation. Our experimental evaluation shows that configuration folding can significantly reduce the energy requirements in cases where applications are built from similar sets of components. Furthermore, it indicates that the energy overhead of configuration folding is quickly amortized by the possible savings. Currently, we are investigating how the energy savings can be further increased by modifying parametrizations in a controlled but automatic manner.

Acknowledgements

This work has been partially supported by LIVING++ funded by the BMWi under contract number KF2095019FR0 as well as CONET and GAMBAS, both funded by the European Commission under FP7 with contract numbers FP7-2007-2-224053 and FP7-2011-7-287661, respectively.

References

[1] J. Lester, T. Choudhury, and G. Borriello, "A practical approach to recognizing physical activities," in *Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 3968. Springer, 2006, pp. 1–16.

[2] P. Mohan, V. N. Padmanabhan, and R. Ramjee, "Nericell: rich monitoring of road and traffic conditions using mobile smartphones," in *6th ACM conf. on Embedded network sensor systems, 2008*.

[3] E. Miluzzo, N. D. Lane, S. B. Eisenman, and A. T. Campbell, "Cenceme: injecting sensing presence into social networking applications," in *2nd European conf. on Smart sensing and context, 2007*.

[4] J. E. Bardram, "Applications of context-aware computing in hospital work: examples and design principles," in *ACM symposium on Applied computing, 2004*.

[5] T. Gu, Z. Wu, X. Tao, H. K. Pung, and J. Lu, "epsicar: An emerging patterns based approach to sequential, interleaved and concurrent activity recognition," in *IEEE Intl. Conf. on Pervasive Computing and Communications, 2009.*, march 2009, pp. 1 –9.

[6] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell, "Soundsense: scalable sound sensing for people-centric applications on mobile phones," in *7th intl. conf. on Mobile systems, applications, and services, 2009*.

[7] J. Kukkonen, E. Lagerspetz, P. Nurmi, and M. Andersson, "Betelgeuse: A platform for gathering and processing situational data," *IEEE Pervasive Computing*, vol. 8, no. 2, pp. 49–56, 2009.

[8] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Intl. conf. on Mobile systems, applications, and services, 2008*.

[9] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *16th IEEE intl. conf. on Automated software engineering, 2001*.

[10] D. Bannach, P. Lukowicz, and O. Amft, "Rapid prototyping of activity recognition applications," *Pervasive Computing, IEEE*, 2008.

[11] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "Pcom - a component system for pervasive computing," in *IEEE Conf. on Pervasive Computing and Communications*, 2004.

[12] R. J. J. V. Erich Gamma, Richard Helm, "Design patterns. elements of reusable object-oriented software, publisher: Addison-wesley."

[13] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, November 1962.

[14] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "Yale: rapid prototyping for complex data mining tasks," in *ACM intl. conf. on Knowledge discovery and data mining*, 2006.

[15] A. Rice and S. Hay, "Decomposing power measurements for mobile devices, 2010," in *IEEE intl. conf. on Pervasive Computing and Communications*, 2010.

[16] Z. Zhuang, K.-H. Kim, and J. P. Singh, "Improving energy efficiency of location sensing on smartphones," in *8th intl. conf. on Mobile systems, applications, and services. 2010*.

[17] A. Y. Benbasat and J. A. Paradiso, "A framework for the automated generation of power-efficient classifiers for embedded sensor nodes," in *Intl. conf. on Embedded networked sensor systems, 2007*.

[18] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh, "A framework of energy efficient mobile sensing for automatic user state recognition," in *Intl. conf. on Mobile systems, applications, and services, 2009*.