Analyzing 100 Billion Measurements: A NILM Architecture for Production Environments

NIKOLAUS STARZACHER, PHILIPP WEIDMANN
Discovergy GmbH
Sofienstr. 7a
69115 Heidelberg, Germany
Email: nilm@discovergy.com

***Abstract*—An architecture for Non-Intrusive Load Monitoring is presented along with a software implementation in the Java language. Various characteristics of the architecture that make it suitable for production deployments are discussed. In particular, a streamlined interface allows load monitoring algorithms to be developed by third parties without full source code access, supports both online and offline appliance recognition, integrates with an adaptive scheduler, and is agnostic with respect to the recognition approach. The Java implementation has been field-tested in a commercial, customer-facing application where it continues to manage NILM operations on more than 100 billion individual energy measurements.**

I. INTRODUCTION

THE integration of Non-Intrusive Load Monitoring (NILM)[1] technology into production systems presents formidable challenges beyond the obvious difficulty of developing correct and performant recognition algorithms.

Practical NILM applications often have to deal with missing or incomplete measurements, near real-time business requirements for recognition results, tight bounds on execution time, limited measurement consistency and temporal resolution compared to the laboratory environment, as well as implementation-specific storage and retrieval issues for both measurement and recognition data. Given these obstacles, it comes as no surprise that customer-facing NILM technology consistently fails to live up to the promises of NILM research.

This paper describes a system that aims to solve these and other common problems by abstracting away as many standard tasks as possible into the infrastructure. The result is a modular application that can integrate "raw" algorithms of the kind encountered in academic publications with little additional code. Once an algorithm has been adapted to use the system's interfaces, execution scheduling and output storage are managed automatically. This high-level approach parallels, but is distinct from, the NILM Toolkit (NILMTK)[2]. Whereas NILMTK provides a standardized *development* environment for NILM algorithms with data parsing, preprocessing and accuracy metrics, our architecture is the production counterpart, comprising all that is needed to integrate a pre-tested algorithm into an energy monitoring application.

II. DESCRIPTION OF ARCHITECTURE

A. The algorithm concept

In its most basic form, a NILM *algorithm* is a function that takes a list of *measurements* and returns a list of *recognition events*. Our interface augments this concept with two additional parameters and return values that facilitate computation and integration (Figure 1).

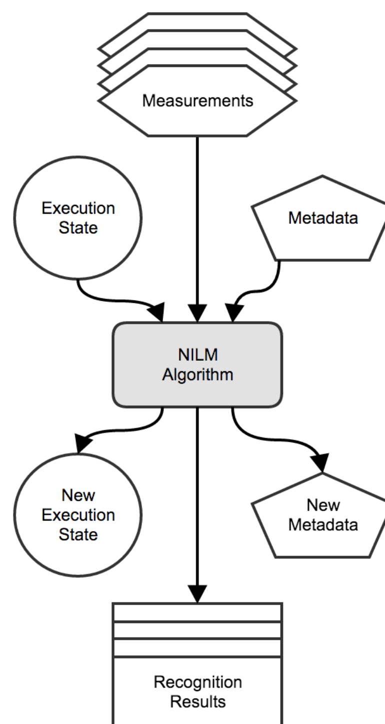

Fig. 1. An algorithm receives measurements, execution state and metadata for a metering endpoint and generates recognition results along with a new execution state to allow resumption of measurement processing. The algorithm can also update the metadata to share data with other algorithms.

1) Endpoint metadata: The metadata parameter passed to the algorithm captures any information about the metering endpoint that is independent of the energy measurements. Examples of useful metadata include a pre-populated list of appliances in the household being analyzed, the number of rooms and people in the household, the floor area, and human feedback on previous recognition results. Such information

may have been provided directly by the household owner, and thus can serve as a "ground truth" for sanity checks performed by the algorithm itself.

What distinguishes metadata from measurements is that only one metadata object exists per endpoint at any given time, representing the system's best knowledge about the endpoint available thus far. In particular, the same metadata is provided to every algorithm analyzing the endpoint. An algorithm can also *return* an updated metadata object to make summarized NILM insights available globally. This allows algorithms to share information independent of the recognition approach. For instance, a refrigerator detecting algorithm might store the total number of refrigerators in the household as a metadata value, which can then be read by another disaggregation algorithm to simplify preprocessing by excluding certain appliance configurations a priori.

2) Processing state: A modern smart electricity meter with a measuring interval of a few seconds generates millions of individual readings every year. Even with a lot of computing power, processing such quantities of data in a single run requires more resources than can usually be allocated when there are thousands of other datasets waiting in the processing queue.

For this reason, a practical NILM architecture must necessarily support splitting the analysis of a set of measurements into multiple invocations of the algorithm. As a side effect, this automatically covers the important use case of *online appliance recognition*, where recognition results are generated while an appliance is still active, or shortly after it is turned off. Online recognition is achieved by feeding individual measurements to the algorithm as soon as they arrive, and processing the results immediately.

But splitting a computation into blocks is a non-trivial task. The main issue is that an algorithm might depend on a characteristic complex feature being present in the load profile, such as a refrigerator's cooling cycle. Unless the complete feature is found during a run, it might not be possible to accurately identify the appliance without additional information.

Our system solves that problem by having algorithms return a *processing state* that encodes partial matches at the end of the measurement timeline that could not be fully resolved during the current invocation. When the algorithm is executed again with a batch of measurements immediately succeeding those in the previous run, that state is passed as a parameter, allowing the computation to continue where it left off. The contents of that state object are specific to each algorithm, and may differ widely between algorithms. Indeed, the most naive state object just contains all measurements from all previous runs, but far more efficient state representations are possible in many practical cases. For an algorithm identifying when an individual appliance was running, the state might be reduced to a flag specifying whether the appliance was on or off at the end of the preceding run. If tracking of partial matches is not required or the measurement batches are sufficiently large, the state object can be entirely empty.

B. The processing cycle

As there will usually be multiple algorithms to be run, a pipelining approach can be employed to reduce the total amount of measurements that need to be loaded. This is important because disk I/O from loading measurements for analysis is often the performance bottleneck of the entire analysis operation.

At a point in time, different algorithms may have processed different subsets of the measurement data available for a metering endpoint. Since most NILM algorithms process measurements in time-ascending order, the progress for each algorithm can be described by the timestamp of the latest measurement the algorithm has processed thus far. By taking the minimum of these timestamps over all algorithms for a given endpoint, one obtains the lower boundary of a time interval that covers all of the intervals required for each algorithm to proceed (the upper boundary of that interval varies, but is typically just the current time, or the timestamp of the latest available measurement).

Thus the system can first load all measurements contained in the aforementioned time interval from the data store, slice the resulting list as prescribed by the individual algorithms' continuation states, feed those slices (along with the states themselves) to the algorithms, and then store the recognition results and updated state objects. This I/O-minimizing strategy is referred to as a *processing cycle* (Figure 2).

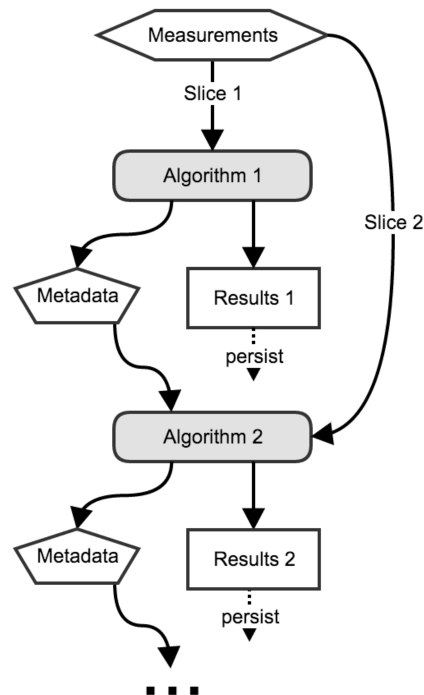

Fig. 2. Slices from a single set of measurements are passed to multiple algorithms depending on their processing state. Metadata generation is chained to avoid merging.

It should be noted that while the distribution of a meter's measurements to multiple algorithms could readily be parallelized, parallelism across meters (that is, having one parallel unit execute all algorithms on a single meter's data

sequentially) should probably be preferred instead. The reason is that after separating the retrieval of measurements from the processing (algorithmic) stage, that stage will easily saturate the available CPU/GPU power when pipelining thousands of meters, enabling the integration with an adaptive execution scheduler that takes system load metrics into account (see following section). Additionally, serializing algorithm runs for each endpoint eliminates the need for metadata merging, as each algorithm consumes the preceding algorithm's metadata object and in turn feeds metadata output to its successor. There is never more than one current metadata object per endpoint.

III. Implementation Details

Our production implementation of the architecture described above uses the Java language. The following are simplified versions of the actual interface and base classes (illustrative only, not intended as functional code):

```
interface Algorithm<O extends Output,
                    S extends State> {
  RunResult<O, S> run(
    List<Measurement> measurements,
    S state,
    MetaInformation metaInformation);
}

class Output {
  long startTime;
  long endTime;
  // To be subclassed depending on algorithm
}

class State {
  // To be subclassed depending on algorithm
}

class RunResult<O extends Output,
                S extends State> {
  List<O> outputs;
  long startTime;
  long endTime;
  S state;
  MetaInformation metaInformation;
}

class Measurement {
  long time;
  Map<String, Object> values;
}

class MetaInformation {
  Map<String, Object> values;
}
```

A. Automatic serialization of output

Like the measurements themselves, algorithmic results should be stored in a database to make them available for other applications, such as serving them over a web API. This is readily accomplished using any serialization framework that just blindly iterates over the `Output` subclass fields.

However, explicitly declaring the `startTime` and `endTime` fields as shown above and treating them separately during serialization has two advantages. First, a temporal index can be generated to support fast lookups of recognition results, which is crucial when displaying NILM output in a user-facing application. Second, algorithms can generate *speculative* output. For example, an algorithm could identify a feature at the end of the current timeline as the possible start of a refrigerator's cooling cycle. It then generates a `Refrigerator` object (subclass of `Output`) representing the partial recognition event (and in particular capturing the beginning and speculative end of the cycle in `startTime` and `endTime`, respectively). This object can be stored for later use just like a complete recognition event would. If the next run of the refrigerator detection algorithm, having received additional measurements, determines that the "partial cooling cycle" was in fact the beginning of a full cycle (or that it was not), *it can update previous output* by making the `RunResult`'s `start/endTime` interval contain the partial cycle. Algorithms need not be conservative about generating recognition data even at low confidence levels, because any speculative output that later turns out to be incorrect can be adjusted or deleted retroactively. This allows NILM results to be persisted, and thus made available to the end user, almost as soon as they are generated, strengthening the "live" aspect of the system.

In summary, I/O proceeds as follows for a single algorithm:

1) New measurements, the previous processing state and the current metadata object are loaded.
2) The three are passed to the `Algorithm` implementation's `run` method, yielding a `RunResult` object.
3) The `RunResult`'s new processing state (algorithm/endpoint-specific) and metadata object (endpoint-specific) are persisted appropriately.
4) All previously persisted (algorithm/endpoint-specific) output objects that fall within the `RunResult`'s `start/endTime` interval are deleted.
5) The `RunResult`'s output objects are persisted.

B. Adaptive execution scheduling

The component deciding which algorithms to run, when to run them and in what order, which endpoints' data to run them on and how much of that data to process during a single run is referred to as the *execution scheduler*.

Such a scheduler might take several forms, some of them very simple. The most basic solution consists of a loop that loads each metering endpoint's new measurements, feeds them to all algorithms, persists the results, and then repeats that process. Another possibility would be to leverage the operating system's job scheduling facilities, such as `cron` on Unix[3], in order to ensure each meter's measurements are regularly analyzed.

In production NILM systems, however, there are usually multiple constraints that necessitate a more sophisticated approach. Most importantly, resources are limited and must be prioritized based on whichever combination of metering endpoint, algorithm and measurement subset is most needed at a given moment. Scheduling criteria might include:

- The duration since an algorithm/endpoint pair was last processed. This is probably the dominant criterion and is easily implemented using a standard priority queue.
- The likelihood for individual algorithms to generate useful output from the latest data. Most refrigerators will

average at least one cooling cycle per hour, but a washing machine might run only once per week.

- The potential impact of a delayed recognition event. Missing a hot plate accidentally left on can be dramatically worse than failing to detect a light switch in a timely manner.
- The average duration of a successful recognition event. If a pool pump typically runs for five consecutive hours and the algorithm designed to detect it just executed without returning a result, there is no need to check again immediately afterwards.
- The current system load. High load might mean lower-priority analyses should be rescheduled to run at a later time.
- Customer interaction with endpoint data. While a customer is logged into the application, it makes sense to prioritize their endpoint(s) to deliver information to the frontend faster.

C. Algorithm chaining

Traditional NILM algorithms normally expect time series of raw (power) measurements as input.[4][5][6] Algorithms designed for very complex tasks, such as determining how many people are likely to be present in a household from the electricity consumption alone, can benefit from working with *other algorithms' output* instead. This leads to the idea of algorithm chaining.

At a minimum, an architecture supporting chaining has to replace the list of measurements in the `Algorithm` interface's `run` prototype with a list of generic input data points:

```
interface Algorithm<I extends Input,
                    O extends Output,
                    S extends State> {
  RunResult<O, S> run(
    List<I> inputs,
    S state,
    MetaInformation metaInformation);
}

class Input {
  // To be subclassed depending on algorithm
}
```

Additionally, the actual algorithm chains to be executed must be defined. An example of such a chain could be an algorithm detecting basic heating elements via their thermostat signature, followed by an algorithm identifying compound appliances containing such elements, like dishwashers, ovens and washing machines, and finally and algorithm turning high-level appliance recognition events into an activity profile for the entire household.

Chaining is a potentially powerful approach but also introduces new complexities into the system, namely storage of intermediate data, validation of results, and chain management. As such, chained algorithms have not yet found their way into our production architecture.

ACKNOWLEDGMENT

Our colleague SERGIU HLIHOR was instrumental in the implementation of the system described herein, particularly the scheduler, and fine-tuning its performance to the characteristics of our analysis servers.

REFERENCES

[1] G. W. Hart, "Nonintrusive appliance load monitoring," *Proceedings of the IEEE*, vol. 80, no. 12, pp. 1870–1891, 1992.

[2] N. Batra, J. Kelly, O. Parson, H. Dutta, W. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava, "NILMTK: An Open Source Toolkit for Non-intrusive Load Monitoring," in *Fifth International Conference on Future Energy Systems (ACM e-Energy)*, Cambridge, UK, 2014.

[3] IEEE and OpenGroup, "crontab - schedule periodic background work," 2001. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

[4] C. Laughman, K. Lee, R. Cox, S. Shaw, S. Leeb, L. Norford, and P. Armstrong, "Power signature analysis," *Power and Energy Magazine, IEEE*, vol. 1, no. 2, pp. 56–63, 2003.

[5] M. Zeifman, C. Akers, and K. Roth, "Nonintrusive appliance load monitoring (nialm) for energy control in residential buildings: Review and outlook," in *IEEE transactions on Consumer Electronics*. Citeseer, 2011.

[6] M. L. Marceau and R. Zmeureanu, "Nonintrusive load disaggregation computer program to estimate the energy consumption of major end uses in residential buildings," *Energy Conversion and Management*, vol. 41, no. 13, pp. 1389–1403, 2000.