The Weight-Watcher Service and its Lightweight Implementation

Benoît Garbinato*, Rachid Guerraoui**, Jarle Hulaas**,
Alexei Kounine**, Maxime Monod** and Jesper H. Spring**
*Ecole des HEC
Université de Lausanne
CH-1015 Lausanne, Switzerland
**School of Computer & Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

***Abstract*—This paper presents the Weight-Watcher service. This service aims at providing resource consumption measurements and estimations for software executing on resource-constrained devices. By using the Weight-Watcher, software components can continuously adapt and optimize their quality of service with respect to resource availability. The interface of the service is composed of a Profiler and a Predictor. We present an implementation that is lightweight in terms of CPU and memory. We also performed various experiments that convey (a) the trade-off between the memory consumption of the service and the accuracy of the prediction, as well as (b) a maximum overhead of 10% on the execution speed of the VM for the Profiler to provide accurate measurements.**

I. Introduction

Weight-Watcher diet programs have been very popular the last decade as presumably effective ways of losing weight. Diets restrict the amount of food or give advice on how to choose the right type of food that a human should ingest. To this end, the weight-watcher usually relies on the foods' *nutritional facts labels*, which give information on the different nutrients included and the corresponding human daily needs. The listed nutrients are *calories*, *fat*, *carbohydrates*, *proteins*, maybe *vitamins* and so on. Based on this information, the consumer can decide whether or not to eat this or that aliment. This kind of declaration does normally not exist for software components. That is why we developed the present Weight-Watcher service, whose role is to dynamically elaborate the same kind of nutritional facts labels for pieces of code in the context of resource-constrained devices.

Consider a mobile device (the *provider*) streaming audio to several other devices (the *receivers*) at some predefined quality. As the number of receivers grows, the provider will gradually become more prone to resource availability problems: its hunger for processing power and network bandwidth will reach a level where concurrency with other services and applications will cause intermittent, but intolerable interruptions of the audio stream. In dedicated server environments, some of these issues may be addressed by proper ahead-of-time dimensioning and planning of priorities; this cautious approach is however rather unlikely in the world of mobile consumer devices that we address here. Moreover, exogenous factors like network congestion will anyway have to be dealt with at run-time. Other resource types on which there typically will be a high contention are CPU and memory: in our example scenario, if the user of the provider device additionally wants to activate interactive applications like an agenda or a game, he should be able to designate to which ones the system should allocate resources first. A desirable behaviour – that the Weight-Watcher approach enables – would then be that the system and the various applications automatically adjust to yield the best overall resource distribution.

If the provider's policy is not to limit, but to maximize the number of receivers, it must adapt gracefully,[1] by degrading its quality of service (or *application fidelity* [12], [13], [16]), i.e., by increasingly compressing the stream. But first of all, in order to implement its resource-aware behaviour through timely adaptations to the fluctuations of its resource environment, the software running on the provider's device must be informed on the amount of resources currently available, as well as the amount of resources required by the piece of code that is currently to be executed (from here on, the piece of code on which the Weight-Watcher gives a prediction in terms of resource consumption will be named an *action*). The problem we address is thus to detect *when* a given software component should adapt itself. The Weight-Watcher service provides resource consumption predictions that can be compared to the current levels of resource availability. The comparison is thus the base information on which decisions can be made, be it by a system-level scheduler or by resource-aware applications.

One tricky aspect of the Weight-Watcher service is that it must itself consume the least possible amount of resources, especially `CPU` and `Memory`. Moreover, our Weight-Watcher service itself shall also adapt its own resource requirements at runtime, which results in a change of quality of service, i.e., in its prediction accuracy. Using the Weight-Watcher service,

The work presented in this paper was sponsored by the European IST PALCOM project (Swiss OFES No 03.0495-1)

[1] Depending on the communication protocol and the audio format, the receivers may also have to adapt explicitly; in the present example, we consider that this is not necessary.

programmers can turn their applications into resource-aware services and the Weight-Watcher itself, if configured so, can also behave in a resource-aware manner.

The principle of the Weight-Watcher is to provide *history-based* resource consumption predictions, meaning that the predictions rely on resource consumption measurements following the execution of any action. All kinds of actions, e.g., methods or event handlers, can be profiled as long as they are executed more than once (as in any history-based learning approach, e.g., [12], [13], [19]). In this paper, we handle the following `Memory`, `CPU`, `Network`, `Energy`, and `Time` resources, even though the techniques presented are not limited to these specific resources. Two sub-services provide the resource consumption informations: measurements are output from the Profiler, and predictions are output from the Predictor.

In order to provide resource consumption measurements following the execution of an action, we implemented a dynamic Profiler (by modifying the KVM [17] Java virtual machine) which gives perfect measurements, at the VM level,[2] for `CPU`, `Memory`, `Network` and `Time`, resulting in a slowdown as low as 6.46% – 9.86% (see Section VI-A). In comparison, the slowdown induced by Java bytecode instrumentation techniques [1], lies on average around 20% [1] to 40% [10]. On the other hand, the accuracy of the Predictor is a function of the resources it is allowed to use, as explained in Section VI-B. This can be entirely automatic, or tuned programmatically (by an application), depending on the resources left in the system (see Section IV-A).

II. The Weight-Watcher Service

The Weight-Watcher service is composed of two subservices, the Profiler and the Predictor:

1) The Profiler provides measurements about the amount of resources an action has consumed during its last execution, *after* the execution. It can be viewed as *resource accounting*, with the restriction that it should itself require a very low amount of resources.
2) The Predictor provides estimation about the amount of resources an action will consume, *before* its execution. The Predictor *remembers* the execution history of the action (depending on the state of the system and the parameters of the action at execution time), *combines* the measurements together and builds an estimation of the amount of resources an action will require for its execution. The Predictor can be tuned in many ways to change its accuracy, as exposed in Section IV.

A. Definitions

Resources. In this paper, we consider resources as being the number of bytes that the code has dynamically allocated (`Memory`), the number of CPU cycles or bytecodes executed (`CPU`), the energy, in micro Joules, consumed during the execution (`Energy`) and the time spent executing the action (`Time`). In our system, predictions depend on measurements, therefore the prediction values will only concern the list of resources that were chosen to be profiled.

Resource Profiles. A *resource profile measurement* (rpm) is the set of resource amounts that an action has consumed, known *after* its execution. If the Profiler is configured to measure `Energy` and `Memory` consumption of the action A, a possible resource profile measurement of A after its first execution is $rpm_{A,1} = \{442.6\ mJ,\ 2048\ bytes\}$.

A *resource profile prediction* is an estimation of the resource amounts the action will require upon its next execution, it is the output of the Predictor, and a possible prediction for the execution of action A for its $(i+1)^{th}$ execution is $p_{A,i+1} = \{522.3\ mJ,\ 4096\ bytes\}$. The Predictor outputs resource amounts of chosen resources, corresponding to the resource profile measurements. It accepts *parameters*, as exposed in Section IV to make it aware of variables that change the resource consumption of the action to predict.

Therefore, a *resource profile* (rp) represents the set of resources in which the application programmer is interested for a particular action, e.g., $rp_A = \{$`Energy, Memory`$\}$. The resource profile contains a list of resources (names), whereas both resource profile measurements and predictions contain lists of corresponding resource amounts (values).

Prediction Errors. The *prediction error* is defined as the difference between the resource profile prediction (p_{i+1}) and the actual resource profile measurement after the execution (rpm_{i+1}): $p_{err,i+1} = |p_{i+1} - rpm_{i+1}|$, and thus the relative error is defined as $p_{err,i+1}/rpm_{i+1}$.

B. Interfaces

Via two interfaces (the Profiler and the Predictor), the Weight-Watcher outputs resource profile measurements of past executions and resource profile predictions for upcoming executions. The usage of the interface is illustrated in Figure 1. The interface of the Profiler contains the following operations:

- `Profiler.reset(resource_profile rp)` sets the resource accounting counter of each resource in the resource profile back to 0.
- `Profiler.getCount(resource_profile rp)` outputs the amount of each resource in the resource profile consumed since the last `Profiler.reset()`. For instance, `Profiler.getCount({Memory})` outputs the exact amount of bytes allocated since the last `reset()`.

For illustration (Figure 1), consider the resource profile measurement of action A with a resource profile $rp = \{$`CPU, Memory`$\}$. First, the profiling counters are reset for each resource in rp: `Profiler.reset({CPU, Memory})`. Second, the action A gets executed if the Scheduler accepts to execute it (e.g., if enough resources are available), and third, the rpm is assigned to the values given by the profiler, containing measurements for each resource in the resource profile:

[2]Meaning that resources consumed by native code are not accounted.

```
// the action A is identified by its id: aid
// the action resource profile is rp={CPU, Memory}
...
try {
   p = Predictor.query(aid); // gets the prediction
   Scheduler.canExecute(p); // can throw exception
   Profiler.reset(rp); // resets the Profiler
   ...
   ... // the actual action A execution
   ...
   rpm = Profiler.getCount(rp); // gets the measurements
   Predictor.update(aid, rpm); // updates the prediction
}
catch (...) { // an exception raised by the
            // Scheduler.canExecute() method
   ... // here is the callback that can implement
   ... // a strategy for reacting to the scheduler
   ... // notifications
}
```

Fig. 1. Use of the interfaces of the Profiler and Predictor

`rpm=Profiler.getCount({CPU, Memory})`. The *rpm*, if used as exposed when encapsulating the execution of an action, is then taken as input for the Predictor.

The interface of the Predictor contains the following operations:

- `Predictor.query(actionID aid)` outputs a prediction for a given action,
- `Predictor.update(actionID aid, resource_profile_measurement rpm)` updates the prediction of a given action with the output of the Profiler (`rpm`).

For a given action, the Predictor gets first setup with different combining strategies (i.e., the way the Predictor takes into account past measurements to provide predictions, see Section IV-B) and parameters values (i.e., the state in which the system is before executing the action, see Section IV). The method `Predictor.query()` is the main entry point of the Predictor. It is used to know the amount of resources an action A will require for its execution, before its execution. With this prediction, a decision making process can be run to decide if this particular action should be executed or not (`Scheduler.canExecute(p)`). If yes, the Profiler resets, then the action gets executed, and a rpm is output from the Profiler. Now that a new measure is available, the predictions for an upcoming execution can be refined via `Predictor.update()` as shown in Figure 1.

The Scheduler can thus base scheduling decisions by comparing the prediction and the actual amount of resources available on the device. If the resource demand is greater than what is currently left in the system or if the resources left are detected to be shrinking (e.g., using threshold values) the Scheduler can (1) postpone execution (until required resources are available), (2) trigger resource shortage notification (so that resource related errors can be avoided), or (3) pick an alternative implementation (e.g., downgrade the level of service). The alternative implementations might be downloaded from the network, already loaded in main memory (i.e., ready to execute) or on stable storage (i.e., ready to be loaded). This issue is out of scope of the paper and is not further discussed.

III. History-based Prediction

The role of the Predictor is to estimate, before the execution of the action, its upcoming resource consumption. In [2], the estimation of the memory consumption of a method is given by a parametric function, where the parameters of the function are the actual method parameters. Let us formalize this function as $f_{\texttt{Memory}}(\{x_1, \ldots, x_n\}, m)$, where the x_i are all the parameters of the method m and f provides an estimation of the `Memory` allocated for an execution of m. In [2], the $f_{\texttt{Memory}}$ function is created out of offline code analysis, whereas we numerically approximate the same function, at runtime, for every resource in the rp, i.e., not only `Memory`, by basing our prediction on actual measurements (the resource profile measurements, or rpms) output by the Profiler. Thus the estimation function f that we approximate is a generalization of $f_{\texttt{Memory}}$ such that $f_{\forall r \in rp}(\{x_1, \ldots, x_n\}, m)$.

To achieve the history-based prediction, the action first gets executed and profiled. The Predictor *remembers* this rpm_1, by storing it in memory. When the piece of code is about to be reexecuted the prediction is in fact the first stored measurement given by the profiler (rpm_1). After the second execution a new rpm_2 is generated by the Profiler. Both measurements (rpm_1 and rpm_2) can thus be *combined* to refine the prediction (see Section IV-B).

An action that has a constant behavior in terms of resource consumption is for example a method that computes a Celsius temperature out of Fahrenheit (`float fToCelsius(float fVal)`). In this case the estimation function and therefore the predicted value, is constant for each resource r for any execution: $f_{\texttt{r}}(\{\texttt{fVal}\}, \texttt{fToCelsius}) = rpm_{\texttt{fToCelsius},i} = p_{\texttt{fToCelsius},i+1}\{r\} = c_r$. More generally, for constant actions (m is a constant action), the following holds: $\forall x_i \forall i,\ f_{\texttt{r}}(\{x_1, \ldots, x_n\}, m) = rpm_{m,i+1} = p_{m,i+1}\{r\} = c_r$. In theory, for such a simple behavior the memory cost for recording the profiled value is equivalent to one `int` per resource (resp. `float` according to the resource unit).

In contrast, the execution of some other action can be influenced by a set of parameters. In such a case two different executions of the same portion of code might not be the same in terms of resource consumption, which suggests that these parameters should be taken into account in the prediction. Figure 2 presents two actions for which the resource consumption depends on parameters. Specifically, The action `convertToCelsius` in Figure 2(a) has a resource consumption that is directly proportional to the parameter `temp.length`. On the other hand, the memory consumption of the action `getAudioChunk` presented in Figure 2(b) depends on several parameters. In case data was not lost, the memory consumption is proportional to $2 \cdot dSize$ if the chunk is stereo and to $dSize$ if the chunk is mono.

This same action could be split into smaller ones, that would have a close to constant execution pattern: an action for the `isLost==true` behavior and another one for `isLost!=true`. The latter action could again be split into

```
void convertToCelsius(int[] temp) {
    for (int i=0; i<temp.length; i++) {
        temp[i]=fToCelsius(temp[i]);
    }
    // The CPU consumption of the action depends
    // on the array size (temp.length)
}
```

(a) Single parameter: $nbTemp$

```
void getAudioChunk(Data d) {
    Header h = d.getHeader();
    boolean s = h.isStereo();
    int id = h.getId();
    boolean isLost = (id != (currentId +1));
    if (isLost) {
       // Detection of lost data -> Dedicated handling
    }
    else {
        int dSize = h.getDataLength();
        if (s) {
            int [][] buf = new int[dSize][2];
            // processing of bi-channel audio signal
        }
        else {
            int [] buf = new int[dSize];
            // processing of mono channel audio signal
        }
    }
}
```

(b) Several parameters: $isLost$, id and s

Fig. 2. Actions from which the execution patterns depend on input variables and which thus have non-empty sets of parameters.

two actions, one for the treatment of the stereo-channel signal and another one for treatment of the mono-channel signal, and so on. Of course it is cumbersome for a developer to unfold loops manually or even split an action into smaller ones, which is the main reason why the Predictor needs to take parameters (e.g., `isLost`, `id`, `s`) into account for storing and building predictions.

IV. Lightweight Weight-Watcher Implementation

For obvious memory reasons, the Predictor does not keep track of every past parameter value and corresponding resource consumptions. Thus, it must sample continuous parameters to remember past executions, i.e., parameter values that come close are considered equal and will get the same prediction. Taking the example from Figure 2 (a) two executions of `convertToCelsius()` with arrays of different sizes (the parameter of the action being `temp.length`) might return the exact same prediction. How close two parameter values are is defined by: (1) the parameter bounds, and (2) the sampling precisions.

The simplest data structure and the one that has the smallest overhead for storing samples of one single continuous parameter is an array. The size of this array depends on three factors: the minimum value of the parameter (min), its maximum value (max), the number of intervals (k) between the min and the max and the number of resources (r) in the resource profile taken into consideration. If measures are stored as integers, for a continuous parameter having a $min = 0$, $max = 500$, and $k = 50$, the width of each interval ($\Delta = (max - min)/k$) is 10 and the size of the array is then $50 \cdot \texttt{sizeOf(int)} \cdot r$. Δ is further refered to as the *precision* of a certain parameter. This means that every action executing with parameter values in range $[10..19]$ are considered the same, in other words, the prediction for an upcoming execution in this particular interval will be the same.

For one parameter, the size of the array is defined by $k \cdot r \cdot t$, and is generalized as $\prod_{p=1}^{nbPar} k_p \cdot r \cdot t$ for multiple parameters, where r is the number of resources, $NbPar$ the number of parameters, k_p their corresponding intervals and t the size of the type of data stored.

An analysis of this function trivially shows that the memory used for storage is mostly influenced by the number of parameters ($nbPar$) and the desired precision for the prediction of values of each parameter (k_p). Both t and the number of resources r are relatively small, meaning that it is not by modifying t and r that the array memory consumption will be influenced. Thus, if the service must dynamically adapt its resource consumption (the memory consumption of the array in that case), it must be able to tune the k_p service parameters to still be able to take a constant $nbPar$ number of parameters into account.

A. Adaptive Array

A programmer might not always be able to know a priori the exact range of values that a parameter might take during all executions of a certain action. Cases can happen when a parameter takes a value which is outside its initial range. If in a particular execution the parameter value v falls outside of its $[min..max]$ range, two possibilities arise for storing the value:

1) Extending the size of the array and keeping the precision (Δ) constant.
2) Keeping the same array size and decreasing the precision Δ.

Keeping the precision (Δ) constant. In the first case, the range of a parameter is increased while Δ stays constant, resulting in an increase in the array size ($k_i \nearrow$). Considering the example from last section where a parameter is initialized with the values $min = 0$, $max = 500$ and $k = 50$ ($\Delta = 10$). The action is now executed with $v = 512 \notin [0..500]$. The range is therefore forced to grow to $[0..520]$ to include the new v (two intervals of size $\Delta = 10$ are added) and the number of intervals reaches $k = 12$. This strategy results in an increase of the memory taken by the resource profile (directly related to k as exposed in Section IV) especially when the out of range value is very far from the min (resp. max). The number of additional intervals k_{add} is defined as $\lceil (v - max) \ / \ \Delta \rceil$ (resp. $\lceil (min - v) \ / \ \Delta \rceil$).

Keeping the array size constant. In order to keep the size of the array constant, the range of a parameter is modified by decreasing the array precision ($\Delta \searrow$). Looking at the previous example, when the action is executed with $v = 512$, several solutions can be followed:

- Keeping the Δ of every cell but the last one constant, $\Delta_{0..48} = 10$ and increase the Δ of the last cell to take

the parameter value into account, $\Delta_{49} = 22$. The last cell of the array of the concerned parameter has a worse precision than the other cells (because it is larger) and the total range of values is changed to $[0..512]$, allowing the new value to be stored in the array.
- Increase the Δ of every cell: $\forall i, \Delta_i = 11$, in which case the total range is changed to $[0..550]$. In order to do this, the values for each new cell need to be recomputed. As a cell in the new array will be a linear combination of cells spanning the same range in the old array, dedicated calculation for computing the values must be executed to repopulate the array.

These reactions to a parameter value that is out of range of the prediction array can be issued by (1) the system in situations where the need of either precision on measurements or available resources is predominent (automatic adaptivity) or (2) by the programmer in order to have control over the content of the resource profile.

B. Combining Strategies

A combining strategy is represented by the function $p_{i+1} = g(p_{1..i}, rpm_{1..i})$ meaning the prediction for the upcoming $(i+1)^{th}$ execution can be computed with the prediction of the last executions $p_{1..i} = \{p_1, \ldots, p_i\}$ and/or the resource profile measurements $rpm_{1..i} = \{rpm_1, \ldots, rpm_i\}$ of the past executions. To have an optimal prediction, the strategies should minimize the prediction error under the constraint that the combining strategies must be as *frugal* as possible, i.e., being the least consuming in terms of `CPU` and `Memory` usage.

Overwriting Strategy (OS). The first and simplest strategy to update its prediction p_{i+1} is to state that the $(i+1)^{th}$ execution will have the exact same resource need as the i^{th} output from the Profiler. In other words, the resource profiler measurement rpm_i is used as is in order to predict the next execution, see (1) in Figure 3.

(1) Overwriting Strategy (OS):	$p_{i+1}{=}rpm_i \ (i \geq 1)$
(2) Adapting Strategy (AS):	$\begin{cases} p_2{=}rpm_1 \\ p_{i+1}{=}\frac{p_i + rpm_i}{2} \end{cases} (i \geq 2)$
(3) Low-Pass Filter (LPF):	$\begin{cases} p_2{=}rpm_1 \\ p_{i+1}{=}\frac{8 \times p_i + 2 \times rpm_i}{10} \end{cases} (i \geq 2)$
(4) Global Average (GA):	$\begin{cases} p_2{=}rpm_1 \\ p_{i+1}{=}\frac{\sum_{j=1}^{i} rpm_j}{i} = \frac{p_i \cdot (i-1) + rpm_i}{i} \end{cases} (i \geq 2)$

Fig. 3. Combining strategies

Overwriting the prediction with the rpm at every execution states that an action stabilizes completely over time, or that the action does not depend on any parameter. It is the most *frugal*, in the sense that no memory on the history of measurements is kept and that the actual computation is simply a value replacement in the prediction array.

Adapting Strategy (AS). The second strategy is to do an average of the last (i^{th}) prediction p_i with the corresponding measure rpm_i. The strategy is defined by (2) in Figure 3. The last rpm influences the prediction twice as less as in the overwriting strategy, adding an addition and a division to the complexity.

Low-Pass Filter (LPF). The low-pass filter gives predefined weight to both the prediction (80%) and the actual measurement (20%) as (3) in Figure 3. It adds two multiplications to the complexity of the adapting strategy. Note that for every strategy until now, a variable containing the measurement from the profiler rpm_i and the prediction itself p_i was enough to compute the new prediction p_{i+1} (p_i and p_{i+1} correspond to the same slot in the array).

Global Average (GA). A global average is also proposed as (4) in Figure 3. Note that every past resource profiles do not need to be stored, as the last prediction p_i and i are enough to reconstruct the sum from 1 to $i-1$. This basically means that not only the last prediction must be kept but also i, the number of measurements the profiler has output for that particular prediction. In fact, an array for storing the prediction and an additional array (of same size) must be used for storing the corresponding number of iterations, i.e., doubling the memory usage of the combining strategy. In number of mathematical operations, it has one multiplication less, and adds only one subtraction $(i-1)$ to the low-pass filter, but in practice, the strategy is more expensive as more accesses in memory must be made (for getting i) instead of using constants as in the low-pass filter.

The prediction errors of each combining strategy are summarized in Figure 4 and will be exposed as a performance metric in Section VI.

(1) OS:	$p_{err,i+1} = \lvert rpm_i - rpm_{i+1} \rvert$	$(i \geq 1)$
(2) AS:	$p_{err,i+1} = \left\lvert \sum_{j=1}^{i} \frac{rpm_{(i+1-j)}}{2^x} - rpm_{i+1} \right\rvert$	$(i \geq 2)$
(3) LPF:	$p_{err,i+1} = \left\lvert \sum_{j=1}^{i} \frac{2 \times 8^{j-1}}{10^j} rpm_{(i+1-j)} - rpm_{i+1} \right\rvert$	$(i \geq 2)$
(4) GA:	$p_{err,i+1} = \left\lvert \frac{\sum_{j=1}^{i} rpm_j}{i} - rpm_{i+1} \right\rvert$	$(i \geq 2)$

Fig. 4. Prediction errors

C. First execution issue or sharing resource profile as initial data

It is only after the first execution of the action that an rpm is output and thus a prediction can be created[3]. We consider the following solutions to this problem of having initial predictions:

- Share resource profile: even though resource units are not completely portable (e.g., heterogenous VM, CPUs, devices can lead to different rpm for the same action), they can be used as an initial resource profile measurement rpm_0 for the first prediction $p_1 = rpm_0$ and then refined by the different combining strategies.
- Use existing solutions, as exposed in [12] by first executing a training phase, that consists in executing the action various times to get first measurements, then optionally run an offline learning phase and finally use the predictions at runtime.

[3]To be exact, for a predefined parameter range R, there exists only an rpm thus a prediction after an execution of the action in the same condition, i.e., with a parameter value $v \in R$.

V. IMPLEMENTATION

The Weight-Watcher service is composed of (1) modifications to the KVM in order to implement the Profiler and (2) Java classes implementing the Predictor.

A. Profiler

We propose generic modifications to a VM that implement the Profiler. The dynamic Profiler accounts for `Memory` (in bytes), `CPU` (in number of bytecodes), `Time` (in μs) and `Network` (in bytes/s) while the program is actually running. For doing so, counters are added to the VM threads, that are incremented while the bytecodes are executed by the virtual machine. The `CPU` counter must be incremented as many times as bytecodes are executed in the current thread (easily achieved in an interpreted VM as KVM [17]), whereas the `Memory` counter is incremented by the size of the data that is allocated by the current thread (modifications to the VM memory management of KVM).

For `CPU`, a counter (an array of 256 `int`s) keeps track of which bytecodes are executed and how many times, so that it is summed up when `Profiler.getCount({CPU})` is called, providing the actual number of bytecodes executed since the last `Profiler.resetCount()`. It is also possible to output the `CPU` counter array (which bytecode was executed how many times, and not only the total number of bytecodes executed) to get a more fine-grained measurement of the execution and could be used as a time analysis factor if a per-bytecode time consumption model is available.

`Energy` (in μJ) is deduced from a per-bytecode energy consumption model [11], i.e., a table containing for each bytecode a corresponding energy cost (aggregating CPU instructions and memory energy costs). An array of 256 `float`s contains the per-bytecode energy consumption which is multiplied with the internal `CPU` counter array to get an estimation of the action `Energy` consumption.

`Time` is the time spent in the given piece of code excluding the time executing higher priority tasks that could have preempted the current action. The implementation of the `Time` counter incrementation is slightly more difficult than `CPU` accounting as threads can be preempted in the VM.

The `Network` resource is not profiled in the VM itself, but at the library level; it basically adds up the number of bytes sent out by the action, in concordance with the `Time` consumption. This resource will e.g. help to detect network congestion, as exposed in the introductory example, when used with blocking protocols as HTTP or TCP.

In KVM, memory usage for primitive types is given as 1 byte for a `byte`, 2 bytes for a `char`, 4 bytes for an `int` or a `float` and 8 bytes for a `double` or a `long`. These primitive types can be encapsulated in `Objects` or contained in arrays. An `Object` in KVM has an overhead of 12 bytes and an array (`arrayStruct`) 16 bytes.

Memory usage of simple data structures can therefore be deduced from the primitive types and overheads and are summarized in Table I. Note that the ceiling part of sizeof(T[x]) is due to data alignment on a 32-bit architecture.

T[x]	sizeof(T[x]) (in bytes)
`byte[x]`	$16 + 4 \cdot \lceil x/4 \rceil$
`char[x]`	$16 + 4 \cdot \lceil x/2 \rceil$
`int[x]`	$16 + 4 \cdot x$
`float[x]`	$16 + 4 \cdot x$
`Object[x]`	$16 + 4 \cdot x$
`double[x]`	$16 + 8 \cdot x$
`long[x]`	$16 + 8 \cdot x$
`StringBuffer()`	$72 =$ `StringBuffer(16)`
`StringBuffer(x)`	$40 + 4 \cdot \lceil x/2 \rceil$
`String(charArray)`	$40 + 4 \cdot \lceil$`charArray.length`$/2 \rceil$

TABLE I
SIZE (IN BYTES) OF SOME SIMPLE STRUCTURES

B. Predictor

The implementation of the predictor was entirely done in Java. It takes as inputs the rpm from the Profiler which is built into the KVM (C code exposing hooks in Java). The main implementation issue was to create an adequate data structure for the prediction array, that could efficiently simulate an array of N dimensions, where $N = NbPar + r$, with the constraint that any dimension, i.e., parameter, must be able to grow and shrink (i.e., changing the minimum, respectively maximum value or increasing, respectively decreasing k) at runtime (see Section IV-A). In fact, the multidimensional array has $NbPar$ dimensions for indexing on every parameter (the size of each parameter array is k_p) and storing r different values for each resource to predict. The implementation of the combining strategies, and their corresponding memory costs are summarized in Table II. The results of the two first columns of this table were generated taking the example of a resource profile with one single parameter having one single interval ($k = 1$) spanning the range $[10..20]$. The numbers given correspond to the whole updating procedure, which consists of finding a cell in the multidimensional array and updating the previous value with a new one using one of the four combining strategies. In this process, the most expensive operation is the localization of the value to change, however since it is the same for all four situations, the only visible differences are related to the costs of the combining strategies themselves. The values in parentheses convey the relative cost compared to the most frugal strategy (OS).

Strategy	Bytecodes	Time [μs]	Memory cost (array)
OS	273 (1)	9.348 (1)	$\mathcal{O}(NbPar + 4r\prod_{p=1}^{nbPar} k_p)$ (1)
AS	279 (1.02)	9.473 (1.01)	$\mathcal{O}(NbPar + 4r\prod_{p=1}^{nbPar} k_p)$ (1)
LPF	283 (1.03)	9.597 (1.02)	$\mathcal{O}(NbPar + 4r\prod_{p=1}^{nbPar} k_p)$ (1)
GA	332 (1.21)	11.565 (1.23)	$\mathcal{O}(NbPar + 8r\prod_{p=1}^{nbPar} k_p)$ (>2)

TABLE II
COMBINING STRATEGIES COSTS (BYTECODES, CPU TIME AND MEMORY)

It is obvious that querying and updating the resource profile before and after the execution of every action has a cost. However the resource overhead added by these operations is not significant compared to the resources exploited while running a typical action. Figure 5 shows the total costs (in bytecodes) for querying and updating the resource profile using the two

extreme combining strategies *overwrite* and *global average*. These results are issued by recording CPU consumption for the following three steps: querying the resource profile for a prediction, executing an action and updating the resource profile after the action is executed. The action taken in the example is the Minimal Spanning Tree (MST) computation algorithm of the JOlden package [3]. The only parameter on which the execution depends is the number of vertices of the graph. This parameter is initialized with the range $[1..50]$ and $k = 10$. Three series of 10 executions are performed with 10, 20 and 30 vertices for each series. The cost implied by a prediction query is around 200 bytecodes whereas the updates are between 320 and 440 bytecodes. The variation in the costs of the updates is due to the selection of different cells ($k = 10$ possible cases) in the array: for each cell, the cost for computing its index is slightly different. Figure 5 clearly shows that the overhead for querying and refining a prediction is much lower than the execution of the action itself.

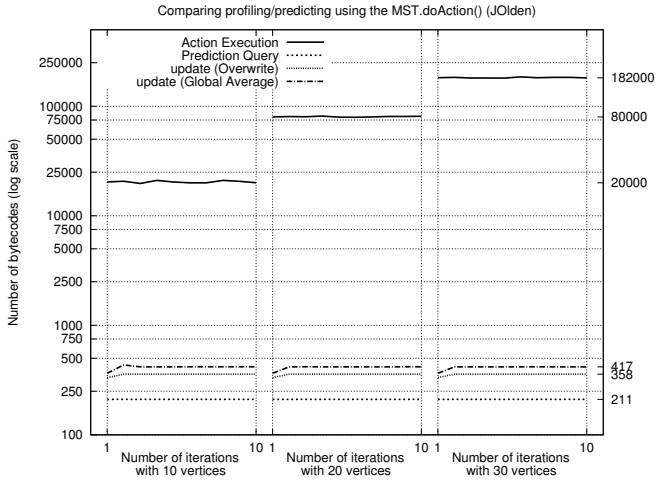


Fig. 5. Costs of operations on the resource profile depending on the combining strategy used.

C. Cost of array adaptation

The previous example is only valid under the assumption that the parameter value of all executions falls into the range initially defined for the parameter. In case the parameter value falls outside this range, strategies for adapting the parameter range need to be used as already discussed in Section IV-A. A similar example as the one presented in the previous section is taken: the resource profile contains one parameter with the range $[1..50]$ but this time $k = 50$ (array with 50 cells for the initial range), thus with $\Delta = 1$. This time, the number of vertices is uniformly distributed in the range $[20..30]$ for the first 20 executions after which max changes to 200. This is done to demonstrate the situation where the parameter value (200) is out of the parameter range $[1..50]$ and thus the array must be adapted. Keeping a precision ($\Delta = 1$) that is constant the array gets extended to 4 times its initial size to have a new range of $[1..200]$. The total cost, in `CPU` of the operation is 5813 bytecodes which thus becomes non-negligible comparatively to the action itself. Other strategies could be used to perform such operations, however either the resource consumption of the system or the quality of the predictions need to be traded-off against one another. Therefore, the adaptation of the parameter range should ideally not occur: parameters should be initialized with reasonable values by the programmer.

D. Memory Footprint Cost of the Weight-Watcher

The modifications done to the KVM, to implement the Weight-Watcher accounts for 16 kilobytes. The modifications in the VM, for adding low-level counters and implementing the energy consumption model [11], i.e., the Profiler, accounts for 2 kilobytes whereas the rest of the addition implementing the Predictor represent 14 kilobytes of Java classes.

VI. Performance

Performance of the Weight-Watcher service is twofold: how slower is the execution of Java programs on top of the modified KVM (adding the native dynamic Profiler) and how accurate are the predictions?

A. Profiler

Using CaffeineMark 3.0 [15] benchmark for embedded devices, the KVM speed decrease caused by the dynamic profiling is going from an overall score of 1159.27 points to an overall score of 1052.27 points, that is a 9.23% decrease in speed. The detail of each test is exposed in Figure 6. Using JGrande 2.0 benchmark [6], the KVM speed slowdown is 6.46% as illustrated in Figure 7(a) for JGrande simple section 1 tests, and Figure 7(b) for JGrande section 2 tests.

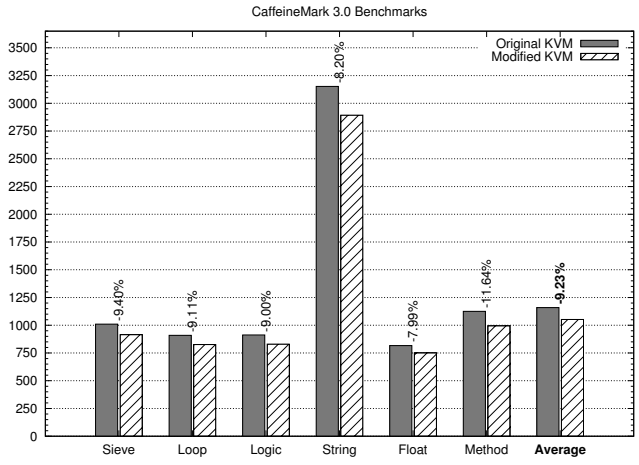
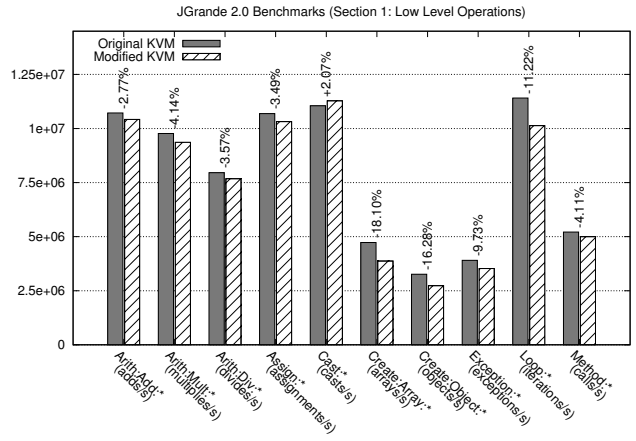
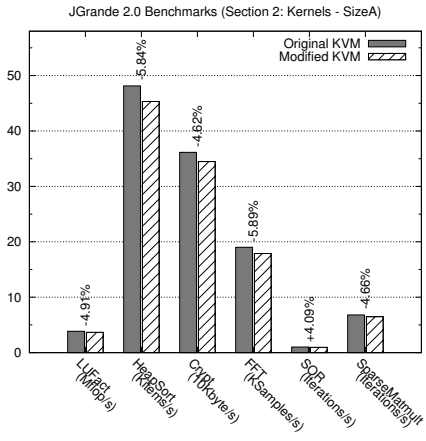


Fig. 6. CaffeineMark 3.0 benchmarks showing relative performance decrease (in points) of the modified KVM compared to the original KVM.

The JOlden [3] benchmark shows a time execution increase of 9.86% as illustrated in Figure 8. For comparison, the averaged performance slowdown introduced by bytecode self-accounting [1], that is for accounting bytecodes only, is centered around 20% [1] and 40% [10], depending on the publication.



(a) Section 1: Low Level Operations



(b) Section 2: Kernels

Fig. 7. JGrande 2.0 benchmarks showing relative performance decrease of the modified KVM compared to the original KVM.

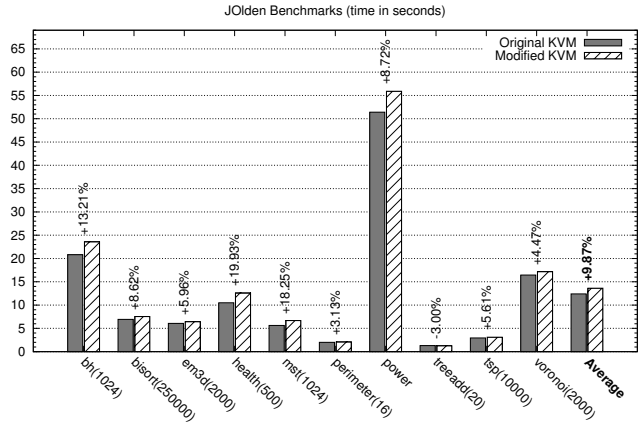


Fig. 8. JOlden benchmarks showing relative time execution increase between the modified KVM and the original KVM.

The overhead of profiling, i.e., keeping track of several integers and incrementing them while the program is executing is relatively low for the following reasons: (a) the cost of accounting is paid in native code (as opposed to [1] where the accounting is done at the bytecode level, after having rewritten the bytecodes), (b) the accounting itself is kept simple (the integers to increment are attached to the running thread, supporting data locality, thus data caching).

B. Prediction Errors

In order to compare the prediction errors exposed in Figure 4 (Section IV-B) and related to the four different combining strategies presented above, a concrete example is done using the Minimal Spanning Tree computation algorithm from the JOlden package [3] as the action. The only parameter used in the resource profile is again the number of vertices (n) of the graph. This parameter is initialized with the range $[10..20]$ and $k = 5$ ($\Delta = 2$). Thirty iterations of the algorithm are done in which n is taken uniformly at random in the range $[10..15]$ and the error is computed at each iteration as $\epsilon = 100 \cdot \frac{|p_{i+1} - rpm_{i+1}|}{rpm_{i+1}}$.The result is presented in Figure 9, showing that each of the combining strategies has a learning phase induced by the number of intervals k of the parameter: there is no prediction ready (p_1 undefined) at the beginning producing a prediction error of 100%. However, as this stage is passed, the different combining strategies can be compared. The *overwrite* strategy is the one producing the highest error peaks since the prediction only depends on the last iteration whereas the error corresponding to the *global average* strategy gives the best results since all of the previous predictions have an influence on the current one. The *low-pass filter* and *average* strategies give similar results.

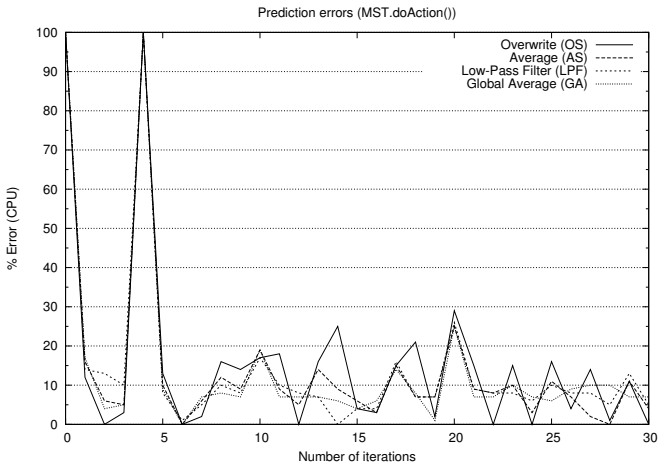
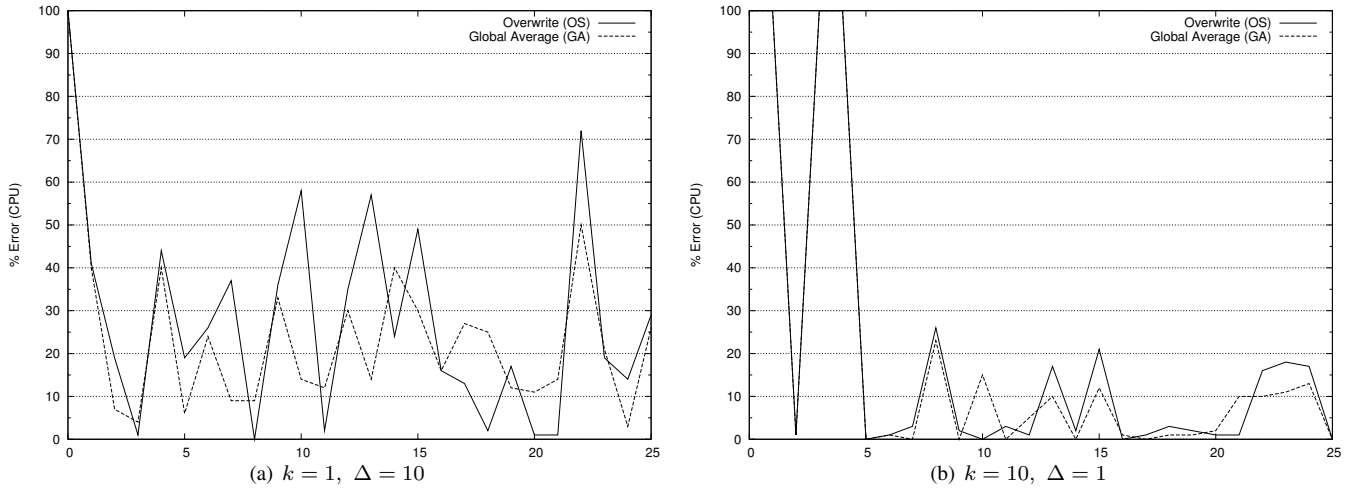


Fig. 9. Predictions are made based on the MST algorithm ($k = 5, \Delta = 2$).

Figure 10 compares the prediction errors of the *overwrite* and *global average* strategies in different setups, $k = 1$ (Figure 10(a)) and $k = 10$ (Figure 10(a)) during 25 iterations for the example described in the previous paragraph. The average prediction error is illustrated in Figure 11 for all four strategies. In this case, as $k = 1$, ($\Delta = 10$) there is only



(a) $k = 1,\ \Delta = 10$ (b) $k = 10,\ \Delta = 1$

Fig. 10. Different prediction errors depending on k (thus Δ) for *overwrite* and *global average* strategies.

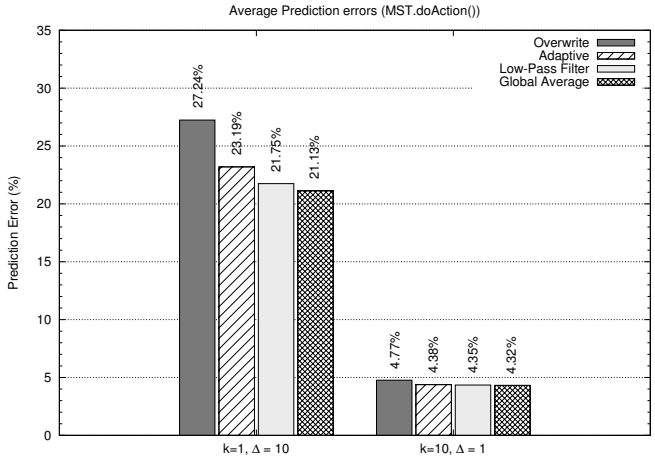


Fig. 11. Average prediction errors (10'000 iterations) depending on k (thus Δ) for different combining strategies.

one integer to store in the prediction array, i.e., a memory cost of 4 bytes. It is also the only value which can be used in next iterations to predict the resource consumption of the system. With this very simple prediction setup, the *overwrite* combining strategy gives errors going up to 70%. In fact, the very worst case, for this strategy is when an execution with $n = 10$ is directly followed by an execution with $n = 15$. The graphs show that the complexity of the combining tasks is proportional to their accuracy, as summarized in Figure 11.

Figure 10(b) shows the prediction errors of the same action, with $k = 10$ intervals of size $\Delta = 1$ ($[min..max] = [10..20]$). A prediction array of size 10 must be kept in memory, which represents 40 bytes. The graph clearly shows that sampling the parameter increases the precision of the prediction drastically. In fact, the only drawback (apart from the increased memory consumption) is that 5 executions were needed to initialize the array (recalling that n is chosen uniformly at random in the range $[10..15]$) for which the error equals 100%, which explains the 5 peaks at the beginning of the graph. An interesting aspect to notice is that one can naturally think that the error should be 0 after the five first steps used for setting the values in the array. However the MST algorithm is not totally deterministic since it uses random numbers to compute distances between edges. Therefore even if two executions are done with the same parameter, their CPU consumption will be close, but not precisely the same, leading to the errors observed even when $\Delta = 1$.

The prediction errors are averaged, for each combining strategy, with 10'000 iterations (Figure 11). The histogram shows (1) that the overall accuracy of combining strategies are proportional to their complexity, and (2), that with a very small Δ, all combining strategies behave comparably.

Global average, in general, is superior in terms of accuracy, which seems somehow negligible when its complexity, both in terms of memory and CPU consumption (see Table II), is compared to the second best combining strategy: *Low-Pass Filter*.

VII. Related Work

Predicting resource consumption of code is hard (in some cases undecidable [2]), and typically requires a lot of resources for its own purpose, thus making it a real challenge on resource-constrained devices. Moreover, the estimations predicted for a piece of code are typically not portable from one device to another and thus can typically not be computed in advance and shared amongst devices: internal object layout and header size are implementation specific, and, above all, battery and CPU consumption of a piece of code are device specific. It is shown, for instance in [4] that there is a strong correlation between the number of bytecodes executed and the elapsed CPU time, but this correlation is application-specific and obviously depends on the given VM/OS/hardware combination. However, in [9], an attempt is made to define a set of portable resource metrics which are converted to platform-specific values thanks to statically computed conversion factors.

Static analysis of memory [2], [5], [8], [18] and time [7], [14] can provide upper bounds of memory, respectively time usage of a given piece of code, providing strong guarantees that the code will never exceed the estimation. Determining memory upper bounds may improve memory management, e.g., for stack-based allocation of dynamic objects, or for creating parametric memory-allocation certificates [2]. Worst-case execution times are key in computing scheduling schemes that satisfy all timing constraints [7]. However, static load-time code analysis is itself very demanding in terms of resources, and is therefore not an ideal candidate for resource-constrained devices, as it may cause significant latencies (i.e., service downtime). In [2] most of the static analysis including the execution of the *core components* (finding creation sites, computing control-state invariants, inductive variables and Ehrhart polynomials) took close to 30 seconds on an Intel Pentium IV 3GHz CPU. In contrast, the approach we consider here consists in loading and executing the code on the fly and performing the analysis at runtime (during the first executions), giving resource consumption approximations after a few executions.

In [1], [2], the amount of memory that is allocated by native code or by the virtual machine itself cannot be measured, respectively estimated. Section V-A shows that applying modifications at the VM level allows to quantify the memory that is allocated by the VM itself, e.g., the overhead of data structures.

The work in [13] is close to ours since the goal also is to make programs change behavior depending on resource predictions. Nevertheless, their predictions are based on (1) desktop linux kernel outputs and (2) history of executions. The first is not targetting embedded devices and the second relies on log files (stable storage) and statistical machine learning, which are both way too resource demanding for the embedded devices we target.

VIII. Concluding Remarks

This paper introduces the Weight-Watcher service. This service aims at providing resource consumption measurements and estimations for software executing on resource-constrained devices. As a consequence, the service enables software components to continuously adapt and optimize their quality of service according to resource availability. We presented an implementation of the service that includes a Profiler (in the KVM) as well a library of Java classes encapsulating resource prediction. The evaluation shows that there is still room for improvement on the implementation of the Profiler. In particular, it could be interesting to include preparation sequences in order to reduce the cost of profiling in interpreted virtual machines. It would also be interesting to precisely capture the trade-off between the memory consumption of the Predictor and the accuracy of the predictions.

References

[1] W. Binder and J. Hulaas. A portable cpu-management framework for java. *IEEE Internet Computing*, 8(5):74–83, 2004.

[2] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, June 2006.

[3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *PACT '01: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, September 2001.

[4] A. Camesi, J. Hulaas, and W. Binder. Continuous Bytecode Instruction Counting for CPU Consumption Estimation. In *QEST '06: (3rd International Conference on the Quantitative Evaluation of SysTems)*. IEEE Computer Society Press, 2006.

[5] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory usage verification for oo programs. In *SAS '05: Proceedings of the 12th Static Analysis Symposium*, pages 70–86, 2005.

[6] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmarking suite. In *Proceedings of the joint ACM-ISCOPE conference on Java Grande*, pages 106–115, Palo Alto, 2001.

[7] C. Ferdinand. Worst case execution time prediction by static program analysis. *IPDPS '03: Proceedings of the 17th International Parallel & Distributed Processing Symposium*, 03:125a, 2004.

[8] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 185–197, New York, NY, USA, 2003. ACM Press.

[9] E.-N. Huh, L. Welch, B. Shirazi, and C. Cavanaugh. Heterogeneous resource management for dynamic real-time systems. In *HCW 2000: 9th Heterogeneous Computing Workshop*, page 287, 2000.

[10] J. Hulaas and W. Binder. Program transformations for portable cpu accounting and control in java. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 169–177, New York, NY, USA, 2004. ACM Press.

[11] S. Lafond and J. Lilius. An energy consumption model for an embedded java virtual machine. In *ARCS '06: Proceedings of the 19th International Conference on Architecture of Computing Systems*, pages 311–325, Frankfurt, March 2006.

[12] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *WMCSA '00: Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications*, pages 31–. IEEE Computer Society, 2000.

[13] D. Narayanan and M. Satyanarayanan. Predictive resource management for wearable computing. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 113–128, New York, NY, USA, 2003. ACM Press.

[14] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.

[15] Pendragon Software Corporation. *CaffeineMark 3.0 for Embedded Devices*. http://www.benchmarkhq.ru/cm30/info.html.

[16] M. Satyanarayanan and D. Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7(6):601–607, 2001.

[17] Sun Microsystems. *J2ME Building Blocks for Mobile Devices, White Paper on KVM and the Connected, Limited Device Configuration (CLDC)*, May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[18] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized live heap bound analysis. In *VMCAI '03: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–85, London, UK, 2003. Springer-Verlag.

[19] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.