Scavenger: Transparent Development of Efficient Cyber Foraging Applications

Mads Darø Kristensen
Interactive Spaces, Aarhus University
Aarhus, Denmark
Email: madsk@cs.au.dk

***Abstract*—Cyber foraging is a pervasive computing technique where small mobile devices offload resource intensive tasks to stronger computing machinery in the vicinity.**

This paper presents Scavenger—a new cyber foraging system supporting easy development of mobile cyber foraging applications, while still delivering efficient, mobile use of remote computing resources through the use of a custom built mobile code execution environment and a new dual-profiling scheduler. One of the main difficulties within cyber foraging is that it is very challenging for application programmers to develop cyber foraging enabled applications. An application using cyber foraging is working with mobile, distributed and, possibly, parallel computing; fields within computer science known to be hard for programmers to grasp. In this paper it is shown by example, how a highly distributed, parallel, cyber foraging enabled application can be developed using Scavenger. Benchmarks of the example application are presented showing that Scavenger imposes only minimal overhead when no surrogates are available, while greatly improving performance as surrogates become available.

***Keywords*-Distributed systems, pervasive computing, mobile environments**

I. Introduction

Cyber foraging was introduced by Satyanarayanan [1] in 2001 and further refined by Balan *et al.* [2] in 2002. The term covers the opportunistic use of available computing resources by small, mobile devices, and as such is applicable for all kinds of resources that *surrogate* computers, i.e., the stronger computers offering up their services, may be in possession of. This could be resources such as CPU power, storage, network connectivity, display capabilities, printers etc. In this paper, and in the Scavenger system presented herein, the focus is on foraging for CPU power.

Mobile computing devices are increasingly common—in a matter of years almost everyone will be carrying some kind of smart phone. While these devices are becoming more powerful with regards to processing power, the limiting factor remains battery capacity. A modern smart phone may be fully capable of performing quite resource intensive tasks, e.g., simple image manipulations such as sharpening an image, but doing so is still quite slow and it drains the energy of the device rendering it unusable in a mobile setting. Such tasks can be solved efficiently by utilising cyber foraging to offload the CPU intensive work to nearby surrogates.

Consider the following use-case: A tourist is sitting in a café going through the pictures she has taken earlier in the day. The pictures were taken using the multiple megapixel camera in her smart phone, and she is browsing them to select the ones that she wants to upload to her online storage account, so that her friends and family may see them. Before uploading them she applies some filters to them—some need sharpening, others red-eye reduction, and yet others may need their brightness/colour/contrast adjusted. All of these operations are applied only on small previews of the photographs on her smart phone, but when she presses the "apply" button cyber foraging is used to apply the image operations to the actual images. Her mobile device automatically scans its environment, finding a couple of surrogates provided by the café and some laptop computers owned by other customers, and quickly offloads both the image operations and the uploading of the resulting picture to these devices, leaving her phone free for her to use—and, more importantly, leaving her phone's battery at an acceptable level so that she may use it for the rest of the day. If no surrogates are available at the café, her mobile device will ask her to choose between performing the operations locally or queueing the operations until later on as surrogates become available.

In a scenario such as this one, cyber foraging is not only a benefit—it is almost a necessity. As will be shown in the evaluation in Section V, asking a current smart phone to apply just three image operations to a single five megapixel image takes more than two and a half minutes, whereas performing the same operations using cyber foraging can be done in less than 17 seconds, and that result is obtained using relatively resource poor surrogates.

As noted by Porras *et al.* in [3], a number of software components must be in place in order to realise a scenario such as the one described above. Firstly, a discovery mechanism must be in place, so that the mobile devices may discover and receive information about available surrogates. Secondly, a way of partitioning an application into local code and remote executable tasks must be devised; preferably in a way that minimises the burden on the application developer. Thirdly, a scheduler capable of doing cost assessment is needed so that the tasks are performed in the right place—that place being either locally or at one or more of the available surrogates. Finally, some way of actually performing the identified tasks on a remote host is needed; and preferably this should work on any surrogate regardless of architecture

978-1-4244-5328-3/09/$25.00 ©2009 IEEE

and software availability. I.e. some kind of mobile code approach should be employed.

This paper presents the Scavenger system and discusses how it approaches these challenges. The key contributions of Scavenger presented in this paper is 1) the cyber foraging system as a whole, 2) a novel dual-profiling scheduler enabling intelligent use of resources even in unknown/unprepared environments, 3) a mobile code approach towards task distribution and corresponding execution environment for mobile Python code, and 4) a development model that makes the process of creating cyber foraging enabled applications transparent to the applications programmer.

Having a cyber foraging system capable of performing all of these complex tasks is all well and good, but it is futile without an easy way for application programmers to utilise the system. Cyber foraging entails distributed, and in many cases parallel, computing; two fields within computer science that are known to be hard for programmers to fully grasp. Furthermore, in order for cyber foraging to be really usable, the system should be capable of using mobile code so that functionality may be pushed onto surrogates as needed—thus adding another level of complexity to application development. In previous cyber foraging systems [4]–[7] the burden put on the application programmer is quite heavy. Scavenger, on the other hand, is designed to be easy for programmers to use, and in this paper it will be shown by example how highly mobile, distributed, parallel applications may be built, without putting a heavy burden on the application programmer.

The structure of this paper is as follows: In Section II related work is discussed. Section III presents the Scavenger cyber foraging system with special focus on the benefits of using a mobile code approach, and on how scheduling is done using the dual-profiling scheduler. Section IV demonstrates how easily a cyber foraging enabled application may be developed using Scavenger, Section V shows some experimental results using the developed prototype to show that the application does indeed work satisfactorily, and to show off some of the aspects of the scheduler. Finally, the paper is concluded in Section VI.

II. Related Work

In the years that have passed since the introduction of the term cyber foraging, a number of different systems have been developed within academia. These systems have had very different foci with regards to e.g., level of mobility, development model, and deployment method—some using per client virtual machines [6], [7], others relying on pre-installed services reachable via RPC [4], [5], and yet others using a more flexible mobile code approach [8].

Some of the earliest examples of cyber foraging systems were the twin systems, Spectra [5] and Chroma [4], developed at CMU. Both of these systems used pre-installed RPCs to perform remote tasks, meaning that the mobility of the client devices were restricted to areas where the needed functionality was already installed. The mobile code approach in Scavenger alleviates this need to prepare surrogates beforehand. Spectra and Chroma both have schedulers capable of choosing between available surrogates by continuously updating and consulting a history based profile that is maintained for each known surrogate—an approach that works well considering the limited mobility of their system. In this paper a novel dual-profiling history based scheduling approach is presented, and it is described how this approach yields good performance in both known and unknown environments. Another difference is that Spectra and Chroma seem to be working with only one profile per task, whereas Scavenger works with two-dimensional profiles to account for the fact that task complexity may vary with input size or value. The developer creating applications using Spectra and Chroma has to manually partition the application into remote executable RPCs (that have to be pre-installed on the surrogates), and on run-time the scheduler will tell the application where to perform the different tasks. In Scavenger the amount of rewriting needed is kept at a bare minimum; all that is needed is that remote executable functions, or tasks, are annotated as such—the rest, both installation and invocation, is then handled by the Scavenger library.

Some systems, e.g., the system by Goyal and Carter [7] and Slingshot by Su and Flinn [6], have experimented with using OS-level virtual machines (VMs) on surrogates, giving a client access to an entire VM hosted at a surrogate. While this approach is as flexible as using mobile code when it comes to deployment, the mobility of such systems is naturally rather low, seeing as an entire VM must be spawned and any needed functionality fetched and installed *before* any cyber foraging can take place. Scheduling of tasks is non-existent in these systems; as soon as the client application receives access to the virtual machine, it is left up to the application itself how this resource should be utilised. The same goes for development; the systems provide access to a VM, and the development model, i.e., how the application communicates with and uses this VM, is application specific.

The systems have different approaches when it comes to developer support; ranging from completely automatic approaches that study an application's execution history graph and offloads parts of the application based on this. This is the approach taken by Hunt and Scott in the Coign system [9] and by Messer *et al.* [8]. Other systems, as described above, are at the opposite extreme, offering nothing more that access to a virtual machine. Finally, some systems have taken the middle ground offering some middleware support for cyber foraging, where the developers have to adhere to a specific API. This approach it taken by Spectra, Chroma, and Scavenger. This is done because, as noted by Flinn

et al., "*a little application-specific knowledge can go a long way.*" [5]; meaning that although it is in some cases possible to employ cyber foraging without rewriting an application, the addition of distributed and parallel capabilities may very well mean that some parts of the application can be solved more efficiently by making very small rewrites, e.g., by unrolling a loop into a bunch of parallel tasks.

As noted by Paluska *et al.* in [10], applications operating in a pervasive computing environment must be ready to adapt to the ever changing resource levels. This adaptation must happen at run-time, since the application programmer can not be expected to predict all possible resource combinations, that the application will have to function in. The approach taken by Paluska *et al.* is to create a high-level decision making system where the developer of the application states the *goals* of a given task, and then a number of *techniques* capable of reaching that goal by using available resources as provided. The set of techniques are extendable at run-time, so that new kinds of resources may be used. In Scavenger, working in the limited domain of offloading CPU intensive tasks, a mobile code approach has been taken, meaning that resources may be "taught" how to perform a task, alleviating the need for resource specific adaption.

Cloud computing is an area of research that is closely related to cyber foraging. Using cloud computing computationally intensive work can be pushed over the Internet into "the cloud", where the tasks may be scheduled in a data centre using known cluster computing techniques. This removes the burden of resource discovery and scheduling from the client device and thus simplifies the process of using remote execution considerably. Cloud computing is not always a good solution to the problems that cyber foraging aims to solve though, as noted by Satyanarayanan *et al.* [11]. Because of the high latencies of a WAN link when compared to a wireless LAN connection, and because of the fact that the bandwidth of a wireless LAN is typically two orders of magnitude higher than the wireless Internet bandwidth available to a mobile device, cloud computing is not always feasible. In an application such as the one described in the introduction of this paper, the high latency of a WAN link would mean that a considerable delay would be incurred if applied to the operations on the smaller preview versions of the images. And, if applied to the larger, original versions of the images, the bandwidth limitations would mean that the time spent transferring data would by far outweigh the benefits of using remote execution.

III. The Scavenger Cyber Foraging System

Scavenger consists of two independent software components: the daemon running on surrogates enabling them to receive and perform tasks, and the library used by client applications. It is possible for a device to be running both the daemon and applications using the client library, thus enabling the use of co-operative cyber foraging, where devices work together for the greater good.

Both the client library and the daemon are written in Python, and the execution environment, which is written using Stackless Python, is capable of performing mobile Python code. Stackless Python is used exclusively on surrogates, so client devices need not support Stackless. Regular Python support is needed on client devices, which is supported on all major PC platforms and also by a large number of mobile platforms. We currently have Scavenger demonstrators running on Nokia's Maemo based devices, Apple's iPhone, HTC's Android phones, netbooks etc.

In Section III-A the architecture of Scavenger is briefly described, Section III-B describes the use of mobile code in Scavenger, and Section III-C discussed the challenges posed when scheduling in a highly mobile, heterogeneous environment.

A. Architecture

A high-level view of Scavenger's architecture is depicted in Figure 1. A surrogate must install and run the Scavenger

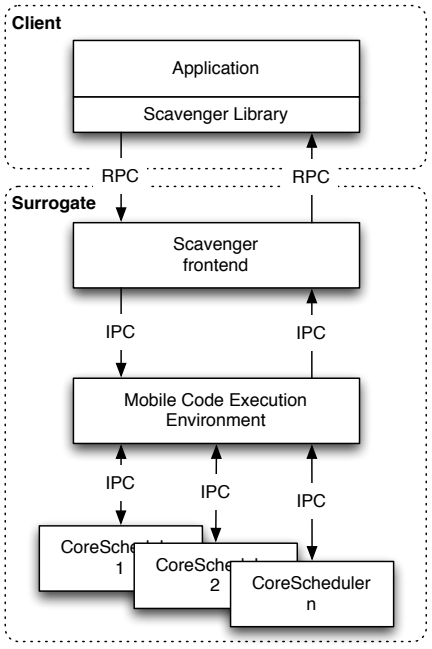

Figure 1. A high-level view of Scavengers architecture.

daemon, and in order to use such resources an application merely needs to include the Scavenger library and adhere to some simple rules—as will be described shortly.

The daemon consists of a small front-end offering remote access to the mobile code execution environment through some RPC entry points. This front-end is also responsible for device discovery, which it does by using the Presence service discovery framework [12], that has been developed

especially for use in cyber foraging settings, where up-to-date information about available resources is critical. Discovery is used not only by client devices; surrogates also collect information about other surrogates available in the environment. For now, this information is used when surrogates fetch intermediate results from each other, but in the future this information could be used for re-scheduling and task migration, enabling the surrogates to hand over tasks to other surrogates. As a part of the Scavenger library clients also use Presence to discover available surrogates.

All client applications must use the Scavenger library. This library offers two ways of working with cyber foraging: 1) a manual mode, where the application may itself ask for a list of available surrogates, install code onto these surrogates, and invoke this code.; and 2) a fully automated mode, where the application programmer only needs to annotate his remote executable functions, and then Scavenger will take care of the scheduling. In this paper the focus is on development using the automatic mode, and on the kind of scheduling done using this mode.

B. Mobile Code

Behind the Scavenger front-end lies the mobile code execution environment that offers dynamic installation and execution of Python code. Using this, task deployment in Scavenger becomes the simple task of 1) checking whether the needed functionality is already installed on the surrogate, and 2) installing it before invoking it if necessary. This is handled automatically by the Scavenger library. If true mobility is to be supported, using mobile code is a necessity. Using pre-installed tasks only works in the limited cases where all needed tasks have already been installed on all potential surrogates, and using virtual machines is far too heavyweight if it is to be used in a mobile environment, where the user may only be within range of a surrogate for as few minutes at a time. By using a trusted mobile code environment, such as the one Scavenger provides, it is possible to use pre-installed functionality, if such exists, and if the needed functionality is unavailable it may simply be installed on-demand by the mobile clients.

Scavengers execution environment is designed to fully utilise multi-core machines. A Scavenger daemon can be configured to use any number of cores, and for each core a core scheduler is spawned that handles tasks performed on that specific core. Tasks are, in the current implementation, submitted to these core schedulers in a round robin manner. That the decision of how many cores to utilise is a configuration parameter, means that a user running a Scavenger daemon is capable of limiting the amount of resources to dedicate. Thus a laptop with two cores can be offering up its services as a surrogate, and at the same time be completely usable by its local user because only one core is used by Scavenger.

While the execution environment supports installation of code, it is not a mobile code environment in the sense that code must always be given along with task input, in order to perform a task. In many cases a client using a surrogate will be interested in invoking the same task multiple times, using different input data for each invocation. Furthermore, if multiple client devices are running the same application, the tasks defined within will be the same, and it thus makes sense for the surrogates to store these tasks for future use once they have been installed. This is handled in Scavenger by giving each task a unique identifier, and when invoking a given task the client initially asks the surrogate whether that task is already installed. These task identifiers *may* be defined manually by the task developer, but in most cases they will be generated automatically by the Scavenger library, and to avoid naming clashes the MD5 sum of the actual task source is part of the auto-generated identifier.

Working with mobile code has a lot of implications to security—especially in a cyber foraging scenario, where untrusted clients must be allowed to install and run their own code. To make sure that malicious clients may not access the private data of a surrogate computer, or use a surrogate to launch attacks on other computers on the network, a number of measures have been taken in Scavenger to secure the execution environment. Before mobile code is accepted into the execution environment, it is validated using both a black-listing and a white-listing approach. Built-in language features that are deemed dangerous, such as many of the reflective features of Python, are black-listed, and module imports are white-listed, thus only allowing code that imports trusted modules from the standard library. More measures are in place to secure the execution environment, but further details are outside of the scope of this paper; see [12] for more information.

C. Scheduling in a mobile, heterogeneous environment

Task scheduling in a mobile, heterogeneous environment, such as the pervasive computing environment targeted by Scavenger, is a very complex operation. Many factors must be considered in order to select the right place to perform a given task. The “right” place in this case being either locally or at a surrogate, and, if remote execution is chosen, the current best of the available surrogates must be chosen.

Consider the pervasive computing environment depicted in Figure 2.

An example ad hoc network is shown in this figure—such as one that may emerge between the devices of customers in a café. As this example shows, there is a great heterogeneity of devices with three different brands of smart phones, two laptop computers, a PDA, and even a stationary PC. Scavenger, utilising the cross platform deployable Python language, runs on all of these platforms. All devices in this scenario may act as both surrogates and clients in a pure peer-to-peer fashion, but in most cases users of mobile

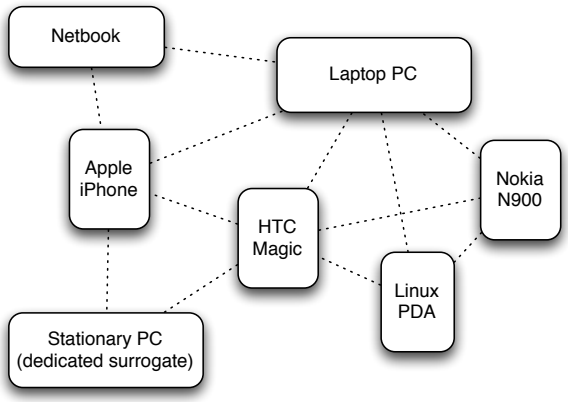

Figure 2. An ad hoc networked pervasive computing environment within which cyber foraging could take place.

devices such as smart phones and PDAs will not be willing to share resources because of severe battery life limitations.

Given a collection of potential surrogates, a number of factors must be taken into consideration, when considering where to place a task. 1) The relative speed and current utilisation of the surrogates, 2) network bandwidth and latency to the surrogates, 3) task complexity, and 4) in- and output size and possibly also input location. Some of these factors may be measured, some may be considered static, while yet others have to be specified by the application programmer.

The scheduler within Scavenger takes all of these factors into account when deciding where to place a given task. Talking about the "strength" of a device with regards to performing CPU intensive tasks is tricky. Due to the heterogeneous nature of the devices participating in cyber foraging, the architectural differences mean that mere clock speed is no good as a measure for performance. To accommodate for this Scavenger uses a CPU benchmarking suite to score both surrogates and clients. The suite in use is nbench[1], and the score yielded by this benchmark is used as a strength indicator. As for the current utilisation, in Scavenger this is expressed as the number of tasks currently running on the surrogate, and this information is included in the periodically emitted discovery packets.

As for information about the network connections between the client and the surrogates, and between surrogates as well, this is statically defined per device in Scavenger. I.e., both clients and surrogates include information about the network media they are using in their discovery packets. Even though the devices must be on the same network in order to see each other, they may not necessarily be using the same network media; e.g., in a managed Wi-Fi scenario, where some surrogates may be connected to an access point using a wired connection, while the mobile devices use an IEEE 802.11 wireless connection. Using information about the network media in this case, the scheduler knows that it can quickly send intermediate results between the wired surrogates, while it has to pay a higher price if the wireless network is traversed. Static configuration of network bandwidth has been chosen because of the inherent difficulty in continuously measuring bandwidth without imposing severe amounts of traffic on the network.

Using the information about expected bandwidth of interconnecting links, the scheduler considers the cost in time of using remote execution; i.e., the time that must be spent transferring in- and output data if a remote surrogate is chosen. Since scheduling happens at run-time, information about input size is readily available, but information about output size needs to be given by the application programmer. How this is done in Scavenger is described in detail in Section IV.

Whenever a task is handed over to the scheduler, it initially inspects the task to decipher information about task complexity. Task complexity in this context is not the big O asymptotic running time of the task; is the expected running time of the task when run on a specific surrogate—not something that can be estimated by merely looking at the task code. Real runs of the task is needed to get such information, i.e., profiling data must be available. Profile data is collected in Scavenger whenever a task is performed, so given a task identifier the scheduler can take a look in its local profile to find the estimated running time of that task. Two profiles are maintained for each task: the task-centric profile, where a globally applicable *task weight* is stored, and the peer-centric profile, where a profile is maintained for each (peer, task)-pair that has been encountered.

Scavenger's dual profiles are one of the main contributions of the system. Peer centric profiles, that store information about how exactly that peer performs with regards to a specific task, are no doubt the most precise way to profile task complexity. But when working with a highly mobile scenario, where surrogates with high probability are unknown to the mobile client before use, being able to make an informed guess about how unknown surrogates will perform given a task is of high value. As mentioned, Scavenger's scheduler maintains a history based, peer centric profile, and this profile is the first that is contacted when profile data about a (task, peer)-pair is needed. In the likely case that such peer specific information is unavailable, the task centric profile may be consulted. The task centric profile is also a history based profile, and it contains the assessed task weight of the given task. This task weight is calculated as shown in Equation (1); where $T_{duration}$ is the duration of the task in seconds, $P_{strength}$ is the aforementioned peer strength (nbench rating), and $P_{activity}$ is the number of other tasks running on the same core during execution.

[1]http://www.tux.org/~mayer/linux/bmark.html

$$T_{weight} = T_{duration} \times \frac{P_{strength}}{P_{activity}} \quad (1)$$

Task duration is measured in seconds and by scaling this with the expected strength of the surrogate while performing the task a measure, that can be used to reason about expected running time on other surrogates, is obtained. The assumption here is e.g., that a task that takes one second to perform on an idle surrogate with an nbench rating of 40, should take about two seconds to perform on an idle surrogate with a ration of 20. That this assumptions holds to a satisfactorily degree, can be seen in Table I.

	2 GHz G5	733 MHz G4	1 GHz Pentium 3	900 MHz Celeron M
Brightness	50.980	50.188	54.600	52.755
Colour	54.012	53.605	57.223	54.249
Contrast	54.946	55.100	59.229	54.723
Sharpen	109.859	126.243	106.860	98.404
Blur	82.545	95.976	83.081	73.399
Invert	21.208	27.360	32.347	28.038
Scale	91.072	102.974	44.648	35.300

Table I
TASK WEIGHT MEASUREMENTS. IDEALLY VALUES SHOULD BE EQUAL ACROSS ROWS.

To produce the data in Table I four different machines performed seven image manipulation tasks and reported the weight (using the metric in Equation (1)) they would assign to that task. All tasks were performed 50 times and the weight reported is the average of these runs. The machines in use are very different with regards to processor architecture; having both a PowerPC G4 and G5, an Intel Pentium 3, and an Intel Celeron M processor. Even with those differences in architecture, it can be seen that the assigned weights are quite similar, and when used in a history based profile these provide a good starting point for the scheduler, that can be used when no (peer, task) specific knowledge is available. Notice though, although the different machines tend to agree on the weight of most of the tasks, in some cases the architectural differences shine through. Consider for example the last row, where an anti-aliasing scale operation was performed. In these tests the PowerPC based (G4 and G5) machines reported weights that were twice or almost three times as high as the Intel based machines. This shows that while using this "task weight" based on benchmarking scores does in most cases provide good results, it is no silver bullet. Compared to having no knowledge at all about the running time of task execution on unknown surrogates, it makes for a more informed scheduling in unknown environments.

Apart from using dual profiles, the profiles themselves are also two-dimensional. Task complexity is in most cases not a constant; it may vary with input size, value, or both. In Scavenger this is reflected by using two-dimensional profiles. When writing a task the developer is asked to describe which input parameter(s) determine the complexity of the task. This can be written as an expression combining input parameter values and/or sizes, and at run time this expression is evaluated to yield a single value, v. Using v, which could e.g., be the size of an input file, Scavenger maintains its two-dimensional profiles by inserting the collected profile data into the matching interval. Updating the profile is done using the following simple algorithm:

1) If this is the first run, simply create a new bucket for v and insert the profile data (task weight) there.
2) If buckets exist, find the bucket numerically closest to v and compare the profile data to that bucket's average:
   a) If the profile data differs less than a certain percentage insert it into this bucket.
   b) If the variation in profile data is too large create a new bucket for the data if, and only if, v also differs more than a certain percentage from the interval covered by the bucket.

By updating the profile in this way the profile data is capable of adjusting to variations in complexity, while only maintaining as few buckets as possible. Consider the depiction in Figure 3.

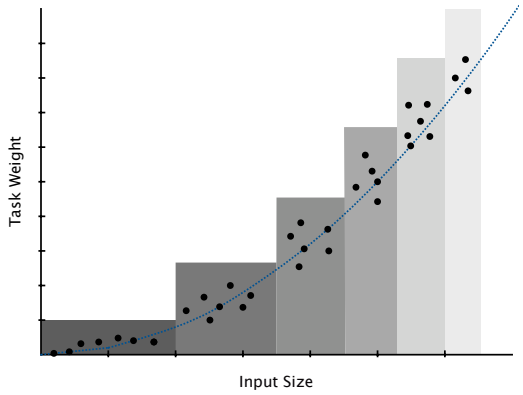

Figure 3. Two-dimensional profile. The shaded boxes are the buckets created, the dotted line is the predicted running time, $O(n^2)$, of the task, and the data points (dots) are actual weight measurements.

When the complexity of the task rises fast many buckets will be created to reflect this, and when the increase is slow only very few buckets are maintained. When doing lookups in the profile the bucket with the value closest to the current v is chosen.

IV. DEVELOPMENT USING SCAVENGER

The application that will receive cyber foraging capabilities in this section is the image browser/editor for mobile devices described in the introduction. Operations such as the ones supported by this application are quite heavy for modern mobile devices to perform, but they are capable of performing them on their own, when no surrogates are available. In the following it will be shown how is to

add use of cyber foraging to an existing application when using Scavenger. It is therefore assumed that the application has already been developed and is running on the mobile platform of choice, and what will be shown here is merely the changes needed to apply cyber foraging.

As mentioned in the previous section, the Scavenger library has two operating modes: manual and fully automatic. In this example only the automated mode will be demonstrated. The automated mode works by the use of Python *function decorators*. Decorators in Python are higher order functions that wrap the function they are attached to, so that when the decorated function is called the decorator is called first. Using this built-in language feature adding cyber foraging can be as simple as adding a decorator to a function that may benefit from remote execution. There are a few rules though, that must be adhered to for the remote execution to work:

1) The function must be self-contained, i.e., it must not call other functions or methods defined elsewhere in the application.
2) Modules used from within the function must be imported within the function itself, so that these modules may also be imported and used at the surrogate.

A small example function adhering to these rules is shown in Figure 4.

```
1  @scavenge
2  def sharpen(image, factor):
3      from PIL import ImageEnhance as IE
4      factor = 1.0 + factor
5      return IE.Sharpness(image).enhance(factor)
```

Figure 4. A simple task defined using a Python decorator.

In this figure the function "sharpen" is decorated using the "scavenge" decorator. This decorator enables use of automatic cyber foraging whenever the sharpen function is called. What the decorator does is to 1) fetch the function source, 2) generate a unique id for that function by hashing the code into a MD5 sum, 3) ask the scheduler to schedule the task immediately. The scheduler then 4) checks whether surrogates are currently available, 5) chooses the most eligible surrogate (which may in fact be the local device), 6) installs the function code onto the chosen peer if needed, 7) performs the task on the peer returning the result.

All of these steps are completely automated by using the scavenge decorator, leaving the developer free to worry about other, non-distributed parts of the application. The functions used must be self-contained, and this is reflected in the example as import statements are done within the function body, whereas the regular Python coding style would be to have these import statements in the top of the source file. Also the function body does not call external function, apart from the functionality imported from the Python Imaging Library (PIL), which is a standard module found on all surrogates. That the function must be self-contained does not mean that developers may not define new functions and classes—as long as these functions and classes are defined within the function body of the decorated function it will work just fine.

The scavenge decorator can be used in this simple manner, but in order to give the scheduler optimal working conditions, some arguments should be given to the decorator: an expression relating the output size to the size and/or value of the input parameters, and an expression designating which input parameter(s) are determining for the complexity of the task. In Scavenger these arguments can be given directly to the decorator as two strings. Within these strings the input parameters of the function may be referred to by position; e.g., the expression "$len(\#0)$" would refer to the length of the first input parameter. Anything expressible in a Python expression can be written into these strings. This gives a very powerful mechanism for expressing how output size and task complexity relates to the size, value, or any other property of any number of input parameters.

Apart from these arguments one more optional argument is accepted by the decorator. This argument is a simple boolean value; if set to true the result data from the task is left at the surrogate, and only a remote data handle is returned. If set to false, the result data is fetched and returned to the client application. These remote data handles may be used as input parameters for future task invocations, and the scheduler will then take data locality into account when scheduling the task. Working with remote data handles is completely transparent to the developer. Wherever a task input is needed a remote data handle may be given in its place. The Scavenger library will detect this automatically and schedule accordingly. Using these arguments the full implementation of the sharpen task is shown in Figure 5.

```
1  @scavenge('len(#0)', 'len(#0)', True)
2  def sharpen(image, factor):
3      from PIL import ImageEnhance as IE
4      factor = 1.0 + factor
5      return IE.Sharpness(image).enhance(factor)
```

Figure 5. The actual source code of the sharpen task in the image editor.

In the image editor a chain of image operations are often applied to the same image. The last parameter of the decorator has therefore been set to true, to keep the result data at the surrogate for use when the next task in the chain is scheduled.

It is that simple to add cyber foraging capabilities to an application using Scavenger's automated mode. The client application is completely oblivious to whether or not remote execution is performed, it just uses a self-contained function to do its resource intensive work, and the Scavenger library takes care of the rest. As for parallel usage of resources,

a client application should handle that in the usual way, by spawning worker threads to handle multiple concurrent tasks. The Scavenger library is completely thread safe, and is capable of scheduling multiple tasks in parallel over multiple surrogates. And, if no surrogates are available, the heavy work will be done within these worker threads on the client device.

Using the scavenge decorator, a cyber foraging enabled image editor prototype for Nokia Internet Tablets has been developed. All that needed to be changed in the application to incorporate Scavenger was to add the decorator to twelve image operations, other than that the application is left unchanged. A screen shot of this application is shown in Figure 6.

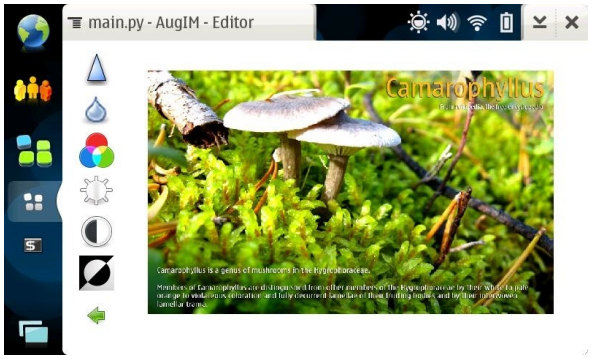

Figure 6. The Augmented Image Manager (AugIM) demonstrator.

The source code for this demo application can be found alongside the Scavenger source. The next section offers a brief look at the performance benefits that can be reaped by using Scavenger.

V. Experimental Results

Extensive benchmarks and experiments have been performed to measure the performance increase gained when using Scavenger, and also to validate the scheduling approach when compared to a handful of other schedulers. Reporting on all of this experimental work is outside of the scope of this paper; instead this section focuses on the performance increase gained when using Scavenger in the mobile image editor developed in the previous section.

For these experiments the client application was installed on a Nokia N800 Internet tablet, sporting a 320 MHz ARM CPU. The surrogates in environment are all reachable by the client over an IEEE 802.11b (Wi-Fi) network connection, and the surrogates are a PowerMac with a 2 GHz G5 CPU running OS X Leopard, an Ubuntu Linux desktop PC with a 1 GHz P3 CPU, and an Asus Eee netbook with a 900 MHz Celeron-M CPU. These surrogates are all fairly slow compared to modern machines, but when compared to the mobile device they are lightning fast, and, as will be shown shortly, using even old machines like these as surrogates yields big benefits for the mobile devices utilising them.

For the first benchmark a linear string of three image operations, expressed as Scavenger tasks, are performed. The operations are an image sharpening, colour adjustment, and finally a contrast adjustment. These operations are performed both on the original five megapixel image and on a thumbnail version, roughly half a megapixel in size, which is the one shown in the UI of the mobile application. All tests have been run 50 times and the measurements reported here are the averages of these runs. The test was initially run without surrogates to measure the local execution time, and then with each surrogate turned on in turn. If all surrogates were turned on at the same time, the scheduler would soon find that the PowerMac was the strongest, and all tasks would be sent to that same surrogate. The results of the test are shown in Figure 7.

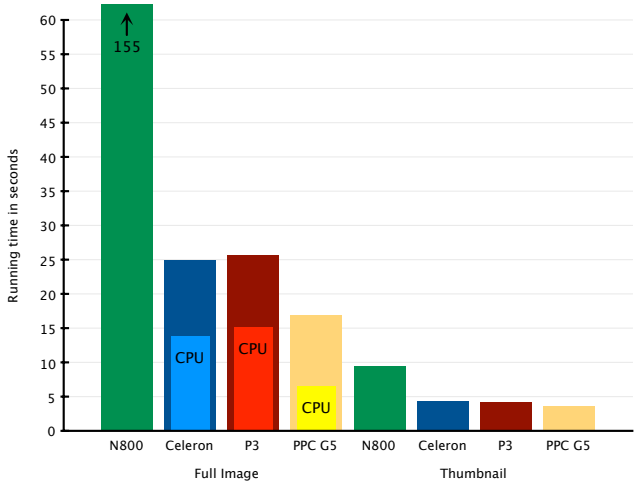

Figure 7. Results of the first benchmark. Each bar shows the total running time as experienced by the client when performing the task. Within three of the bars a lighter bar is depicted which illustrates the fraction of the time spent doing CPU bound work.

A number of things can be concluded looking at this figure. For one it is immediately clear, that employing cyber foraging is essential when working on the original, five megapixel image. If the mobile device itself tries to perform the tasks it takes more than two and a half minutes—time in which the mobile device is left completely unresponsive because its CPU is 100% utilised. In fact, even performing the operations on the thumbnail version if the image is quite resource intensive for the mobile device, using up around nine seconds to perform all three operations. This is detected by the scheduler, and these preview operations are thus also forwarded to surrogates when available. The bars depicting the running time of handling the original image when using a surrogate all have a lighter shaded bar within them. This illustrates the amount of CPU time spent actually performing the operations on the surrogate, and the rest of the running time can be accredited to network overhead. This network overhead of roughly eight to ten seconds cannot be

reduced further by adding stronger surrogates, which sets a hard lower bound on the obtainable benefits when working with such relatively large in- and outputs. In a setting where e.g., an IEEE 802.11g connection were in use, this network overhead would of course be brought down and stronger surrogates would be more beneficial.

The second benchmark discussed here shows off the possibility of using multiple surrogates in parallel; again without having to do anything special on the client side, as this is handled entirely by the Scavenger scheduler. In this test the same image manipulation operations are performed, but this time the operations on the original image have been queued because no surrogates were initially available. Now the mobile device has entered an area with surrogates, and the queued tasks are thus scheduled. In this test all three surrogates are on-line, and what will be shown in the test is that the tasks will be spread out between the surrogates so that their resources are used in parallel.

The test has been run 50 times, and in almost all of these runs the division of labour was like so: the first two tasks were given to the strongest surrogate, the PowerMac, for processing because this surrogate is a little more than twice as fast as the others. The last task would then typically be given to one of the two remaining surrogates, with the P3 Linux desktop handling most of these because of it being slightly faster than the Eee netbook. After scheduling these first three tasks, the tasks doing sharpening of the image, the remaining tasks, doing colour and contrast adjustments, were almost all scheduled at the surrogate holding their data because of the data locality measures in the scheduler. That not all colour and contrast tasks were placed at the surrogate holding the input image, can be accredited to the fact that the PowerMac and the Linux desktop PC are connected by a 100 MBit LAN. The scheduler knows this, and the calculated overhead of moving the input data between these two machines is therefore quite small, leading to schedules where it may make sense to move task processing from one surrogate to the other.

The results of running these benchmarks show that the average running time of performing these tasks is about 34 seconds. Looking at the results from the previous test, it can be seen that performing the tasks in serial instead of in parallel, e.g., using the strongest surrogate, would take about 50 seconds. Using multiple surrogates in parallel has thus had the expected positive effect on the running time, whilst the developer has been kept blissfully unaware of the complicated details. The benefits of using more surrogates will increase with the number of concurrent tasks, e.g., if more than three tasks were concurrently scheduled in this environment all three surrogates would be used.

The experiments presented are but a small selection from a large set of experiments done within the Scavenger system. These tests are based on the simple image editor described in Section IV, and they show how easily serious performance benefits can be obtained using the Scavenger cyber foraging system.

Apart from the obvious benefit of having increased performance with regards to running time, initial studies conducted using Scavenger have also shown that energy is indeed preserved by offloading resource intensive tasks. Our initial experiments show, that e.g., offloading the image operations working on the large, original images energy savings of up to 86% may be obtained, while offloading the operations on the smaller preview versions of the images reduced the power consumption by 24%. These results have been obtained while running Scavenger on a Nokia N810 Internet tablet; for further information about these experiments see [13].

VI. Conclusion

This paper has presented Scavenger; a cyber foraging system with a new approach towards task distribution and scheduling. Scavenger's mobile code approach has been presented, as has its novel dual-profiling scheduler using adaptive history-based profiling. The scheduler works with multiple factors when selecting where to place tasks; considering both data locality, network capabilities, device strength, and task complexity. The entire source code of Scavenger is released as open source, and practitioners within the field of pervasive computing are encouraged to fetch it and experiment with it in their own computing environments. The source may be found at: http://www.interactivespaces.net/projects/Locusts/

The paper has also shown how cyber foraging enabled applications may be created using the Scavenger system. In this system the process of writing highly distributed, parallel cyber foraging applications has been simplified into adding a single decorator to resource intensive code that may benefit of remote execution. This is a big step forward for the usability of cyber foraging as a general pervasive computing technique—to some degree answering one of the original questions posed by Satyanarayanan in [1] while defining the term cyber foraging.

Finally, this paper has shown through experiments that 1) even for modern mobile devices applying cyber foraging to common tasks such as simple image manipulations yields very large speedups—even when the surrogates are relatively old computers as was the case in these experiments, and 2) that Scavenger as a cyber foraging system is capable of greatly increasing the performance of mobile applications.

Future work within the Scavenger system entails mainly experimentation with the dual profiling scheduler to further prove the viability of the task-centric profiling approach. Also experimentation concerning energy usage of the mobile clients will be performed—perhaps augmenting the scheduler such that energy usage may become a factor when task placement is chosen.

ACKNOWLEDGEMENTS

This paper has been funded by a research grant from the Danish Research Council for Technology and Production Sciences. The author would like to thank professor Jari Porras for helpful insights into the problem domain. Thanks also goes to Niels Olof Bouvin, Mikkel Baun Kjærgaard, and to Nokia for providing us with mobile hardware. Finally thanks goes to the anonymous reviewers, who provided valuable insight into the problem domain.

REFERENCES

[1] M. Satyanarayanan, "Pervasive computing: vision and challenges," *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, 2001.

[2] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC*. New York, NY, USA: ACM Press, 2002, pp. 87–92.

[3] J. Porras, O. Riva, and M. D. Kristensen, *Middleware for Network Eccentric and Mobile Applications*. Springer, 2009, ch. 16, pp. 353–372.

[4] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*. New York, NY, USA: ACM, 2003, pp. 273–286.

[5] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 217–226.

[6] Y.-Y. Su and J. Flinn, "Slingshot: deploying stateful services in wireless hotspots," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2005, pp. 79–92.

[7] S. Goyal and J. Carter, "A lightweight secure cyber foraging infrastructure for resource-constrained devices," in *Mobile Computing Systems and Applications, 2004. WMCSA 2004. Sixth IEEE Workshop on*, 2004, pp. 186–195.

[8] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 43–51.

[9] G. C. Hunt and M. L. Scott, "The coign automatic distributed partitioning system," in *Operating Systems Design and Implementation*, 1999, pp. 187–200.

[10] J. M. Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward, "Structured decomposition of adaptive applications," *Pervasive Computing and Communications, IEEE International Conference on*, vol. 0, pp. 1–10, 2008.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, pp. 14–23, 2009.

[12] M. D. Kristensen, "Scavenger – mobile remote execution," University of Aarhus, Tech. Rep. DAIMI PB-587, 2008.

[13] M. D. Kristensen and N. O. Bouvin, "Using wi-fi to save energy via p2p remote execution," in *Pervasive Computing and Communications Workshops, 2010. PerCom Workshops '10. Eight Annual IEEE International Conference on Pervasive Computing*. IEEE Computer Society, 2010.