Using Code Collection to Support Large Applications on Mobile Devices

Lucian Popa
Politehnica University Bucharest
lucianpopa@cs.pub.ro

Costin Raiciu
Politehnica University Bucharest
costinraiciu@cs.pub.ro

Radu Teodorescu
Lehman Brothers, New York
teodores@cs.ucdavis.edu

Irina Athanasiu
Politehnica University Bucharest
irina@cs.pub.ro

Raju Pandey
University of California, Davis
pandey@cs.ucdavis.edu

ABSTRACT

The progress of mobile device technology unfolds a new spectrum of applications that challenges conventional infrastructure models. Most of these devices are perceived by their users as "appliances" rather than computers and accordingly the application management should be done transparently by the underlying system unlike classic applications managed explicitly by the user. Memory management on such devices should consider new types of mobile applications involving code mobility such as mobile agents, active networks and context aware applications. This paper describes a new code management technique, called "code collection" and proposes a specific code collection algorithm, the Adaptive Code Collection Algorithm (ACCAL). Code collection is a mechanism for transparently loading and discarding application components on mobile devices at runtime that is designed to permit very low memory usage and at the same time good performance by focusing memory usage on the hotspots of the application. To achieve these goals, ACCAL uses properties specific to executable code and enhances conventional data management methods such as garbage collection and caching. The results show that fine-grained code collection allows large applications to execute by using significantly less memory while inducing small execution time overhead.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management – *garbage collection, virtual memory.*

General Terms

Algorithms, Management

Keywords

Code Collection, Garbage Collection, Caching

1. INTRODUCTION

The proliferation of mobile devices has exposed the Computer Science community to a wide range of new problems including the development of new usability models as well as support for these models in the context of high resource constraints.

Unlike desktop computers that were gradually embraced by the general public as hi-tech equipment, requiring a level of expertise to operate and a certain level of maintenance, most mobile devices evolved from simple appliances and are expected to continue to perform as low/no-maintenance tools even as their feature-set expands. While most computer users understand and accept that in order to use computer one needs to install and keep up to date specific applications, most cell phones and even PDA users expect their devices to be fully functional out-of-the-box. Only a small fraction of the users is aware of the physical resources (such as memory or CPU) associated with their devices. In order to further increase the features associated with mobile devices without having to change the common user perspective, mobile device middleware should be able to transparently maintain and update device applications according to user's needs but without explicit user interaction [6, 26, 32].

Context aware applications represent a major direction for mobile devices evolution [6, 7]. Mobile devices are the ideal personal access points for the surrounding information environment. The devices can provide the user with context related information, such as flight information within an airport or a product catalog inside a store [27]. In order to serve such purposes the device should be able to download and execute relevant application code for each context. A key challenge in such a dynamic environment is the middleware's ability to manage software components on memory-constrained devices. Given the wide range of potential applications and the inherent limited resources of mobile devices, mobile middleware must provide an automated way of discarding unused software modules to release memory strain when availability is low. Furthermore, it must minimize time penalties generated by the re-downloading of discarded modules by distinguishing between heavily and rarely used software modules. A middleware that is consequent in discarding heavily used software components will generate huge execution time overhead, failing its purpose. The ideal mobile system middleware should be able to provide the application components needed by the user when they are needed, while maintaining the balance between memory and network traffic needed to download the new components.

This paper introduces a new application management concept, called **Code Collection.** Code Collection enables a resource-constrained device to execute large applications by dynamically loading code units needed by the application, and by discarding

Mobicom'04, Sept. 26–Oct. 1, 2004, Philadelphia, Pennsylvania, USA.

those no longer needed. Rather than managing (loading/unloading) code at the library or application granularity level, Code Collection relies on a finer granularity such as class or even method level. We further propose a specific Code Collection algorithm, the Adaptive Code Collection Algorithm (ACCAL), which uses a combination of garbage collection and caching techniques to identify the software components most likely to be used in the future. The algorithm uses runtime information to find the dynamic dependencies between the currently executed module and the rest of the code units. ACCAL adapts the collection techniques to the specificity of code management: code units represent constant information, which can be retrieved from an application server when it is no longer available on the mobile device. This relaxation brings the problem closer to the area of data caching.

We have implemented the code collection technique in a Java runtime environment optimized for mobile devices called Dynamic JUCE (this architecture is thoroughly described in *Section 3*). The code collection enabled architecture was deployed on Motorola StarCore SC140 DSP core. We have also implemented a Location Based Services simulation and analyzed the performance of code collection in this context. The results show that fine-grained code collection allows large applications to execute by using significantly less memory while inducing small execution time overhead. The tests also show that our approach clearly outperforms the performances of classical caching algorithms in the same context by minimizing the runtime overhead of an application (the execution time and the size of the downloaded code).

This paper is organized as follows: In Section 2 we provide background information on the code management problem. Section 3 describes the Dynamic JUCE architecture and its implementation on the StarCore DSP. Section 4 describes the Adaptive Code Collection Algorithm in detail. Section 5 presents the experimental results of our code collection mechanism in the context of a Location Based Services application and discusses the applicability of this mechanism. Section 6 provides a brief overview of related work. We finalize in sections 7 and 8 with a summary of the concepts and work presented in this paper and a list of possible extensions.

2. BACKGROUND

This section presents an overview of existing code management and caching algorithms and discusses the applicability of such techniques in mobile environments.

2.1 Code management techniques

Traditional code management techniques usually work with code libraries as units and are implemented at the Operating System (OS) level [14]. In traditional systems, code management (through linkers and loaders) is used primarily for managing sharing modules among different programs. Once loaded, the modules typically remain a part of the program during the entire execution of the program. Loadable kernel modules provide another mechanism for loading and removing program components on the basis of demand and usage of the components. The level of granularity of code management in these systems is coarse (library or module). In addition, the policies used for loading and unloading software components are simple.

The Java virtual machine [4] uses classes as a unit for code management. Code loading policies can be implemented at the virtual machine level (through the system class loader) or at the user level (through user-defined class loaders). The Java virtual machine may only unload classes with unreachable class loaders, which must be user-defined. The Java 2 Micro Edition's Connected Limited Device Configuration version[15] only loads classes through the bootstrap class loader (as a part of the restrictions imposed by the sandbox security model) making class unloading impossible.

Even if the techniques we have presented so far work fine on most systems, the particular characteristics of mobile devices make the applicability of the code management techniques limited. First, mobile devices run very lightweight or no operating systems without any support for virtual memory. Classical code-management techniques cannot be applied in this case. Second, Java class management, in particular garbage collection on classes, does not take into consideration the distinctive properties of code, for instance the locality property [8].

Other research in mobile code has correctly identified the need of using fine-grained code mobility to update seamlessly mobile applications [31, 32].

Our code-management technique differs from the above approaches in three important ways:

- *Granularity*: The technique uses methods as a unit for code allocation and de-allocation. This allows us to control accurately the usage of memory in memory-constrained devices.
- *Policies*: The technique uses several heuristics to capture patterns of code usage at runtime, and allocate and de-allocate code units based on these patterns.
- *Scope*: The technique allows collection of methods that may still be referenced in a running program. This allows us to run Java programs in small amount of memory, at a slower rate.

2.2 Caching and garbage collection

Unloading certain blocks of code relates to other memory management techniques such as caching or garbage collection. Code collection is conceptually related to caching, as it provides a way to manage a large amount of data in a small amount of memory and to ensure a minimal number of miss penalties in the future. Classical caching (page replacement) algorithms [5] such as First In First Out (FIFO), Last In First Out (LIFO) or Least Recently Used (LRU) are, therefore, candidates for code collection. However, we show in this paper that an algorithm using runtime information and specific properties of code-blocks outperforms these algorithms.

Code collection can also be thought of as an instance of garbage collection, performed on code-units. However, the focus in garbage collection algorithms is primarily on discovering unused objects and on reclaiming space within a specific application. Code collection algorithms, on the other hand, focus on the *relative* usage of code across multiple applications and must not release all the code that can be discarded. Such algorithms rely on heuristics to detect hotspots and to limit the extent of the collection.

3. SYSTEM ARCHITECTURE

Dynamic JUCE is an architecture designed to run large Java applications on memory-constrained devices. ACCAL is implemented as a runtime service of Dynamic JUCE, an architecture based on JUCE [1].

JUCE is a Java runtime environment specially designed for ubiquitous devices, based on a transparent distributed computing paradigm that moves the load away from mobile devices to general-purpose hosts (JUCE gateways). It assumes that the hosts and the devices can communicate with each other through a communication link. JUCE uses the additional processing power and memory capabilities of the gateways to compensate for the limitations of embedded devices. JUCE relies on remote Just In Time (JIT) compilation and method migration to run Java applications on mobile devices.

Dynamic JUCE builds on several aspects of the JUCE architecture and meets the most important requirements of mobile architectures: it relies on code mobility and makes application management completely transparent. Its main features are a highly modular architecture that greatly eases the task of deploying the platform on a new processor and a platform independent communication protocol (running on a two-level software protocol stack and able to run on top of any available network protocol). Dynamic JUCE implements code collection, providing an efficient way of running large applications on memory constrained mobile devices.

In order to run an application, the Dynamic JUCE gateway compiles the application, and "pushes" the main method binary on the remote device. As the main method executes and references other methods, the gateway compiles the referenced methods, and transfers them to the device. If, at some point during the execution, the total size of the methods exceeds the upper memory threshold at the device, the code collection component simply unloads some methods, freeing memory. Collected methods needed at a later point during the execution of the program are sent again to the device. The primary goal of the collection algorithm is to minimize the re-loading of collected methods (the total size of the re-downloaded methods will be referred hereafter as miss penalty).

3.1 The Dynamic JUCE Gateway

The Dynamic JUCE gateway consists of the Compilation Unit, the Class Repository and the Dynamic JUCE server (see *Figure 3.1)*. The main purpose of the gateway is to service requests from the device. The gateway communicates with the mobile device by means of a network connection and a layered communication model that uses the platform independent XDR standard.

The Dynamic JUCE server handles device requests for compiled methods. Once a method request arrives, an appropriate request is made to the Compilation Unit, which compiles the method. Then, the compiled method is sent back to the mobile device. The Compilation Unit (CU) receives a method compile request, locates the method and compiles it for the requesting device. The CU is adapted to work with multiple devices by dynamically choosing the appropriate back-end code generator at runtime. The current version of the Dynamic JUCE platform contains a back-end code generator for the SC140 DSP core.

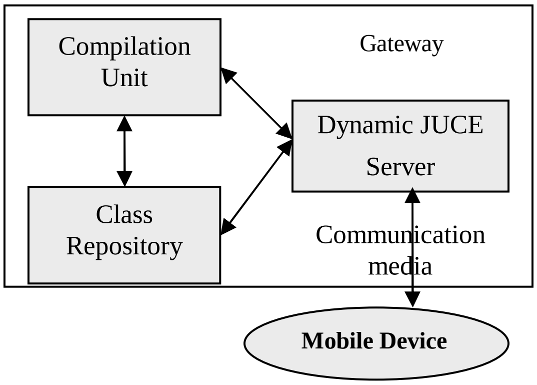

Figure 3.1 Dynamic JUCE Gateway Layout

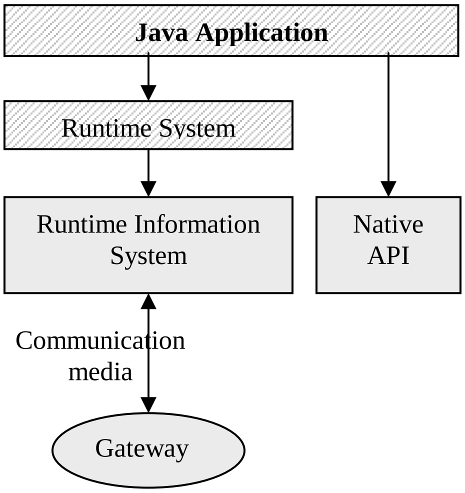

Figure 3.2 Dynamic JUCE Device Layout

The Class Repository mainly consists of a list of applications that can be run on the device.

3.2 The Dynamic JUCE Device

The structure of the device resident framework is shown in Figure 3.2. Remotely compiled Java applications are run on the device with the help of the Runtime System. The Runtime System includes device-resident components (Static Runtime System) and dynamic components loaded transparently when they are needed (Dynamic Runtime System). When memory is scarce, the latter can be collected just like regular methods. The NativeAPI contains a set of device-specific primitives used by Java applications. Usually, the primitives in NativeAPI handle display or communication capabilities, but can also be extended to take advantage of the particular features of the embedded platform.

3.2.1 Runtime Information System

The Runtime Information System (RTIS, Figure 3.3) is the core of the architecture's device-resident part. RTIS is responsible for class management and code management on the device and for providing code collection. The Runtime System uses RTIS to load methods or class information required by the executing application.

To perform class management, RTIS holds a list of class information[1] entries. A class information entry is initialized when the first reference to a class is made and enriched with bits of class information as more references are made to the methods of the class. This way, RTIS stores only the class information that is effectively used by the program (usually, a small fraction of the total class information) to ensure minimal memory usage.

The requests made by the Runtime System through the Access Interface are serviced by the RTIS either by using locally available class information or by querying the gateway for unavailable pieces of class information.

When the progressive loading of methods raises the size of the code memory above the upper memory threshold, the Runtime Information System triggers the collection process. The collection process uses runtime information on methods to collect the least likely methods to be used in the future. If a device is very low on memory, the collection process is used frequently and usually the number of method misses in the RTIS is larger, resulting in more downloaded code. Devices with more memory use collection less frequently and accordingly the number of code-misses and downloaded code is much smaller. By successive method loading and collecting RTIS actually implements a transparent balancing mechanism of the application and Dynamic Runtime System components between the gateway and the device. Dynamic balancing is used at different rates during the execution of the application, depending on the device's resource availability.

RTIS includes an optional Persistent Storage module, capable of saving and restoring software modules on devices capable of persistent storage. Based on runtime information acquired by RTIS core, application components are selectively saved in the device's persistent data area; future executions of the same components save time by loading code from the fast persistent memory instead of the network. This mechanism allows Dynamic JUCE to efficiently scale to a wide class of devices. At the extremes, a device equipped with enough storage to hold an entire application will be able to run that application very fast (applications are stored in binary format on the device) while a device without storage will run the application by incrementally downloading it from the application server.

Through transparent code loading and collection, RTIS makes it possible to run large applications on very limited devices. Through code collection and persistent storage RTIS offers flexibility by dynamically balancing between execution time and mobile storage space.

3.3 Implementation Notes

We have implemented Dynamic JUCE on Motorola's network-ready MSC8101 processor (based on StarCore SC140 DSP Core). The StarCore SC140 DSP Core is at the heart of mobile communication products [24] and mobile architectures [23]. A wide range of applications is suitable to be run on the MSC8101 chip, from voice encoding and decoding to packet switching algorithms in routers or even mobile user level applications. The current implementation of the Dynamic JUCE architecture can prove an important starting point for the deployment of large Java applications on MSC8101.

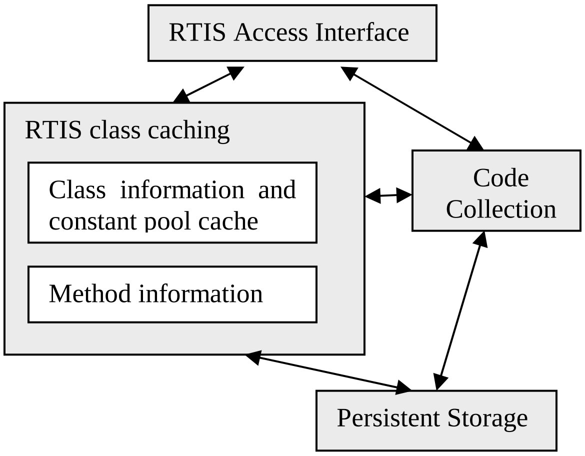

Figure 3.3 RTIS Layout

The main purpose of this work is to prove the effectiveness of code collection in a mobile environment. The existing implementation lacks support some of the features of the Java programming language (such as I/O, native methods, multi-tasking or static members of classes). Some of these limitations (the ones supported by the device) can be easily overcome by taking advantage of Dynamic JUCE's service-oriented architecture. We intend to extend the set of features supported by Dynamic JUCE in future work.

4. ADAPTIVE CODE COLLECTION ALGORITHM

A code collection algorithm's goal is to select and discard code units in a way that minimizes the number of code-unit downloads while keeping the memory usage under a given threshold. To serve this purpose, our Adaptive Code Collection Algorithm identifies the code units that are most likely to be invoked again, and based on that, decides which should be kept in the memory and which can be discarded. While it is impossible to make an exact prediction of application behavior [5], there are a number of factors that can provide useful hints on the future usage of a certain unit.

The ACCAL operates in two distinct phases: **runtime** phase and **collection** phase. During the runtime phase, the algorithm collects information on an application's code access patterns. During the collection phase, it uses the acquired information to discard code-units.

4.1 Runtime Phase

During normal execution, the algorithm maintains the last invocation timestamp and the *Popularity Index* (PI) for each code-unit the device's memory.

The popularity index reflects the probability for a unit to be invoked again in the future. The PI captures two factors relevant to a code unit's popularity: the number of invocations and the popularity of its caller units. The combination of the two factors matches the spirit of combining caching and garbage collection

[1] Class Information is a data structure recording information (including name, instance size, static size, loaded methods and other constant-pool information) for each class that has been actively referred by the application

techniques: the number of invocations being specific to caching metrics (the access count) and code unit dependencies being primarily used in garbage collection.

The reasoning behind this approach is that while the number of invocations is a direct indication of a code-unit's popularity, the popularity is also influenced by invocation dependencies between units. A code-unit invoked primarily by code-units with low popularity is less likely to be invoked again than a code-unit with more popular invokers even if the two have been equally invoked up to the current time. Thus if a more popular code-unit invokes a less popular one, the popularity index of the invoked code-unit should be further increased.

Every time a code unit is invoked its PI is updated according to the following formula:

$$PI = PI + self\_increment + inherited\_popularity(PI, callerPI)$$

Where *self_increment* is a constant used to factor in the number of times the code unit is invoked and *inherited_popularity* is a function used to factor in the popularity of the caller unit. i*nherited_popularity* is defined by the following formula:

$$inherited\_popularity(PI, callerPI) = correlation\_coefficient \times (callerPI - PI) \times \sigma(callerPI - PI)$$

Where symbols have the following meanings:

- *correlation_coefficient* – is a constant that reflects the impact that invocation dependencies should have on the value of Popularity Index. By varying the ratio between the *self_increment* and *correlation_coefficient,* the algorithm can behave more "cache like" (i.e. utilizing the invocation frequency) or more "garbage collector like" (i.e. utilizing the code dependencies).
- $\sigma(x)$ – represents the "step" function and it takes the value 0 for x < 0 and 1 for x>=0. The function is used because *inherited_popularity* must have the value zero when the invoker function has lower popularity than the invoked function.

It is important to note that *inherited_popularity* is zero for recursive function calls, which reflects the fact that recursive dependencies are irrelevant from Popularity Index perspective.

4.2 Collection Phase

The collection phase consists of the computation of the Collection Index for each code unit followed by the actual code collection in the ascending order of their collection indexes.

The collection process scans through the loaded code-units and eliminates the ones with the Collection Index lower than a certain threshold referred to as Elimination Coefficient (EC). The Elimination Coefficient is calculated for each collection phase based on the current distribution of Collection Indexes. The purpose of the Elimination Coefficient is to mediate between the overhead of collection and miss penalties caused by additional units download. A low value for EC would lead to an aggressive code collection that is likely to generate a higher future miss penalty (code units downloaded more than once), whereas a high value for EC would reduce the amount of memory freed by each collection phase and implicitly increase the frequency of the code collection process.

The Collection Index represents the Popularity Index biased by the unit's last usage time and its size. This favors the collection of bigger code units over smaller ones in addition to favoring code units that have been recently used. The collection index adds information known at the collection time to decrease further the probability of a future miss.

The Collection Index (CI) is computed for each code unit at collection time based on the following formula:

$$CI = PI - aging\_factor \times \Delta t + size\_dependency\ (unit\_size)$$

Where the factors have the following meanings:

- *PI* – is the Popularity Index computed during Execution Phase.
- *aging_factor* – is a constant used to factor the time elapsed since the last invocation of the unit
- Δt – the time that passed since the last invocation of the unit. The algorithm uses a virtual timer that is incremented for each code unit invocation. Each unit has its time stamp updated upon invocation.
- *size_dependency* –is a function that enables the collection algorithm to prioritize the collection of bigger size code units.

The *size_dependency* is proportional to the reverse of unit's size as follows:

$$size\_\ dependency = size\_factor \times (mean\_size / unit\_size)$$

with:

- *size_factor* – is a constant that defines the unit's size in the collection decision
- *mean_size* – is the arithmetic mean of the sizes of all units available in memory.

4.2.1 *Constants Used by the Algorithm*

In the rest of this paper we will refer to the constants used for computing the Popularity Index (correlation coefficient and self increment) and for computing the Collection Index (aging factor and size factor) as ACCAL's constants. Using these constants makes ACCAL easily configurable for various types of memory availabilities and types of applications. Different types of applications and different memory sizes may render different optimal values. However, values that perform well on most cases exist, as we show in the Analysis section.

4.3 ACCAL Implementation in Dynamic JUCE

This section points out the distinctive particularities of the Dynamic JUCE implementation of ACCAL, embedded in the Code Collection unit and resident on the device.

4.3.1 *Code Collection Semantics*

In the current implementation, the collection is triggered at certain points during an application's execution, when there is not enough available memory. Previous research [10] has proven that determining the right time to perform garbage collection is quite important to the performance of an application. Consequently, determining the right time to perform code collection has an important impact on the performance of applications running on the Dynamic JUCE platform. Different implementations of the algorithm can choose to trigger the collection periodically or use heuristics like memory allocation rate to determine the best suitable time**.** The performance gain of using such heuristics is still to be assessed.

In their attempt to minimize the execution time overhead of garbage collecting, generational collectors try to limit the number of frequently garbage collected objects by splitting the object space into several generations and scavenging frequently only young areas [29]. Our code collection semantics allow, in an extreme case, the collection of all the modules not currently executing on the mobile device. Such aggressive collection can lead to a large number of code misses and it must be tempered. Instead of dividing memory space into generations and aggressively collecting space in young generations, our collection algorithm prefers an alternative heuristic: it uses a dynamic factor (the Elimination Coefficient) to determine the number of code-units one code collection should free.

ACCAL uses two memory thresholds, specified by the environment implementers, to determine whether the memory is occupied (the upper memory threshold) or free (the lower memory threshold). The collection process keeps (if possible) the memory usage below the upper limit. Collection is considered unnecessary when memory usage is below the lower limit. After each code collection, the code-memory usage is situated between the two thresholds.

The collection process spans two different stages. In the first stage, the code-units with Collection Indexes smaller than the value of EC will be collected as long as the size of the occupied memory is larger than the lower memory threshold. The first stage ends when all code units with CI values smaller than EC are collected or the memory usage is below the lower memory threshold.

If the size of code-occupied memory is still larger than the upper memory threshold after the first stage, the second collection stage is performed. It continues to collect code-units until the occupied memory drops below the upper memory threshold or until all the unused methods have been collected.

4.3.2 Architecture Dependent Characteristics

ACCAL is optimized to take advantage of the Dynamic JUCE's method level granularity by using *methods as code-units*. The usage of smaller code-units (i.e. methods instead of classes) allows better memory usage and increases the efficiency of the algorithm. Other research explores using a even finer grained code-mobility unit, at instruction or even declaration level [31]. We consider that in this particular case using such granularity would be highly inefficient.

The Dynamic JUCE implementation of ACCAL regards as "unused" all the methods that are not on the execution stack of the device. In an extreme case, an application can run if all the methods on its execution stack fit in memory. A different meaning to the "unused" property of code units can have interesting effects: an application could run by keeping in memory only the currently executing method and not all the methods on the call stack. However, the memory and execution time overhead associated with the automated context saving and restoring mechanism makes such an approach seem unfeasible.

ACCAL has been tested with a Location Based Services simulation; the results of the tests and comments on their implications are listed in the upcoming Analysis section.

5. ANALYSIS

This section analyzes the characteristics of ACCAL embedded in the Dynamic JUCE architecture and running on the StarCore DSP. We first test how the algorithm adapts to various memory availabilities to prove its flexibility regarding the ratio between the occupied memory and execution time. Next, performance comparisons are made against classical caching algorithms, like LRU and FIFO. Time measurements are used to analyze the component factors of the execution, including collection time, communication time, gateway computation and device local computation. We try to infer from these results the performance of our architecture running on different network connections. We then measure the impact each of our heuristics have on overall performance and analyze a method to obtain best performance. Finally, we discuss on the applicability of our solution.

It is important to note that the results presented in this section do not dependent on particular features of the StarCore environment. Thus, we expect to obtain similar results on any network-enabled mobile architecture.

A typical context-aware application has been developed to allow the profiling of the ACCAL. The application outlines the way our algorithm adapts to different memory availabilities and types of mobile devices and also points out the advantages of this solution compared to other approaches in the same field. The context aware application we have implemented is a typical mobile application requiring code collection; however, to discuss ACCAL's properties on a wider range of application types, new applications must be developed and a similar series of tests must be performed. Our sample application shows that ACCAL performs well on a typical location-based application. To extrapolate these results to the full range of mobile applications we need to examine a significantly larger number of application types.

5.1 Experimental Methodology

The context-aware application was run using the Dynamic JUCE architecture. The device resident components of the Dynamic JUCE architecture are implemented using the C programming language and compiled with Metrowerks CodeWarrior for StarCore v2.5 C Compiler. The Dynamic JUCE gateway is implemented using the Java and C programming languages; the gateway is deployed on a general purpose PC running the Linux Operating System. The device and gateway communicate by using the UDP/IP protocol.

In the upcoming sections memory benchmarks refer to the amount of memory used by code and not to amount of memory used by data, unless stated otherwise. Our time sampling is discrete and uses the count of processor cycles; the DSP's clock frequency has been configured at 200 MHZ. The memory limits specified in the following sections have been chosen to reflect the typical resource limitations encountered by the location based services in real environments.

The I/O operations (like user input or console output) on the MSC8101 development board are intended for debugging purposes. These operations are executed remotely (on the host

PC) and they are highly time-consuming. In order to achieve accurate time measurements we benchmarked the application without user input and display support. However, this does not affect the validity of the measurements. The network link between the device and the gateway can also generate flawed time measurements in case of congestions or other possible sources of latency. To avoid network traffic delays, we used a direct network link between the device and the gateway. The running time of an application was also profiled on the gateway to make sure different executions of the same application have minor time differences (that can be disregarded) and assumed the error-rate of the measurement has a Gauss distribution. By using this profiling environment, the processor cycles measurements between two executions of the same application in identical conditions have rendered differences of at most 2%.

5.2 Test Application

We envision a world where mobile users roam around geographical areas carrying smart devices, able to manage and download applications on the fly, without user intervention. The world becomes an "application space" where there is a high level of interaction between users and the surrounding environment. A transposition of the application space in reality is the “ubiquitous city”, where at different locations in space (or time) several applications are available, such as store leaflets, virtual museum or city guides, programs with information on public transportation, software support for virtual ad-hoc communities, applications for environment access (climatic control, displays, industrial gadgets) and so on. A fluctuating number of such applications are dynamically downloaded and executed on the user’s device as he or she moves around the city, taking account user’s preferences, location and other context related information.

Designing, implementing and evaluating realistic mobile (and in particular context aware) applications such as the ones we have previously described is hard. Since our goal is to prove the effectiveness of Code Collection, there are other important factors we must consider when creating a valid test application besides its inherent complexity:

- The mobility patterns of the users
- The typical size of the applications compared to the available mobile resources:
- The bandwidth and latency of the network routes between mobile devices and application providers

Given these factors, we have tried to produce a simulation and several execution scenarios that we think are characteristic for location based applications involving code mobility.

5.2.1 Location Based Services Simulation (LBSSim)

Our application is a simulation of a LBS system. The mobile device runs on the StarCore DSP, while the Dynamic JUCE architecture’s gateway models the various service providers. The position of the device and its movement are simulated in code (however, this must not be considered a limitation: a real device would simply use information provided by another API using a GPS service, for example). Based on its current location, the device can download and execute location-specific services. These services are in fact applications residing on the Dynamic JUCE gateway that are downloaded on-demand and executed. When, through progressive accumulation of code, the device’s available memory becomes very low; code collection frees up parts of the downloaded applications.

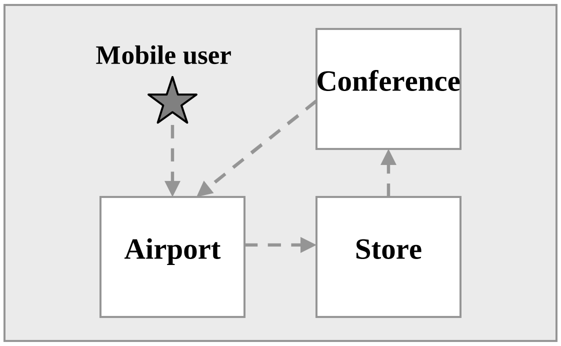

Figure 5.1 Geographical layout of the LBS simulation (first scenario)

The simulation assumes the user moves around in a region where various services are available in distinctive areas. When the user enters an area where a service is available, it can download the service and execute it. There are currently 3 typical service providers [26, 27] simulated in this application:

- Airport Guide – available in an airport, offers flight and gate information. It can provide the user with directions to certain points of the airport (such as gates) by using the current location of the user.
- Conference Application – allows attendees to access the proceedings electronically, select the talks they want to attend and provide context sensitive directions to the various conference rooms.
- Store Assistant – provides the mobile user with a list of the items on sale, special offers. The system features context sensitive commercials and can also provide the user with directions.

To test our collection algorithm, we consider two scenarios that describe distinct mobility patterns of the user. In the first scenario, we assume the user to be a researcher, which arrives by plane in our region, attends some conference proceedings, does some shopping in the region and then leaves by plane. Thus, the device is initially in the airport where it loads and executes the Airport Guide; afterwards it moves at the conference where it runs the Conference Application, moves on to the store and uses the Store Assistant and finally reuses the Airport Guide. The second scenario follows a researcher that also attends some conference proceedings, goes back to the airport, where it enters a store while awaiting his flight. Notice in this second case that the Airport Guide and Store Assistant overlap when the user is in the store. The profiling this LBS Simulation in these two scenarios has provided all the data in the upcoming sections.

The applications we use for testing are synthetic in that they do not implement all the functionality of the location based applications we have described. However, this does not affect the overall behavior of our architecture, since the call sequence and the sizes of the methods are the only factors influencing the collection decision, and not the operations performed by the methods.

5.3 Memory Footprint

This section traces the memory footprint of the LBS simulation for various memory limits in the two execution scenarios. The memory limits have been specifically chosen to reflect ACCAL's behavior in conditions ranging from very low to large memory sizes, in respect to the memory requirements of the LBS simulation. The values of ACCAL constants' used for profiling were selected by fine-tuning, as we show in the next section.

The code size of all the downloaded services in the first execution scenario is 17.782 KB and was profiled by using four different memory limits as shown in Figure 5.2. The execution times are shown in Figure 5.3. The first memory limit, set at 6 KB, outlines the behavior of the algorithm in an environment very low on memory. As the graphic shows, collection is performed very often and most the 6 KB of code memory are usually occupied.

As we loosen the memory constraint to 8 KB, collection is performed less frequently. Also, the application runs faster (approximately 170 million cycles) in this case mostly due to the reduction of miss penalties (this improvement is also an effect of less frequent collection, but this factor has a much smaller impact, as we show in the upcoming sections).

The next case uses a limit of 11 KB, further reducing the number of code collections and improving the execution time. In this case all the methods are downloaded only once and the miss penalty is zero. This emphasizes one of the major benefits of code collection, the ability to run a program with very little time penalty by using significantly less code-memory. The optimal size of the code memory (i.e. the size where insignificant overhead is added to the application) depends on the locality of the application.

The final case is when the memory is large enough and collection is not performed at all. This is the fastest way to execute the application (around 600 million cycles). However, the improvement in the running speed is negligible (only about 1 mil. processor cycles), compared to the previous case (i.e. the 11 KB memory). This difference is also a cost indicator for the code collection performed within the 11 KB of memory.

The second execution scenario, with a total size of 17,902 KB, was profiled by using four different memory limits as shown in Figure 5.4 and the results are, as expected, similar.

Notice that the improvement in execution time between the 8 KB memory limit and the 11 KB memory limit is around 100 million cycles. This improvement is significantly smaller the improvement between the 6 KB and 8 KB memory limits (~450 million Cycles). The speed improvement is even smaller between the two highest memory limits (11 KB and 20 KB), being approximately 5 million cycles. These figures point out the locality rule we have focused on: if there is enough memory to hold most of the used code, the performance of the application will not be dramatically affected by code collection. A similar situation can also be observed in the first scenario.

During the execution, we have traced the collection of several methods. We have noticed that the algorithm did not collect common methods for all providers (like printDirections – a method used to guide the user to a certain location). This improves the start-up and execution time of the LBS application. This prediction could be used in practice by the LBS providers,

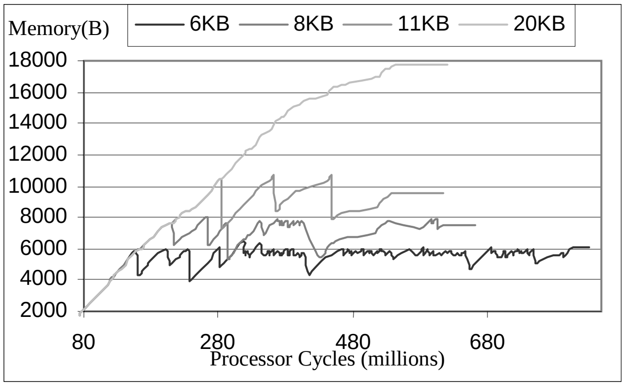

Figure 5.2 LBS Memory Footprint (First Scenario)

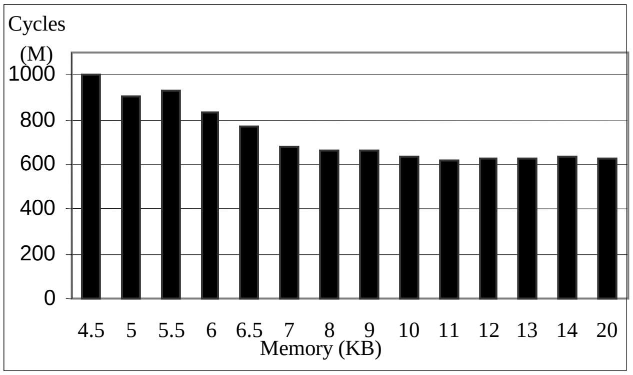

Figure 5.3 Application Execution Time (First Scenario)

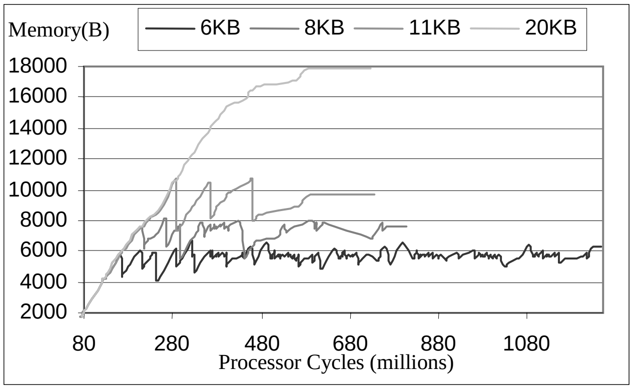

Figure 5.4 LBS Memory Footprint (Second Scenario)

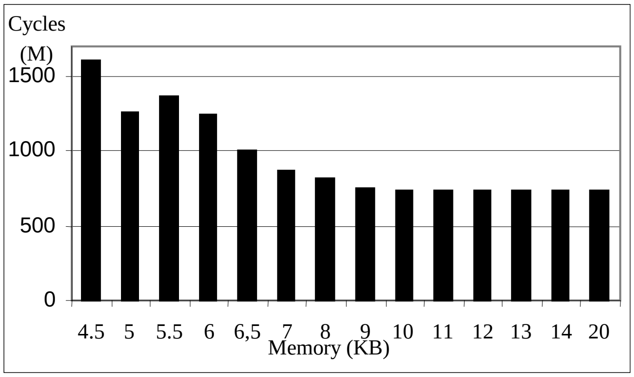

Figure 5.5 Application Execution Time (Second Scenario)

which may use a common set of base services for their applications.

We conclude that Dynamic JUCE is able to run the three applications (included in our LBS simulation, first scenario) with a total size of 18 KB of code memory in 6 KB of physical memory with a time penalty of 37% (relative to a code collection free execution scenario) or in 8 KB of physical memory with a time penalty of only 7%. The performance/cost ratio of the 8 KB case is excellent: users expect mobile applications to use very little memory and in the same time run as fast as possible. However, the value of this ratio is application and execution environment dependent and truly excellent values are difficult to obtain on the full range of mobile applications.

5.4 Performance Analysis

In this section, we test the performance of ACCAL against those of classical caching algorithms such as LRU or FIFO, analyze the impact of each constant in ACCAL's behavior and analyze methods to fine-tune the constants for better efficiency. We conclude by making qualitative observations on the differences between our approach and a well-established approach for Java on mobile devices, J2ME.

5.4.1 Comparison with Classical Caching Algorithms

We analyzed the performance of ACCAL against those of two classical algorithms: LRU and FIFO.

The results for the LBS simulation (scenario 1) running with different collection algorithms are displayed in Figure 5.6 and compare the scaled miss penalties of the algorithms (similar results were obtained for the second execution scenario). The scaled miss penalty is the ratio between the size of the code downloaded during an execution (with collection) and the total size of the code without performing any collection (i.e. the sum of all method sizes). An ideal scaled miss penalty is 1 when all necessary data is downloaded only once.

For a fair comparison, our implementation of LRU uses EC-based collecting just like ACCAL does. We have also implemented a memory-limit collection for LRU, but this implementation produced poor results. FIFO uses the lower memory threshold to determine the amount of memory to free since the Elimination Coefficient cannot be evaluated in this algorithm. The value of the lower memory threshold was intentionally picked very high to minimize the miss penalty.

As expected, Figure 5.4 shows that ACCAL, by using code-specific properties and combining caching and garbage collection techniques, achieves a significantly lower value for miss penalty than LRU or FIFO. It is interesting to note that ACCAL, LRU and FIFO have similar complexities in both resource usage and memory consumption in the runtime phase, while ACCAL has a slightly slower collection phase (induced by the sorting of the coefficients in O(n), where n is the number of methods). Since n is usually manageable (the usual order of magnitude is 10^2) ACCAL is a viable solution for real-life implementations of code collection.

5.4.2 Time Measurements

This section evaluates the performance of our implementation from two points of view: the latency experienced by the user waiting for the application to load and the overhead induced by collection.

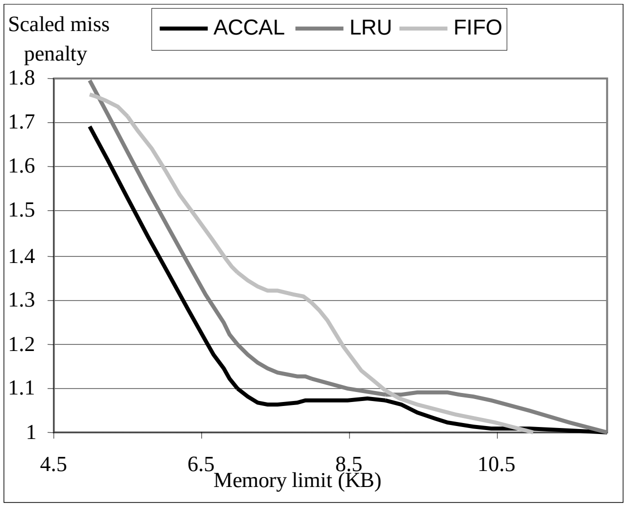

Figure 5.6 Performance comparison

Table 5.1 Time Measurements

Mem (KB)	**Total Time(s)**	**Waiting Time(s)**	**Local Time(s)**	**Collection Time(s)**	**Gateway Time(s)**
6	5.994	3.729	2.265	0.0205	3.421
7	4.231	2.501	1.731	0.0117	2.256
8	3.971	2.316	1.655	0.0089	2.080
9	3.646	2.108	1.538	0.0050	1.886
10	3.611	2.082	1.529	0.0033	1.861
11	3.567	2.055	1.513	0.0028	1.836
12	3.575	2.057	1.514	0.0021	1.837
20	3.604	2.056	1.548	0.0001	1.833

Table 5.1 lists time measurements for the LBS Simulation (second execution scenario) using various memory sizes. Total Time is the application's execution time (measured on the device). Waiting Time accounts for the time spent by the device waiting for gateway responses (i.e. the latency experienced by the user). Local Time is the amount of time the device uses for (local) computing (the Total Time is the sum of Waiting Time and Local Time). Gateway Time measures the response time of the gateway (i.e. the total time spent by the gateway to compile methods and resolve constant pool requests). The Waiting Time is the sum of the Gateway Time and the actual time spent on communication (the actual transmission time plus the overhead of the gateway's kernel and device's Ethernet communication library). The total size of the communication is sensibly larger than the size of the downloaded methods, due to protocol overhead (Dynamic JUCE Protocol, UDP, IP and Ethernet).

The data shows that the collection process has an almost insignificant impact on the overall execution time. As expected, as the memory size decreases, the collection time increases. Overall, the collection time accounts for approximately 0.3% of the local execution time.

In our measurements, the Gateway Time has a major effect on the user's waiting time while communication time has a smaller impact. An optimized implementation of the gateway could reduce Gateway Time to an almost constant value (depending only on the number of requested methods and not on their sizes) by pre-compiling and saving methods into a hash table. This optimization would make code collection a desirable feature of mobile runtime systems (such as JUCE [1]): the application executes in less code memory and in the same time the runtime overhead is small enough so that it is not noticed by the user.

The effective communication time (the difference between the Waiting Time and the Local Time) is quite small compared to the Total Time. It should be noted, however, that besides the transmission time it includes packet processing overhead on both the device and the gateway. Since we used a direct 100 Mbps Ethernet link, we estimate (based on the total size of the network packets) the actual transmission to have an order of magnitude of 10^{-2} s.

Changing the network connection affects the transmission time. If the platform is deployed in a 3G mobile environment, the estimated time for downloading an application of 100 KB varies from 0.4 seconds up to 5.5 seconds, depending on the user's position and movement. It seems likely that faster, location-specific connections will be available in the surrounding environment and 3G mobile communication will be used as a carrier only in remote areas. An example of a high-speed local wireless connection is IEEE's 802.15 Wireless Personal Area Networks (WPans), with speeds up to 55 Mbps. In this environment, the estimated communication time is also 10^{-2}s.

In our execution environment, the Waiting Time is mostly influenced by the Gateway Time. However, in a prospective implementation using a wireless network connection and an optimized gateway, the Waiting Time will be mainly affected by the transmission time. As the Gateway Time and the transmission time vary progressively with the communication size, the evolutions we have measured can be qualitatively extrapolated to a wireless implementation.

The local computation time on the device varies with the memory threshold because of the computational overhead associated with communication.

We have to mention that performance depends on many factors that vary extensively from one execution environment to another. Among them are the gateway and device hardware platforms, the communication speed and overhead and the efficiency of the Dynamic JUCE implementation. For example, we profiled the Dynamic JUCE gateway on two hardware configurations and obtained significantly different ratios between the Local Time and the Waiting Time. The Gateway Time of the faster gateway (a PIII processor) was around 0.6 seconds while the time of the slower gateway (a PII processor used throughout the Analysis section) was a little over 2 seconds, running the LBS Simulation in the second execution scenario in 8 KB of code memory.

Therefore, we cannot infer relevant quantitative measurements for a future industrial implementation, but rather give qualitative references. The only quantitative measurements that can be extrapolated and used in other running environments from our measurements are miss penalties.

Table 5.2 The impact of individual heuristics on the ACCAL's performance

Mem Size (KB)	Code Size (B)	Self Increment=0		Correlation Coefficient=0		Aging Factor=0	
		Code Size (B)	Improve (B)	Code Size (B)	Improve (B)	Code Size (B)	Improve (B)
6	26812	27128	316	27308	496	25742	-1070
7	19532	20596	1064	21870	2338	23122	3590
8	19088	19272	184	19088	0	21180	2092
10	18106	19088	982	18768	662	19854	1748
11	17782	17782	0	17782	0	18106	324
12	17782	17782	0	17782	0	17782	0
Average Improvement (B)			424.3		582.7		1114

5.4.3 The Significance of ACCAL's Heuristics

To accurately measure the contribution of each constant in the collection decision, we have profiled the size of all the methods downloaded by the execution environment when collecting with ACCAL (using the values obtained by fine-tuning for ACCAL's constants) against those obtained by ACCAL without using one of its constants (i.e. the value of the constant is set to 0) in the first execution scenario. Each heuristic proved to have beneficial effect in the collection process.

The experimental data (listed in Table 5.2) show that usage count ensures an average reduction of the downloaded size of over 400 bytes, while call graph dependencies and aging produce even better improvements. In the presented case, the most important improvement is triggered by the presence of the aging factor (i.e. its absence induces the greatest load penalty).

The benchmarked results correspond to the first execution scenario of the LBS Simulation. Other mobile applications (with different code locality, dependency graphs etc) will probably give different results and the relative importance of the factors could be different. However, it can be inferred from these results that all the heuristics ACCAL uses have a beneficial effect on the performance of the algorithm.

The size dependency has not been included in Table 5.1 since its purpose is to minimize the running time and not the total size of the downloaded methods. Its value is dependent on (and should be chosen accordingly) the delay of the communication media specific to the embedded device. If the communication overhead is very low, the size dependency should have a minimum influence in the collection process and implementers may use it as the decisive factor when all the other heuristics render equal results or may even not use it at all.

Given these circumstances, there are few cases in our test environment where the size dependency plays an important part in the collection process. One such example is in the LBS Simulation's second execution scenario using a code memory of 5 KB: the variation of the size dependency from 2 to 4 negatively affects the collection process (miss penalty increases with 1 KB) and has a positive effect on execution time (6 million cycles faster). Similar results have been obtained for other applications.

We consider size dependency an important factor in the collection process; in most cases network traffic induces significant overhead and its presence should prove beneficial to the overall performance of the application.

5.4.4 Elimination Coefficient (EC)

We assigned EC the arithmetic mean of the lowest Collection Index and the all the Collection Indexes' arithmetic mean. This value was chosen on experimental grounds. We have included in Figure 5.7 a graphical representation of the methods' Collection Indexes and EC for two distinct collection phases of the LBS Simulation (second scenario) running with 8 KB of code memory (leftmost in the graphic) and 9 KB (rightmost). The Elimination Coefficient's role is to temper the aggressiveness of the collection process by removing the methods with low Collection Indexes on one hand, and to free enough memory so that collection is not performed very frequently (resulting in additional overhead) on the other.

To measure EC's contribution, we matched ACCAL's performance when collecting down to fixed memory limits against those obtained with EC collection using the first execution scenario of our LBS Simulation. To perform collection as accurately as possible, we select a high value for the lower memory threshold (95% of the code memory). This ensures a lower miss penalty by collecting very frequently a small number of methods. Figure 5.8 is a comparison of the memory footprints of the LBS simulation running in two different code memory limits and collected by the two versions of the algorithm.

Time comparisons for ACCAL collecting with or without EC and using two lower memory thresholds (60% and 95% of the total size of code memory) are presented in Figure 5.9. The results show that, as expected, aggressive collection results in worse performance (the execution time of code collection without EC and lower threshold 60%). Also, collection based on EC produces results similar to collecting with the upper memory threshold in most cases (6 KB, 6.5 KB, 7 KB, for example) while in others performs better (4.5 KB, 5 KB, 7 KB, 8 KB and 11 KB, for instance). On average, frequent collection incurs 1 KB smaller miss penalty than EC, while the overall execution time with EC is approximately 70 million cycles faster.

The experimental data show that the value selected for EC strikes a good balance between collection granularity and runtime overhead, having an overall beneficial effect on the performance of the collection process.

5.4.5 Fine Tuning ACCAL's Constants

ACCAL, through its constants, can be adjusted to improve its performance for various types of applications and devices. The data presented in this section describes ACCAL's variation in performance for different values of its constants. We used as performance indicator the miss penalty (measured as the total size of the downloaded methods) and profiled the LBS application (first execution scenario) running in a code memory of 7 KB. We used an initial configuration of the constants and varied only one constant at a time. After obtaining minimum values for all the constants we spiraled back and measured again the performance. The configuration in the adjacent graphics uses the following values: self_update=2, correlation_coefficient=1, size_factor=1 and aging_factor=0.1 as base values.

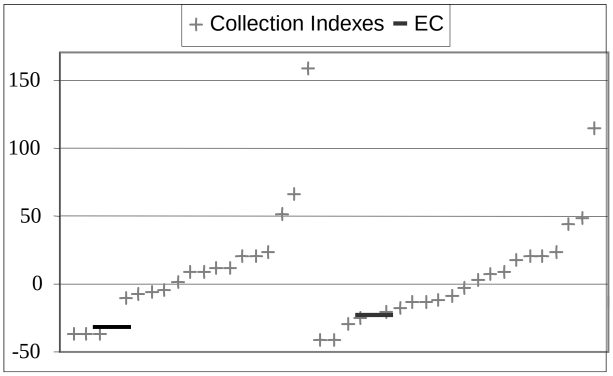

Figure 5.7 Coefficient and EC values (Distinct collection phases)

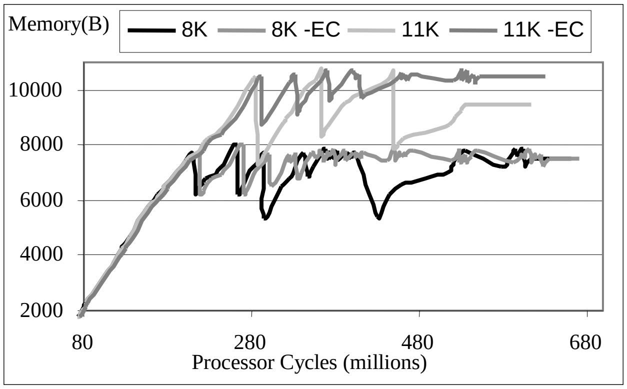

Figure 5.8 ACCAL's Memory Footprint with/without EC

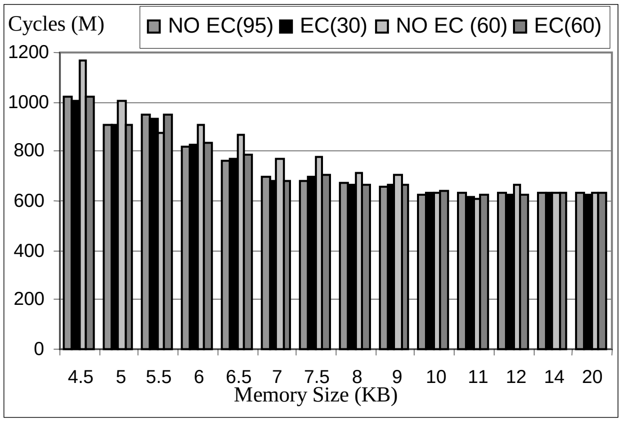

Figure 5.9 Application Execution Time with/without EC

The experimental data include minimum values for each of the constants. The size_factor was profiled using time instead of downloaded memory, since its purpose is not to minimize the size of the downloaded code but to minimize the running time. The size_factor also induces modifications in the downloaded code size (by affecting the collection decision), but we have obtained experimental results that associate lower execution times to larger downloaded code size.

It is obvious that the values we used do not always provide the minimum memory usage and execution time and no such values can be obtained for the full range of mobile applications and memory sizes. This is mainly because the size of the code memory is a discrete function (rather than continuous) and the performance of the collection algorithm depend mostly on when the collection is performed during the application and what methods will be called in the future. However, a class of values that perform well on most applications exists and the ones we used for profiling are one of its members.

As the graphics show for the 7 KB memory limit, the best set of approximate values for the constants of the algorithm should be: self_increment between 1 and 3, correlation_coefficient around 1, size_factor around 1 and aging_factor between 0.5 and 1 and other isolated values like 0.25. We analyzed the values of the constants on minimum miss penalty for various memory limits and developed some mean values that generally perform well. These values are self_update = 2, correlation_coefficient = 1, size_factor = 1 and aging_factor = 0.25. We considered these to be reference values and we used them to profile the application throughout all the tests presented in this paper. By using these values the algorithm outperformed the classical caching algorithms in all the profiled cases.

Fine-tuning depends significantly on other parameters, such as the values of the upper and lower memory thresholds defined as percentile values of the overall code memory size. We used in our tests an upper memory threshold of 95% and a lower memory threshold of 30%.

5.5 Notes on Performance Analysis

We have tested ACCAL on different scenarios of the LBS simulation, simulating different user mobility patterns. Each scenario is characterized by the size of the code common to all LBS services. Even if the algorithm produced similar results for both cases, we noticed that it performed better as the size of the code common to all LBS services grew larger. In such circumstances, the code-related heuristics used by ACCAL play a decisive role in obtaining superior performance.

5.5.1 Power Consumption

A complete analysis of the applicability of our solution cannot overlook energy consumption. We are unable to provide experimental data since there are no tools that measure the energy consumption of the (plugged) MSC8101 development board. Furthermore, since our test platform uses a direct Ethernet link between the gateway and the device, with energy parameters fundamentally different from that of wireless communication, acquired experimental data could not be extrapolated to wireless environments and would have no real value.

5.5.2 Java on Mobile Systems: Other Approaches

The best-known architecture for running Java on memory-poor mobile devices is the Java 2 Micro Edition Platform with its Connected Limited Device Configuration (CLDC). We have performed tests to measure the memory requirements of our LBS simulation in the Linux based reference implementation [22] of the J2ME environment. The total size of the class information loaded by the virtual machine is about 24 KB (including method information), 8 KB more than the 17 KB Dynamic JUCE downloads for the same application. Even if the two memory sizes cannot be thoroughly compared (Java methods are represented in byte-code, Dynamic JUCE methods are in StarCore binary code) and the sizes of the runtime environments are different, we can observe that an application running in the J2ME virtual machine cannot run within 6KB of code memory. With the cost of a minor penalty, Dynamic JUCE is able to run the same Java application using much less code memory than J2ME.

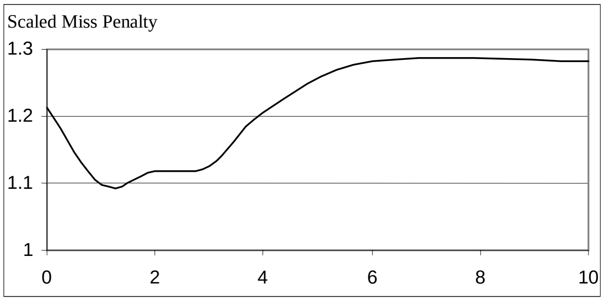

Figure 5.10 Performance variation for self_increment

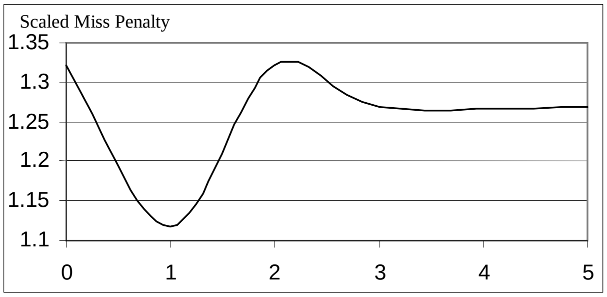

Figure 5.11 Performance variation for correlation_coefficient

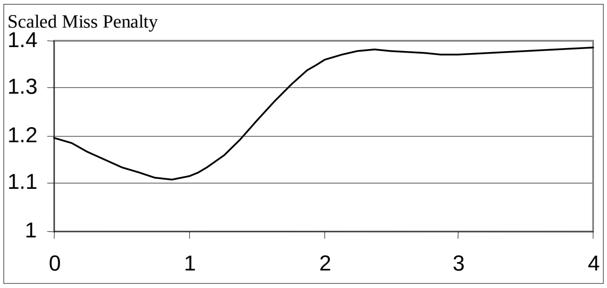

Figure 5.12 Performance variations for size_factor

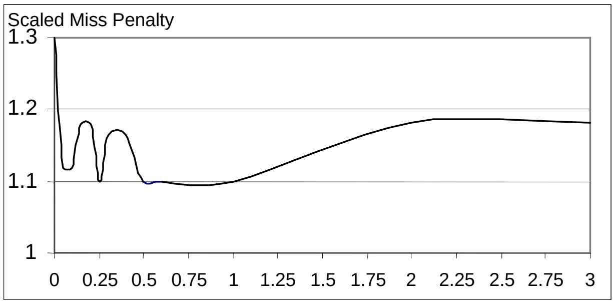

Figure 5.13 Performance variation for aging_factor

6. RELATED WORK

To our best knowledge, the intrinsic properties of code have not been used yet to dynamically improve the execution of typical mobile applications on memory-poor devices. Instead, when targeting such devices, current approaches generally focus towards minimizing the size of the runtime environment, core classes or even application code, such as Java Micro Edition Platform [17] or prefer to load into memory only the code that is actually used, at a method level granularity, such as JUCE [1]. Optimizations of mobile applications, such as the ones described by Krintz, et. al in [19] use specific properties of applications to minimize startup time and therefore performance.

The heuristics in our collection algorithm use a cost model partly similar to the one presented by Kortuem et al in [18]. The cost model presented in [18] serves as an automated cache replacement policy for handling on-demand delivery of software on mobile devices and uses information on application size, dependencies between applications and network bandwidth. However, the static heuristics used in this approach are insufficient if applied in dynamic execution environments. Our code collection enhances the cost model with runtime statistics and uses caching at a different granularity level to achieve better memory usage. Another cost-based cache replacement algorithm is described in [12]. Just like ACCAL, LNC-R uses size and usage count to assist the collection process. LNC-R has been designed as a cache for database queries differs from ACCAL who uses code specific properties in the collection process.

Optimizations meaning to improve the performance of running Java applications also relate to our work. We exploit the locality property of programs in the same way as the HotSpot technique does. Whether implemented fully in software [8] or in software and partly in hardware [16,17], this technique speeds up Java programs by detecting the most used pieces of code (hotspots) of an executing program and compiling them in native code for faster execution. In another study related to HotSpots, Hu et al. use them in [25] to achieve leakage energy reduction for instruction caches.

We use heuristics to determine the amount of memory one collection should free. This approach relates to the runtime optimization of Java programs by tuning the frequency of garbage collection and extent of garbage collection. Brecht et al, for example, research in [10] on the impact of the frequency of garbage collection and heap growth on the performance of Java applications. Reinforcement learning techniques are used by Andreasson et al. in [11] to determine the best suitable garbage collection algorithm and the time it should be run.

Method level or even finer grained mobility has been recently explored to allow the automatic update of software on mobile devices [31, 32]. Our approach differs from this research in several aspects: first, it envisions a dynamic execution scenario in which code mobility is an integrating part of the execution process and not a rarely used form of software update and second it proposes a means of dealing with resource constraints associated with mobile devices through code collection.

Finally, garbage collection of code is researched in [28] at a basic block level to address specific constraints of embedded devices. However, the code collection used in this work is a primitive garbage collection algorithm that fails to account the distinctive characteristics of code units. Furthermore, using basic blocks as unit for code management raises some questions about the applicability of this approach in mobile architectures.

7. SUMMARY

We have presented a predictive code collection technique that allows the running of very large applications on resource-poor mobile devices. This technique can be successfully used for code management in virtual machines or runtime systems involving code mobility.

We embedded code collection in a flexible environment permitting, besides low memory usage, a flexible ratio between the occupied memory and the execution time. Our implementation targets a real life, last generation, DSP processor, the StarCore SC140 DSP.

ACCAL, the predictive algorithm we developed for code collection, is designed to use the distinctive features of code-units such as temporal and spatial locality and usage frequency to minimize the number of code-unit misses. The heuristics we use are based on intuition and have been refined by profiling. The experimental data prove that it clearly outperforms classical caching algorithms like LRU or FIFO used for the same purpose. Another important feature of our code collection enabled platform is the ability to keep the memory occupied by code under a certain threshold (specified by ACCAL's upper memory threshold).

Code Collection is an important step towards transforming the vision of the world as an application space to reality. By collecting code, an inherently limited mobile device can execute applications that, if not collected, would exceed by far its physical memory capabilities. Our algorithm optimizes this technique to incur minimum execution time penalty to mobile applications.

8. FUTURE WORK

Possible extensions to this work include optimizing the current implementation, developing automated techniques for coefficient selection and testing on real applications that are representative for the notion of "application space".

There is a potentially large number of improvements that can be made to our implementation. These include extending the set of Java features our platform supports and refining the current gateway and device implementations for better efficiency. An interesting extension is using efficient static or dynamic profiling techniques to predict method loads and to begin method download in advance, with the purpose of minimizing communication penalty.

Machine learning techniques could be used to dynamically change the values of the coefficients for better adaptability and efficiency, therefore relinquishing the need of representative test applications for profiling. The feasibility of this approach must be carefully evaluated as it is directly related to the ratio between the potential increase in efficiency and the runtime overhead it induces.

9. ACKNOWLEDGEMENTS

We would like to thank Motorola DSP Center Romania (MDCR) for supporting this work and address special thanks to our MDCR contact persons, Cornel Margina and Cristi Caciuloiu, and to Virgil Palanciuc who has helped us fix difficult bugs in the StarCore back-end code generator. We would also like to thank the anonymous reviewers for their insightful comments and suggestions.

10. REFERENCES

[1] R. Teodorescu, R. Pandey – *Using JIT Compilation and Configurable Runtime Systems for Efficient Deployment of Java Programs on Ubiquitous Devices*, in Proceedings of Ubicomp 2001

[2] *MSC8101 Reference Manual Second Revision*, Motorola Corporation, 2002

[3] *SC140 DSP Core Reference Manual Second Revision*, Motorola Corporation, 2001

[4] T. Lindholm, F. Yellin - *The Java Virtual Machine Specification*, Second Edition, Prentice Hall, 1999

[5] A. S. Tannenbaum - *Operating Systems Design and Implementation, Second Edition*, p. 331-343, Prentice Hall, 1997

[6] A. Fuggetta, G. P. Picco, G. Vigna - *Understanding code mobility*, in Proceedings of IEEE Transactions on Software Engineering, 1998

[7] A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster- *The Anatomy of a Context-Aware Application* ,Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom), 1999

[8] *The Java HotSpot™ Virtual Machine v1.4.1*, d2, A Technical White Paper, Sun Microsystems 2002

[9] P. Lee, G. Necula - *Research on proof carying code for mobile security*, DARPA Workshop on Foundations for Secure Mobile Code, 1997

[10] T. Brecht, E. Arjomandi, C. Li, and H. Pham - *Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications*, in the Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2001

[11] E. Andreasson, F. Hoffmann, O. Lindholm - *To Collect or Not To Collect? Machine Learning for Memory Management*, in Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02), 2002

[12] P. Scheuerman, J. Shim, R. Vingralek – *WATCHMAN: A Data Warehouse Intelligent Cache Manager*, in VLDB Journal, 1996

[13] M. Román, C. Hess, R. Cerqueira, A. Ranganat, R.H. Campbell, K. Nahrstedt - *Gaia: A Middleware Infrastructure to Enable Active Spaces*, UIUC Technical Report, 2002

[14] J. R. Levine – *Linkers and Loaders*, manuscript chapters, chapter 10, http://www.iecc.com/linker/

[15] *Connected Limited Device Configuration Specification*, version 1.1, Sun Microsystems, 2003

[16] M.C. Merten, A.R. Trick, E.M. Nystrom, R.D. Barnes, W.W. Hwu - *A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots*, In Proceedings of the 27th Annual International Symposium on Computer Architecture ,2000

[17] Y. Ha, R. Hipik, S. Vernalde, D. Verkest, M. Engels, R. Lauwereins and H. De Man - *Adding Hardware Support to the HotSpot Virtual Machine for Domain Specific Applications*, in Proceedings of Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream 12th International Conference (FPL), 2002

[18] G. Kortuem, S. Fickas, Z. Segall - *On-Demand Delivery of Software in Mobile Environments* in Proceedings of the Workshop on Nomadic Computing 1997

[19] C. Krintz, B. Calder, H. Bok, L. Benjamin, G. Zorn - *Overlapping Execution with transfer Using Non-Strict Execution for Mobile Programs*, Proceedings of the Eighth International Conference on Architectural support for programming languages and operating systems, 1998

[20] *OpenJIT: A reflective compiler for Java*, http://www.openjit.org

[21] H. Ogawa , K. Shimura - *OpenJIT, An Open-Ended, Reflective JIT Compiler Framework for Java*, in Proceedings of ECOOP 2000 - Object-Oriented Programming: 14th European Conference, 2000

[22] *Java 2 Platform, Micro Edition Connected Limited Device Configuration* Version 1.1 Reference Implementation, Sun Microsystems, 2002

[23] *Mobile Extreme Convergence: A Streamlined Architecture to Deliver Mass-Market Converged Mobile Devices Rev. 1*, Motorola Corporation, 2003

[24] *StarCore Licenses Its DSP Platforms to Infineon to Power Next-Generation Mobile Communications Products*, news at http://www.starcore-dsp.com, 10th October, 2003

[25] J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin, M. Kandemir - *Exploiting Program Hotspots and Code Sequentiality for Instruction Cache Leakage Management* in Proceedings of the International Symposium on Low Power Electronics and Design, 2003

[26] Licia Capra, Wolfgang Emmerich and Cecilia Mascolo - *CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications* in Proceedings of IEEE Transactions on Software Engineering, 2003

[27] A. Asthana and M. Cravattsand P. Krzyzanowski. *An indoor wireless system for personalized shopping assistence.* In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, 1994

[28] D. Schell, R. Veldema, M. Philippsen-*Basic Block based Garbage Collection for Code,* Submitted to Conf. on Languages, Compilers, and Tools for Embedded Systems 2004

[29] P.R. Wilson – *Uniprocessor garbage collector techniques,* in Proc. Of Int. Workshop on Memory Management, 1992

[30] W.D. Clinger, L.T. Hansen - *Generational Garbage Collection and the Radioactive Decay Model*, in Proc. of SIGPLAN Conference on Programming Language Design and Implementation, 1997

[31] C. Mascolo, G. P. Picco, and G. C. Roman - *CODEWEAVE: Exploring Fine-Grained Mobility of Code*, Automated Software Engineering Journal, To Appear

[32] C. Mascolo, L. Zanolin_, W. Emmerich - *XMILE: an XML based Approach for Incremental Code Mobility and Update*, in Automated Software Engineering Journal, April 2002