Fine-grained Interlaced Code Loading for Mobile Systems

Luk Stoops, Tom Mens[1], and Theo D'Hondt

Department of Computer Science
Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

{luk.stoops, tom.mens}@vub.ac.be
http://prog.vub.ac.be

Abstract. In the advent of ubiquitous mobile systems in general and mobile agents in particular, network latency becomes a critical factor. This paper investigates interlaced code loading, a promising technique that permutes the application code at method level and exploits parallelism between loading and execution of code to reduce network latency. It allows many applications to start execution earlier, especially programs with a predictable startup phase (such as building a GUI). The feasibility of the technique has been validated by implementing a prototype tool in Smalltalk, and applying it to three applications and a wide range of different bandwidths. We show how existing applications can be adapted to maximally benefit from the technique and provide design guidelines for new applications. For applications that rely on a GUI, the time required to build the GUI can be reduced to 21 % on the average.

1. Introduction

An emerging technique for distributing applications involves *mobile code*: code that can be transmitted across the network and executed on the receiver's platform. Mobile code comes in many forms and shapes [10]. Mobile code can be represented by machine code, allowing maximum execution speed on the target machine but thereby sacrificing platform independence. Alternatively, the code can be represented as bytecodes, which are interpreted by a virtual machine (as is the case for Jini [1] and Smalltalk [5]). This approach provides platform independence, a vital property in worldwide heterogeneous networks. The third option, which also provides platform independence, consists of transmitting source code or program parse trees. Note that the side effect of platform independence is that an extra compilation step is necessary before the code can be executed on the receiving platform.

An important problem related to mobile code *is network latency*: the time delay introduced by the network before the code can be executed. This delay has three possible causes. The code must be (1) loaded over a network to the target platform, (2)

[1] Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium)

eventually checked for errors and/or security constraints and (3) possibly compiled or transformed into an intermediate representation. Step (1) is in general the most time-consuming activity, and can lead to significant delays in the startup of the application. This is especially the case in low-bandwidth environments such as the current wireless communication systems or in overloaded networks. Therefore we need to tackle the load phase if we wish to reduce network latency.

In this paper we propose *interlaced code loading*, a promising technique that introduces parallelism between loading and execution of code to reduce the overall network latency. The technique allows us to start up the code before it is completely loaded.

Our experiments involve adapting and running real code and consequently our results are not obtained as part of some simulation technique. Only the network transmission with different transmission rates is simulated in order to evaluate the technique on load channels ranging from very low to very high bandwidths. We also provide some design level guidelines for application developers to take advantage of the loading technique.

The paper is structured as follows. Section 2 presents some basic observations of current network and computer architectures and introduces the technique of interlaced code loading. Section 3 describes the experiments conducted to validate our approach and discuss our findings. Section 4 presents some related work. Next we conclude and present our future work.

2. Proposed Technique

2.1. Basic observations

A first important observation is that code transmission over a network is inherently slower than compilation and evaluation[2] and this will remain the case for many years to come. The speed of wireless data communications has increased enormously over the last years and with technologies as HSCSD (High Speed Circuit Switched Data) and GPRS (General Packet Radio Services) we obtain transmission speeds of 2Mbps [2]. Compared with the raw "number crunching" power of microprocessors where processor speeds of Gbps are common, transmission speed is still several orders of magnitude slower. We expect that this will remain the case for several years to come since, according to Moore's Law [11], CPU speeds are known to double every year.

A second observation is that actual computer architectures provide separate processors for input/output (code loading) and main program execution.

A third observation is that for many applications, if we launch the application over and over again, its program flow after the start will always be the same for a certain amount of time. This time interval is called the predictable deterministic time zone. Most notably those applications that communicate with the user by a graphical user

[2] We utilize the more general term evaluation to describe execution or interpretation of code.

interface (GUI) spend a lot of time building this GUI, and this process is the same each time the application is started. As soon as the user interacts for the first time with the application, the program flow becomes less predictable. Many applications without a user interface also seem to follow a predictable process during startup until their first interaction with an unpredictable environment such as the connection with external systems, generation of a true (non pseudo) random number etc...

The time needed to load, build and display the GUI is called *the user interface latency*. Loading the GUI code first can be very beneficial. The idle time where the system has to wait for user interaction can be exploited to load the rest of the code. This idle time is not negligible. For example, it takes approximately three seconds to select a command using a mouse interface [3].

As a final observation, typical source code contains a lot of low priority chunks for which loading can be deferred until the last moment. A typical example is exception handling (unless exceptions are used to structure the program flow).

2.2. Interlaced Code Loading

The Interlaced Graphics Interchange Format (GIF) [13] is an image format that exploits the combination of low bandwidth channels and fast processors by transmitting the image in successive waves of bit streams until the image appears at its full resolution. We propose interlaced code loading as a technique that applies the idea of progressive transmission to software code instead of images. The proposed technique splits a code stream in several successive waves of code streams. When the first wave finishes loading at the target platform its execution starts immediately and runs in parallel with the loading of the second wave.

In a JIT compilation environment there is an extra compilation phase needed and therefore there are three processes that could potentially run in parallel: loading, compiling and evaluation. Extra timesavings will only occur if different processors are deployed for the compilation and evaluation phase. Nevertheless, even if the same processor shares the processes of compilation and evaluation, the use of JIT compilation is advantageous for the proposed technique. Since the program flow of a classic compilation process is highly predictable it guarantees that during this phase no unpredictable branches will occur, allowing a smooth parallel process between compilation and loading. In other words, incorporating a compilation phase increases the predictable deterministic time zone that is often found at the start of a program.

We deliberately chose for a JIT compilation approach because of its advantages in a low bandwidth environment: (1) Source code has a smaller footprint than the corresponding native code; (2) Source code preserves a high level of abstraction, thus enabling more powerful compression techniques; (3) JIT fits nicely in the proposed code interlacing technique since, as explained before.

However, to anticipate possible criticism that the results heavily depend on this extra compilation step we did not apply parallelism between the loading and compilation phase. All the obtained results were obtained from parallelism between the running application and the code loading only. In our setup the compilation phase is part of the load process. If we also apply parallelism between compiling and code loading the time gain will increase even more.

An important question is: what is the ideal unit of code to be split into successive waves? We propose to use as unit those program abstractions where the code was build from. For example, in Smalltalk likely candidates at different levels of granularity would be: statements, methods, method categories, classes, class hierarchies, class categories, etc… The unit of code should be sufficiently small to achieve a high flexibility in the possible places where the code can be split, which in turn enables a higher degree of parallelism. In object-oriented languages, it seems most appropriate to use methods as unit of code[3]. Especially for well-written object oriented programs that adhere to the good programming practice of keeping methods small, the splitting flexibility remains high.

Before we can start to cut the code into different chunks we need to **permute** the source code in such a way that the code that will be executed first will be loaded first as well. After the cutting, we need to apply some glue code in the form of **semaphores**. Semaphores temporally suspend the application if the next chunk of code is not yet loaded.

The algorithm used to permute the source code is based on the dependencies between the different Smalltalk entities. A **method** cannot be loaded and compiled if the method's **class** description is not already available in the system. So in the Smalltalk environment a method depends on its class description. In the same spirit we notice that a class depends on its **superclass**, a class depends on its **namespace**, a class initialization method depends on its class description and depends possibly on semaphore code that eventually can prevent its invocation. A class also depends on the availability of relevant **shared variables** and, if the class is a subclass of ApplicationModel, the availability of the associated window specification **resource**. These dependencies are not complete to cover all the possible Smalltalk applications but were considered to be sufficiently comprehensive to cover all the dependencies in the actual experimental setup.

3. Experiments

3.1. Setup

We describe some experiments to illustrate a generic approach of interlaced code loading and to provide a proof of concept. A prototype tool was implemented in Smalltalk (more specifically, VisualWorks Release 5i.4), a popular object-oriented language that allows fast prototyping.

As a practical validation we tested our approach on three applications each exhibiting some typical but distinct behavior. We feel that these three are representative for a

[3] While code loading is achieved at method level in Smalltalk, this is not the case in Java for security reasons: the unit of Java code loading must be a class.

whole range of typical mobile applications and suffice for a proof of concept. Nevertheless, experiments on a larger scale are needed to validate this approach for other types of applications as well.

Benchmark: (ver: 5i.4) (80 kByte, 7 classes) A program that comes with the VisualWorks environment adapted in such a way that after its Graphical User Interface (GUI) appears, it launches a standard test immediately, thereby simulating prompt user interaction.

CoolImage: (ver: 2.0.0 5i.2 with fixes) (184 kByte, 60 classes) An extended image editor that draws on a non-trivial graphical user interface.

Gremlin: (ver: Oct 7 '99) (65 kByte, 4 classes) An application that lets an animated figure pop up from time to time without the need for a user interaction, representing non-GUI applications.

To test these applications we designed a code loader to simulate different transmission rates. Essentially the code loader waits for the amount of time needed to load the file containing the code, under different network bandwidths before effectively loading the code from disk and passing it on to the compiler.

For this setup six transmission rates were simulated: 2400 bps (very low bandwidth), 14.4 kbps (slow modem), 56 kbps (fast modem), 114 kbps (GPRS) en 2 Mbps (UMTS). These different transmission rates were complemented by the rate obtained without network latency: 41 Mbps in our setup.

3.2. Permuting the source code

To simplify the permutation process somewhat, on this first setup we assumed that the code flow is completely deterministic. In other words, we assumed that for each run of the code the application behaves always the same way, hereby neglecting possible different user inputs or other random events. This makes the permutation process straightforward since it suffices to determine the method invocation sequence once and rearrange the methods accordingly. The static structure of the permuted file will then reflect more closely its dynamic behavior.

Finding the ideal breakpoints is less straightforward. Profiling tools together with the dynamic behavior statistics, obtained as a side effect during the permutation process can give us some hints as where to split the code. In our initial experiment we will resort to some simple heuristics, such as cutting the file into equal pieces.

The permutation process, which is completely automated in our setup, consists of several distinct steps (Fig. 1). To obtain the necessary *method invocation sequence* the *original source code* is instrumented with extra code that logs the time of invocation of each method. The instrumentation is accomplished by the *source code instrumenter* component. Then the *instrumented source code* is evaluated. The output is ignored at this time but the instrumented methods will generate the necessary log information, in this case an XML file that contains the *method invocation sequence*.

In another phase, which could be carried out in parallel with the steps described previously, the *original source code* is parsed by the *source code parser* component and the resulting descriptions (class, methods, comments and other descriptions) are stored in an intermediate *repository*.

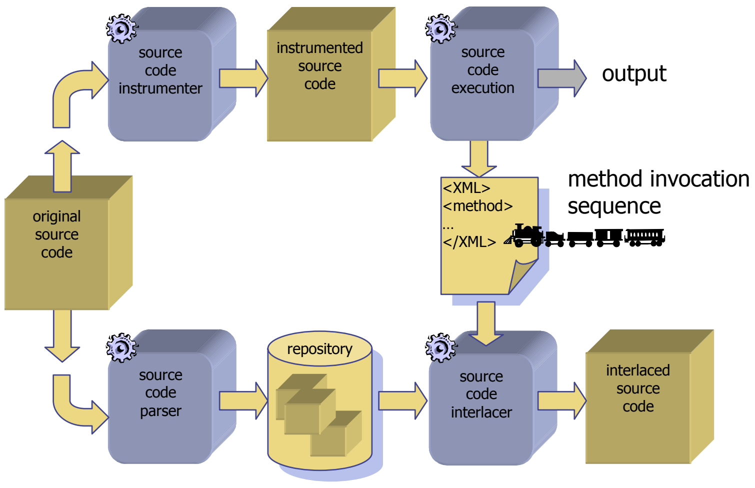

Fig. 1. Permuting the source code

In the final step a *source code interlacer* parses the XML file to retrieve the dynamic sequence of the method invocations and uses this information to assemble a new *interlaced source code* file that reflects this invocation sequence.

For each tested application, the source code was automatically interlaced using the steps outlined above. For logging purposes a few extra lines of code were manually added to log the time the application needs to complete execution and also the time needed to produce its GUI or its first token of existence to the user. The application is loaded, compiled and run as is and then via a load channel simulating a number of different bandwidths, to gather the normal timing information referred to as "normal end" and "normal GUI" in the figures later. Next, the application is cut in four pieces. The following procedure is applied:

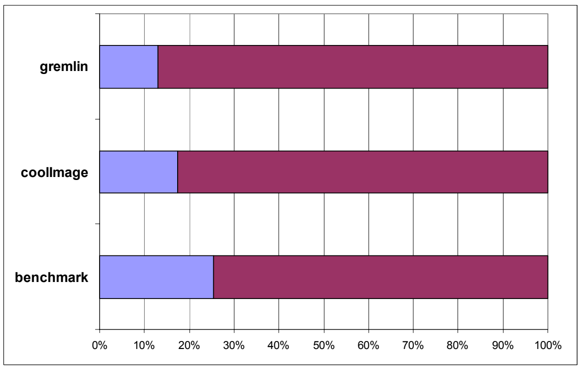

Fig. 2. Percentage of code visited before the graphical user interface becomes available

By examining the interlaced code, it is fairly easy to determine the relative part of code visited by the evaluator to build the GUI (Fig. 2). For the user the emergence of the GUI is often the first indication that the underlying application is loaded and ready to go. To favor a quick emergence of the GUI we will try to make the first cut immediately after the GUI code. In this way we can exploit the inevitable user delay [3], during which the system waits for user interaction, to load the rest of the code. If the method that finishes off the GUI is in the first half of the source code, as in our three test applications, then the first cutting place will be after that method. The remaining code is then equally divided in the three remaining parts: part 2, part 3 and part 4.

Three semaphores are then added at the end of the three loose ends of part1, part2 and part3. The semaphores are added to the last method at the beginning of its method body to avoid possible return messages and therefore to be sure that it will be executed. The methods, in which the semaphores reside, are possibly invoked more than once. This means that the semaphore must be disabled after its first use. In this setting this is done by enclosing each semaphore in a conditional structure in such a way that the semaphore is bypassed after it's first use:

```
Interlacer.S1Active ifTrue: [Interlacer.S1 wait.
Interlacer.S1Active := false]
```

The application is then loaded, compiled and run again in an interlaced style for each of the simulated channel bandwidths and the new timing results are gathered. These are referred to as "interlaced end" and "interlaced GUI" in the figures later.

Each timing result is calculated as the average of three timing runs to be able to flatten occasional variations caused by the operating system or programming environment such as garbage collection.

3.3. Timing results

For each of the three test applications result times were measured with the six different bandwidths. For each of these bandwidths the time was measured in a normal set up (first load all the code and then compile and run) and an interlaced set up where the compilation and start of the code takes place after the first part is loaded. For both loading types we measured the time it took for the GUI to display itself and the total time to complete the loading, compilation and evaluation of the application.

The experiments where carried out on a Dell® Inspiron 8100 computer with Intel® Pentium® III Mobile CPU AT/AT compatible processor at 1GHz processor speed and 256 Mb RAM running Windows® 2000 and VisualWorks 5i4.

3.3.1. Benchmark

Benchmark is an application that runs selectable tests on the VisualWorks environment. For this test the application was adapted in such a way that after the GUI pops up the application immediately runs a number of standard tests.

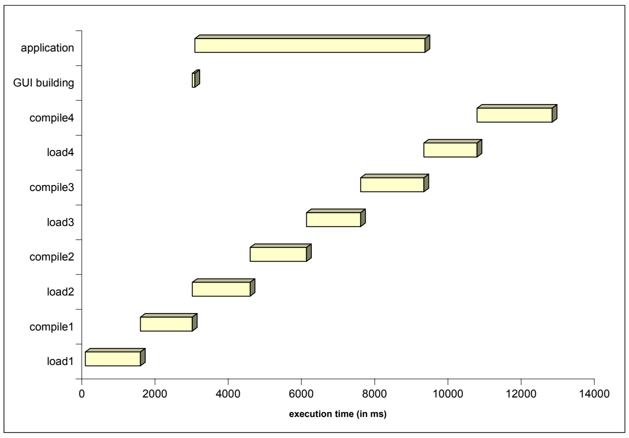

Fig. 3. Parallel execution Benchmark @ 114 kbps

Fig. 3 shows the parallel processes achieved at a bandwidth of 114 kbps where load and compilation times are at the same order of magnitude. *GUI building* indicates the first part of the evaluation process where the GUI is built. The second part of the evaluation process is indicated in the figure by *application*. The evaluation process has to share the processing power with the compile phases but can run in parallel with the load phases (except for load1). Note from Fig. 3 that the evaluation process can take advantage of the relatively long periods of load2 and load3 to be able to finish early. It will even finish before all the code is loaded. This means that all the code that remains to be loaded is not needed for the actual execution. Hence, we may stop loading the rest of the application.

Table 1. Timing results (in ms) for Benchmark application

Bandwidth (kbps)	2.4	14.4	56	114	2048	42308
normal GUI	279268	51184	18074	12352	7016	6562
normal end	280255	52175	19069	13361	8133	7526
interlaced GUI	74327	13341	4669	2995	1722	1341
interlaced end	221564	40291	14279	9279	6062	7609
GUI ratio	26.61%	26.06%	25.83%	24.25%	24.54%	20.43%
end ratio	79.06%	77.22%	74.88%	69.44%	74.54%	101.10%

Timing results are depicted in Table 1 and Fig. 4. The first row of Table 1 (normal GUI) shows the time in milliseconds it normally takes to render the GUI for the different bandwidths. The second row (normal end) shows the time in milliseconds the application normally needs to end. The third and fourth rows (interlaced GUI and interlaced end) show the same time if the application is deployed in an interlaced code loading fashion. Finally the bottom rows (GUI ratio and end ratio) show the relative amount of time gained by interlacing to present the GUI and to finish the application.

In Fig. 4, the x and y scale are logarithmic to accommodate the wide range of bandwidths. Note also from this figure that, if the application is loaded via a network (all rows except the last one where no network latency was simulated), the application itself ends earlier (on average 75% of the original time needed) if deployed in an interlaced mode

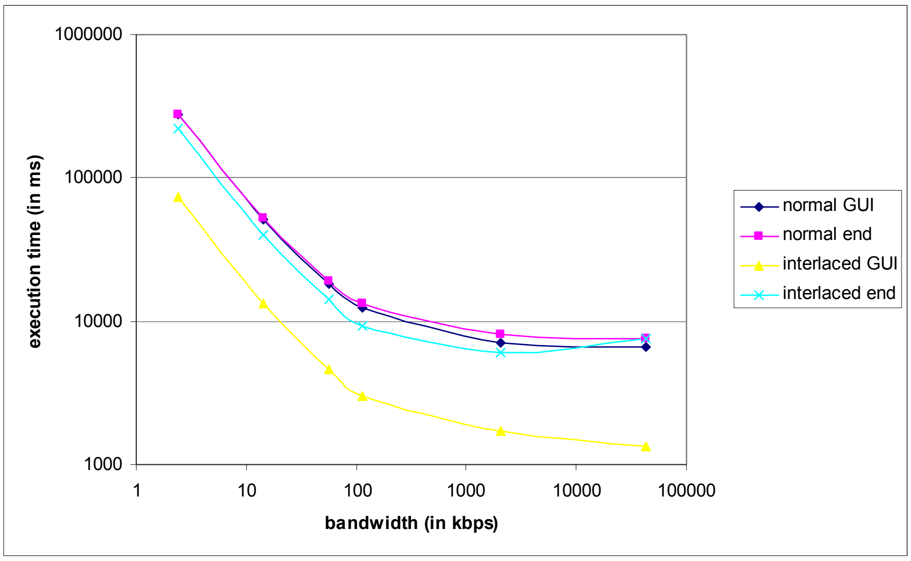

Fig. 4. Timing results for Benchmark application

3.3.2. CoolImage

CoolImage is the largest application of the three, which draws a large GUI and then waits for user interaction to draw icons. As a result, the end of the loading and compile phase is practically the same for the interlaced and normal deployment simply because in this test no action takes place after the GUI building.

Table 2. Timing results (in ms) for CoolImage application

Bandwidth (kbps)	2.4	14.4	56	114	2048	42308
normal GUI	640540	114905	38978	25348	13624	12666
normal end	640545	114909	38982	25351	13628	12673
interlaced GUI	115005	22832	8847	6663	4780	4224
interlaced end	638613	115496	39135	25192	13888	13037
GUI ratio	17.95%	19.87%	22.70%	26.29%	35.09%	33.35%
end ratio	99.70%	100.51%	100.39%	99.37%	101.91%	102.87%

Table 2 contains the timing results. The end ratios are almost equal to 100 %. This indicates that the time the application itself needs to end, in this case the time to load and compile all its code behind its GUI, will not vary. However, the appearance of the GUI in the interlaced deployment is much faster and in the same order as the other tests (on average 25% of the original time needed).

3.3.3. Gremlin

Gremlin is an application that runs in the background of the VisualWorks environment and pops up an animated figure from time to time at the border of the active window. When the application is launched, the animated figure pops up for the first time and a help window shows up. Table 3 shows the delays of the Gremlin application.

Table 3. Timing results (in ms) for Gremlin application

Bandwidth (kbps)	2.4	14.4	56	114	2048	42308
normal GUI	230743	46392	19563	14713	10469	10262
normal end	230745	46394	19565	14715	10471	10264
interlaced GUI	51601	15194	11932	11005	10283	10439
interlaced end	225385	39441	12777	11005	10283	10439
GUI ratio	22.36%	32.75%	60.99%	74.80%	98.23%	101.72%
end ratio	97.68%	85.01%	65.31%	74.79%	98.21%	101.70%

Since the Gremlin application starts with a popup of an animated figure and during the rest of its life it just does the same thing over and over again at different time intervals it means that all the resources needs to be in place before the application can start. This is reflected in Table 3 by the fact that only for bandwidths lower than 56 kbps the *GUI ratio* is lower than the *end ratio*, i.e., the first popup can finish earlier than the complete loading and compilation process. For bandwidths greater than 56 kbps it is the popup process itself that will determine the end of the process.

The poor results of the GUI ratio obtained with the Gremlin application lead us to the question whether it is possible to adapt the design of the application in such a way that interlacing could be applied more advantageously. If we could change the application in such a way that it would not depend any more on all of its resources, for its first token of life, this would do the trick.

To achieve this, we adapted the Gremlin application so that after it is launched only the help window appears (containing an explanation of the behavior of Gremlin and stating that the first popup is scheduled within 5 minutes). This is only a minor change to the main behavior of the application but as Table 4 shows there is now a significant time gain possible for the GUI building (now the text window) and the end of the application (now the loading and compilation of the source code but before the first popup).

We came to the conclusion that small changes at the design level sometimes suffice to get a significantly better behavior in an interlaced loading environment.

Table 4. Timing results (in ms) for adapted Gremlin application

Bandwidth (kbps)	2.4	14.4	56	114	2048	42308
normal GUI	223468	39420	12568	7890	3544	3150
normal end	223470	39422	12569	7892	3546	3152
interlaced GUI	44183	8261	3049	2223	1413	1228
interlaced end	224902	39441	12596	7968	3557	3197
GUI ratio	19.77%	20.96%	24.26%	28.17%	39.88%	38.98%
end ratio	100.64%	100.05%	100.22%	100.97%	100.29%	101.45%

3.4. Discussion

3.4.1. Speedup

From the results presented in the tables it becomes clear that everywhere where the *GUI ratio* and/or *end ratio* are below 100 % a speedup was achieved. Note that Benchmark is the only application where none of the graphs coincide with each other. (see Fig. 4). This is because the Benchmark application is the only example that runs some time-consuming benchmark tests after the appearance of the GUI. The two other applications do not immediately use the processor after building the GUI.

Fig. 5 shows the relative amount of time needed to present the GUI compared with a normal non-interlaced setup for the different bandwidths. If we neglect the original non-adapted Gremlin application we find that an average speedup of 21% is obtained.

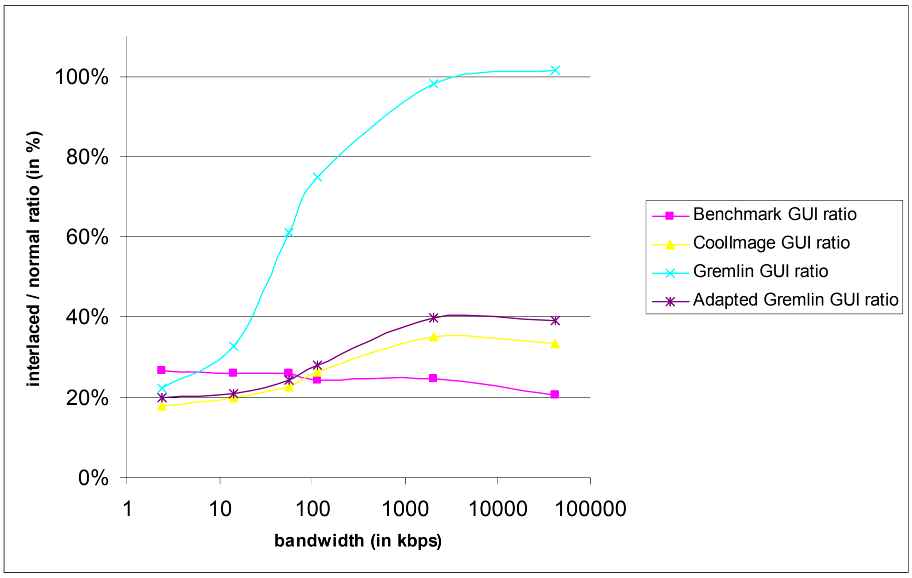

Fig. 5. Time needed to build GUI compared with original time

For applications were the GUI building takes a relatively large part (such as CoolImage and Gremlin) the speedup gain achieved by interlacing seems to decrease as loading speed increases. In the extreme case of Gremlin where the GUI building needs all the resources in place the application takes even a slightly longer time to execute. This is because the extra semaphore code in the source code and the code to guide the interlaced loading process yield an extra overhead, and are responsible for time ratios higher than 100 %.

3.4.2. Interlacing Guidelines

As became apparent in the Gremlin case it can be advantageous to adapt existing programs to make full use of the power of interlaced loading. Especially when writing new applications from scratch it is possible to follow guidelines that lead to an optimal interlaced code loading. More research is needed to device these guidelines but some of the obvious ones are:

- Keep programming modules independent from each other (i.e., use low coupling and high cohesion).
- Start as soon as possible with building the GUI.
- Keep the code and the resources needed to present the first user interface as small as possible. Mostly this is the GUI the user is confronted with at startup.
- If necessary, enhance the GUI (e.g., extending the GUI menu), at a later time.
- Postpone heavily resource-dependent actions as long as possible.
- Postpone multithreaded processes as long as possible.

3.4.3. Dealing with semaphores

As mentioned before, precautions must be taken to prevent methods to be triggered that are not loaded yet. Although it is possible to catch these exceptions on the level of the virtual machine or even on the level of the operating system, for this setup we chose for the generic approach of adding semaphores in the source code.

It can be assumed that for every application there will exist an ideal number of pieces to split the code in to obtain a maximum speedup. If the number of pieces increases so will the total size of the code since each piece of code will need extra statements to present the semaphore code. And if the code size increases so will the loading time and since the extra code needs to be evaluated too, also the evaluation time. Times that we wanted to decrease in the first place. Furthermore there will be an extra overhead at the receiver and sender platform to administrate the loading, compiling and evaluation of the different parts.

More experiments are necessary to determine the optimal number of parts, but as shown in the examples a simple heuristic of cutting the source code in four pieces and trying to put the first break at the point where the first GUI is built provides already significant results.

Provisions need to be made to disable the semaphores once they have served their purpose for the first time. Placing them in a conditional branch that bypasses them after first use seems to be a valid option and this is the choice that we took in the experiments of this paper. If the method in which the semaphore is placed is triggered a significant number of times, complete removal of the semaphore code after its first use can be considered. Access to a precompiled version of the same method without the semaphore code can speed up that process. Deploying garbage collection agents to remove unused semaphores in the background is another possible approach. On the other hand, if we are dealing with mobile code that moves continuously from host to host it may be advantageous to keep the semaphores in place.

4. Related Work

There are a number of different techniques that have been proposed in the research literature to reduce network latency: code compression, exploiting parallelism, reordering of code and data, and continuous compilation.

Code compression is the most common way to reduce overhead introduced by network delay in mobile code environments. Several approaches to compression have been proposed. Ernst et al. [4] describe an executable representation that is roughly the same size as gzipped x86 programs and can be interpreted without decompression. Franz [7] describes a compression scheme called slim binaries, based on adaptive methods such as LZW [14], but tailored towards encoding abstract syntax trees rather than character streams. The technique of code compression is orthogonal to the techniques proposed in this paper, and can be used to further optimize our results.

Exploiting parallelism is another way to reduce network latency. Krintz et al. [8] proposed to simultaneously transfers different pieces of Java code in parallel, to ensure that the entire available bandwidth is exploited. Alternatively, they proposed to parallelise the processes of loading and compilation/execution, a technique that is also adopted by this paper. Compared to our paper, Krintz et al. also suggest parallelisation at the level of methods, and their experiments yielded even better results than ours: transfer delay could be decreased between 31% and 56% on average. An important difference with our approach is the implementation language (Java instead of Smalltalk). Moreover, because of the limitations of the Java virtual machine security model, Krintz et al. simulated their experiments. Additionally, they only considered two different bandwidths while we explored a wider range of 6 different bandwidths in this paper.

Reordering of code and data is also essential for reducing transfer delay. Krintz et al. [9] suggest splitting Java code (at class level) into hot and cold parts. The cold parts correspond to code that is never or rarely used, and hence loading of this code can be avoided or at least postponed. With verified transfer, class file splitting reduces the startup time by 10% on average. Without code verification, the startup time can even be reduced slightly more.
To determine the optimal ordering of code, a more thorough analysis of the code is needed. This can be done either statically, using control flow analysis, or dynamically, using profiling. Both techniques are empirically investigated in [8] to predict the first use ordering of methods in a class. These techniques are directly applicable to our approach as well. More sophisticated techniques for determining the most probable path in the control flow of a program are explored in [6].

Continuous compilation and **ahead-of-time compilation** are techniques that are typically used in a *code on demand* paradigm, such as dynamic class loading in Java. The goal of both compilation techniques, explored in [9] and [12], is to compile the code before it is needed for execution. Again, these techniques are complementary to our approach, and can be exploited to further optimize our results.

5. Conclusion

Network latency becomes a critical factor in the usability of applications that are loaded over a network. As the gap between processor speed and network speed continues to widen it becomes more and more opportune to use the extra processor power to compensate for the network delays.

Performance of an application is most commonly measured by overall program execution time but in a mobile environment performance is also measured by invocation latency. Invocation latency is the time from application invocation to when execution of the program actually begins. From the viewpoint of the user the most crucial latency is the user interface latency, being the time a user has to wait between his demand and a user interface reaction of the system. Exploiting parallelism between loading and execution proves to reduce user interface latency considerably (21% of the original time on average in three applications tested). Besides this reduction of the user interface latency also the overall program execution time can be significantly reduced (75% of the original time in the Benchmark application).

Except for the simulation of a large range of transmission rates our experiments do not rely on simulation techniques what makes us confident about the obtained results.

6. Future Work

An industrial research project (funded by the Belgian government) that will start end 2002 is situated around mobile code and low bandwidth environments. This setting will give us the real live test environment to validate our approach further on different platforms and will allow us to get more detailed results. Experiments with interlaced code loading will be performed on a larger scale, including applications and benchmarks that will reflect the typical mobile agent application behavior. The results and lessons learned will be distilled in interlacing design guidelines to guide developers willing to take advantage of low user interface latency.

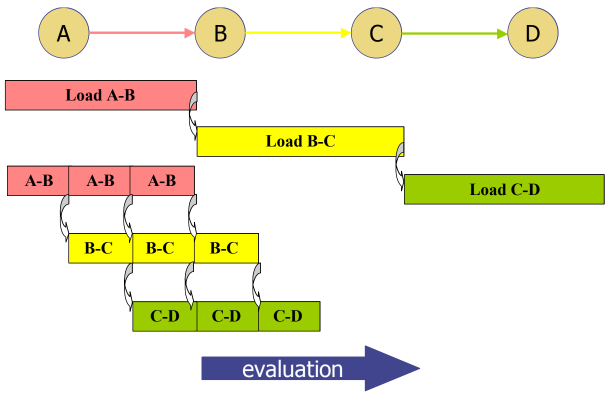

Fig. 6. Mobile agent traversing a multi-hop network in an interlaced mode

We will also apply interlaced code loading on mobile agents operating in multi-hop networks. This environment promises an even more substantial decrease of the invocation latency. See Fig. 6 that compares a classic and interlaced code loading in a multi-hop network. In the example the code is split in three parts.

Further we will look for a more formal approach to decide where to cut the original code and how and where to add semaphores or other guarding systems. Genetic algorithms may provide us the right tool to find the most opportune cutting places.

Acknowledgments

We thank Alexandre Bergel, Karel Driessen, Johan Brichau, Franklin Vermeulen, Gert van Grootel and the anonymous referees for reviewing the paper. We also thank the Programming Technology Lab for their valuable comments.

References

1. K. Arnold, B. O'Sullivan, R.W. Scheiffer and J. Waldo, A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999
2. S. Barberis, *A CDMA-based radio interface for third generation mobile systems*. Mobile Networks and Applications Volume 2 , Issue 1 ACM Press June 1997
3. R. Dillen, J. Edey and J. Tombaugh. *Measuring the true cost of command selection: techniques and results.* CHI '90 proceedings ACM Press April 1990
4. J. Ernst , W. Evans , C. W. Fraser , T. A. Proebsting , S. Lucco , *Code Compression*. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation. Volume 32 Issue 5, May 1997
5. A. Goldberg and D. Robson, *Smalltalk-80 The Language*, Addison-Wesley Publishing Company ISBN 0-201-13688-0 1989
6. R. Jason, C. Patterson, *Accurate Static Branch Prediction by Value Range Propagation* Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 67-78. (La Jolla, San Diego), June 1995
7. M. Franz and T. Kistler. *Slim Binaries.* Comm. ACM Volume 40 Issue 12, December 1997
8. C. Krintz, B. Calder, H. B. Lee, B. G. Zorn, *Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs*. Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, California U.S., October, 1998
9. C. Krintz, B. Calder and U. Hölzle, *Reducing Transfer Delay Using Class File Splitting and Prefetching,* Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications, November, 1999
10. D. Milojicic, *Mobility processes, computers and agents,* ACM Press 1999
11. G. Moore, *Cramming more components onto integrated circuits,* Electronics, Vol. 38(8), pp. 114-117, April 19, 1965.
12. M. P. Plezbert , Ron K. Cytron, *Does "just in time" = "better late than never"?* Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, p.120-131, Paris, France, January 15-17, 1997
13. D. Siegel, *Creating killer web sites,* Indianapolis: Hayden Books. 1996
14. J.Ziv and A.Lempel, *A Universal Algorithm for sequential Data compression,* IEEE Transactions on Information Theory Vol 23, No.3, May 1977