Efficient Application Migration under Compiler Guidance

Kun Zhang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
kunzhang@cc.gatech.edu

Santosh Pande
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
santosh@cc.gatech.edu

Abstract

Mobile computing based upon wireless technology as the interconnect and PDAs, Web-enabled cell phones etc. as the end devices provide a rich infrastructure for anywhere, anytime information access. Wireless connectivity also poses tough problems [11, 12]. Network nodes may be mobile and the connectivity could be sporadic. In many cases, application mobility involving migration from one network node to another could provide interesting possibilities. However, the migration process is expensive in terms of both time and power overheads. To minimize the migration cost, an efficient strategy must decide which parts of the program should migrate to continue execution and at which program point the migration should take place.

In this work we develop a compiler framework to achieve the above two goals. First, the potential migration points are decided by analyzing the call chains in the code. Then the compiler determines what parts of the program are dead at these points. At run time, using the current context of the call chain, a decision on whether to migrate now or later is taken. Such a decision depends mainly upon the cost of migration involved at the current program point vs. at a later potential migration point. Our experiments with multimedia applications show that both the migration state and the latency are significantly reduced by our techniques over the base case of migration with full state in the absence of any compiler guidance. Thus, the key contribution of the paper is to provide an efficient migration methodology removing barriers to application mobility.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors – *Compilers, optimization.*

General Terms Algorithms, Languages, Performance, Measurement.

Keywords Application Migration, Compiler, Mobile Computing, IEEE 802.11

1. Introduction

Mobile computing (mostly based on wireless technology) provides a rich infrastructure for anywhere, anytime information access. Mobile devices such as personal digital assistants (PDAs) as well as web enabled cell phones are widely used over a wireless network. Mobile computing is becoming increasingly prevalent as it allows the users to have their electronic work follow them whenever a networked environment exists. However the wireless technology poses tough challenges to be overcome [11; 12; 19]. For example, the mobile communication is bad; the bandwidth in mobile computing is relatively low and the latency is relatively high; and mobile computers are susceptible to high error rates and sudden link failures.

On the other hand, application mobility is an important asset that could be realized; one of the ways to achieve application mobility is to migrate a partially executed application. Such a mechanism is especially suited for applications on partially connected mobile computers such as laptops and palmtops. For example, a time consuming application executing on a machine which may be prone to failure due to battery capacity, potential memory bottlenecks discovered at run-time etc. can migrate off to a more robust remote machine [10]. In this way, the application can still be available on other machine and it can resume its execution there.

Application migration can also improve an application's performance based on the network conditions and the local resource of the mobile devices. An application migration framework exposes interesting geo-locality issues to be factored into the application semantics improving its precision. For example, migrating an application to a local server which has more up-to-date real time information on traffic conditions might lead to a better route suggestion than just using cached data obtained during the last connection inside a car navigation system. Migration could be an important mechanism pertaining to the *availability* of a system. For example, imagine an application which involves stock market trading; assume that a user is in the middle of a stock transaction using his PDA or cell phone. His handset is running out of power. He may not want to simply postpone his transaction. If availability guarantee has to be seamlessly achieved, it is best to migrate the entire transaction and its state to another tethered device to continue this transaction. Similarly, continuous process based systems operate on the premise of availability and recoverability - an important means to achieve both is to first migrate the execution on another system and then attempt to recover and continue execution. Sometimes it is better to migrate an application to the data host than migrate the data to the host of the application since the data cannot be relocated (due to trust or proprietary reasons) or the movement of data would cause longer lag (e.g., a large database). In short, migration can be applied in many different types of situations.

Research pertaining to classical migration is presented in [1; 7; 8; 20]. However, migration on embedded devices can be very expensive. Migration may entail a considerable time lag and consume a tremendous amount of power [2] due to an application's large state and the low bandwidth between mobile machines, which are often battery powered. So the key question in migration is: How to minimize the latency in migration? The answer depends on three factors: Which parts of an application should migrate? At which program point the migration can minimize the transmission time? How to devise an effective mechanism to realize the above two goals?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-018-3/05/0006…$5.00.

It is important to devise such a guidance mechanism to do this properly since it may take a long time (as we see through our experiments) if one does it naively. In this work, we answer these questions by building a compiler + run-time solution. The answers will help provide insights into the compiler's role in migration, which has mainly been an operating system/middleware's concern.

This paper is organized as follows. Section 2 describes the background and motivation for application migration; section 3 represents our framework; section 4 illustrates the algorithms in detail; section 5 shows the results of our experiments; section 6 discusses the related work; finally section 7 concludes our paper.

2. Background and Motivation

Application migration can be roughly classified into two categories according to the architecture and operating system of the machines: homogeneous application migration and heterogeneous application migration [20]. Homogeneous migration can only take place between machines running the same operating system on the same architecture. Heterogeneous migration can occur between different architecture and operating system. In this work, we focus on homogeneous application migration between embedded systems based on ARM architecture.

The necessary component required for resuming an application is called the *application state*. The components of an application state depend on the machine where the application is running and may show substantial difference on different machines. However, essential parts of an application state remain the same across the implementations. A typical application state includes virtual memory, opened files, machine state and environment data etc.

Migration before an application starts its execution is quite different from migration after the application has executed for some time. In the first case, migration is easy since we do not have to consider the application state. In this work, we target the second case in which the application state must be analyzed carefully in order to minimize the time lapse experienced by an application between suspension and resumption. In the following, we will elaborate on such migration that takes place in the middle of an application's execution.

Migration allows seamless execution to continue on another machine. The main process can be divided into five parts [2; 21]: suspend the application on the source machine; gather the state and represent it using some meta-data (this process is often referred to as *serialization*), transfer the application state to the destination machine; re-establish the state by loading it; and resume the execution on the destination machine. The most expensive part involves serialization and transmission of the serialized state to another machine. So the best way to minimize the migration time is to minimize the application state that gets migrated.

There are many different factors impacting migration such as the number of applications running on the machine, CPU utilization, file systems, trust, network connectivity, or any combination of the above indicators. Other factors to consider include where to migrate geographical proximity to the resources, operating system capabilities and specialized hardware/software features [20]. Providing the whole systemic solution is not feasible nor is the goal of this paper. In this paper, we address the issue of migration dictated only by the state serialization and data transmission time given other constraints being satisfied. The scenario is that the middleware has taken migration decision after considering other factors and it now seeks help from the program's runtime system (orchestrated by the compiler) on how and where to carry out the migration. Under the guidance of the runtime system, the program is prepared for migration through serialization and then migrates under the middleware's supervision.

Several mechanisms have been designed and implemented to move an application among the nodes of a network in a distributed system [7; 8]. However, the current application migration schemes proposed in the OS literature mainly deal with issues about where to migrate, which applications to migrate, and how to select meta-representation for migration, but do not deal with the critical issues of *which parts of the state to migrate* and *when to migrate in the middle of the program's execution*? Such decisions involve knowledge of program properties that affect the state. Migration becomes more attractive if such an analysis is undertaken and is coupled with runtime system guidance. For example, current approaches do not undertake *file and code liveness analysis*, usually resulting in all of the opened files and code being migrated to another machine. Such an approach which transfers all the data and code entails considerable time and is not feasible esp. for applications involving large data. An analysis that allows determining which parts of the state should migrate and when to migrate makes a big difference as shown in this paper. We achieve such a solution in this paper through a combination of static compiler analysis and runtime system guidance.

3. Framework Overview

In this work, we target distributed embedded systems in the setting of an 802.11 [14] local area wireless network environment. The embedded device is a widely used PDA, the Compaq iPAQ based on Intel's StrongARM processor. IPAQ is a small multimedia-centric PDA that can act as a lightweight entertainment station for movie, music and more. It is a versatile device enabling the use of wireless communication. It usually has SDRAM (Synchronous DRAM), non-volatile Flash ROM and a high performance, low power demand CPU. More Flash ROM can also be plugged in as additional storage. The applications are generally written in either the C or C++ programming language.

In our approach, the compiler analysis determines the potential migration points and the program components to migrate. Our framework is shown in Figure 1. First, we instrument the program to keep track of the file operations, stack size, heap size and the time consumption on the paths of the program. The file operation could be one of open, read, write and close. The instrumented application is executed using the training sets to collect the profile data. Second, the reaching graph for the application is built. The reaching graph represents the reachability relationship between the functions and the file operations. The reachability relation shows whether or not it is possible to have a call sequence from (current) function F to certain operations on file f. File f is said to be live in F if such a call sequence exists. If a file is live in (current) function F, it must be carried since there may be an execution path leading to it in the future execution; otherwise it may be discarded during migration. We use function pointer analysis to discover multiple aliases of function pointers. The static call chains are built using this information. By accounting for these factors, the reachability analysis offers a safe mechanism

to determine whether or not a file should be carried during migration. Next the code and global data liveness is analyzed. It is a simple backward data flow analysis such as variable liveness analysis in [16]. The heap liveness information is also collected. Next, by combining the code, global variable and heap analysis with stack size and file liveness information, the cost of migration at potential migration points is decided for a given function. Finally, the migration handler is inserted by the compiler in the code providing guidance as a part of runtime system.

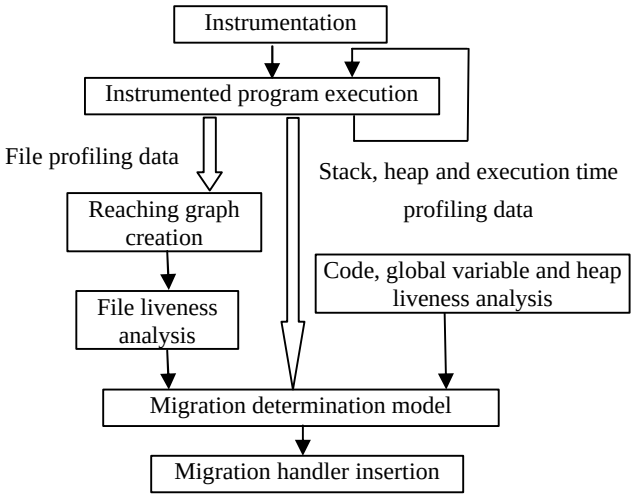

Figure 1. Framework overview

Next we will explain how our mechanism works in the dynamic execution of a program. When a program is running, the operating system invokes a migration interrupt to inform the program to prepare for migration. The migration handler is triggered by the runtime system upon receiving this interrupt. The handler determines the current execution context and predicts future paths to potential migration points. Using the application's current state and profile data, the migration handler predicts the most likely future execution path. Along this path, it will determine earliest such point where the delay benefit/cost ratio is maximized. Delay benefit is the time that can be saved at that migration point and cost is the expected execution time to that point. The migration handler then makes a decision on whether to migrate now or continue execution based on the cost-benefit tradeoffs at the potential migration point. If execution continues, the run-time system checks if the predicted execution path is followed or not. If it is not followed, it re-evaluates the prediction and makes a new decision on migration. At the migration point, the dead files are discarded; the dead data and code is also discarded.

4. Migration Algorithms

4.1 File Analysis

Most media applications make use of a number of large files. Discarding many of these files during migration may generate a tremendous time and energy savings. Most programs use a potential set of files whose name-set can be found through analysis. A real example (Mpeg2 encoder) is used through this section to illustrate how our approach discovers this information.

Before we go into the file analysis, some important conditions of our work are presented. The set of all files used in an application is called its *file universe*. A program's file universe is said to be bounded if we can derive all files names and their locations through hard-coded files names. That means that potential files which will be used by the program are derivable by analysis before its execution. Our framework is not intended for applications whose file universe is unbounded (such as OS file system support, for example). The applications we work on have a bounded file universe and can be found using closure.

For example, there are fifteen files used in the execution of the Mpeg2 encoder. They are: options.par, rec_files (including 0.y, 0.u, 0.v, 1.y, 1.u, 1.v, 2.y, 2.u, 2.v, 3.y, 3.u and 3.v), stat.out and output.m2v. The description of these files is shown in Table 1. In this example, the bitstream of YUV components is encoded. To make it easy, suppose only 4 frames for encoding. Mpeg2 encoder uses a pre-set convention that the file "options.par" contains the names of the "rec_files" which hold the frame data.

File name	File description
options.par	Options of encoding (input file)
rec_files	YUV components to be encoded (input file)
stat.out	Statistics file (output file)
output.m2v	Encoded file (output file)

Table 1. Files and their description for Mpeg2 encoder

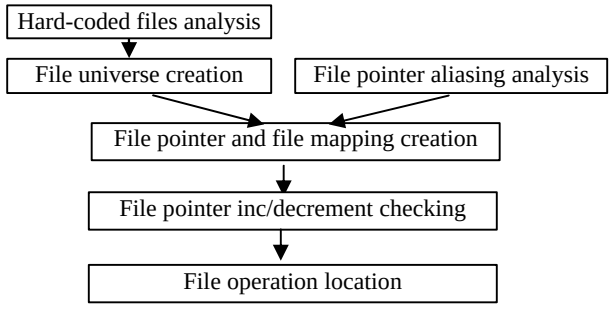

Figure 2. File operation analysis

The file operation analysis process is shown in Figure 2. First, our approach takes a transitive closure using hard-coded names in the program and constructs the file universe. In other words, our technique works if we can construct the file universe of the application. Note that we need not know the content of the files. At the same time, potential aliases of file pointers are discovered. Second, we have file pointer analysis to accurately generate the file-pointer—filename correspondence. Typically, files are opened and read sequentially by incrementing the file pointers. We perform a check on the file pointer to discover any rewinding/resets/potential decrements. If there is no potential reset/decrement operation, only the remainder of the file starting from current file pointer will be read in the future execution. Thus it will be safe to carry only the remainder of the file from where the current pointer is. Finally, by analyzing the operation on file pointers, we can know when the file is opened, read, written and closed. In the following section, we show how file liveness analysis is performed based on these file operations information.

4.1.1 Reaching Graph

First, a reaching graph is built representing the function precedence relationship based on file operations. A reaching graph is a directed graph that contains two types of nodes: functions and file operation tuples. A node label could represent a function or a file operation tuple. The node label is the function name or a 2-tuple {file_operation, file_name}. The file operation tuple node is specially used to store the exact program point where the file operation is located.

The edges of the graph can be built in three ways. When any two nodes are invoked sequentially in the same function, they are connected with a directed edge indicating the order of execution or precedence relation. The second type of edges are from the last callee of a function to all of its caller's pointed to nodes. These edges show that the next function to be executed should be the next sequential function after the caller in the program. Then the edge from the caller to the other nodes except its last callee can be removed from the reaching graph. If a node N is the first callee of a function F, there is an edge from F pointing to N. Notice that F may have more than one first possible callee since F may call them in a switch statement or an if-else statement. Thus, the edges in the reachability graph can show an invocation or precedence relation.

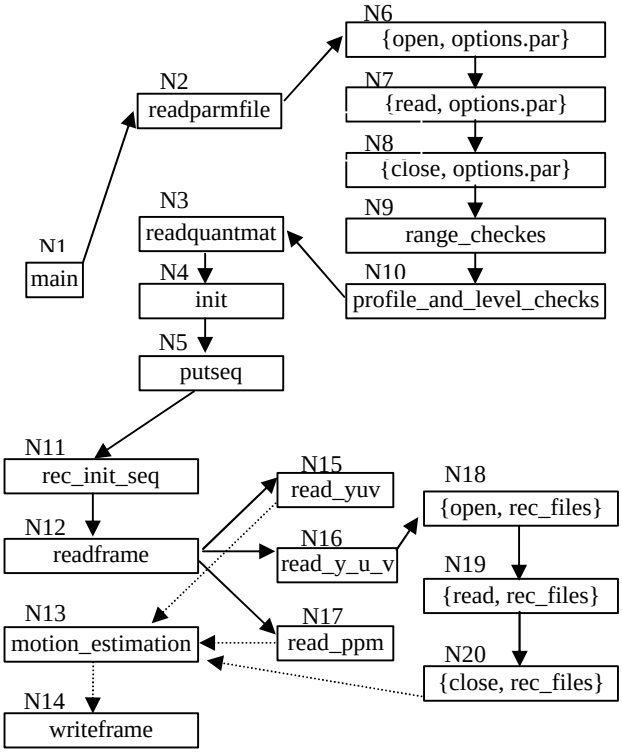

Figure 3. Sub reaching graph for Mpeg2 encoder

The reaching graph for the Mpeg2 encoder is shown in Figure 3. To save space, not all functions are shown. Only the ones that may involve file operations are included. Functions without file operations do not impact our file analysis algorithm. The dashed line indicates that the caller calls the callee indirectly, which means there are some other function calls between the two functions. Notice that there are two types of nodes in the graph: function call nodes bearing the label of the function calls and file operation nodes bearing the tuple (file-operation, file-name) as the label. The directed arrows either show precedence or a caller-callee relationship. The precedence relation illustrates the order of execution. There are edges from N12 to N15, N16 and N17 since they are called in a switch statement in N12. At first, there is an edge from N12 to N13 since N13 is called after N12 in N5. N15, N16, N17 are all last callees of N12, so edges are added from these nodes to N13. Then the edge between N12 and N13 is deleted. An edge from N20 to N13 is inserted and the edge from N16 to N13 is removed.

4.1.2 File Analysis Algorithm

This subsection details our file live range analysis algorithms. First, some definitions are introduced. In this subsection, G denotes the reaching graph of the program.

Definition1: A file is said to be *live* in a node N of G if there exists a reference to the file on a path starting from the entry of N. Otherwise, the file is said to be *dead* in N.

Notice if a file is dead, the program is not used anymore; it can be discarded during migration. This observation results in possible time saving in application migration.

Definition2: A node B is *reachable* from a node A in G if there exists a directed path from A to B in G.

Next, we identify a file's effective live range and the migration points in a program.

1) A file f's effective live range set is denoted by FLR[f]. The algorithm to compute FLR[f] is shown in Figure 4. Initially, FLR[f] contains only those nodes that contain file operations on f. We then consider the nodes in G from which we can reach any node in FLR[f] and include them in FLR[f]. The process is repeated until convergence (defined as either termination or transitive closure). In other words, FLR[f] denotes the effective live range of a file consisting of all the functions from which an execution can lead to a potential operation on f.

```
FLR[f] = {N | N contains file operations on f}
change = true;
while(change)
   change = false;
   For N∈ G
      If ( ∃ M∈ FLR[f]) && (M is reachable from N in G)
         && (N∉ FLR[f])
         FLR[f] = FLR[f] ∪ {N};
         change = true;
      EndIf
   EndFor
EndWhile
```

Figure 4. Algorithm for identifying FLR[f]

File f	FLR[f]
options.par	N1, N2, N6, N7, N8
rec_files	N1- N12, N16, N18 – N20

Table 2. FLR[f] for Mpeg2 encoder

Consider the example in Figure 3 again. In this reaching graph, the nodes containing file operations on f form FLR[f]. Then FLR[f] is expanded according to the reachability information. For example, at first, FLR[options.par] contains nodes N6, N7 and N8. N2 can reach N6 and N1 can reach N2, so they are also

included in FLR[options.par]. Similarly, FLR[rec_files] is obtained. Among the files used by the Mpeg2 encoder, stat_out and output.m2v have long live ranges. Because they are live through almost the entire program and they involve file output due to the write operations, they cannot be discarded during migration. Therefore, we will not discuss them anymore. The other files' live ranges are shown in Table 2.

2) Determine LF[N], the set of live files in a node N. It is easy to see that LF[N] = {f: N∈ FLR[f]}. This set is only used to conduct the LFNW set as following.

3) Next, we identify LFNW[N], a subset of LF[N], as those files in node N such that there is no write operation on them along any path starting from the entry of N. The algorithm for identifying LFNW[N] is described in Figure 5.

```
LFNW[N] = LF[N];
For f∈ LFNW[N]
   For M∈ FLR[f]
      If (M is reachable from N in G)
         && (∃ a write operation on f in M)
         LFNW[N] = LFNW[N] – {f};
         Break;
      EndIf
   EndFor
EndFor
```

Figure 5. Algorithm for identifying LFNW[N]

The significance of LFNW[N] is as follows. LFNW[N] denotes only those files which could be read on some path starting at N. It excludes those which could be written. The files that could be written should migrate, but those in LFNW[N] need not migrate if function N itself becomes unreachable in the current call context. One more clarification is that our analysis focuses only on those files that are safe not to be carried since they will be never needed in execution after migration.

Function name	LF[F]	LFNW[F]
N1, N2, N6, N7, N8	rec_files, options.par,	rec_files, options.par
N3 – N5, N9 – N12,	rec_files, stat.out,	rec_files

Table 3. LF[N] and LFNW[N] for Mpeg2 encoder

The LF[N] and LFNW[N] sets for the Mpeg2 encoder are illustrated in Table 3. Notice that we add stat.out and output.m2v to show the difference of LF[F] and LFNW[F]. Stat.out and output.m2v are live throughout the whole program, so they are included in LF[F]. But there exist write operations on them, so they are not contained in LFNW[F]. From this table, we can observe that there are many opened files in the program and the time duration between when the files are opened and closed is relatively short. This feature affords the opportunity to delay the migration to a better program point since if the interrupt comes at a point too close to the open operation on a file, we can delay migration to the point when the file is closed. Through experiments, we observed that this does not demand too many cycles. For example, suppose readframe receives the interrupt. The migration may be delayed to writeframe instead of readframe saving the time for migrating three of the rec_files.

For convenience, we illustrate all the set definitions and their descriptions in Table 4.

Definition	Description
G	Reaching Graph of the program
FLR[f]	File live range of file f
LF[N]	Set of live files in node N
LFNW[N]	Set of live files in node N and there is no write operation in or after N

Table 4. Definitions and their descriptions

4.2 Code and Global Variable Analysis

In this subsection, we show how to identify dead functions and global variables through data flow analysis.

Definition 3: A function (global variable) is *dead* at a program point if there does not exist any path from the current execution point to the function (global variable). Otherwise it is *live*.

For a function F, denote in[F] to be the set of live functions at the entry of F; out[F] to be the set of live functions at the exit of F, dead_in[F] to be the set of dead functions at the entry of F; Universal_set to be the set of all the functions in the program; gv_in[F] to be the set of live global variables at the entry of F; gv_out[F] to be the set of live global variables at the exit of F; gv_dead_in[F] to be the set of dead global variables at the entry of F; and gv_universal_set to be the set of all the global variables in the program. The algorithm to collect dead function and dead global variable information is shown in Figures 6 and 7 respectively. Notice that before performing global variable analysis, the variable aliasing is performed. Both of the two algorithms are similar to live variable analysis [16] except that there is no kill set and the analysis unit is a function instead of a basic block. Because we use a conservative approach, the kill set is unnecessary. When a function receives the migration interrupt, if a function or a global variable is dead before entering the function, they need not be carried during migration.

```
Initialization:
   in[F] = out[F] = NULL;
   gen[F] = {F'|F' is called in F} ⋃ {F}
Data flow equations:
   in[F] = gen[F] ⋃ out[F];
   out[F] = ⋃ F' is reachable from F in[F']
Finalization:
   dead_in[F] = universal_set – in[F]
```

Figure 6. Code analysis

```
Initialization:
   gv_in[F] = gv_out[F] = NULL;
   gv_gen[F] = {v | v is a global variable referenced  in F}
Data flow equations:
   gv_in[F] = gv_gen[F] ⋃ gv_out[F];
   gv_out[F] = ⋃ F' is reachable from F in[F']
Finalization:
   dead_gv_in[F] = gv_universal_set – gv_in[F]
```

Figure 7. Global variable analysis

A binary code can be viewed as a sequence of functions. A table is built to keep track of the address of each function in the binary

code. According to the liveness of a function, the code can be divided into live parts and dead parts. A live/dead part consists of sequentially connected live/dead functions. Live and dead parts appear alternately in the code. When the code migrates to another mobile machine, only live parts are transmitted. The function calls to the dead functions in the migrated code are removed and the remaining code will be rebuilt on the destination machine.

4.3 Heap Analysis

Currently we only deal with heap objects that are allocated and freed by the programmer using the malloc() and free() system function calls. First, we do heap object pointer aliasing analysis. A heap object is said to be live if a malloc() call is invoked and no free() call is invoked on any pointer pointing to this heap object. Otherwise, it is said to be dead. Through our experiments detailed in the next section, we can observe that large heap object may be freed when the application proceeds executing enabling us to save time in migration if the program reaches a later migration point.

4.4 Migration Points Determination Model

In this phase, we describe how to set up a model to determine the most likely migration point based on the analysis of the program and the time estimation. The goal of the algorithm is to delay migration so that the time consumption on the files, stack, global variable and code is less than the additional time to get there.

First, we need to estimate *delay_cost*, the cost of additional time for executing the program to a later migration point. Let MP[F] be the functions which are reachable from F, where F is the function receiving the migration interrupt. And MP[F] is the set of all possible migration points starting from F. The delay_cost for each function in MP[F] is evaluated as follows.

Suppose we are currently in the execution of a function F and want to know the most likely path from F to M∈ MP[F]. Each execution path from F to M is labeled with prefix, suffix, frequency and execution time. The prefix P is the function sequence in the application's stack when it reaches F. The suffix S is the function sequence when the program reaches M excluding the prefix. The frequency is the execution times of the path with P and S as its prefix and suffix. The execution time is the time consumption on the path from F to M matching prefix P and suffix S. The most likely path from F to M is defined as the path with the highest frequency that matches the function sequence (prefix) in the application's current stack. Next the estimated time from F to M (delay_cost) is determined by the execution time on such a path. If there is no such a path from F to M in the profiling data, delay_cost is set to infinity since that may represent a dynamically infeasible path.

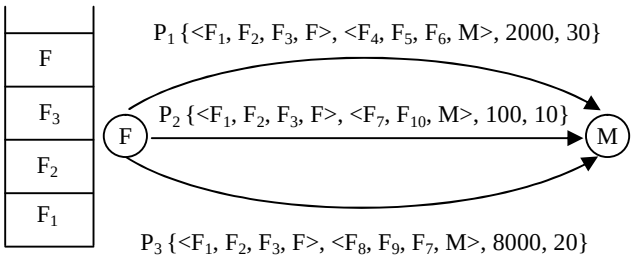

Figure 8. Current stack **Figure 9.** Matched paths from F to M

The example in Figures 8 and 9 show how our prefix and suffix matching scheme works. First, the paths (P_1, P_2, P_3) whose prefix matches with the application's current stack <F_1, F_2, F_3, F> are selected. Among these paths, P_3 is executed most frequently (8000 times), so it is chosen as the most likely path. The delay_cost from F to M is 20 (msec).

Next, the time saving at a later migration point is denoted by *delay_benefit*. Delay_benefit is computed according to the transmission delay plus the serialization overhead. The pure transmission speed is not enough to estimate migration time since it only tackles the cost of transferring data from the sender's buffer to the receiver's buffer. The cost of moving this data from the receiver's SDRAM or Flash ROM to its buffer still needs to be taken into account, which is called serialization overhead. Bryan Carpenter et. al. [18] measured the serialization cost for marshalling data in a Java interface to MPI. They showed that the overhead ranged from 0.027 m s/byte to 100 m s/byte, depending on the data type.

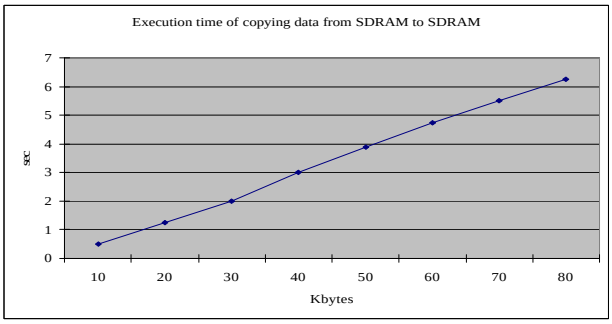

Figure 10. Serialization: Copying data from SDRAM to SDRAM

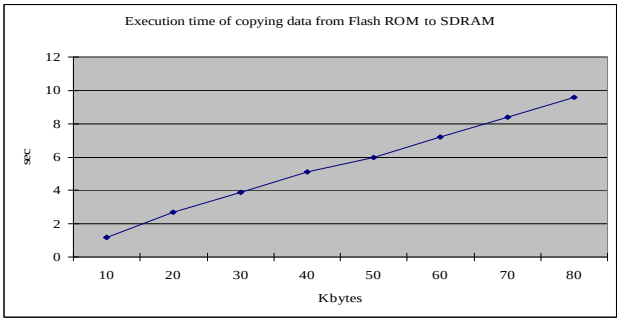

Figure 11. Serialization: Copying data from Flash ROM to SDRAM

In this work, we build a serialization overhead model by the simulation on an iPAQ, which is based on the ARM architecture. The Flash ROM in an iPAQ consists of a fixed number of blocks, each with 32 pages of 512 or 2048 bytes of main data [25]. The experimental results for serializing SDRAM or Flash ROM resident data are shown in Figures 10 and 11. It shows that the data serialization time increases with the size of moved data linearly. The slope of the line represents the bulk transfer capacity. They are 0.124 and 0.071 respectively. That means it costs 0.124 (0.071) seconds to move 1kbytes data from the Flash ROM (SDRAM) to the buffer. In an application's state, the global variables, stack and code usually stay in SDRAM while the opened files lie in the Flash ROM. So delay_benefit should be

(unit_trans_time + 0.071) × (stack_size[F] - stack_size[M] + global_variable_size[F] – global_variable_size[M] + code_size[F] – code_size[M] + heap_size[F] – heap_size[M]) + (unit_trans_time + 0.124) × (file_size[F] – file_size[M]), where the interrupt comes into F and M is a function in MP[F] used for deciding actual migration point. Unit_trans_time in the above equation represents unit time needed to transmit the serialized data over the wireless link. The migration interrupt handler performs the trade off analysis for all the functions in MP[F] to determine the best potential later migration point.

From the above discussion one can see that in case a migration interrupt is received in function F, it may be possible to continue execution until a function M that belongs to MP[F] and migrate once this function is called. Thus, one could generate a migration call in M.

But it is not sufficient to generate such a call and simply continue execution from F. There is no guarantee that execution will reach M. Thus, M only represents a *potential* migration point if execution proceeds along the respective call chains. But we must monitor and change the decision to migrate in case execution proceeds along a different chain in the dynamic sense as discussed above. Recall that we computed M as the *most likely* migration point that could be reached in the given calling context for migration cost tradeoff analysis. We used profiling information to arrive at the above. If the program does not follow the predicted (most likely) path, we must provision for it.

To solve this problem, we use the suffix to do the re-estimation of migration point and the tradeoff. Suppose the application receives the interrupt at function F and it was predicted by the interrupt handler that it will go along a certain path to reach a migration point M. Assume that somehow the program's execution does not match with the predicted path and it reaches a function F' not on the predicted path, then we have to re-analyze the migration decision. We re-evaluate it as follows. We first predict the execution path from F' and the corresponding migration point. Let M' be the most beneficial migration point in MP[F']. If the sum of the actual execution time from F to F' plus the estimated execution time from F' to M' is greater than the delay_benefit at M', we decide to migrate immediately in F'. Otherwise, we decide to continue execution until M'. We continue this process in case, again, path prediction to M' is in error and so on until the application migrates.

5. Experimental Results

This section shows the experimental results indicating the state and latency saving offered by our migration framework. Our experiments are based on the ARM architecture, which is very popular for embedded systems such as PDAs and mobile handhelds such as iPAQs. The wireless networks are 802.11 wavelan and the transmission speed in such a network is assumed to be 1Mbps during a migration. In fact, the transmission speed is usually much smaller than 1Mbps due to the network traffic volume and some link failures. The optimizations are implemented in gcc cross-compiler and the handlers are inserted in the run time system. We simulated our programs using SimpleScalar [15].

We ran a series of experiments to determine if our application migration algorithm could save significant amounts of time during migration. To evaluate it here, we focus on multimedia applications. We have eight multimedia applications. They are compiled using the gcc cross-compiler with instrumentation. These benchmarks are trained using training set, which includes five to ten different inputs to collect profiling data. Then another input (called ref input), which is not in the training set, is used for the evaluation. The ref inputs of these applications are shown in Table 5. For the benchmark mesa, the demon (osdemon), which is included in MediaBench, is used to get the result.

Applications	Input set
Adpcm(en)	clinton.pcm (in MediaBench)
Epic	lana.tif (in MediaBench)
G721(en)	clinton.pcm (in MediaBench)
Gsm	clinton.pcm (in MediaBench)
Jpeg(en)	input_large.ppm (in MiBench)
Mesa	None
Mpeg(en)	Bitstream of YUV components (in MediaBench)
Pegwit	public_key, encryption_noise_file, plain_text (in MeadiaBench)

Table 5. Ref input set of multimedia applications

We performed three sets of experiments. In the first experiment, all of the application state migrates completely without any analysis, which is the base line for the latter two experiments. The second one conveys how our application state analysis algorithm works for these applications. In other words, in this setting the application migrates immediately at the point of interrupt (now) except that dead code, global variables and read files that are no longer needed are dropped. The third one is focused on determining the best migration point. Among these three experiments, consistency in interrupts was maintained (i.e., the interrupts happened at the same dynamic program point) while comparing the optimized and unoptimized versions.

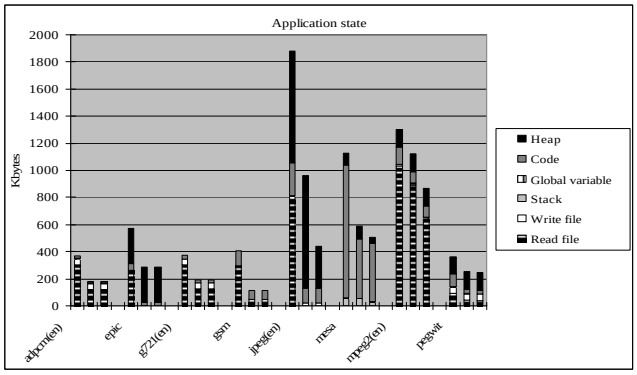

Figure 12. Application state

The application state of the eight multimedia applications is shown in Figure 12. In this graph, we have three columns for each benchmark. They represent the results of migration now without any analysis (first column), migration now with analysis (second column) and migration later with analysis (third column). Each column is split into six parts, showing the sizes of the heap, code, global variables, stack, write files (the file which is at least written once by the application) and read files (the file which is only read by the application).

The first experiment (first column for each application) proves that the application state could be quite big (up to 1882 Kbytes). This may result in long latency in low bandwidth networks and

may cause high overhead in migration. Among the state, stack and global variables constitute the smallest parts and do not contribute much during migration. The dominant factor in an application's state depends on its own feature. For example, the read file in Adpcm(en), G721(en) and Mpeg(en) entails most of the cost during migration, which enables us to save time if part of the file becomes dead. For Jpeg(en), the read file, heap and code are the main parts of its current state and dropping these helps here. There is a chance of saving time by reducing the read file size or heap size in Pegwit and Epic. This justifies that devising a framework that takes a unified view of the state rather than just focusing on data or code is superior.

In the second experiment (second column for each application), the application state is decreased by a large value due to our algorithm. The benefit mostly comes from read files, code and heap. For example, in Epic and Jpeg(en), almost all the read files become dead when the interrupt comes in and there is no potential file pointer reset/decrement on all future possible paths, so the remainder of the file can be removed safely. About half of the mesa code is dead in migration, which can also be discarded safely. On an average the application state is decreased by 45.39% ranging from 13.95% to 71.85% using our migration scheme over the naïve method of carrying all of the state. This scheme shows that it is highly beneficial to migrate without dead state.

In the third experiment, we show how delaying the migration to a later program point saves additional state and time in some cases. Comparing with the second experiment, we can see that later migration algorithm works better in terms of time saving. For example, Mpeg2(en) has many frames as input files. As the program continues running, more and more frames can be thrown away since they are not needed any more. Since reading a file from Flash ROM or SDRAM is less expensive than transferring it through serialization and communication, we are afforded the opportunity to delay migration to a more beneficial later program point where more read files become dead. For Jpeg(en) and Epic, when the program receives the migration signal, most of the heap space allocated is going to be freed. If the application migrates later, less state needs to be carried. There is, of course, cost involved for continuing execution. However, the cost is quite small compared with the benefit we get in migration time saving. Our algorithm works well for such programs that read files into a buffer and then the data is processed. This scheme is not suitable for G721(en) since reduction in state progresses more slowly than execution -- the input file is read for a data item at a time and the major cycles are spent on its encryption, which consumes time. On average, 5.07% additional migration time is saved at later migration point, ranging from 0% to 23.62%.

Figure 13 shows how much time migration would take. We can see that the time latency is long, ranging from 33 seconds to 176 seconds. Among the delay, serialization takes up almost all of the migration time. It is due to our relative high transmission speed (1Mbps) assumption. This assumption is conservative. If the transmission speed was lower, savings would be even more due to larger difference in cost of transmission. The time reduction in Epic and Jpeg(en) is more obvious than the application state decrease since most of the application state reduction lies in the file size decrease, which cost longer time than other data (such as heap, code, global variable). It illustrates our migration algorithm achieves excellent time reductions. On average, the time saving for the second experiment (migrate without dead state) is 48.94%, ranging from 13.62% to 75.81%. For the third experiment (migrate later and removing more dead state), it is 55.34%, ranging from 33.04% to 81.34%.

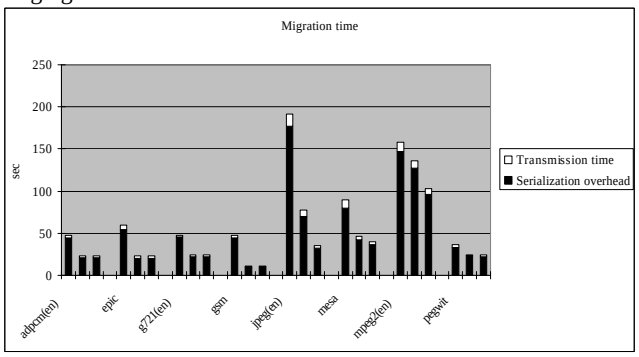

Figure 13. Migration time

We also performed a sensitivity study to determine the effect of migration interrupts coming at different points of program execution. We generated four interrupts more or less uniformly spaced and checked the effects on the state. We observed that many files get read right in the beginning of execution (and thus become dead in first few thousand or so cycles) and thus all interrupts coming after those have the same effects. Also, most files are not written until the end and due to symmetric reason other interrupts coming in between do not impact either. We found that for just one application that the data was being read and written continually in a loop throughout. Since a part file was becoming dead (which was discovered by incrementing file pointer), the deadness increased with later interrupts, but at the same time more was added in a live (write) file.

The results of the sensitivity study are illustrated in Figures 14 and 15. They show active application state and migration time at respective interrupt points comparing the base case with our optimized schemes. On average, the application state decreases by 47.61% for the second experiment and 51.26% for the third experiment. There is 51.11% time saving for the second experiment and 54.29% for the third experiment.

We also evaluate the correctness of our path prediction algorithm. We experimented with eight applications and four interrupts were generated for each application. So there were 32 interrupts in total. The path prediction failed only for one interrupt and after one re-evaluation, it became correct. So the correctness of our path prediction algorithm is 32/33 = 97%.

Benchmark	Overhead (s)
Adpcm(en)	0.131
Epic	0.663
G721(en)	0.376
Gsm	0.894
Jpeg(en)	1.423
Mesa	1.046
Mpeg(en)	1.391
Pegwit	1.034

Table 6. Overhead

The overhead of our algorithm is shown in Table 6. The overhead includes the time to decide which part to migrate, to calculate the delay benefit and delay cost, and to determine the potential migration points. On average, the overhead is 0.870 seconds. We can observe that the overhead is much smaller compared with the overall time spent and saved in migration.

6. Related Work

There has been a great amount of research work on process migration in distributed systems [1; 20]. Emerald [17] provides an object level migration mechanism to support fine-grained object mobility. Sprite's [8; 9] and COOL [3; 4] supports a high degree of transparency.

Scalability is one big concern of process and object migration techniques [13]. Application migration operates at a higher level. It can be applied to the systems in wide area networks. Application migration is either transparent or under user's control [5, 6].

Krishna A. Bharat and Luca Cardelli [5] implemented homogeneous application migration at the programming language level. The biggest strengths of their implementation are that the details of migration are completely hidden from the application programmer, and; arbitrary user interface applications can be migrated by a single "migration" command. J. Hall et. al. [26], proposed an efficient migration algorithm. Their approach was driven by the change in network configuration. Our strategy is driven by the application's state. To our knowledge this is the first work that provides a compiler guidance framework for deciding what to migrate and when to migrate to achieve tremendous savings in the amount of state migrated and the time it takes to migrate.

Compiler assisted program analysis has also been around for a long time. For example, the compiler generated potential checkpoint code to maintain the desired checkpoint interval [22; 23]. Plank [24] et. al., proposed a compiler assisted automatic memory exclusion in checkpointing systems. Our strategy is different from theirs since our analysis is based on the application state instead of the pure structure of one program. Also, we do not assume periodic state savings as is the case in incremental check-pointing schemes. In addition, we perform analysis to determine whether to migrate right away or continue for dropping additional state.

7. Conclusion

In this work, we tackle the problem of doing efficient application migration. We propose a compiler assisted migration point determination framework based on profile information to save data transmission time in application migration. The basic idea is to determine a best migration point to save time in application migration. At run time, we guide the migration decision through a combination of tradeoff analysis that gets triggered. The framework not only determines the most profitable migration points, but also determines what should be dropped during migration to minimize serialization and communication overheads.

We have empirically shown such a compiler driven approach does save on the state to be migrated and does reduce the cost of migration. The time saving due to our framework is significant over a scheme that migrates all of the state naively (45.39% saving in time ranging from 13.95% to 71.85%). For some applications, it is even beneficial to continue execution (being lazy) to drop more state, (ranging from additional saving of 0 % to 23.62% with an average of 5.07%). Overall, our approach improves the efficiency of migration in a significant way by dropping dead state, and in some cases, by continuing to execute expending a few more cycles to thin it further. Due to the tremendous savings achieved by our scheme, we believe that it will eliminate the biggest barrier to migration: high overheads and will facilitate application mobility in future systems.

8. References

[1] Alfonso Fuggetta, Gian Pietro Picco and Giovanni Vigna. Understanding Code Mobility. *In IEEE Transactions on Software Engineering*. 1998.

[2] Robin Kravets, Karsten Schwan and Ken Calvert. Power-aware Communication for Mobile Computers. *In Mobile Multimedia Communications*. 1999.

[3] Rodger Lea, Christian Jacquemot and Chorus systemes. COOL: System Support for Distributed Object-oriented Programming. *In IEEE Transactions on Software Engineering*. 1993.

[4] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont. Overview of the CHORUS Distributed Operating Systems. *In Workshop on Micro-Kernels and Other Kernel Architectures*. 1992.

[5] K. Bharat and L. Cardelli. Migratory Applications. *In Proceedings of the Eighth ACM Symposium on User Interface Software and Technology*. 1995.

[6] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A Resource Query Interface for Network-aware Applications. *In 7th IEEE Smposium on High-Performance Distributed Computing, IEEE*. 1998.

[7] Artsy, Y. and Finkel, R. 1989. Designing a Process Migration Facility: The Charlotte Experience. *In IEEE Computer*. 1989.

[8] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *In Software: Practice and Experience*. 1991.

[9] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson and Brent B. Welch. The Sprite Network Operating System. *In Computer Magazine of the Computer Group News of the IEEE Computer Group Society*. 1988.

[10] Joseph TARDO and Luis VALENTA. Mobile Agent Security and Telescript. *In* Proceed*ings of the 41st IEEE International Computer Conference*. 1996.

[11] G.H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *In IEEE Computer*. 1994.

[12] T. Imielinsky and B.R. Badrinath. Wireless Computing: Challenges in Data Management. *In Communications of the ACM*. 1994.

[13] Deborah Estrin, Ramesh Govindan, John Heidemann and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. *In Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. 1999.

[14] http://grouper.ieee.org/groups/802/11/index.htmly

[15] The SimplesSalar - arm power modeling project. http://www.eecs.umich.edu/~panalyzer

[16] Alfred V. Aho, Ravi Sethi and Jefferey D. Ullman. Compilers, Principles, Techniques, and Tools. *Addison Wesley*. 1986.

[17] E. Jul, H. Levy, N. Hutchinson and A. Black. Fine-Grained Mobility in the Emerald System. *In ACM Transactions on Computer Systems*. 1988.

[18] B. Carpenter, G. Fox, S.H. Ko and S. Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. *In Proceedings of the ACM 1999 conference on Java Grande*. 1999.

[19] Robert Gray, David Kotz, Saurab Nog, Daniela Rus and George Cybenko. Mobile agents for mobile computing. *Technical report PCS-TR96-285*.

[20] ARamon Lawrence. A survey of process migration mechanisms. *Technical report, University of Iowa*. 1998.

[21] J.M. Smith. A survey of process migration mechanisms. *Technical report, Columbia University*. 1995.

[22] Chung-Chi Jim Li, Elliot M. Stewart and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software -- Practice and Experience, 24(10):871—886*. October 1994.

[23] M. Beck, J. S. Plank, and G. Kingsley. Compiler-assisted checkpointing. *Technical Report CS-94-269, University of Tennessee at Knoxville*. December 1994.

[24] James S. Plank, Micah Beck and Gerry Kingsley. Compiler-Assisted Memory Exclusion for Fast Checkpointing. *In IEEE Technical Committee on Operating Systems and Applications Environments - Special Issue on Fault-Tolerance*. 1995.

[25] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee and Sang Lyul Min. Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory. *In Proceedings of the fourth ACM international conference on Embedded software*. 2004.

[26] Joseph Hall, Jason Hartline, Anna R. Karlin, Jared Saia and John Wilkes. On algorithms for efficient data migration. *In Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. 2001.

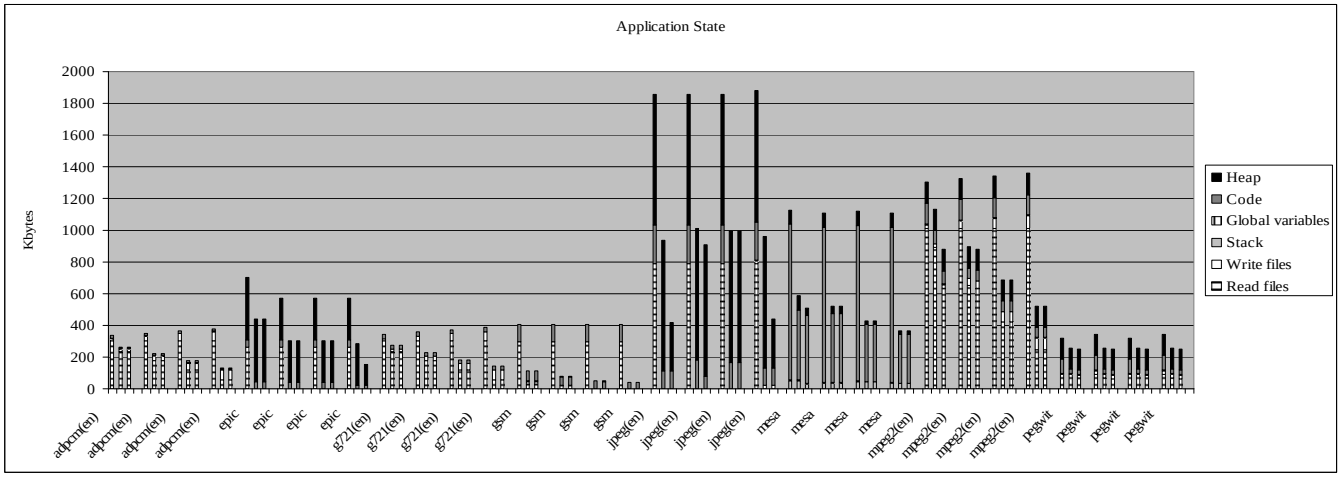

Figure 14. Application state

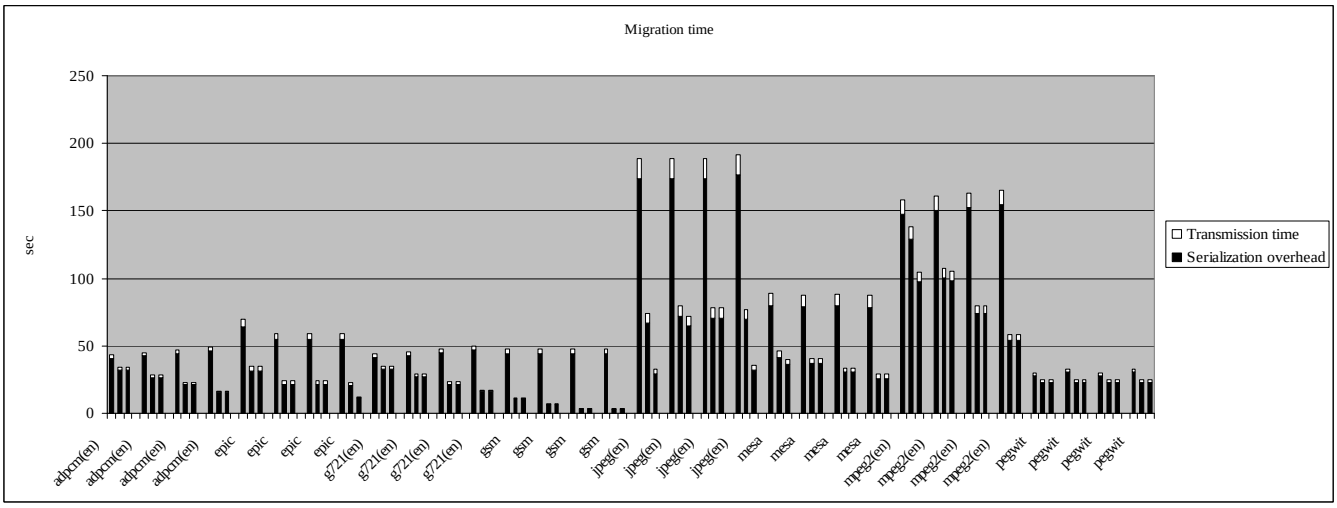

Figure 15. Migration time