Computing and Informatics, Vol. 30, 2011, 1131–1146

# A SHADOW-LIKE TASK MIGRATION MODEL BASED ON CONTEXT SEMANTICS FOR MOBILE AND PERVASIVE ENVIRONMENTS

Feilong Tang

*School of Software*
*Shanghai Jiao Tong University, Shanghai 200240, China*
*e-mail:* tang-fl@cs.sjtu.edu.cn

Can Tang

*Department of Finance*
*Heilongjiang University, Harbin 150080, China*

Minyi Guo

*Department of Computer Science and Engineering*
*Shanghai Jiao Tong University, Shanghai 200240, China*

Shui Yu

*School of Information Technology*
*Deakin University, Burwood, VIC 3125, Australia*

Song Guo

*School of Computer Science and Engineering*
*The University of Aizu, Fukushima 965-8580, Japan*

**Abstract.** Pervasive computing is a user-centric mobile computing paradigm, in which tasks should be migrated over different platforms in a shadow-like way when users move around. In this paper, we propose a context-sensitive task migration model that recovers program states and rebinds resources for task migrations based on context semantics through inserting *resource description* and *state description sections* in source programs. Based on our model, we design and develop a task migration framework *xMozart* which extends the Mozart platform in terms of context awareness. Our approach can recover task states and rebind resources in the context-aware way, as well as support multi-modality I/O interactions. The extensive experiments demonstrate that our approach can migrate tasks by resuming them from the last broken points like shadows moving along with the users.

**Keywords:** Task migration, pervasive computing, context-aware computing

## 1 INTRODUCTION

Pervasive computing is a new distributed computing paradigm that provides mobile users with preferred services at anytime and anywhere. Owing to the high mobility of users, pervasive softwares have to be migrated among extremely heterogeneous platforms. As a result, context-sensitive task migration is an important enabling technology to achieve the attractive human-centric goal of pervasive computing [1, 2]. Task migration in pervasive environments should be aware of and self-adaptive to software and hardware platforms, mobility, network connection and resource constraint of pervasive devices. However, none of existing proposals on task migration have sufficiently addressed the new requirements for mobile and pervasive applications.

In this paper, we investigate the problem of context-sensitive task migration, especially focusing on the approach adjusting execution and display behaviors based on run-time context automatically to enable task migration in a shadow-like way. We demonstrate the key issues in task migration considering the following scenario. A student Steve watches an online movie in his lab on a Windows-based PC that is connected with Internet through a wired network. He, then, takes a bus to attend a meeting held in another campus a few minutes later. On the bus, he resumes playing the movie from the last broken point on his Mac-based iPhone through a wireless network. We outline the characteristics of such task migrations as follows:

**Heterogenous software and hardware platforms.** Not only video players and operating systems but also devices, before and after the migration, are highly heterogeneous.

**Delay-sensitive migration process.** Long delay is always undesirable to users.

**Lightweight task migration.** Data migrated actually should be as little as possible because of the limited bandwidth of wireless networks and the requirement of the prompt migration.

**Multi-modality representation.** A lower-resolution movie is preferred, after this migration, to adapt to the small screen of the iPhone as well as the lower bandwidth of wireless networks.

This paper is motivated to solve the above issues by providing a context-sensitive task migration service for users moving in different contexts. In the paper, we propose a shadow-like task migration model based on context semantics by capturing various migration-related contexts. Compared with existing works, our model adds sections of *state description* and *resource description* which are used for state recovery, resource rebinding and multi-modality interaction. We then develop a lightweight task migration framework xMozart based on our task migration model, which can migrate tasks among different platforms in a shadow-like way. Furthermore, the migrated tasks can resume execution from their broken points, and interact with users through multiple modality interfaces.

The rest of this paper is organized as follows. In Section 2, we review related work. Section 3 proposes a semantics model and a context-sensitive task migration model. In Section 4 we present a shadow-like task migration framework xMozart built on our migration model. The implementation and performance evaluation are presented to demonstrate the effectiveness of our model and framework in Section 5. Finally, Section 6 concludes this paper with a discussion on our future work.

## 2 RELATED WORK

Existing research on task migrations falls into three categories, *desktop-level migration, application-level migration* and *process-level migration* [3–5, 9, 10].

The *desktop-level migration*, like Windows Remote Desktop and XWindow, transfers the remote desktop visualization to a target platform. After the migration, users still use the original computer systems in a remote way that requires a stable connection with a sufficient bandwidth. Furthermore, this method lacks the control on the migration granularity.

The *application level migration* scheme includes two methods. The first one packets source files and then transfers them to a target node. The limitation is that it is not practicable to migrate tasks among heterogeneous systems. The second method is based on the C/S pattern, where the source node is a server and the target platform is a client. The application-level UI rather than the whole desktop is migrated to the target node. In this paper, we use the C/S pattern to build a context-sensitive migration model and a migration framework because

1. it can migrate tasks among heterogeneous platforms through persisting programs in the source node and then recovering them in the destination, and
2. it is easy to incorporate the context with the migration process, enabling the shadow-like task migration.

The *process-level migration* focuses on the kernel level. It only migrates some active execution images from a source node to a target. Although this migration

mechanism has the highest flexibility, it is very complex due to the kernel-level state maintenance, especially when tasks are migrated among different operating systems [6].

Mozart programming system supports multiple platforms: UNIX and Windows. It consists of two main components: Oz language and programming interfaces [7]. Oz is a multi-paradigm programming language that supports declarative programming, object-oriented programming, as well as constraint programming. Mozart supports GUI programming through QTK toolkit [8], an extension of TK module. QTK uses Oz records to introduce a partly declarative programming approach, where UIs are calculated on the fly and loaded at runtime. This is a desirable property for our migration scenarios.

## 3 CONTEXT SEMANTICS BASED TASK MIGRATION MODEL

### 3.1 Task Migration Model Based on Sections

We model an Oz based program as a set of sections. As shown in Figure 1, an original Oz program (i.e., *file.oz*) is re-organized as three sections (i.e., *file."oz"*): *main Oz section, state description section* and *resource description section* before a migration. The latter two sections introduced in our model are responsible for recovering the runtime context after the migration.

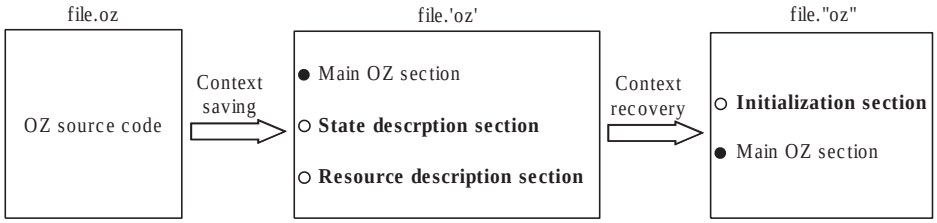



Fig. 1. Section based task migration model

More specifically, the *main Oz section* is the original Oz program. *State description section* is used to keep and recover various state information during the task migration. *Resource description section* prescribes the migrated Oz program on how to access data resources, e.g., video files, in the target node.

In our model, data resources are not actually migrated. Instead, they are stored in the source node or a third-party server. After a migration, our xMozart deserializes the state and resource sections and maps them to a set of Oz records, which form the *initialization section* of the *file."oz"*. Migrated tasks firstly execute the initialization section to recover various states and then rebind data resources. As a result, Oz programs can be resumed from their individual broken points.

### 3.2 Semantics Model for Context-Sensitive Task Migration

#### 3.2.1 Semantics Extension for Oz Language Compiler

As aforementioned, our migration model introduces new sections: resource description section, state description section and initialization section. To enable Oz compiler to understand these new elements, we extend the Oz compiler, focusing on

1. section based programming,
2. new reserved words for state recovery and resource rebinding, and
3. multi-modality interfaces.

This section introduces semantics support for state variables, resource variables, and multi-modality interfaces. In the next section, we will present how to enable Oz compiler to support section based programming.

#### 3.2.2 Semantic Support for State Variables

State variables are used to describe the task execution context and to recover the migrated task state to the same one before the task is migrated. We introduce a new reserved word *StateAttribute* to mark state variables.

State variables fall into two categories: *local* and *global state variables*. The *global variables* can be easily extracted because any code can access them. We employ a compiler-generated procedure to collect all global states and then serialize them in the state description section. By comparison, however, it is very complicated to get local states because any procedure can not guarantee to extract all *local state variables*. So, we propose a shadow function technique for keeping these local state variables, which will be presented in Section 3.3. Based on a state declaration, for example "*Declare WatchPoint* {*[StateAttribute], x*}", the extended compiler will handle the state of WatchPoint according to its category (global or local). If WatchPoint is a local variable, the shadow function for the WatchPoint will resume the video play from the broken point.

#### 3.2.3 Semantic Support for Resource Variables

In our model, we do not actually migrate data resources (e.g., a video file) to avoid undesirable transmission latency. Instead, we use resource variables to indicate how to access these data files after tasks are migrated.

To indicate resource variables, we introduce a new reserved word *ResourceAttribute*. We extend the Oz compiler to accept this reserved word when parsing Oz source files. Similarly, a resource variable declaration looks like "*Declare ImageHandler [ResourceAttribute], ...*". With the resource declaration, the migration module of our xMozart will be notified of initiating the serialization when a migration is triggered.

**Algorithm 1: Resource Rebinding**

```
if (originalHandler is an URI)
   rebindingHandler=originalHandler;
else { if (originalHandler is a relative LRI)
           originalHandler=pathPrefix+originalHandler;
       rebindingHandler="//"+hostName+originalHandler; }
```

Fig. 2. Resource rebinding

We categorize all resources as two categories: *local* and *global resources*. The local resources are stored in the source node while the global resources refer to the ones provided by third-party servers. The global resources can be globally accessed using the *URI* (Uniform Resource Identifier), e.g., "*//HostServer/Resources/truck.gif*", where HostServer is the server accessible from the Internet. Correspondingly, the local resources are accessed through the *LRI* (Local Resource Identifier). LRI resources may be represented in an absolute path, like "*C://Resources/truck.gif*", or a relative path, for example "*Resources/car.gif*".

We propose an algorithm to rebind resources, as illustrated in Figure 2, where *originalHandler* and *rebindingHandler* refer to resource handlers before and after a migration, respectively. Our xMozart intercepts the incoming resource resolution request in the target node, maps it using *algorithm1* and returns the new resource handler. Essentially, our xMozart acts as a proxy between a source node and a target node for the resource rebinding.

### 3.2.4 Semantic Support for Multi-Modality Interfaces

Due to the heterogeneity of pervasive devices, I/O interfaces used in a source node may be inappropriate or even unavailable in a target node. Multi-modality interfaces are designed to interact with users through appropriate I/O means for better user experiences, which are very important in pervasive environments. For this purpose, we introduce a new reserved word *Interchangeable*, which is declared like "*Declare InputModule [Interchangeable]*", to indicate multi-modality interfaces.

We extend the Oz compiler to understand this reserved word when parsing Oz source files. Our xMozart provides multi-modality support for extended Oz programs through separating application logic with I/O modules. These I/O modules are managed by a standard adapter through a general interface.

## 3.3 Shadow Functions and Local State Recovery

As aforementioned, any single procedure can not access all local state variables due to their locality. We use a shadow-function-based mechanism to collect the value of local state variables. In our model, a shadow function refers to the procedure that the compiler generates for recovering the specified local states. The basic idea

behind the shadow function is to save variable-value pairs in a predefined symbol table during the program execution and then to recover the variables through reading the symbol table after a migration. We show how to generate a shadow function using the following example.

for(int $i = 0$; $i < 10$; $i$++) {
    ...; // normal codes
    shadow function; // to persist the variable $i$. }

In the example, we use the shadow function to keep the last value of variable $i$. As a result, every loop will write the state variable $i$ in the state section of the extended Oz program. After a migration, the program will be resumed at the very beginning of the interrupted iteration.

### 3.4 Oz Setter Generation

Our model saves state and resource information in the initialization section. After a task migration, the task will firstly execute the initialization section to recover various states and then rebind resources as discussed in Section 3.2.3. This section presents how to generate Oz setters for the state recovery.

We map state variables in the variable-value pairs, i.e., Oz setters. As shown in Figure 3, *algorithm2* generates Oz setters for global state variables. It firstly deserializes variable-value (simplified as *V-v*) pairs and set up a hash table HT, and then inserts these V-v pairs in the initialization section. Consequently, each variable and its value in the state description section is mapped as an "Oz setter" record, e.g., TruckPositionX = 50. When a migrated program is resumed, all the generated Oz setters will be executed sequentially to recover the corresponding states.

**Algorithm 2: Oz Setter Generation**

```
deserialize V-v pairs;
set a HT;
while (HT≠∅)
   generate Oz setter V=v;
   insert V=v in initialization section;
   HT=HT-{V=v};
endWhile
```

Fig. 3. Oz setter generation

By comparison, local state recovery is more challenging. We employ the code injection technique to achieve the local variable recovery, illustrated as follows.

int $i = 5$; //code injection here to recover variable $i$
for( ; $i < 10$; $i$++){ // the original $i$ assignment will be suppressed
    ...; //normal execution }

## 4 TASK MIGRATION FRAMEWORK

Our task migration model extracts resource handlers and serializes computing contexts in a source node, and then rebinds resources and recovers the contexts in a target node. In this section, we present a task migration framework xMozart based on our model, focusing on two aspects: migrate management and multi-modality adaptation.

### 4.1 Migration Management

Migrate management modules initiate and control the task migration process, shown in Figure 4. On receiving a task migration request, the *Migration Trigger Handler* module halts the current program execution. The *State Serialization* module, then, submits the executing program to compiler-generated functions. The latter collects the values of state variables and resource variables, and then stores them in the state description section and resource description section, respectively. From then on, xMozart will prepare the two endpoints for the task migration.

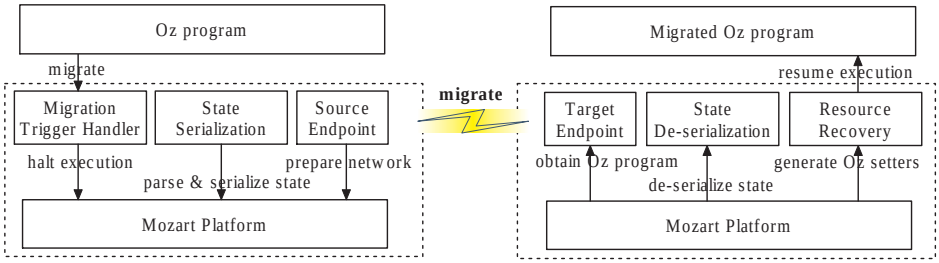



Fig. 4. Task migration framework xMozart

Migration process transfers the file."oz" with three sections from a source node to a target one. *De-serialization* module in the target node recovers runtime states by extracting the resource handlers and variable states from the resource description section and the state description section, respectively. Using resource handlers, the *Resource Recovery* module can address resources through our algorithm1 (Figure 2). At this moment, an Initialization section in file."oz" is formed and the migrated program is ready to resume its execution. Figure 5 shows the whole task migration process.

### 4.2 Multi-Modality Adoption

Multi-modality adoption provides appropriate I/O interaction means based on the contexts (see Figure 6). We separate I/O interfaces with program logic to implement the multi-modality adoption using two key modules: Device Discovery and Evaluation (DDE) module and I/O Adapter Management (IOAM) module.

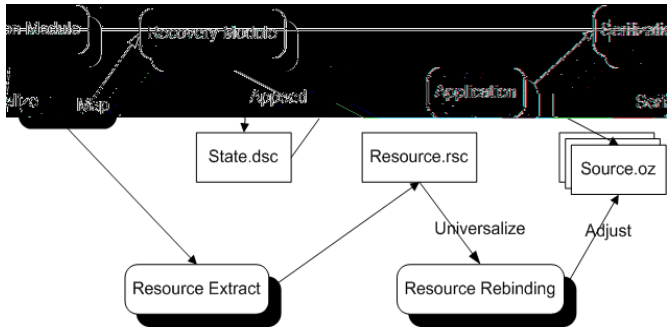



Fig. 5. Task migration process

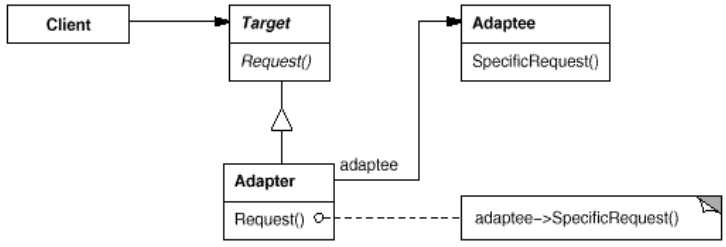



Fig. 6. Synchronization among multiple modality interfaces

The IOAM module manages all the supported I/O adaptors. These I/O adaptors can be added in a plug-in way. For instance, our framework provides the standard input module such as a keyboard, and at the same time it supports voice input as well.

The DDE detects which kind of interaction is the best and whether the corresponding devices are available. Based on the detection results, the DDE notifies users which I/O interface will be used. Usually, the DDE enumerates all potential I/O capabilities based on the context and comes up with its recommendation.

## 5 IMPLEMENTATION AND EVALUATIONS

We have developed two prototypes to evaluate our task migration model and framework. Computers are configured with Intel T5870 2.00 GHz CPU with 2 GB memory and Intel E8600 3.00 GHz CPU with 2 GB memory, respectively. They are connected through our campus network and are installed with Windows 7 and Ubuntu Linux, respectively. We adjusted the router rate at 128 kbps, 256 kbps and 512 kbps to

simulate different network bandwidths during the experiments. We conduct testing and evaluation of our task migration approach in the following aspects.

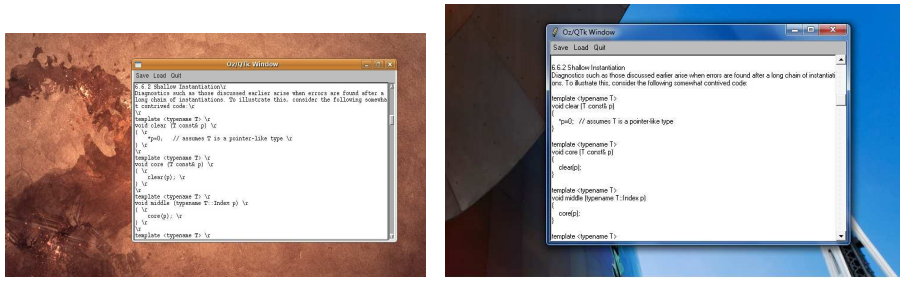
### 5.1 Application Migration

Distinguishing from other related work, our approach supports application state recovery and resource rebinding in a shadow-like way. To evaluate the effectiveness of our xMozart, we have tested the state recovery and resource rebinding of our scheme through migrating a text compiler across both homogeneous and heterogeneous platforms.

| *Items* | *Type* | *Denotation* |
|---|---|---|
| C++Primer.txt | resource variable | [ResourceAttribute] |
| Page Number | state variable | [StateAttribute] |
| Line Number | state variable | [StateAttribute] |

Table 1. Key variables in the text compiler migration

*Task migration between two heterogeneous platforms.* In this experiment, networked nodes run on a Windows and a Linux operating system, respectively. Firstly, a text compiler that runs in the Linux-based workstation opens the file named as C++Primer.txt and locates it at a specified page and line. We, then, trigger a task migration process. In this experiment, key variables used in this task migration are set as in Table 1, where C++Primer.txt is a resource variable and is restored in our Linux platform, while the other two are state variables, which are the page number and line number that are being read.



a) b)

Fig. 7. Text compiler migration, a) Text compilation before migration, b) Text compilation after migration

The compiler screens before and after the migration are shown in the Figures 7 a) and 7 b), respectively. We find that the text compiler does migrate to the Windows-based PC, and the file C++Primer.txt is opened again and located at the same page and line.

*Task migration between two homogeneous platforms.* In the experiment, we migrate a classic Mozart sample, the TruckRace [11] program, between two Linux-based workstations. In the TruckRace program, 3 trucks start running after a click. We mark corresponding variables based on Table 2. The TruckRace.mp3 is a resource variable and plays a music as trucks keep on running. The picture Truck.gif is also a resource variable. The RaceTime refers to the time that the trucks start running. It is set as a state variable. Finally, the TruckPositions is set as a state variable that represents the current locations of the 3 trucks.

| *Items* | *Type* | *Denotation* |
|---|---|---|
| TruckRace.mp3 | resource | [ResourceAttribute] |
| Truck.gif | resource | [ResourceAttribute] |
| RaceTime | state | [StateAttribute] |
| TruckPositions | state | [StateAttribute] |

Table 2. Item types and denotations in TruckRace

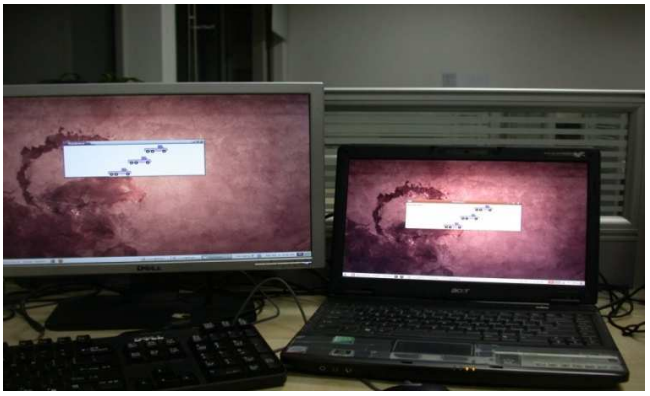


Fig. 8. State recovery in TruckRace

We find From Figure 8 that TruckRace is migrated from a Linux workstation to another. The states (i.e., locations of trucks) are recovered and the resources are rebound so that the music can be played and picture can be shown in the target Linux workstation.

### 5.2 Average Migration Latency

We define *average migration latency* as the average time interval from a migration request in a source node to program rerunning in a target platform. It is relevant to the size of modified Oz source program (i.e., file.oz) and the network bandwidth.

We use 19 Oz programs to test the average migration latency. As Figure 9 illustrates, we conduct our experiments with the bandwidth of 128.0 kbps, 256.0 kbps and

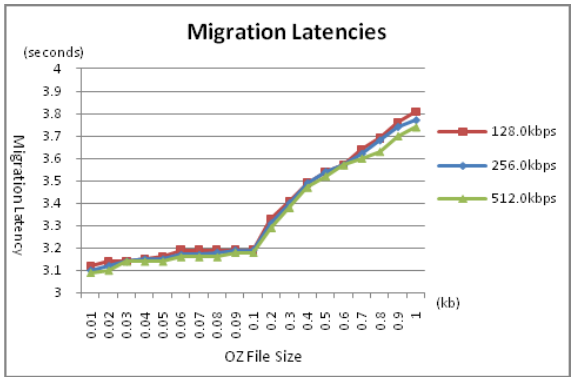



Fig. 9. Average migration latency

512.0 kbps, respectively. The size of Oz programs is set in the range [0.01 kb, 1.0 kb], which, we think, can cover most applications. Figure 9 shows that the migration latency grows significantly with the increase of program size. One reason is that the larger a file size is, the longer its transmission takes. Another more important reason is that a larger file typically needs more time for state recovery and resource rebinding.

### 5.3 Multi-Modality Interaction

Multi-modality interaction is another advantage of our approach that provides most desirable I/O interfaces for users. We have built our second prototype to exhibit the ability of our xMozart that can support multi-modality interaction through the above TruckRace scenario.

The source and target workstations use a standard mouse and a speech recognition system as input devices, respectively. We add a new reserved word InputModule to represent a car control module with interchangeable interfaces in this experiment. As a result, those two I/O adapters are successfully discovered and managed by our IOAM.

In the source workstation, we convert the events of left and right click on a mouse into a “start” and a “stop” text, respectively. In the target workstation, however, similar conversion is achieved by the speech recognition system. We find that after a migration, the TruckRace can be resumed in the target platform, i.e., the trucks start running or stoping when we speak “start” or “stop”, respectively. Furthermore, the TruckRace program has no response when we speak other words. In summary, our approach can support the multi-modality interaction very well.

### 5.4 Persisted Data Control

Our approach persists resource and state variables before a migration. The persisted data, then, are moved to a target platform, consuming the network bandwidth significantly. In this experiment, we test how the amount of persisted data grows with the size of the migrated programs.

We modified 27 classical Oz programs through adding new resource and state variables. The experimental results are shown in Figure 10 a). There is not a significant relationship between the persisted data and the program size because many large programs possibly do not need proportionable resource and state variables. Figure 10 b) indicates that the amount of persisted data is proportional to the number of declared variables. The reason is that more declared variables need more data persistence. On the other hand, in Figure 10, we notice that all the persisted data are below 1 KB. So, data persistence does not affect the performance of the networks and systems significantly.

## 6 CONCLUSIONS AND FUTURE WORK

We present a context semantics based task migration model and a migration framework xMozart. Our approach has the following advantages. Firstly, it can migrate tasks among homogeneous and heterogeneous platforms. Secondly, migrated tasks can resume their execution in the target platform from the last broken point. Thirdly, our approach does not actually migrate data resources. Instead, it rebinds the data resources in the target node, which saves the network bandwidth and reduces the average migration latency. Finally, it supports multi-modality interfaces for desirable user experience. The experiments demonstrate that our approach can migrate tasks like a shadow as users move in changing contexts.

As a part of our future work, we are going to improve our shadow function mechanism for local variables to further shorten the average migration latency.

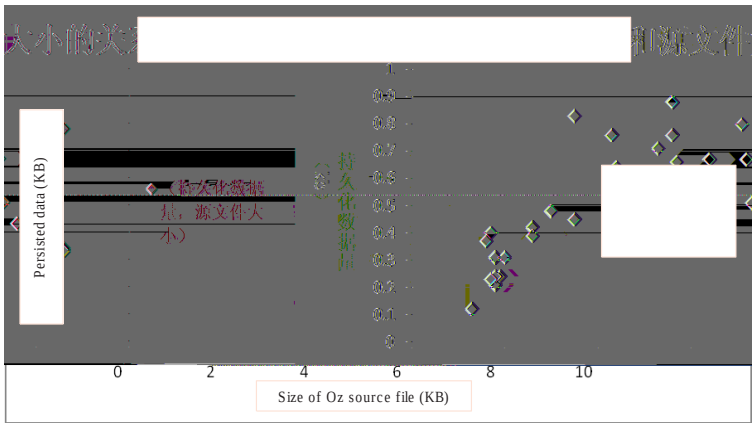
### Acknowledgements

This work is supported by the National Natural Science Foundation of China (NSFC) with Grant Nos. 61073148 and 60725208. An earlier version of this paper was presented in IMIS 2010 [12].

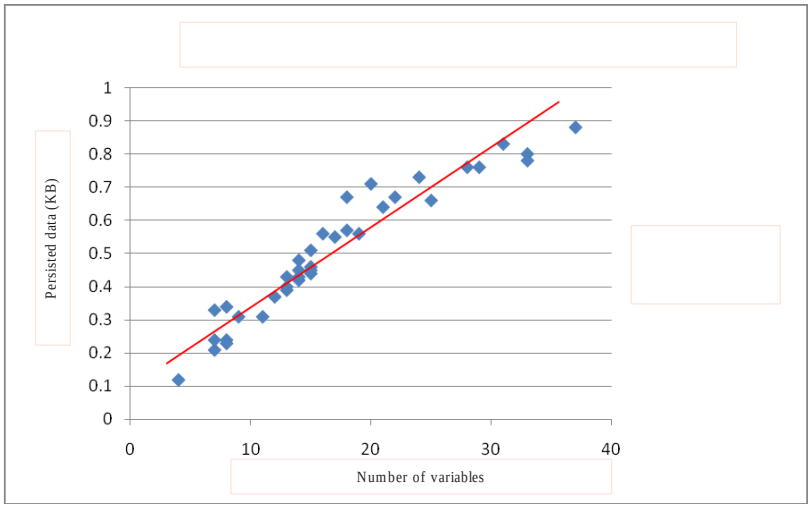
Feilong Tang would like to thank The Japan Society for the Promotion of Science (JSPS) for providing the excellent research environment during his JSPS Postdoctoral Research Fellow (ID No. P 09059) Program in Japan.

## REFERENCES

[1] Yu, Z. W.—Zhou, X. S. et al.: Supporting Context-Aware Media Recommendations for Smart Phones. IEEE Pervasive Computing, Vol. 5, 2006, No. 3, pp. 68–75.




a)




b)

Fig. 10. Persisted data control, a) Persisted data and Oz program size, b) Persisted data and declared variables

[2] Kiriyama, S.—Wakikawa, R.—Xia, J. W. et al.: Context Reflector for Proxy Mobile IPv6. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), Vol. 1, 2010, No. 2/3, pp. 36–51.

[3] Chang, X.—Cheung, S. C.—Chan, W. K. et al.: Heuristics-Based Strategies for Resolving Context Inconsistencies in Pervasive Computing Applications. Proceedigns of ICDCS '08, 2008, pp. 713–721.

[4] Chu, H.-H.—Song, H.—Wong, C. et al.: Roam: A Seamless Application Framework. The Journal of System and Software, Vol. 69, 2004, pp. 209–226.

[5] Kafle, V. P.—Inoue, M.: Locator ID Separation for Mobility Management in the New Generation Network. Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), Vol. 1, 2010, No. 2/3, pp. 3–15.

[6] Jonathan, M. S.: A Survey of Process Migration Mechanisms. Technical Report CUCS-324-88, Computer Science Department Columbia University New York, NY. 10027.

[7] Smolkal, G.: The OZ Programming Model. Proceedings of EURO-PAR '95, Vol. 966, 1995, pp. 5–8.

[8] Grolaux, D.—Roy, P. V.—Vanderdonckt, J.: QTk-A Mixed Declarative/Procedural Approach for Designing Executable User Interfaces. Proceedings of EHCI 2001, 2010, pp. 109–110.

[9] Gitzenis, S.—Bambos, N.: Joint Task Migration and Power Management in Wireless Computing. IEEE Transactions on Mobile Computing, Vol. 8, 2009, No. 9, pp. 1189–1204.

[10] Oikonomou, K.—Stavrakakis, I.: Scalable Service Migration in Autonomic Network Environments. IEEE Journal on Selected Areas in Communications, Vol. 28, 2010, No. 1, pp. 84–94.

[11] Miller, M.—Schulte, C.: Truck Race. Availaible on: `http://www.mozart-oz.org/home/doc/demo/trucks.html`.

[12] Wang, M.—Shen, Y.—Tang, F. L.—Guo, M. Y.: xMozart: A Novel Platform for Intelligent Task Migration. Proceedings of IMIS 2010, pp. 800–805.

**Feilong Tang** received his Ph. D. degree in Computer Science and Technology from Shanghai Jiao Tong University (SJTU) in 2005. Currently, he is a JSPS (Japan Society for the Promotion of Science) Postdoctoral Research Fellow. He works with the Department of Computer Science and Engineering of SJTU, China. His research interests include grid and pervasive computing, wireless and sensor networks, and distributed transaction processing.

**Can Tang** studies finance and computer science at Heilongjiang University, China. Her current research interests focus on computational finance and distributed computing.

**Shui Yu** received his B. Eng. and M. Eng. degrees from University of Electronic Science and Technology of China in 1993 and 1999, respectively. He received his Ph. D. (Computer Science) from Deakin University in 2004. He is currently a lecturer of the School of Information Technology, Deakin University, Australia. His research interest includes networking theory, network security and mathematical modeling.

**Minyi Guo** received his Ph. D. degree in Computer Science from University of Tsukuba, Japan. He is now a Full Professor at the Department of Computer Science and Engineering, Shanghai Jiao Tong University, China. His research interests include pervasive computing, parallel and distributed processing, parallelizing compilers and software engineering.

**Song Guo** received the Ph. D. degree in Computer Science from the University of Ottawa, Canada in 2005. He is currently an Assistant Professor at School of Computer Science and Engineering, the University of Aizu, Japan. His research interests include protocol design and performance analysis for communication networks