JACET

J. ADV COMP ENG TECHNOL, 5(4) Autumn 2019

A Version Numbering Scheme for Informational Objects Used in VM Live Migration

Majid Tajamolian[1], Mohammad Ghasemzadeh[2]
1- Computer Engineering Department, Yazd university, Yazd, Iran.
2- Computer Engineering Department, Yazd university, Yazd, Iran.(m.ghasemzadeh@yazd.ac.ir)

Received (2019-02-22) Accepted (2019-08-08)

Abstract: *Various numbering schemes are used to track different versions and revisions of files, software packages, and documents. One major challenge in this regard is the lack of an all-purpose, adaptive, comprehensive and efficient standard. To resolve the challenge, this article presents Quadruple Adaptive Version Numbering Scheme. In the proposed scheme, the version identifier consists of four integers. These four numbers from Left to Right are called: "Release Sequence Number", "Generation Number", "Features List Number", and "Corrections List Number" respectively. In the article, special values are given for the quadruple numbers and their meanings are described. QAVNS is an "Adaptive" scheme; this means that it has the capability to track the different versions and revisions of files, software packages, project output documents, design documents, rules, manuals, style sheets, drawings, graphics, administrative and legal documents, and the other types of "Informational Objects" in different environments, without alterations in its structure. The proposed scheme has the capability to monitor changes in the types of informational objects, such as virtual machine memory, in the live migration process. The experimental and analytical results indicate the desirability and effectiveness of the proposed scheme in satisfying the desired expectations. The proposed scheme can become a common standard and successfully applied in all academic, engineering, administrative, legislative, legal, manufacturing, industrial, operational, software development, documentary and other environments. The standardization of this scheme and its widespread usage can be a great help in improving everyone's understanding of the numbering of versions & revisions.*

Keywords: *Quadruple Adaptive Version Numbering Scheme, Informational Object, Virtual Machine Memory, Virtual Machines Live Migration.*

How to cite this article:
Majid Tajamolian, Mohammad Ghasemzadeh. A Version Numbering Scheme for Informational Objects Used in VM Live Migration. J. ADV COMP ENG TECHNOL, 5(4) Autumn 2019 : 245-254

I. INTRODUCTION

Cloud computing is one of the important aspects of information technology in recent years. Moei Emamqeysi et al. in their article [1] quote from Chen et al. paper [2]: "Cloud Computing from the perspective of the National Institute of Standards and Technology is a model which by using its users will be able to receive configurable shared resources such as networks, provider, storage, application and services when they apply. These sources with minimal management effort or without any need for interaction with a service provider, promptly can be prepared and released".

To achieve "minimal management effort" (as quoted above), cloud computing uses various techniques such as *Virtual Machines Live Migration*. In this field, there are many approaches in selecting an appropriate Physical

Machine (PM) as a destination for a migrating VM. The PM ranking parameters include: Performance efficiency, Communication cost between VMs, Power consumption, Workload, Temperature efficiency and Availability [3]. Datacenter operators heavily rely on live-migration to perform maintenance operations over production infrastructures [4].

One of the important issues in virtual machines live migration is how to monitor the changes of virtual machine memory in the migration process and decide on how to migrate based on it. In this regard, it is useful to have a scheme that can efficiently display and track virtual machine memory changes as the versions of an informational object.

In the world of information technology, different numbering schemes are used to track the various versions of editions, software packages, and documents. The structure of many of these designs depends on the taste of the relevant project managers. Due to the lack of standardization of numbering schemes, there are many ambiguities in the minds of individuals (especially audiences outside of the projects). Also, the concepts found in many of these schemes are only suitable for a particular type of project output and are inappropriate for other types. Also, because of this lack of standardization, the definition of precise and measurable criteria based on the values that are available in the current versioning schemes identifiers is not possible. This leads to difficulty & imprecision of automation of some of the manual procedures such as "the decision making on how VM live migration should occur" (i.e. the authors' research interest).

In this article, a new numbering scheme is presented, which, along with the strengths of the existing schemes, also covers their weaknesses and is also adaptive (multipurpose applicable). The scheme can be effectively used in the field of our active research which is "Virtual Machines Live Migration". Of course, the application of numbering schemes in the field of virtual machines and cloud computing is not unprecedented [5].

The following is an outline of this article: In the section 2, a variety of existing schemes for the version numbering are introduced. In the section 3, the structure and operation of the proposed scheme in details are presented. In the section 4, the results of the implementation of the proposed scheme in two real work environments are reported and evaluated. At the end (section 5), conclusions and suggestions are presented.

II. AVAILABLE SCHEMES

Currently, many different version numbering schemes are being used by software & other digital content developers & vendors. They can be divided into four broad categories:

1. Schemes with the order-based numeric identifiers
2. Schemes with the identifiers based on the publication date
3. Schemes with the alpha-numeric code identifiers
4. Other miscellaneous schemes

1. Schemes with the order-based numeric identifiers

These schemes are the most common type in the software industry. In these schemes, each version number is based on one or more numerical levels that increase in order. In most cases, going from left to the right, the value of the digits is reduced as what occurs in a regular decimal number. Of course, definitely there are not just two decimal places in such schemes, but there may be several levels, and also the separator instead of the dot is one of the other characters such as dash (-). If the changes are large in relation to the previous version, the leftmost digits are changed; if the changes are moderate, the middle digits, if the changes are low, then the next digits, and if too small, the rightmost digits will change. The amount of change in the digits depends on the publisher's opinion, and it is not certain that it is only one unit. For example, when version 5.5 of the Internet Explorer 5.0 was released, it suggests that the changes in the previous version were relatively high. Another point to be taken into account in these schemes is that the levels are not necessarily single digit. For example, in the Drupal project, we have the version number 7.61 [6].

Another approach in this category is to use two major and minor sequential numbers, along with an alpha-numeric string such as a, b , and rc (Release Candidate) for the type of publication. A "software release sequence" that uses this

approach can be displayed as below:

$$0.7 \rightarrow 0.8 \rightarrow 0.9 \rightarrow 1.0a1 \rightarrow 1.0b1 \rightarrow 1.0b2 \rightarrow 1.0rc1 \rightarrow 1.0$$

Figure 1: A Software Release Sequence that uses AlphaNumeric approach

Other approaches like the following are also included in this category (the items in [] means they are optional):

major.minor[.build[.revision]]
major.minor[.maintenance[.build]]

Many open source projects use such schemes [7]. In these schemes, like all of the above, the definition of what change is "major" or "minor" exactly depends on the publisher's opinion. What defines "build" and that the difference between a "revision" and a "minor change" again depends on the publisher's perception. Therefore, the problem of lack of a standard understanding between the publisher and the audience is evident in these schemes.

The latest effort in this category was presented by Mr. Preston Werner, which is known as "Semantic Versioning". His scheme is based on a three-part structure, in the form of "version. revision.change", or as is described in version 2.0.0 of his scheme, in the form of "major.minor. patch" [8]. Though his scheme has solved many problems with the other order-based schemes, his proposed scheme still suffers from weaknesses that in the followings we mention some of them:

- There is no description of the scheme's application in non-software production environments.
- It's not clear distinctly that in what situations "minor" and "patch" fields will be changed.
- Failure to distinguish between situations in which a publication has both of minor and patch changes simultaneously, and in the event that only minor changes occurred.
- Lacking the ability to inform audiences enough about the maturity level of the software package
- Unable to perform "Lexicographical Ordering" on the name of the objects containing their version ID. Note that in many operating environments, the Lexicographical Ordering is necessary for objects and their respective files [9].
- Raemaekers et al. report that the current mechanisms in Semantic Versioning for signaling interface instability are not used properly [10].

The "semantic versioning" scheme seems to be just suitable for use in the production of software that has the "Application Programming Interface" (API) and has defined it accurately. Of course, it should be noted that the mentioned scheme in relation to "Dependency Management" and the prevention of falling into the trap of "Dependency Hell" has a positive impact on software projects, which is why, despite the little time it has been presented, much attention has been paid to it by the activists in this field.

2. Schemes with IDs based on release date

In these schemes, the identifiers are related to the release date. For example, the famous WINE software project used the combination of year, month, and day (like Wine 20040505) as a version identifier [11]. The popular distribution of GNU/Linux operating system, Ubuntu, also uses the similar version ID scheme. For example, Ubuntu 18.04 has been released in April 2018 [12]. Another example is the Street Fighter EX computer game, which at the start shows its version identifier as a date plus a geolocation code (such as ASIA 961219).

Most of the time when using the date in the version identifier, the scheme follows the ISO 8601 standard [13], [14]. Table 1 shows the months of year and their ordinal values as mentioned in the ISO 8601 standard:

Table 1: Calendar months & their ordinal values

Calendar month number	Calendar month name	Number of days in the month	Ordinal dates of the days in common years	Ordinal dates of the days in leap years
01	January	31	001-031	001-031
02	February	28 (leap year 29)	032-059	032-060
03	March	31	060-090	061-091
04	April	30	091-120	092-121
05	May	31	121-151	122-152
06	June	30	152-181	153-182
07	July	31	182-212	183-213
08	August	31	213-243	214-244
09	September	30	244-273	245-274
10	October	31	274-304	275-305
11	November	30	305-334	306-335
12	December	31	335-365	336-366

As part 2.3 of Annex B of the standard says, date and time of day should follow formatting like

Table 2:

Table 2: Date and time of day format

Basic format	Extended format	Explanation
YYYYDDDThhmm	YYYY-DDDThh-mm	complete ordinal date — reduced accuracy time of day
YYYYMMDDhhmm,m	YYYY-MM-DDhh:mm,m	complete calendar date — reduced accuracy time of day with one digit decimal fraction for minute — no time designator
YYYYWwwDThh,hhZ	YYYY-Www-DThh,hhZ	complete week date — reduced accuracy UTC of day with two digit decimal fraction for the hour

Of course, in most of the versioning schemes based on date and time, the "Basic format" is used instead of the "Extended format".

Tables below comprise some other samples as mentioned in part 1.3 of Annex B of the standard:

Table 3: Combinations of calendar date and local time

Basic format	Extended format	Explanation
19850412T101530	1985-04-12T10:15:30	Complete

Table 4: Combinations of ordinal date and UTC of day

Basic format	Extended format	Explanation
1985102T235030Z	1985-102T23:50:30Z	Complete

Table 5: Combinations of week date and local time

Basic format	Extended format	Explanation
1985W155T235030	1985-W15-5T23:50:30	Complete

3. Alphanumeric-based identifiers

In this set of schemes, alphanumeric codes are used to create the version identifier. For example, a number of well-known Microsoft company products, titled MS-Windows XP, MS-Windows Me, MS-Windows Vista, and the Flash MX product of Macromedia Company, are mentioned.

Another example of this bunch of schemes can be found in the popular distribution of the GNU/Linux operating system called Debian. Although this distribution uses major & minor sequence numbers for its stable versions, but along with them, it uses code names that are inspired by the characters of "Toy Story" animation. For example, at the time of composing this article, the last stable version of the Debian distribution is 9.5 which also known as Stretch. The testing versions of this distribution have identifiers based on the date of issue, and its unstable version is always called Sid (the so-called evil son of the neighbor in the aforementioned animation) [15].

In most of these schemes, a specific and uniform rule is not used and the promotional and commercial aspects of the product are overcome by scientific, uniform and standard logic.

4. Other misc. schemes

Of course, we also encounter with unusual and miscellaneous schemes that mostly imply some fantasy aspects. For example, the famous TeX word processor uses a unique method. After publishing version 3 of the software that it has been achieved largely maturity and stability, subsequent updates are displayed by adding a digit to the end of the previous version number with the aim of asymptotic approach to the value of Pi number (Π). In fact, the actual version number of the TeX software is the same as the number of digits of its version identifier! At the time of writing this article, the version number of the software is 3.14159265.

Similarly, the version number of Metafont interpreter software, which like TeX is published by Professor Donald Knuth, is asymptotically approaching to the number e. Currently the version number for this software is 2.7182818.

Obviously, such schemes, in spite of their sense of attractiveness and originality, are mostly fancy, and do not make the audience aware of the type and extent of changes in each version relative to the previous version.

III. THE PROPOSED SCHEME

In order to overcome the problems described in the previous section, we propose a quadruple scheme for the numbering of the version identifiers that is adaptable (multi-purpose usable). "Adaptability" means that the scheme has the capability to track the various versions and revisions of files, software packages, project output documentation, designs, rules, manuals, style sheets, drawings, graphics, administrative and legal documents, and the other similar things in different environments, without changing its structure. So from now on, in this article, in order to emphasize the multipurpose usability,

we will use the word "object" for all of the above. We call our proposed scheme "QAVNS", which is an acronym and stands for "Quadruple Adaptive Version Numbering Scheme".

In our proposed scheme, QAVNS, numbering is performed using four integer fields separated by the "." (Dot character), and are named from the left to the right "Release Sequence Number" (RSN field), "Generation Number" (G field), "Features List" (FF field) and "Corrections List" (CCC field), respectively.

Production of an object begins with the assignment of 0.0.0.0 as version ID to it. The zero number at the position of the Release Sequence Number indicates that this is the first creation of the object. The zero number in the position of Generation Number indicates that the object is still in its early stages and does not have the ability to be presented (even as experimental). The zero number in the position of the Features List field indicates that the object doesn't contain the minimum (even incomplete) content or functionality that is noticeable. The zero number at the position of the Corrections List indicates that no corrections have performed on the current version of the object. By adding content or features to it (without making corrections), subsequent versions are 1.0.1.0, 2.0.2.0, 3.0.3.0, 4.0.4.0 and likewise. None of the above quadrants are required to be single-digits; for example, after version 15.0.9.0, we can reach 16.0.10.0.

Changes in each version to the previous versions are referred to as the "Directed Delta" or, in short, the "Delta". It means "a series of basic change operations that if applied on one of the versions, the next one is obtained". Another kind of delta called "Symmetric Delta" is also defined [16], which is not our objective here.

How the degree and type of changes made to successive informational objects are detected, and what sorting is done upon them, depends to a great extent on the area of the informational object usage. As an example, Bauml and his colleagues have proposed a solution for automatic detection and determination of version of software components that are developed according to the OSGi standard [17]. Of course, in this article, we do not enter this category because of the general and adaptive approach we have, and instead provide general criteria for determining the extent and type of changes made to information objects and their impact on the object's version identifier.

If none of the changes made to the object are in the form of adding and/or modifying the functionality or contents, but all are corrections and fixings, instead of the number of the Features List, only the number of Corrections List will be bumped. For example, if after version 11.0.7.0 a number of changes are made to just correct and fix the object, the new version would be 12.0.7.1 and not 12.0.8.0 (which indicates the change only in features and content). Similarly, if all changes in the subsequent version are correction and fixing, the Corrections List will be bumped again and the next version will be 13.0.7.2.

If the changes made to the object related to the previous version are a combination of "adding and/or modifying the functionality or content" and "correction and fixing", then the Features List is bumped and the Corrections List is rolled to "one". So for such a case in the above example, we will have version 28.0.8.1 after the 27.0.7.15. Note that in the above example, if the new version changes related to the previous version is restricted to the addition and/or modification of the functionality or content of the object and there is no change as the correction and fixing, then after version 27.0.7.15 we will go to version 28.0.8.0; i.e. the Features List number will be bumped and the Corrections List number will be rolled to zero. By following this rule, it's easy to see if the new version of the object contains what kind of changes related to the previous version, by comparing the new version with the previous.

- If only the corrections list number is increased, it means that the new version of the object has only a few corrections and amendments to the previous version.
- If the features list number is increased and the corrections list number is reset to zero, it means that the new version of the object compared to the previous version only contains a number of changes of the form of adding and/or modifying the functionality or content.
- If the features list number is increased and the corrections list number is reset to one, it means that the new version of the object, compared to the previous version, contains a number of alterations of the form of adding and/or changing the

functionality or content, as well as some corrections and amendments.

Generation number change from zero to one occurs when the first presentable generation is available. What is meant by "presentable" depends on organizational definitions. Then, if the changes made to the object are substantial and fundamental, it will lead to subsequent changes to the Generation number. For example, from version number 235.1.14.772 it goes to 236.2.0.0 version ID. It is important to note that if the Generation number increases, the Features List number and the Corrections List number must be rolled to the zero.

Unlike some commercial schemes introduced in the second section of the article, in our proposed scheme, the amount of changes made to an object does not affect the pace of changing the four fields of version ID. Therefore, whatsoever the number of corrected bugs between two successive versions is high, the Corrections List number will increase by only one unit. Correspondingly, whatsoever the number of capabilities/contents added or changed between two successive versions is high, the Features List number only increases by one unit. Also, whatsoever fundamental changes are made between two sequential versions of an object is high, the Generation number increases by only one unit. We believe that the expression of the amount of changes by the pace of changing numbers is not an appropriate and exact idea. Instead, providing "Change Log" reports along with any version of an object can clearly tell audiences how much change is made to the previous version.

The change in the "Release Sequence Number" is simply a unit increment in each version. This number is in fact a sequential serial number, the existence of which at the beginning of the version identifier will have the following benefits:

- Specifies the number of submitted versions of the object since its inception.
- Combined with the second field (Generation number), it can indicates the maturity of the object's content. For example, compare an object with the 9.5.0.3 version ID with another object with the 417.5.0.3 version ID. Although both are from the fifth generation and have the same "Features List" and "Corrections List" fields, the comparison of the first field of the two objects shows that the second object is much more mature.
- When updating, the "Release Sequence Number" field tells the user how many releases exist between what he has and what he wants to update. For example, update from version 9.5.0.3 to 34.5.1.1, which represents 25 intervals between existing version and update version.
- If some of the points are observed, this field can also play a facilitating role for "lexicographical ordering".

So briefly, if we want to put the structure of a version identifier as a model in the QAVNS scheme, it's like the followings:

e0RSN.G.FF.CCC

There are three sets of changes upon informational objects:

1. Correction and fixing errors of informational object (we named them as set ε)
2. Adding and/or modifying the functionality or content of informational object (we named them as set λ)
3. Substantial and fundamental changes on informational object (we named them as set Ω)

Suppose Δ_i as the set of changes that are made on version i of an informational object. Then we have:

$$\Delta_i = \{\delta_i \mid \delta_i \in \varepsilon \oplus \delta_i \in \lambda \oplus \delta_i \in \Omega\}$$

Our proposed algorithm for changing of QAVNS fields is shown in Figure 2.

IV. RELATED WORK & EXPERIMENTAL RESULTS

The authors of this article are currently researching virtual machines live migration. We have used the proposed scheme for the numbering of the virtual machine's main memory versions (as an informational object) and tracking the type and extent of its changes.

In the context of virtual machines live migration, two strategies called “pre-copy” and “post-copy” have already been presented; but each of these strategies works well only in some

circumstances. In our research, we present an innovative approach which is based on the concept of "Informational Object", assigning QAVNS revision numbers, and observing its changes.

In this approach, the total virtual machine memory is considered as a QAVNS informational object that is constantly changing. The changes in the version number of the informational object (virtual machine memory) indicate the type and extent of the change. We have added an initial step called "sampling stage" to the three steps that Clark and colleagues have proposed for the virtual machine migration process [18]. At this stage, the behavior of the virtual machine is monitored and the decision-making algorithm we provide can detect the current behavior of the virtual machine based on the results of sampling stage. Then the virtual machine hypervisor can automatically select the appropriate VM migration strategy from both pre-copy and post-copy options. In this regard, we have defined a number of criteria named "length of sampling stage", "virtual machine OS instance number", "number of process changes in virtual machine" and "number of modified memory pages", that the decision-making algorithm works based on their changes.

In order to provide an automated algorithm for choosing a virtual machine live migration method, we provide criteria for automatically detecting virtual machine state. Based on the concept of the "Informational Object" presented in [2], we assume virtual machine's memory as an informational object that is constantly changing and can therefore have a "QAVNS Revision Number" at any moment.

Assume that the *MST* (Migration Start Time) constant represents the start time of the live migration of virtual machine memory.

$$jiffies = \text{count(TimerInterrupts)} \quad (1)$$

$$MST = \text{jiffies at start of migration tas} \quad (2)$$

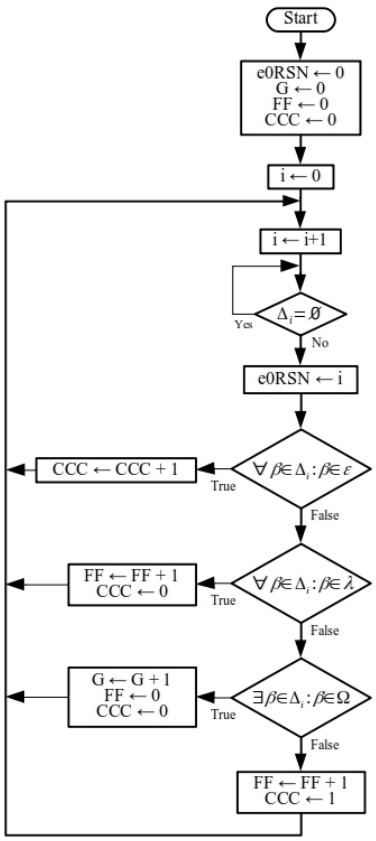


Figure 2: Algorithm for changing QAVNS fields

The suggested mapping of QAVNS fields in the VM memory informational object is as follows:

The 0RSN field in the QAVNS scheme is known as the "Release Sequence Number". Given the nature of the virtual machine memory informational object, we consider this field as a sequence number that has a time nature and its values are extracted from machine timer interrupts. This field represents the number of moments that have elapsed since the start of the migration operation. Therefore, the *0RSN* field of the QAVNS scheme is mapped here as the "jiffies" with the *JS* acronym.

The *GG* field in the QAVNS scheme is known as the "Generation number". Given the nature of the virtual machine memory informational object, we consider this field to be the "Generation Number of the virtual machine operating system". Therefore, the *GG* field of the QAVNS scheme is mapped here as the "Operating System Instance Number" with the OSIN acronym.

The *FF* field in the QAVNS scheme is known as the "Feature List Number". Given the nature of the virtual machine's memory informational

object, we consider this field to be "indicator of the volume of changes in processes of the virtual machine". Therefore, the *FF* field of the QAVNS scheme is mapped here as the "Process Change Number" with the PCN acronym.

The *CCC* field in the QAVNS scheme is known as the "Correction List Number". Given the nature of the virtual machine's memory informational object, we consider this field to be "number of the modified memory pages". Therefore, the *CCC* field of the QAVNS scheme is mapped here as "Dirty Page Count Number" with the *DPCN* acronym.

Therefore, the "Version Identifier Scheme" for virtual machine memory as a QAVNS-based informational object is:

$$JS.OSIN.PCN.DPCN \quad (3)$$

The VM memory revision identifier is displayed at the start of the sampling phase with the index s and at the end of the sampling phase with the index e. So we will have:

Revision identifier at the beginning of the sampling phase:

$$JS_s.OSIN_s.PCN_s.DPCN_s \quad (4)$$

Revision identifier at the end of the sampling phase:

$$JS_e.OSIN_e.PCN_e.DPCN_e \quad (5)$$

We claim that the following algorithm can automatically detect the different operating conditions of a virtual machine and choose the appropriate approach for live migration appropriately. Flowchart of the proposed method and our decision making algorithm is shown in Figure 3.

We simulate pre-copy, post-copy, and our synthetic algorithms by coding on MATLAB and GNU Octave platforms. Simulation shows that using the proposed method and the proposed algorithm can automatically prevent the weaknesses of pre-copy & post-copy methods (if used alone), though utilizing strengths of them in the VM live migration process [19].

In this research, we have used 100 data sets to be implemented in ten different categories of behavior (each category contains 10 data sets). The charts in Figure 4 to Figure 6 show the results of the simulation of pre-copy and post-copy methods on these ten categories of data sets, respectively, for the "Average performance degradation of applications", "Average data transfer volume during migration" and "Average time length of virtual machine migration" criteria:

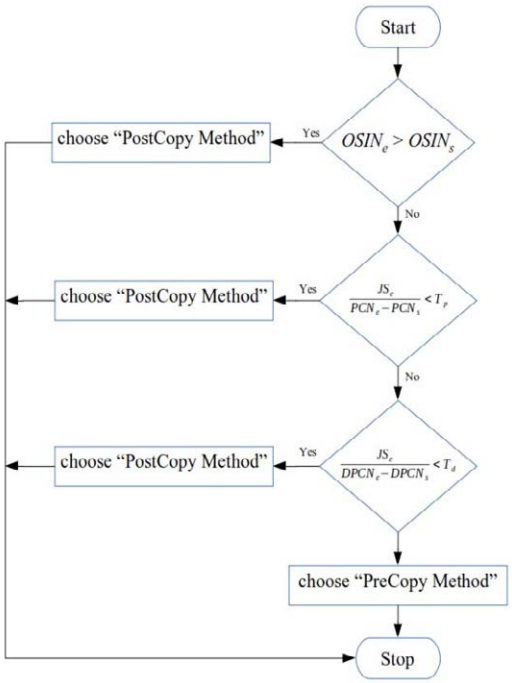
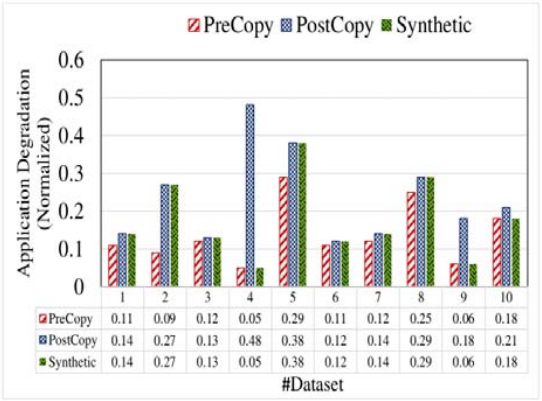


Figure 3: Our decision making algorithm based on QAVNS fields values



	1	2	3	4	5	6	7	8	9	10
PreCopy	0.11	0.09	0.12	0.05	0.29	0.11	0.12	0.25	0.06	0.18
PostCopy	0.14	0.27	0.13	0.48	0.38	0.12	0.14	0.29	0.18	0.21
Synthetic	0.14	0.27	0.13	0.05	0.38	0.12	0.14	0.29	0.06	0.18

Figure 4: Average performance degradation of applications in PreCopy, PostCopy & proposed methods

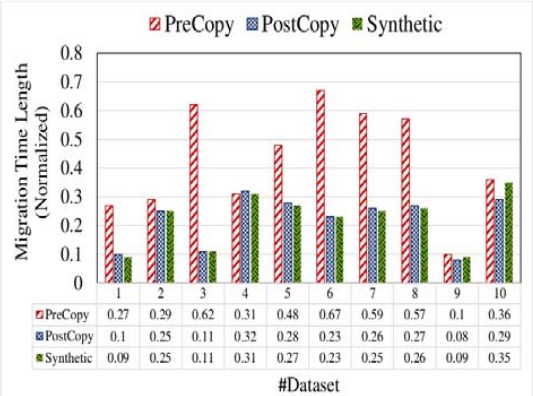


Figure 5: Average data transfer volume during migration in PreCopy, PostCopy & proposed methods

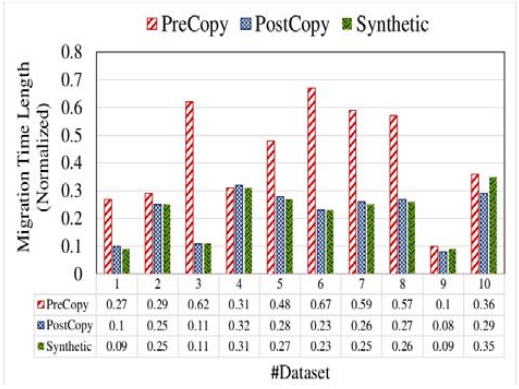


Figure 6: Average time length of virtual machine migration in PreCopy, PostCopy & proposed methods

The Data Sets that we used in our simulation have the structure that is shown in Table 6:

Table 6: Structure of Data Sets of our simulation

Header	VM_RAM (GB) , VM_Bandwidth (Gbps) , JS_Time_Length (ms) , VM_Workload_Percentage
Body	**JS_1, event_code, event_parameters** **JS_2, event_code, event_parameters** **JS_3, event_code, event_parameters** . **JS_i, event_code, event_parameters** ...

Table 7: Events that are defined in our simulation

Event Code	Event Desc.
1	OS Reboot
2	Process Creat/Kill
3	Read Memory page
4	Write Memory page

Table 8 shows the parameters values of each event in our used Data Sets:

Table 8: Parameters of each event in Data Sets

Event Code	Event Parameters
1	-1 (no parameters)
2	1 (Create)
	2 (Kill)
3	Page# for Read
4	Page# for write

The efficiency of the proposed approach is shown in our simulation results. Figure 7 shows number of VM migration fails in our proposed method versus PreCopy & PostCopy methods.

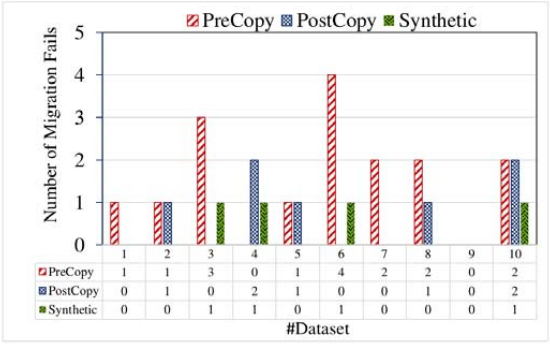


Figure 7: VM live migration fails in the three methods

The scheme (QAVNS) has also been successfully implemented in an academic research environment and a knowledge-based company to track and control technical and administrative informational objects. The use of the new scheme in both environments has led to solving the problems of organizing the IDs of revisions and versions.

In the course of the research, we also gradually developed QAVNS to cover all of our needs as a single scheme to systematize the identifier of revisions and versions of various types of informational objects. The objects we used QAVNS on them in these two environments were: software, technical documents, project reports, administrative documents, contracts, manuals, style sheets and graphic designs.

The experimental and analytical results indicate the desirability and effectiveness of the proposed scheme in meeting expectations. It was also found that QAVNS training to employees and peers in various activities in a short time is

feasible. In addition, this scheme can be used to automate the selection of appropriate methods for the "VM live migration".

V. CONCLUSIONS AND SUGGESTIONS

The proposed scheme (QAVNS) is a useful tool for monitoring the type and extent of changes in the memory of a virtual machine (as an informational object) with the aim of automated selection of a proper method for VM live migration.

In addition, the proposed scheme can become a common standard and applied successfully as a comprehensive and competent scheme in all administrative, operational, artistic, software development, documentary, legislative, publishing and other related areas. The standardization of this scheme will be a great contribution to uniform understandings of everyone about the numbering of versions and revisions. It is suggested that agencies, organizations, companies, educational institutions and manufacturing institutions introduce the QAVNS scheme as a comprehensive standard and apply it at all organizational levels.

As a future work, it's suggested that QAVNS be used as an alternative versioning scheme in other research fields for tracking changes of different informational objects and report its degree of effectiveness.

REFERENCES

1. H. Moei Emamqeysi, N. Soltani, M. Robati, and M. Davarpanah, "A review of methods for resource allocation and operational framework in cloud computing," J. of Advances in Comput. Eng. and Technology (JACET), vol. 3, no. 3, pp. 173–180, Aug. 2017.
2. Z. Chen and J. Yoon, "IT Auditing to Assure a Secure Cloud Computing," in 2010 6th World Congr. on Services, 2010, pp. 253–259.
3. S. R. Hosseini, S. Adabi, and R. Tavoli, "A Near Optimal Approach in Choosing The Appropriate Physical Machines for Live Virtual Machines Migration in Cloud Computing," J. of Advances in Comput. Eng. and Technology (JACET), vol. 1, no. 3, pp. 23–32, Oct. 2015.
4. V. Kherbache, É. Madelaine, and F. Hermenier, "Scheduling Live Migration of Virtual Machines," IEEE Trans. on Cloud Computing, pp. 1–14, Sep. 2017.
5. M. Cavage, D. Pacheco, B. Cantrill, and N. Fitch, "Versioning schemes for compute-centric object stores," Patent US9092238B2, 28-Jul-2015.
6. "drupal 7.61 ReleaseNotes." [Online]. Available: https://www.drupal.org/. [Accessed: 20-Nov-2018].
7. J. R. Erenkrantz, "Release management within open source projects," in Proc. ICSE'03 Int. Conf. on Software Eng., Portland, Oregon, USA, 2003, pp. 51–55.
8. Tom Preston-Werner, "Semantic Versioning 2.0.0." [Online]. Available: http://semver.org/. [Accessed: 09-Jun-2018].
9. A. M. Keller and J. D. Ullman, "A version numbering scheme with a useful lexicographical order," in Proc. 11th Int. Conf. on Data Eng., 1995, pp. 240–248.
10. S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in IEEE 14th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM), Victoria, BC, Canada, 2014, pp. 215–224.
11. "Wine Project History - WineHQ." [Online]. Available: https://wiki.winehq.org. [Accessed: 17-May-2018].
12. "Ubuntu 18.04 ReleaseNotes." [Online]. Available: https://wiki.ubuntu.com. [Accessed: 27-Oct-2018].
13. International Organization for Standardization, "ISO 8601:2004 - Data elements and interchange formats -- Information interchange -- Representation of dates and times," ISO Standard 8601, Dec. 2004.
14. Markus Kuhn, "A summary of the international standard date and time notation." [Online]. Available: http://www.cl.cam.ac.uk/~mgk25/iso-time.html. [Accessed: 28-May-2018].
15. "Debian Releases." [Online]. Available: https://www.debian.org. [Accessed: 30-Apr-2018].
16. R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," ACM Comput. Surv., vol. 30, no. 2, pp. 232–282, Jun. 1998.
17. J. Bauml and P. Brada, "Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee," in Proc. 35th Euromicro Conf. on Software Eng. and Advanced Applicat., Patras, Greece, 2009, pp. 428–435.
18. C. Clark et al., "Live migration of virtual machines," in Proc. of the 2nd conf. on Symp. on Networked Syst. Design & Implementation, Berkeley, CA, USA, 2005, vol. 2, pp. 273–286.
19. Majid Tajamolian, "A New Synthetic Method for Virtual Machine Live Migration," PhD. Dissertation (in Persian), Yazd University, Yazd, Iran, 2019.