Analyzing Software Updates: Should You Build a Dynamic Updating Infrastructure?

Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang

Iowa State University

Abstract. The ability to adapt software systems to fix bugs, add/change features without restarting it is becoming important for many domains including but not limited to finance, social networking, control systems, etc. Fortunately, many ideas have begun to emerge under the umbrella term “dyanamic updating" to solve this problem. Dynamic updating is critical to address certain software evolution needs. Dynamic updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. However, we do not have a technique to analyze whether certain updating solution, based on its costs and benefits, is suitable for an application.
In this paper, we present a quantitative analysis model to fill this gap. Our model is parameterized and it can be instantiated with application-specific valuation functions. Given the software evolution history of the application under consideration, our model allows rigorous comparisons of the value of different software updating schemes (e.g. online vs. offline). We illustrate our model using two case studies inspired from the the evolution history of Xerces XML parser library and Apache httpd web server (Other case studies and evaluation examples are presented in our technical report [Gharaibeh, Rajan and Chang 09]). The proposed analysis scheme can serve system architects in evaluating their current updating scheme. For example, to audit the system’s value during previous development cycles and whether a different updating scheme will generate higher value.

1 Introduction

Software evolution and maintenance is a fact of life [3,14]. Enhancements, security, and bug fixes are routinely made to a software system during its usable life. Long running software systems such as web and application servers, financial software, critical control systems often need to balance evolution and availability requirements. For such systems downtime due to software update is unacceptable and often very costly [13,16,25].

Dynamic software updating has attracted significant interest in the last few years [6,19,23]. This is due to the benefits software updating can provide to long running applications. The interest in dynamic updating is clear from a plethora of research efforts and a specialized workshop (i.e. HotSwUp). Such interest is only expected to continue with the industrial trends towards software as long-running services in service-oriented architectures.

However, adopting any dynamic updating scheme requires deep understanding about its cost and benefits beyond the stated software engineering benefits. To date, dynamic updating literature evaluates such systems in terms of coverage (i.e. what type

of code changes are supported) and performance. For example, Chen *et al.* [6] evaluated their system over a set of server applications. The evaluation was in terms of average server's response time before and during the update process.

What is missing is a formal quantitative analysis that allows us to study such a system in comparison to current static update practices or other dynamic updating systems. We need to answer the question of whether the benefits of dynamic updating justifies its cost (performance or regular fees). In theory, being online 24/7 is a priceless advantage. However, this may not apply to all systems. Given the real history of bugs in a particular software, does the loss of system value due to these bugs justifies the investment in dynamic updating? The answer depends on many factors related to system operations and bug's severity.

The contribution of this work is a quantitative value model that allows us to study the gain from updating systems. Our model is based on Net option-value (NOV) analysis [28]. NOV has been devised to price options in a financial market and has also been used to study the cost and benefit of modularity in designs [2, 15, 27]. Our value model allows us to study the relation between updating system's operational parameters (e.g. cost and timing) and value provided to users. To the best of our knowledge, this is the first attempt to quantitatively formulate and evaluate the costs/benefits of offline and dynamic updating in software systems.

The proposed model can be used in different scenarios. For example, it can be used to audit the system's value during previous development cycles. By using information about added features and their revenue, developers can compare the current update practice and whether a different update strategy would provide higher value. It can also be used to quantitatively compare different dynamic updating schemes. Given the characteristics of two updating schemes such as types of supported updates and performance characteristics, the two schemes can be quantitatively compared using a set of benchmark features.

We have applied our model to two case studies: the evolution of the XML parser library Xerces [29], and 42 bug fixes for Apache httpd server obtained from Bugzilla (Section 3). Other case studies are presented in our technical report [Gharaibeh, Rajan and Chang 09]. Using these case-studies, we studied the model's trends, relative values depending on the selected parameters, and assess its precision. These studies also give us insights on how one would actually go about estimating the parameters that serve as the input to the model. We believe this to be a very useful aide to system developers and maintainers. To summarize, our contributions in this paper are:

- A quantitative model for cost/benefit analysis of updating systems and its formulation. The novelty of the model is in its application of net options value theory to the area of software updates.
- A case study from software evolution of a real-world application that illustrate the use of our model. The main benefit of the case studies is that they give insights into selection of the model parameters.

The rest of this paper is organized as follows. In Section 2 we discuss our quantitative model . We describe our case studies in Section 3. Section 4 presents the related work while Section 5 discuss various aspects and limitations of our evaluation model. Section 6 discusses directions for future investigations and concludes.

2 Quantifying Software Update

This section presents our analysis model. The main idea behind the analysis model is the computation of daily revenue of the system. By understanding how different updating policies affect the daily value, we can calculate the effect on total revenue made by these systems.

2.1 Update Models

We will evaluate the following updating models:

- Model 0: Offline update at release time.
- Model 1: Offline update at feature time.
- Model 2: Dynamic Updating.

The first model (Model 0) represents the base case where updates are performed when a new version is released. The update in this model is performed offline so the service is stopped until the system finishes the updating process. The disadvantage here is that severe bugs will not be addressed in a timely manner.

Model	Revenue
Model 1	(+) time value of feature. (-)cost of updating. It depends on cost of disabling and restarting the service.
Model 2	(+) time value of feature. (-)cost of online updating, which depends on feature complexity. (-)cost of using a modified system that supports online updating.

Fig. 1. Value of Updating to Feature i

The costs and benefits of the last two update models are summarized in Figure 1. Model 1 presents the option for offline updating at feature availability time. In this model, updates are scheduled on the next system restart and applied when the system goes offline. Users are able to install these features instead of waiting for the next release date. Under this model, users will be required to restart their phones, which might cause users to delay applying the patch until a more suitable time.

Finally, in Model 2 the system is dynamically updated when new features are available even if availability occurs before the next release time. However, users might suffer from short-time performance loss during the update process.

In the last two models, we assume that the system is restarted when a new version is released. Models 1 and 2 allow developers to deploy features quickly rather than waiting for the new version release time. However, under these models, a restart is still required at each release.

2.2 Net Options Value Model

Net Options Value (NOV) model quantifies the value of using the system over a certain period of time. In other words, if the value is represented as a function of time, the total value is equal to the integration of the value function over the specified period.

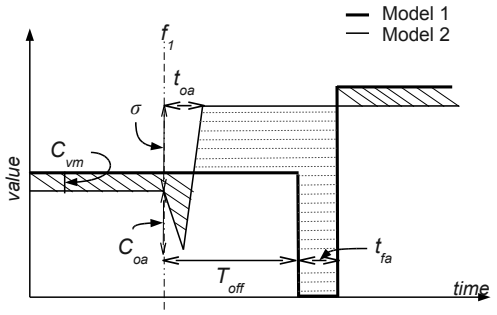

Fig. 2. Net Option Value of Different Updating Models.

To illustrate, consider the scenario in Figure 2. It shows the revenue generated by Model 1 (bold line) and Model 2. Each model's total revenue is equal to the area under its value function. Model 2 has less value initially due to the cost of supporting dynamic updates. However, Model 2 gains value by early adoption of feature and reduced cost of updating. The dip in the Model 2 value represent the cost of the updating process. For Model 1, the dip is more severe since it incurs complete service disruption. The area in the figure shaded by diagonal lines represents the gain achieved by offline over dynamic updating, while areas shaded by horizontal lines represents the gain of dynamic over offline updating. Intuitively, if the area of diagonally shaded region is larger than the horizontal region, then offline updating provides better total revenue and the cost of supporting dynamic updating does not justify its benefits.

In general, net options value [2] is represented as follows:

$$V = S + \sum_i NOV_i - C$$

$$NOV_i = V_i - C_i$$

where S is the base system's value (i.e. before applying new features), V is the net value of the model, C is the model cost, which is paid even if no updates were exercised. NOV_i is the value gained by updating to feature i and C_i is the cost of the update. This formula, although general, does not offer much insight into the specifics of a typical updating system. Thus, we seek a domain-specific formulation of the net-options value analysis starting with a quantitative treatment of the value of Model 1 and 2.

Model 0: Static Update at Release Time For this model the system value increases at release time by an amount equal to added features value. Thus we define the system value (V) for this model at a future release as:

$$V = S + \sum_i \sigma_i$$

where S is the system value at the current release and σ_i is the technical significance (value) of feature i. In other words, the value of the system after installing a new release is equal to its original value (old release value) plus the value of new features.

Model 1: Static Update at Feature Time For this model the system value increases at next restart time by an amount proportional to added features time value. The cost has two components. First, the cost of delaying the update. Second, the cost of restarting the service. Thus we define the system value (V) for this model at a future release time (t_i) until the new version is released T, as follows:

$$V = \sum_{i=1}^{n} NOV_i$$

$$NOV_i = E[U] \int_{t_i+T^i_{off}}^{T} \sigma_i(t)dt - C_R \tag{1}$$

$$C_R = \begin{cases} 0 & t_i + T^i_{off} = t_{i-1} + T^{i-1}_{off} \\ U_L \int_0^{t_{fa}} dt \overbrace{\sum_{j=1}^{i-1} \sigma_j(t_i)}^{*} & \text{otherwise} \end{cases} \tag{2}$$

$$\tag{3}$$

The value function we will use represent the value gained by a single user. It is often necessary to multiply the gained value by the expected number of users to obtain the total value. In the value model, $E[U]$ represents the expected number of users, U_L is the number of users at low-demand time. T_{off} is expected value of time until update is applied, and t_{fa} is the time needed to complete offline update. This value model has two parts. The first part describes how the deployment of a feature increases the system value. The value is equal to the summation of daily revenue of a feature represented by ($\sigma(t)$). The integration bounds represents the period of time the new feature is active. Since this model relies on scheduled restarts, the feature will not be deployed at its release time (t_i) but rather after certain number of days (T_{off}). The second part of the formula presents the cost associated with offline updating. The first case states that if two features are scheduled on the same restart period, we only need to pay the cost once. The second case presents the cost of the restart in terms of lost value (system value so far, labeled with (*)) and the time needed to finish the restart of the system after update (t_{fa}).

Model 2: Dynamic Updating For this model the system value increases at feature availability time by an amount proportional to added feature's time value. The cost has two components. First, the long-running cost of using the updating system. Second, the cost of performing the update. Thus we define the system value (V) for this model at a future release as:

$$V = E[U]C_{vm} \sum_{i=1}^{n} NOV_i$$

where C_{vm} is the ratio of the performance of a dynamic-updating system relative to an offline-updating system and ranges over the period $[0, 1]$, where having the value of one means that there are no long-running overhead. Thus, as $C_{vm} \rightsquigarrow 1$, the operating cost of the system with dynamic updating decreases. Like Model 1, $E[U]$ is the expected number of users. The value gained by Model 2 is offset by the cost of using the updating system.

The per-feature value (NOV_i) is defined as follows:

$$NOV_i = \int_{t_i}^{T} \sigma_i(t)dt - \int_{0}^{t_{oa}} C_{oa}(t)dt \sum_{j=1}^{i-1} \sigma_j(t_i) \tag{4}$$

where t_i is the time of release for feature i, T is the time of next release, $\sigma_i(t)$ the value function of the feature, t_{oa} is time needed to finish the dynamic update, and C_{oa} represents the reduction in system's value during the dynamic updating. Again, this value model represents the gain from deploying the feature (integration of $\sigma(t)$) minus the cost of the dynamic update which is related to update duration and value loss during the update.

2.3 Effect of Operational Parameters

Operational parameters are those used to describe the cost and timing of the update process. Based on the previous valuation models, we will now construct a set of relations that describes the bounds on these parameters that guarantees profitable operation. The original value models can be used to compare total revenue, while this set of relations can be used to calculate the system parameters based on known constraints.

Effect of Updating Overhead In our model, both update systems suffer a value loss during the update. However, the dynamic update system also pays the continuous cost of supporting dynamic updates (C_{vm}). The value of C_{vm} represents the performance overhead from using the dynamic update system. It is known that such overhead must be kept at minimum. However, the question is when does the overhead reverse any gains from the modified system.

In general, the relation between C_{vm} and gain in comparison to the other system can be modeled by equating equations (4) and (3). By assuming n dispersed features, dynamic updating has higher value when its value is higher than the value offered by static updating. After simplification, the effect of update overhead is presented in the following formula:

$$\sum_{i=1}^{n} \overbrace{[E[U](1 - C_{vm}) \int_{t_i}^{T} \sigma_i(t)dt + E[U]C_{vm} \int_{0}^{t_{oa}} C_{oa}(t)dt \sum_{j=1}^{i-1} \sigma_j(t_i)dt}^{\text{effect of dynamic update}}$$

$$\underbrace{-E[U] \int_{0}^{T_{off}^{i}} \sigma_i(t)dt - U_L \int_{0}^{t_{fa}} dt \sum_{j=1}^{i-1} \sigma_j(t_i)dt]}_{\text{effect of static update}} < 0$$

The first half represents the cost of the dynamic update system which consists of the long-running cost and the update cost. The second half shows the offline updating cost consisting of delayed feature deployment and service disruption at update time. Notice that as C_{vm} increases to reach the value of one (no long-running costs), the cost of dynamic update is reduced to the cost of the update process at update time. As C_{vm} decreases, the long running cost increases in a similar amount. Also, note that as either T_{off} or t_{fa} increases, lower values of C_{vm} can be tolerated.

Effect of Delayed Updates For two features f_i, f_j where $t_j > t_i$, applying the two features at t_j has higher value than applying each feature at its time for Model 1 if (here terms have their previously defined meanings):

$$U_L \int_0^{t_{fa}} dt \sum_{k=1}^{i-1} \sigma_k(t_i) > E[U] \int_{t_i+T_{off}^i}^{t_j+T_{off}^j} \sigma_i(t)dt$$

and for Model 2 if

$$\int_0^{t_{oa}} C_{oa}(t)dt \sum_{k=1}^{i-1} \sigma_k(t_i) > \int_{t_i}^{t_j} \sigma_i(t)dt$$

On the other hand, if $\sigma(t), C_{oa}(t)$ do not depend on time (i.e. constant values), these conditions are simplified to:

Model 1:

$$\sigma_i < \frac{U_L}{E[U]} \frac{t_{fa} \sum_{k=1}^{i-1} \sigma_k(t_i)}{(t_j + T_{off}^j) - (t_i + T_{off}^i)}$$

Model 2:

$$\sigma_i < \frac{t_{oa} C_{oa} \sum_{k=1}^{i-1} \sigma_k(t_i)}{t_j - t_i} \tag{5}$$

The later condition relates the value of σ_i to the cumulative system value, update cost and period between features. For example, under Model 2 (5), a feature that is equal to 10% of cumulative value and with a period of one week until next feature, the dynamic update time should be more than 8 hours and 24 min to justify combining the update of these two features.

Coverage of Dynamic Updating Many dynamic updating systems do not support all types of code updates. Therefore, even a dynamic update system requires occasional restarts to serve certain update requests. Generally, we can include this factor as a random event x_i that is related to the ratio of supported updates. Assuming that for any certain feature, there is a probability $p(x_i)$ that the feature can not be updated dynamically. Therefore, the valuation model of Model 2 is changed as follows:

$$NOV_i = p(x_i) NOV_i^1 \tag{6}$$

$$+ (1 - p(x_i|x_{i-1})) max\{NOV_i^1, NOV_i^2\} \tag{7}$$

$$+ (1 - p(x_i|\overline{x}_{i-1})) NOV_i^2 \tag{8}$$

$$\tag{9}$$

In the new model, the NOV of feature i has two factors. First, there is a probability of x_i that a static update is required (i.e. NOV_i^1). Second, if the feature can be applied dynamically, the NOV is the maximum of the dynamic and static NOV. We are using the maximum aggregate to cover the possibility that feature $i-1$ was updated statically (i.e. $p(x_i|x_{i-1})$) and that the new feature is released within the T_{off} period. In this case, we have the option of upgrading feature i dynamically at the regular cost or statically at reduced cost since the restart is already required. Otherwise, the regular NOV of dynamic update is used (i.e. $p(x_i|\overline{x}_{i-1})$)

3 Applying Our Analysis Model

This section applies our analysis model for comparing software updating schemes to Apache httpd [1], a well-known web server. The main objective is to study our model's

trends, relative values depending on the selected parameters, and assess its precision. The Apache httpd case we collected information about bug fixes over a five years period.

We will start by describing the process of selecting the evaluation parameters. Then we will present detailed information about the case study. Finally, we will study the effect of operating parameters on gains achieved by different updating models and how the timing of applying updates affect the system's value.

3.1 Selecting Analysis Parameters

The main challenge in the application of our model is selection of proper value functions. Each feature contributes to the value of a release and each bug reduces the system value until it is fixed. However, assigning proper values of $\sigma(t)$ is not trivial as it requires an understanding of the technical importance of a feature and how it affects the whole system's value. Here, we use a simple heuristic to evaluate a feature's importance. For evaluation purposes, we used a constant value for $\sigma(t)$. However, if more information is available regarding a feature effect on system's value, this information should be represented using a more appropriate function. The value of T_{off} equals the number of days until offline updates are performed (i.e. Sundays). The value of t_{oa} is approximated by αt_{fa}. The value of α depending on code modifications required to implement the feature and was computed by studying code changes. Finally, the value of C_{oa} is set to 0.5. This value indicates that the system loses half of its performance (which is very conservative) during the dynamic update process.

3.2 Xerces Case Study

We selected ten features from two consecutive releases of Xerces XML parsing library. The features provide additional capabilities (e.g. A3: Japanese characters serialization , B4: support for <redefine> attribute), performance enhancement (e.g. A4: improve Deterministic Finite Automaton(DFA) build-time performance) or resolve bugs (e.g. B1). Deploying these features allows the system to increase its revenue through faster processing, wider customer base and support additional types of XML documents. We approximated σ_i values for studied features through a point system. We assume that a system value ($\sum_i \sigma_i$) doubles at every release. Any security-related features is assigned four points, bug fixes and added features are assigned three points, performance enhancement are assigned two points, and finally, any remaining features are assigned one point. Using this approach, each feature's σ_i is equal to its share of points.

For simplicity, we are assuming that a release consists of these features only. Figure 3 shows the parameter values for selected features. The table shows the number of points assigned to each feature as points are used to approximate value gained by deploying a feature (σ). The table also shows the feature's relative complicity (α) as derived from code modification logs. Feature complexity is used to derive the dynamic updating time (t_{oa}). A complex feature requires more updating time than a simpler feature. The table also lists the time in days until the next release is available ($T - t$) and the wait period from the feature release time until the next Sunday (T_{off}) which we used as waiting period for the offline updating.

Feature	Points	σ	α	$T-t$	T_{off}
A1	3	0.214	0.185	54	2
A2	3	0.214	0.012	50	5
A3	3	0.214	0.235	48	3
A4	2	0.143	0.136	20	3
A5	3	0.214	0.432	13	3
B1	3	0.214	0.4	43	3
B2	3	0.214	0.36	42	2
B3	2	0.143	0.1	38	5
B4	3	0.214	0.08	28	2
B5	3	0.214	0.05	11	6

Fig. 3. Xerces Feature's Parameters

We can note that feature complexity follows the trend of feature's value for Xerces. In other words, important features are complex. Therefore, supporting a high-value feature comes at higher cost than a simpler feature, but will provide higher value.

3.3 Apache httpd Case Study

In this case study, we analyzed the history of bug fixes for Apache httpd for versions from 2.0 to 2.3. Figure 4 shows an overview of bugs timeline and their effect on system value. Each bug has a severity level that represents its effect on the system and is assigned by users reporting the bug. In the figure, the system value is decreased by the weight of the bug severity (Figure 5) starting from bug discovery date until it was fixed. In this study, bugs that can be fixed by changing configuration files rather than source code, and bugs that caused a crash at startup were excluded. These bugs require a restart in any update model and thus, will not be included in the analysis between dynamic and static updating. Furthermore, we excluded bugs with severity below normal. The reason behind excluding these bugs is that users do not usually update their servers for below normal bugs.

The value of each bug fix (W) is proportional to its severity. Since severity is selected by users reporting the bug, it reflects the value of the bug fix more accurately compared to value assigned by an external observer. However, it is less obvious how bug severity affects the system quantitatively. For httpd, we assigned different weights to different severity levels (Figure 5)[1]. To compute σ_i, the severity weight is multiplied by β, which is an estimate of value loss. For example, a packet monitoring service may be employed to record transactions affected by the bug. Such system can affect the overall throughput, and thus, reduces the system value (i.e. hits per day).

The value of a bug fix depends on its severity level (user supplied) and its estimated value loss (β). We later show the effect of β and bug severity on system values of different updating models.

3.4 Analysis

We will now apply our analysis model to evaluate the two update models (Model 1 and Model 2). We will study the difference and the effect of operational parameters on revenue.

[1] https://issues.apache.org/bugzilla/page.cgi?id=fields.html

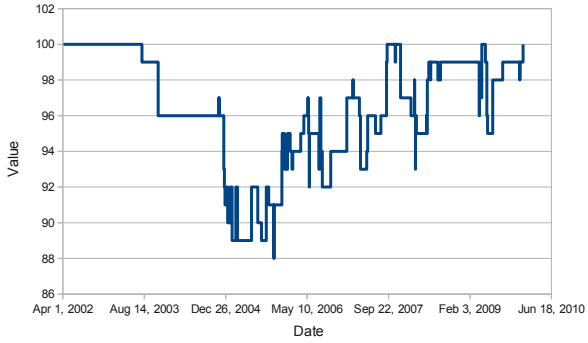

Fig. 4. History of httpd Value

Category	W	Description
Critical	3	crashes, loss of data, memory leak
Major	2	major loss of function
Normal	1	some loss of functionality under specific circumstances

Fig. 5. Apache httpd Bug Categories. Taken from ASF Bugzilla: A Bug's Life Cycle

Revenue Analysis Assuming $E[U] = U_L = 1$, Figure 6 presents the revenue values for Model 2 and Model 1. These values reflect the expected benefits for a single continues user. Note that increasing the number of expected users ($E[U]$) will increase the absolute revenue. However, it has minimum effect on the difference between Model 1 and Model 2 update systems. The main cause of increased value in Model 2 for Xerces is the wait period until restart required by Model 1. For Apache httpd, the wait period for receiving a bug fix is manifolds longer than that for the next scheduled restart. Therefore, in the case of httpd, the long running cost of Model 2 makes it less beneficial on the long run.

Cycle	Model 1	Model 2
Xerces 1.2.3-1.3.0	34.86	37.73
Xerces 1.3.0-1.3.1	28.43	31.64
httpd	2285.15	2269.66

Fig. 6. NOV Calculation when $E[U] = U_L = 1$ and t_{fa} = one min. $C_{vm} = 0.99$. $\beta = 0.05$

Effect of Updating Overhead Figure 7 shows the gain percentage from using Model 2 compared to Model 1 for the studied features from Xerces and bug fixes of Apache httpd. It shows that for Xerces, dynamic updating can provide benefit as long as its performance is above 90% of Model 1 performance. In other words, the long running costs of using dynamic updating for Xerces must not exceed 10% of the system revenue. If supporting the dynamic update system reduces a server's performance (e,g, satisfied

requests per second) by 10%, then this performance loss translates into lost customers, and thus a loss in revenue by 10%. Any gain from early adoption of features will be eliminated by the constant high cost of supporting dynamic updating. Apache httpd presents a different story. Even at highest C_{vm} ratio, there is no benefit from using Model 2. While as expected, the loss of value increases as C_{vm} decreases.

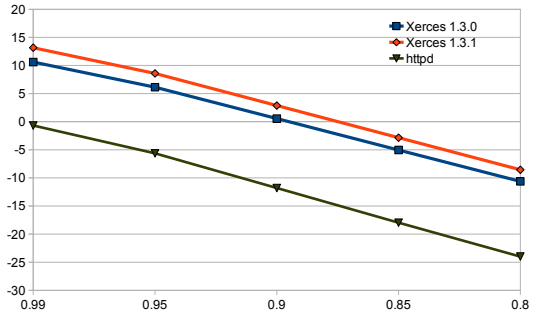

Fig. 7. Effect of C_{vm}. $E[U] = U_L = 1$ and $t_{fa} =$ one min.

Effect of Restart Schedule The main reason that Model 2 can generate better value compared to Model 1 is delayed updates in Model 1, which is related to T_{off}. This parameter represents the period of maintenance cycle in model 1. High value indicates longer periods without restarts and thus reduced cost due to service interruption. On the other hand, low values of T_{off} brings required updates at a faster rate. Figure 8 shows the relation between the value of T_{off} and the gain of model 2 compared to Model 1. At higher T_{off} values, the static update model losses most of its benefits and become closer to Model 0. Xerces case is very sensitive to varying the value of T_{off} due to the short period between features. As T_{off} increases, many features will be delayed. However, Apache httpd bug fixes are well-dispersed in time. Therefore, higher T_{off} barely affect Model 2 gain (from -1% to 1.5%).

With low T_{off} (i.e. daily restarts), the system will closely follow the value of Model 2. It is worthy to note that in all cases, we assumed that static updates occur on low demand times (i.e. restart cost multiplied by U_L rather than $E[U]$). In reality, this assumption may not hold for low values of T_{off}.

3.5 Summary

In this section, we investigated the value of different update models on a set of real-world applications (Xerces and Apache httpd). Our main objective was to study our model's trends and values depending on the selected parameters. Several key insights are worthy to note. First, the long running cost of dynamic updating has an influential role in determining the total revenue. Another observation is the relation between bug fix history and benefits from dynamic updating.

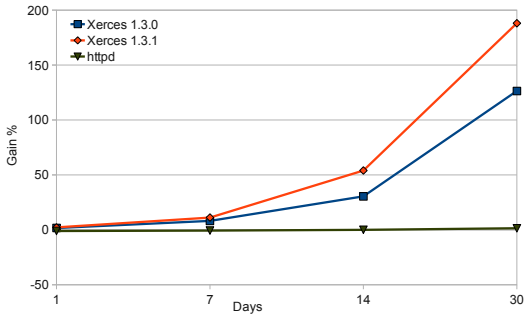

Fig. 8. Effect of t_{off}. $E[U] = U_L = 1$ and t_{fa} = one min. $C_{vm} = 0.99$

4 Related Work

Dynamic updating is gaining increased interest from research and industry. Several research projects have proposed, designed and implemented dynamic updating systems. However, the main evaluation tasks in the literature were performance and coverage. Chen *et al.* [6] and Subramanian *et al.* [26] evaluated their systems in terms of service disruptions during the update process. Evaluation of runtime aspect-weaving tools [7] have also focused on runtime overhead. In this paper, we explored a different evaluation goal and methods. To the best of our knowledge, this is the first exposition into evaluating update systems in terms of running costs and added options value.

Our evaluation model is based on the NOV analysis [5, 12]. NOV analysis is based on the problem of pricing financial options. A financial option presents the opportunity to purchase a commodity at a strike price in the future regardless of price fluctuations, provided that the buyer pays a premium in the present (also known as Call Option). In this paper we used the basics of options analysis to evaluate the benefits of dynamic updating. Updating has a significant resemblance with the problem of option pricing. As options, dynamic updating provides the opportunity to perform a future update at a possibly reduced price given that a premium (i.e. cost of using the dynamic update system) is paid. The body of literature describing this financial instruments is extensive and out of the scope of this paper. However, we note the application of options to software design and especially to design modularity. Baldwin and Clark [2] showed the benefits of modular design in increasing a system's value. They conclude that a set of options over modules are more valuable than options on the whole system. This idea is further utilized in software design research by analyzing which modularization provides the best value. Sullivan *et al.* [27] show the value of design based on information hiding principles by combining NOV analysis and design information.

Similar uses of option analysis can be found in [4, 11, 15]. Our work shares the basic analysis techniques since the problem of quantifying updating benefits can be translated into a modular design evaluation problem (i.e. Updatable systems are modular). However, the case for software updating presents a different set of operational parameters and dependencies on time that are not considered for option analysis for software design. Ji *et al.* [10] used option analysis to evaluate the benefits from designing and issu-

ing new software releases in relation to market uncertainty. Their analysis is concerned with the software developer perspective and analyze the preferred market conditions for releasing an upgrade (additional features). In this paper, we were mainly concerned with how to decide between different upgrading policies. In contrast, our analysis assists system users and updatable systems designers, rather than feature providers, with deriving decisions related to upgrading policies.

The problem of designing dynamically updatable systems has also received considerable attention in the last decade. Oreizy *et al.* [21, 22, 24] and Garlan *et al.* [8] have presented and studied dynamic software architectures. These systems were evaluated based on the performance of resulting application and other code metrics. The model presented in this paper can be applied to evaluate different online updating schemes including those presented by Oreizy *et al.* and Garlan *et al.*. Evaluating dynamic deployment architectures were also presented Mikic-Rakic in her PhD thesis [17], where the goal was to reduce service disruption (i.e. increase value) in distributed systems through better deployment strategies. In this paper, we are not concerned with enhancing a specific updating system, but rather on providing a mechanism to evaluate and compare their benefits.

5 Discussion

We have illustrated how our proposed model can be used to evaluate updating systems and to understand the effect of some operational parameters. This evaluation model is advantageous since it accounts for the value of time and supports the study of time dependent value functions. In our evaluation (Section 3), we treated the feature value function as a constant. In general, assigning values to features is often subjective. However, it would be of interest to study value functions that directly depend on time. For example, functions that model compound interest on feature's value.

We assumed that each applied update is correct and does not fail (i.e. bug-free). This assumption simplifies the formulation. However, a more practical model will incorporate the possibility of failed updates. A failed update can be considered as a feature with negative gain to model value loss during the use of the malfunctioning code. Since failures are unknown before their occurrence, this additional negative-gain feature will depend on a probability distribution that describes bug probability over time. The issue of failed updates have been extensively studied by Mokous and Weiss [18].

Finally, this study evaluated two update models, static(offline) and dynamic. An interesting question is to try to evaluate a combination of several dynamic update schemes depending on the nature of the feature and how they compare in provided value.

6 Conclusions and Future Work

Software updating has several advantages such as runtime monitoring, bug fixes or adding features to long running applications. Therefore, dynamic software updating has attracted significant interest in the last few years [6, 19, 20, 23, 25]. To date, dynamic updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. For example, Chen *et al.* [6] evaluated

their system in terms of service disruptions during the update process and noted the types of code changes that their system can not handle. Such evaluation is sufficient to understand the system performance and coverage. However, we often need other metrics to compare different updating systems. For example, what would be the gain from dynamic updating over offline updating, or what is the gain difference between two dynamic updating systems. To answer these questions, we formalized a quantitative model to evaluate the net revenue gained by the use of different updating models. Using this model, we were able to evaluate the gain from online updating vs. offline updating based on the evolution history of real-world applications. Furthermore, the model can also be used to compare two, updating schemes that differ in their coverage and performance.

An interesting outcome of this analysis was an insight into the perceived value of performance overheads for dynamic update systems. Generally, researchers have been concerned about two kinds of such overheads [7]: first, during update time, and second, constant overhead during the system's normal execution. Our analysis provides a method to analyze and compare these overheads based on their perceived values, which has the potential to aid in the selection of an updating system during software design.

Future work involves extending our analysis model in two main directions. First, the formulation can be extended to model the effect of bug discovery. Often after a feature release a bug is discovered and a second patch is needed to resolve the bug. The extension can model the revenue loss from such activity. Second, in terms of evaluation, we used simple constants to represent feature values. However, modeling real-world economics would require more complex valuation functions.

References

1. Apache httpd. http://httpd.apache.org/.
2. C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, 1999.
3. K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *the Conference on The Future of Software Engineering*, pages 73–87, 2000.
4. Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, U. of Virginia, 2006.
5. A. D. Chandler. *Strategy and Structure*. MIT Press, Cambridge, MA., 1962.
6. H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, 2007.
7. R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08*, 2008.
8. D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, 2004.
9. B. Gharaibeh, H. Rajan, and J. M. Chang. A quantitative cost/benefit analysis for dynamic updating. Technical report, Iowa State University, 2009.
10. Y. Ji, V. Mookerjee, and S. Radhakrishnan. Real options and software upgrades: An economic analysis. In *International Conf. on Information Systems (ICIS)*, pages 697–704, 2002.
11. K. Sullivan *et al.*. Modular aspect-oriented design with XPIs. *ACM TOSEM*, 2009.
12. S. Klepper. Entry, exit, growth and innovation over the product life cycle. *American Economic Review*, 86(30):562–583, 1996.
13. G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP*, 1999.

14. M. Lehman. Software's future: managing evolution. *Software, IEEE*, 15(1):40–44, Jan/Feb 1998.
15. C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, 2005.
16. K. Mätzel and P. Schnorf. Dynamic component adaptation. Technical Report 97-6-1, Union Bank of Swizerland, 1997.
17. M. Mikic-Rakic. *Software architectural support for disconnected operation in distributed environments*. PhD thesis, University of Southern California, 2004.
18. A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.
19. I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44(6):13–24, 2009.
20. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *PLDI*, 2006.
21. P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, 2008.
22. P. Oreizy and R. Taylor. On the role of software architectures in runtime system reconfiguration. In *Intl. Conf. on Configurable Distributed Systems,*, 1998.
23. A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of Java software. 2002.
24. P. Oreizy *et al.*. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.
25. S. Malabarba *et al.*. Runtime support for type-safe dynamic java classes. In *ECOOP '00*, pages 337–361, 2000.
26. S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *PLDI*, pages 1–12, 2009.
27. K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE '01*, pages 99–108, 2001.
28. O. E. Williamson. *The Economic Institutions of Capitalism*. Free Press, New York, NY, 1985.
29. Xerces. XML library: http://xerces.apache.org/xerces-j/.