9-6-2009

A Quantitative Cost/Benefit Analysis for Dynamic Updating

Bashar Gharaibeh
Iowa State University

Hridesh Rajan
Iowa State University

J. Morris Chang
Iowa State University

A Quantitative Cost/Benefit Analysis for Dynamic Updating

Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang

TR #09-27
Initial Submission: September 6, 2009.

Keywords: Dynamic Updating, Cost Benefits Analysis, Software Evolution
CR Categories: D.2.8 [*Software Engineering*] Metrics D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement D.2.3 [*Software Engineering*] Coding Tools and Techniques D.2.13 [*Software Engineering*] Reusable Software

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Quantitative Cost/Benefit Analysis for Dynamic Updating

Bashar Gharaibeh
Electrical & Computer Eng.
Iowa State University
Ames, IA
bashar@iastate.edu

Hridesh Rajan
Computer Science
Iowa State University
Ames, IA
hridesh@iastate.edu

J. Morris Chang
Electrical & Computer Eng.
Iowa State University
Ames, IA
morris@iastate.edu

ABSTRACT

Dynamic software updating provides many benefits, e.g. in runtime monitoring, runtime adaptation to fix bugs in long running applications, etc. Although it has several advantages, no quantitative analysis of its costs and revenue are available to show its benefits or limitations especially in comparison with other software updating schemes.

To address this limitation in evaluating software updating schemes, we contribute a quantitative cost/benefit analysis based on net option-value model, which stems from the analysis of financial options. Our model expresses the relation between added value and paid cost in mathematical forms. We have used this model to evaluate the revenue from dynamic updating in two case studies featuring Xerces and MobileMedia. These studies reveal the set of parameter values that render dynamic updating effective. We also compared two previously published dynamic updating schemes and observed how the perceived performance and coverage of different updating systems affects their relative gain.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Measurement, Performance, Theory

Keywords

Dynamic Updating, Cost Benefits Analysis, Software Evolution

1. INTRODUCTION

Software evolution and maintenance is a fact of life [6, 22]. 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, automatic teller machines (ATMs), critical control systems often need to balance evolution and availability requirements. As Malabarba *et al.* state, "for a large class of critical applications, such as business transaction systems, telephone switching systems and emergency response systems, the interruption poses an unacceptable loss of availability [24]". Stringent availability requirements for such systems dictate minimal downtime and degradation in performance, whereas evolution needs often translate into update and restarting of such systems. As an example, consider the maintenance needs faced by European banks while updating ATMs from national currencies to Euro [21, 26]. The 24-hour service typical for ATMs dictates constant availability, whereas the maintenance needs to convert currencies required immediate software update. Often such maintenance needs are critical and unanticipated [21]. Dynamic software updating helps address such software evolution needs.

Dynamic software updating has attracted significant interest in the last few years [3, 10, 27, 33]. 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 [17]). 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, Subramanian *et al.* [33] 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. Their analysis also included a description of supported version changes for these server applications. Similarly, Chen *et al.* [10] and Gharaibeh *et al.* [14] evaluated their systems in terms of service disruptions caused by 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 and other dynamic updating systems. Furthermore, we need to understand the long-term effects of using dynamic updating and what circumstances may limit its usefulness in long running software systems.

The contribution of this work is a quantitative value model that allows us to study the gain form updating systems. Our model is based on Net option-value (NOV) analysis [36]. 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 [5, 23, 34]. 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 benefits and

Submitted to ICSE 2010.

costs of offline and dynamic updating in software systems.

We illustrate our model through a software evolution scenario taken from the version history of Google's Android mobile platform [1]. This example is used to explain the logic behind our model. For further evaluation, we have applied it to two case studies: the evolution of the XML parser library Xerces [37] and 7 feature releases for a software product-line application called MobileMedia [38] (Section 4). Using these case-studies, we evaluated the dynamic update model and explored how operational parameters affect the gain from dynamic updating in comparison to offline updating. Furthermore, we used our value model to compare the revenue of two, previously published, different dynamic updating systems. Our evaluation reveals the important role of dynamic updating overhead in determining the net revenue of the system. We have found that for the studied applications, any dynamic updating system should not cause more than 10% degradation on revenue. Otherwise, the value gained by timely updates where lost due to the constant cost of supporting the update process. We also observed how the perceived performance and coverage of dynamic updating systems affects their relative gain.

To summarize, our contributions in this paper are:

- A quantitative model for cost/benefit analysis of updating systems and its formulation. The model targets the revenue generated by the update systems.
- A case study from software evolution of real-world applications that illustrates the use of the proposed evaluation model.
- We performed a quantitative analysis between two previously published dynamic updating systems in terms of provided revenue. The two schemes differ in the type of supported evolutionary changes and performance.

The reminder of this paper is organized as follows. Section 2 presents a gentle introduction to dynamic updating. In Section 3 we discuss our quantitative model in parallel with a running example. We describe the two case studies in Section 4. Section 5 presents the related work while Section 6 discuss various aspects and limitations of our evaluation model. Section 7 discusses directions for future investigations and concludes.

2. DYNAMIC UPDATING

The typical process of updating a software system includes restarting the application in order for changes to take effect. However, it is argued that restarts, and hence interruption of service, are highly undesirable and can cause revenue loss [29, 30]. Therefore, a solution that avoids service interruption might guard against revenue loss.

Dynamic updating refers to the process at which software applications are modified amid execution. In this scenario, an update module takes the code changes and applies them to the running application structure (e.g. methods, objects, etc.). Several updating systems have been proposed targeting the theory [15, 32] and implementation [10, 14, 16, 24, 28] of dynamic updating systems. For example, Ginseng [28] is a dynamic software update tool for C where programs are compiled specially so that they can be dynamically patched. POLUS [10] also updates C programs. Hjalmtysson *et al.* [16] introduced a method for updating C++ code in running programs by using dynamic classes, while [24] supports dynamic updates by extending the Java class loader functionality. Also, Sun's Java Virtual Machine(Hotspot) supports dynamic redefinition of classes through its HotSwap capabilities [11]. Furthermore, variations of dynamic updating exists in the form of runtime aspects weaving for aspect-oriented programming [4, 12, 19, 31].

Supporting dynamic updating capabilities requires paying the associated costs. The process of modifying the running application affects its performance during the update period. For example, updating in POLUS [10] and Ginseng [28] reduces the performance by 30% for a the short duration of the update(order of milliseconds). Also, the updater module might cause long term performance loss in the system, which translates to revenue loss. Furthermore, not all code changes can be applied dynamically. Dynamic updating literature evaluate their systems in terms of the mentioned limitations. In other words, what is the performance loss and what type of updates are supported.

3. QUANTIFYING SOFTWARE UPDATE

This section presents the details about the proposed value model and illustrates its use through a running example. The main idea behind the value 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.

We first present a software update scenario taken from the version history of Google's Android platform, which is used throughout this section. We then present a generic value model for software update and its instantiation for three different update models. This allows us to quantitatively represent the gains from each model in terms of its operating parameters. We then use this formulation to compare and contrast the gain that each model can provide for our Android update scenario.

3.1 Updating Android Platform

We will illustrate how our proposed value model works through a simple, yet realistic example. The example is based on the Android mobile platform from Google [1]. Android is an open-source platform for operating smart phones. The platform builds on a Linux kernel as its base and uses a special virtual machine (Dalvik VM) to execute the system and user applications. Most of the system is composed of Java applications specially compiled to execute on top of the Dalvik virtual machine. Applications include web browser, calendar, chat clients and others. Android was first released (V 1.0) on September 2008 and a new release (V 1.5) followed on June 2009. We will study two patches to Android's code base that affected the value of the system.

3.1.1 Scenario I: Improving Security

The first code change was a security patch. A security vulnerability was identified on October 20th 2008. The vulnerability was caused by the use of an outdated package for the browser application. A patch was released on November 1st and users were notified of the availability of software update. The update process requires phone restart and rendered the phone unusable, not even for emergency calls, during the update process. More information about this vulnerability can be found in [2].

3.1.2 Scenario II: Performance Enhancement

The second feature is a performance enhancement to some of the platform application's user interfaces. The patch enhanced the speed of scrolling in the browser and Gmail lists and shortened the camera start-up and image capture times. This patch was not sent to users but incorporated into the next release (1.5). We estimate that these features were added to the code base around April 2009. This feature release scenario presents an interesting question. Users expect their phones to have high availability and performance. Does

the benefits of having a faster phone and less user interruptions justifies the cost of supporting dynamic updating in Android? The main goal of this paper is to formulate this question and to quantitatively compare different updating approaches.

In the next few sections we will define different updating systems precisely and formulate their associated costs and benefits.

3.2 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 where service is stopped until the system finishes the updating process. For example, in the case of the two features for Android, updates will be delivered whenever a new release of the platform is available. Users will install the new release and restart their phones. The disadvantage here is that severe bugs will not be addressed in a timely manner. For example, the Android security patch will be delivered to users as part of the new release, not as an important update.

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.

Figure 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. In the case of the two features for Android, features will be released whenever they are available rather than at the next available release. Users are able to install these features instead of waiting the next release date. Also, 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. In the Android example, when a new patch is available, the system will incorporate the needed change without restarting the phone or blocking users. However, users might suffer from short-time performance loss during the update process.

3.3 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. Let us consider the scenario shown 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 represents complete service disruption. The area on 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.

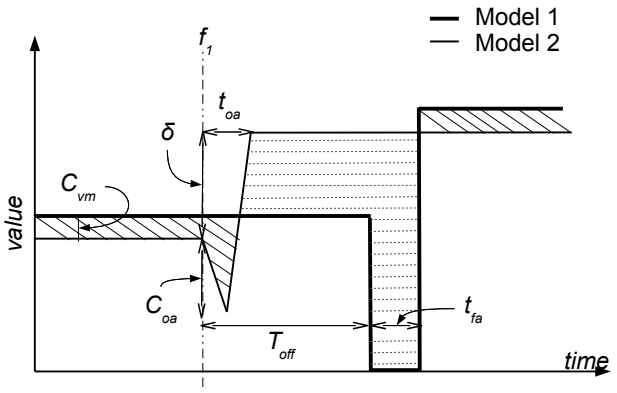

Figure 2: Value of different updating models.

In general, net options value [5] is represented as follows:

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

$$NOV_i = V_i - C_i$$

where 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 described previously.

3.3.1 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.

3.3.2 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 as follows:

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

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

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

(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 above 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 equals the summation of daily revenue of a feature which is 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}).

3.3.3 Model 2: Dynamic Updating

For this model the system value increases at feature availability time by an amount proportional to added features 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 cost of using a modified system that supports dynamic updating and ranges over the period $[0, 1]$, where having the value of one means that there are no long-running overhead. $E[U]$ 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:

$$NOV_i = \int_{t_i}^{T} \sigma_i(t)dt - \int_0^{t_{oa}} C_{oa}(t)dt \sum_{j=1}^{i-1} \sigma_j(t_i) \quad (4)$$

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.

3.4 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.

3.4.1 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 will dispersed features , we conclude that dynamic updating has higher value when:

$$\sum_{i=1}^{n} \overbrace{[E[U](1 - C_{vm}) \int_{t_i}^{T} \sigma_i(t)dt + E[U]C_{vm} \int_0^{t_{oa}} C_{oa}(t)dt \sum_{j=1}^{i-1} \sigma_j(t_i)dt}^{\text{cost of dynamic update}}$$

$$\underbrace{-E[U] \int_0^{T^i_{off}} \sigma_i(t)dt - U_L \int_0^{t_{fa}} dt \sum_{j=1}^{i-1} \sigma_j(t_i)dt]}_{\text{cost of static update}} < 0$$

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.

3.4.2 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

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

and for Model 2 if

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

where terms have their previously defined meanings.

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:

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

Model 2:

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

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 two features.

3.4.3 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:

$$\begin{aligned} NOV_i =& p(x_i)NOV_i^1 && (6)\\ &+ (1-p(x_i|x_{i-1}))max\{NOV_i^1, NOV_i^2\} && (7)\\ &+ (1-p(x_i|\overline{x}_{i-1}))NOV_i^2 && (8)\\ & && (9)\end{aligned}$$

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. $p(x_i|\overline{x}_{i-1})$)

3.5 Android Update Revisited

We will now apply the analysis model to our running example. To that end, the first task is to identify the value function of each feature ($\sigma(t)$). The security-related patch was released on day 60 and resolved a critical security vulnerability that could compromise the system. If the patch is not applied, the system looses value on a daily bases. The loss is proportional to the probability of an attack which grows every day. The loss can be modeled as a exponential random distribution function according to the following formula:

$$\sigma_1(t) = \begin{cases} -\lambda e^{-\lambda t} & t_d < t_1 \\ 0 & t > t_1 \end{cases}$$

Where λ is the probability of an attack and is set to 0.3. t_d is the vulnerability discovery time. In our case, $t_d = 50$ days.

The second feature enhanced the user-perceived performance by increasing response speed of certain applications. In reality, modeling the gain from such feature is hard since it requires knowledge about the market and how this feature can affect potential number of users or their satisfaction. For simplicity, we will assume that this feature increases user satisfaction by 15%. Therefore, the gain can be expressed as follows:

$$\sigma_2(t) = 0.15R$$

Where R is the previous system value, which depends on gain from the previous feature ($\sigma_1(t_2)$).

Finally, we will assume that the system has an initial revenue of \$100 per day. Other system parameters are known or there value can be assumed and are presented in Figure 4. We assume a low-overhead (C_{vm}) dynamic updating scheme with performance close to 99.5% of base system. As for time parameters, we assume that restarts can be initiated four days after feature release (T_{off}) and that a restart cycle (t_{fa}) takes two minuets to complete, while the dynamic update system requires five seconds (t_{oa}) to deploy the feature and causes 50% performance loss during the update (C_{oa}). The phone has a single continuous user ($E[U] = U_L$).

Using our analysis, let us evaluate the total gain for Model 1:

$$\begin{aligned} V =& S + NOV_1 + NOV_2 \\ NOV_1 =& 100 * \int_{14}^{210} \lambda e^{-\lambda t} dt - 100 * \int_{0}^{14} \lambda e^{-\lambda t} dt \\ & - \int_0^{0.0014} .dt * 100 \\ NOV_2 =& \int_{210+4}^{270} 100 * G_1(213) * 0.15 dt \\ & - \int_0^{0.0014} .dt * 100 * G_1(213) \\ V =& 23,619.13\$ \end{aligned}$$

The total gain for Model 2 is:

$$\begin{aligned} V =& (S + NOV_1 + NOV_2) * C_{vm} \\ NOV_2 =& 100 * \int_{10}^{210} \lambda e^{-\lambda t} dt - 100 * \int_{0}^{10} \lambda e^{-\lambda t} dt \\ & - 0.5 \int_0^{0.00006} .dt * 100 \\ NOV_1 =& \int_{210}^{270} 100 * G_1(210) * 0.15 dt \\ & - 0.5 \int_0^{0.00006} .dt * 100 * G_1(210) \\ V =& 23,794.17\$ \end{aligned}$$

Figure 3 shows the daily system revenue. Note that Model 2 gains from early adoption of features while loosing small amounts of daily revenue. In this example, the total revenue from dynamic updating was slightly higher than revenue from offline updating. Although not shown here, but increasing the value of attack probability (λ) increases the gain from using dynamic updating.

4. EVALUATION

This section presents an evaluation of dynamic updating using our formulation. We analysed two cases of software evolution. The chosen cases are Xerces [37] and MobileMedia [13, 38]. Xerces is an XML parsing library that can be found, among many places, in web server applications. MobileMedia on the other hand is a mobile application used to manage images, audio and videos and is similar in nature to the Google's Android platform that we discussed in previous section.

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

4.1 Selecting Analysis Parameters

The main challenge in this section is the selection of proper value functions. Each feature contributes to the value of a release. How-

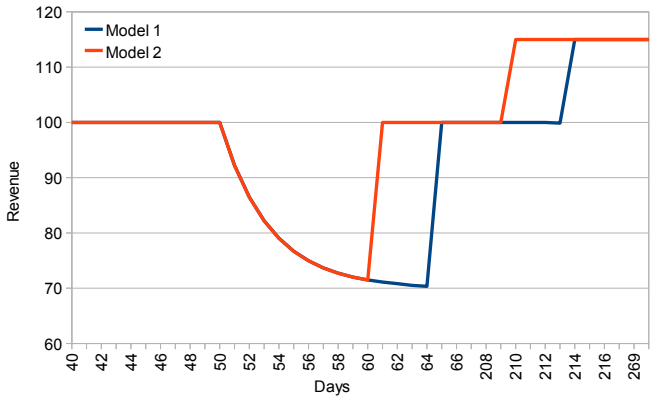

Figure 3: Comparison of daily revenue for the two updating models

Parameter	Value	Parameter	Value
$C_{vm}*$	0.999	$T_{off}*$	4 days
$t_{oa}*$	5 sec.	$t_{fa}*$	2 min.
C_{oa}	0.3	$E[U], U_L$	1
t_1	day 60	t_2	day 210
T	270 days		

Figure 4: System parameters. Values of parameters with * are based on the expected behavior of update systems

ever, 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. Therefore, we employed a simple heuristic to evaluate a feature's importance. For evaluation purposes, we used a constant value for $\sigma(t)$. First, we need to set the limits on σ values. We assume that a system value doubles at every release. Thus, by taking the formula of Model 0 we have:

$$2S = S + \sum_i \sigma_i \Rightarrow S = \sum_i \sigma_i$$

The equation can be further simplified if we start with normalized values (i.e. $V = 2, S = 1$). In this case we have:

$$\sum_i \sigma_i = 1$$

We approximated σ_i values for studied features through a point system. 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.

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.

4.2 Xerces Case Study

We selected ten features from two constitutive releases of Xerces XML parsing library. The features are described in Figure 5. 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.

A0	Dec 6, 2000	Start of V 1.2.3
A1	Dec 8, 2000	Upgraded schema support to the schema CR drafts at a similar level to that which had existed for the WD schema specifications.
A2	Dec 12, 2000	Fix for NullPointerException caused by deferred DOM implementation when in non-validating mode and there are multiple IDs declared on the same element.
A3	Dec 14, 2000	Applied patch from TAMURA Kent: (Japanese characters serialization)
A4	Jan 11, 2001	Updates to DFAContentModel to improve DFA build-time performance.
A5	Jan 18, 2001	Schema Identity Constraints
B0	Jan 31, 2001	start of 1.3.0
B1	Feb 1, 2001	Some massive bug fixes. All subtle but very important.
B2	Feb 2, 2001	Implementation of parsing component (javax.xml.parsers) of JAXP 1.1
B3	Feb 6, 2001	XML Node Normalization
B4	Feb 16, 2001	Initial support for <redefine>.
B5	Mar 5, 2001	XML Notations
C0	Mar 16, 2001	start of 1.3.1

Figure 5: Xerces Features Selected for Analysis: This Information was Obtained by Mining commit logs

For simplicity, we are assuming that a release consists of these features only. Figure 6 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 used as waiting period for the offline updating system.

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

Figure 6: 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.

4.3 MobileMedia Case Study

MobileMedia [13] is an extension of the MobilePhoto [38] application, which was developed to study the effect of aspect-oriented designs on software product lines (SPL). MobileMedia is an SPL for applications that manipulate photos, music and videos

on mobile devices. MobileMedia extends MobilePhoto to add new mandatory, optional and alternative features.

Release	Description	Type of Change
R1	MobilePhoto core [29, 30]	
R2	Exception handling included (in the AspectJ version, exception handling was implemented according to [13])	Inclusion of non-functional concern
R3	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label	Inclusion of optional and mandatory features
R4	New feature added to allow users to specify and view their favourite photos.	Inclusion of optional feature
R5	New feature added to allow users to keep multiple copies of photos	Inclusion of optional feature
R6	New feature added to send photo to other users by SMS	Inclusion of optional feature
R7	New feature added to store, play, and organise music. The management of photo (e.g. create, delete and label) was turned into an alternative feature. All extended functionalities (e.g. sorting, favourites and SMS transfer) were also provided	Changing of one mandatory feature into two alternatives
R8	New feature added to manage videos	Inclusion of alternative feature

Figure 7: Summary of Change Scenarios in the MobileMedia SPL (based on Figueiredo *et al.*'s work [13, Tab.1])

There are a total of seven releases and descriptions of each is shown in Figure 7. For example in release 7 (R7), the optional feature added in a previous release to manage photos was turned into an alternative feature and a new feature to manage music was added.

Feature	Type	Points	σ	α	$T-t$	T_{off}
R2	Mandatory	3	0.214	0.14	23	3
R3	Mandatory	3	0.214	0.1	21	1
R4	Optional	2	0.143	0.04	21	1
R5	Optional	2	0.143	0.11	18	5
R6	Optional	2	0.143	0.11	15	2
R7	Alternative	1	0.071	0.3	0	1
R8	Alternative	1	0.071	0.2	0	1

Figure 8: MobileMedia Feature's Parameters

Figure 8 shows the parameter values for these features. The table lists the same parameters as in Figure 6. Mainly the feature value (σ), complexity(α), time until next release($T-t$) and time until the next Sunday(T_{off}). Contrary to the Xerces case study, less important features (Alternative features) has higher complexity. Therefore, the cost of deploying these features may not justify the gain in the case of this application.

4.4 Analysis

We will now employ our valuation model to evaluate the two update models (Model 1 and Model 2) over the described case studies. We will study the gain in revenue from using Model 2 and the effect of operational parameters on revenue.

4.4.1 Revenue Analysis

Assuming $E[U] = U_L = 1$, Figure 9 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 is the wait period until restart required by Model 1. The cost of waiting increases linearly in relation to the expected number of users. Also, the gain increases in a similar linear fashion. Therefore, the net gain difference does not show large changes when the expected number of users change.

Cycle	Scheme	Revenue
Xerces 1.2.3-1.3.0	Model 2	37.73
	Model 1	34.86
Xerces 1.3.0-1.3.1	Model 2	31.64
	Model 1	28.43
MobileMedia	Model 2	16.87
	Model 1	15.06

Figure 9: NOV Calculation when $E[U] = U_L = 1$ and $t_{fa} =$ one min. $C_{vm} = 0.99$

In all cases, Model 2 update system provides higher revenue than Model 1. The higher revenue from the first Xerces release is due to the long release period. The first release (from 1.2.3 - 1.3.0) was 56 days compared to 44 days for second Xerces release and the short 25 days for MobileMedia. The longer release duration increases the gain from early adoption of features and reduces the cost of waiting in the case of Model 1.

4.4.2 Effect of Updating Overhead

Figure 10 shows the gain percentage from using Model 2 compared to Model 1 for the studied features from Xerces and MobileMedia. It shows that dynamic updating can provide benefit as long as its performance is above 90% of Model 1 performance. In other words, for the studied applications, the long running costs of using dynamic updating must not exceed 10% of the system revenue. For example, 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. Note that this limit (i.e. 10%) is specific to this study and not a general limit. Other update scenarios might show less or more tolerance to the constant overhead of dynamic updating.

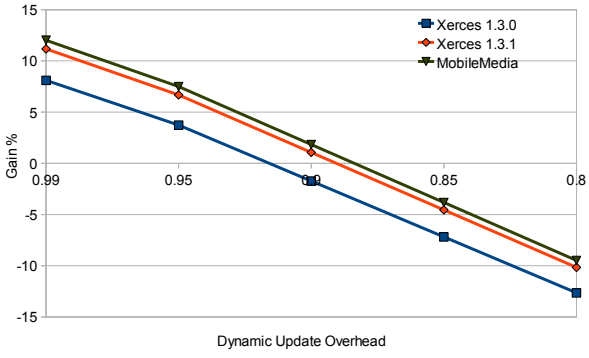

Figure 10: Effect of C_{vm}. $E[U] = U_L = 1$ and $t_{fa} =$ one min.

What this example shows, however, is that our analysis model can be applied to a real world dynamic update scenario to determine the suitability of a candidate update model.

4.4.3 Effect of Restart Schedule

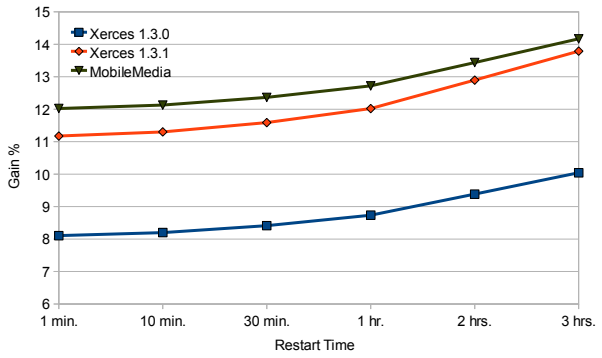

Figure 11: Effect of t_{fa}. $E[U] = U_L = 1$ and $C_{vm} = 0.99$

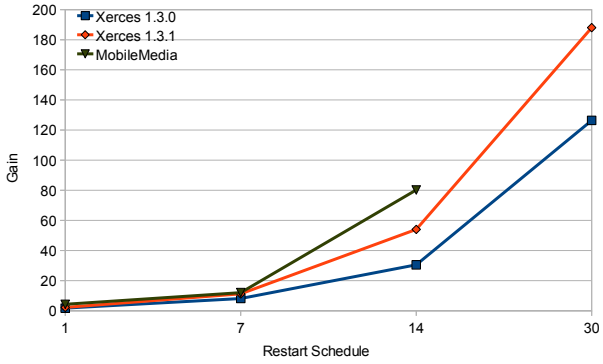

Figure 12: Effect of t_{off}. $E[U] = U_L = 1$ and t_{fa} = one min. $C_{vm} = 0.99$

As seen so far, Model 2 has better value than Model 1. The main reason is the 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 12 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. This is especially true for MobileMedia, where any value of $T_{off} > 25$ makes it behave as Model 0. 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}.

4.5 Comparison of Updating Methods

Until now we have been comparing dynamic updating with the typical offline updating scheme. Now we take the analysis to a different direction. We would like to understand how dynamic updating schemes compare in terms of provided revenue. Previously proposed updating system varies in terms of supported code changes and how these changes are expressed. We will compare between two updating systems. The key objective of the first system is to support dynamic deployment and removal of features expressed via language mechanisms provided to the programmer [12]. The other system is designed specifically for dynamic updating and it uses differences in class structure [33]. This comparison is based on features from Figueiredo *et al.* [13], which describes code changes as aspect-oriented and object-oriented implementations.

Scheme	Revenue	Scheme	Revenue
Updater#2	17.04	Update#1-Weekend	15.97
Updater#1-Daily	16.64	Updater#1 Hypothetical	16.78

Figure 13: Revenue For Different Dynamic Updating Schemes

4.5.1 *Updater #1: Low Constant Runtime Overhead and Negligible Update Overhead*

This update system requires code changes to be described as part of the program. Furthermore, the updater deals with these changes by dynamically weaving them into the running application. An example of such system is presented in [12] from which we will deduce the parameters needed for the analysis.

The research prototype for the selected update system does not support all types of aspects. It only supports the pointcut-advice model [25] at this time, so inter-type declaration can not be processed by the system. Out of the seven releases of MobileMedia, the first release uses inter-type declaration to add new exceptions to the application. As a consequence, this updater will not be able to process the first feature dynamically. Other features are supported by this update system. The updating system used here has a long running cost of 1.5% while update cost is *negligible*.

4.5.2 *Update #2: Negligible Runtime Overhead and Large Update Overhead*

The second update system, designed specifically for dynamic updates, relies on class transformations to change the application structure. Changes are presented in terms of new methods and fields informations. We will consider the system described in [33]. All code changes in the MobileMedia case study are supported by the update system. The system suffers from performance degradation during the update process. We will use the values presented in [33] although they were a result of a different benchmark. The update process requires approximately 15 seconds to complete and the performance is reduced by an average of 28%. The system presented does not show any long-term performance loss.

4.5.3 *Analysis*

Figure 13 shows the revenue of the two update models. The updater#1 has the highest value due to its ability to support all needed changes. The updater#2 inability to support dynamic update of first feature reduces its revenue. This loss is reduced if offline updates are allowed in a daily bases (third row) rather on a weekly bases (second row). The last row shows the revenue assuming that all features can be dynamically applied using updater#2. It shows the effect of long term overhead associated with updater#2.

This analysis reflects on the importance of completeness for software updating systems. Furthermore, it identifies that for most update scenarios it appears to be beneficial to optimize long term overhead from the perspective of dynamic software updating.

4.6 Summary

In this section we investigated the value of different update models on a set of real-world applications. Several key insights are worthy to note. First, the long running cost of supporting dynamic updating has an influential role in determining the net total revenue gained from dynamic updating. In our experiments, the system reached zero gain when the overhead was 10% of the base system performance. Another key observation is the importance of

recognizing the update system performance and capabilities when deciding to support dynamic updating. In our last case study of two dynamic update schemes, We noted how the coverage and performance characteristic of the second update system made it less favorable in terms of total revenue.

5. 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.* [10] and Subramanian *et al.* [33] evaluated their systems in terms of service disruptions during the update process. Evaluation of runtime aspect-weaving tools [4, 12, 19, 31] 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 net option-value analysis [9, 20]. Net option-value 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 [5] showed the benefits of modular design in increasing a system's value. The main conclusion is 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.* [34] showed the value of design based on information hiding principles by combining option analysis and design information. Similar uses of option analysis can be found in [8, 23, 35]. Ji *et al.* [18] used option analysis to evaluate the benefits from designing and issuing new software releases in relation to market uncertainty.

6. 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 4), 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 basics of net option-value analysis used in this paper can also be used to study the value of update systems from a design perspective. We argue that the problem of quantifying benefits of dynamic updates can be translated into a modular design problem. Systems that supports updates, either static or dynamic, are modular. However, their modules and module-dependencies are different. The question is, which modularization provides greater value. Figure 14 depicts this assumption. Model 0 can be considered as an indivisible system with no update options. Model 1 has the update options represented as modules. However, there are restricting dependencies between such modules. Therefore, any module upgrade (i.e. applying an update) affects other modules (by restarting). Finally, Model 2 has the options and they are decoupled through the use of interfaces, which allows easier upgrades to any module.

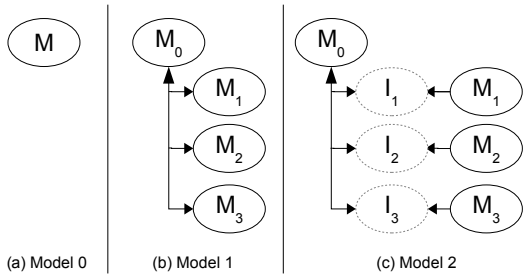

Figure 14: Units of update for different update models

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.

7. 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 [10, 16, 24, 27, 28]. To date, dynamic updating literature evaluates such systems in terms of coverage (i.e. what type of code changes are supported) and performance. Chen *et al.* [10] and Subramanian *et al.* [33] evaluated their systems in terms of service disruptions during the update process and noted the types of code changes that their system can not handle. These evaluation methods are 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 in three update scenarios based on the evolution history of real-world applications. Furthermore, we used the model to compare and contrast two, previously published, 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 [4, 7, 31]: 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.

Acknowledgments

This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CAREER-08-46059. Comments and discussions with Robert Dyer were helpful in developing some of the ideas in this paper. This analysis in this paper has also benefited from the discussion with colleagues over the last few years including Kevin Sullivan, Yuanfang Cai, Michael Hicks, Christa Lopes, and Prem Devanbu.

8. REFERENCES

[1] Android - an open handset alliance project. http://www.android.com.

[2] Android security patch. http://securityevaluators.com/content/case-studies/android/.

[3] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. pages 187–198, 2009.

[4] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pages 86–95. ACM, 2002.

[5] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[6] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, 2000.

[7] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04*, pages 83–92. ACM, 2004.

[8] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, 2006.

[9] A. D. Chandler. *Strategy and Structure*. MIT Press, Cambridge, MA., 1962.

[10] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, 2007.

[11] M. Dmitriev. Safe class and data evolution in large and long-lived java[tm] applications. Technical report, Mountain View, CA, USA, 2001.

[12] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08*, 2008.

[13] E. Figueiredo *et al.*. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, pages 261–270, New York, NY, USA, 2008.

[14] B. Gharaibeh, D. Dig, T. N. Nguyen, and J. M. Chang. dReAM: Dynamic refactoring-aware automated migration of java online applications. Technical report, Iowa State University, August 2008.

[15] M. Hicks. Dynamic software updating. *PhD thesis, Dept. of Computer Science, University of Pennsylvania*, 2001.

[16] G. Hjalmtysson and R. Gray. Dynamic c++ classes, a lightweight mechanism to update code in a running program. *USENIX Annual Technical Conference*, 1998.

[17] HotSwUp. Workshop on hot topics in software upgrades. http://www.hotswup.org/.

[18] Y. Ji, V. Mookerjee, and S. Radhakrishnan. Real options and software upgrades: An economic analysis. In *International Conference on Information Systems (ICIS)*, pages 697–704, 2002.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01*, pages 327–353. Springer, June 2001.

[20] S. Klepper. Entry, exit, growth and innovation over the product life cycle. *American Economic Review*, 86(30):562–583, 1996.

[21] G. Kniesel. Type-safe delegation for run-time component adaptation. In *European Conference on Object-Oriented Programming (ECOOP)*, 1999.

[22] M. Lehman. Software's future: managing evolution. *Software, IEEE*, 15(1):40–44, Jan/Feb 1998.

[23] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, New York, NY, USA, 2005. ACM.

[24] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00*, pages 337–361, London, UK, 2000. Springer-Verlag.

[25] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC '03*, pages 46–60. Springer, 2003.

[26] K. Mätzel and P. Schnorf. Dynamic component adaptation. Technical Report 97-6-1, Union Bank of Swizerland, 1997.

[27] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. *SIGPLAN Not.*, 44(6):13–24, 2009.

[28] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *PLDI*, 2006.

[29] D. L. Oppenheimer *et al.*. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Computers*, 2002.

[30] S. Parker. A simple equation: It on = business on. Technical report, Hewlett Packard, 2001.

[31] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147. ACM, 2002.

[32] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and flexible dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 2006.

[33] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, 2009.

[34] 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.

[35] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *ACM TOSEM*, 2009.

[36] O. E. Williamson. *The Economic Institutions of Capitalism*. Free Press, New York, NY, 1985.

[37] Xerces XML Parser Library from Apache Foundation. http://xerces.apache.org/xerces-j/.

[38] T. Young. Using AspectJ to build a software product line for mobile devices. Master's thesis, Univ. of British Columbia, 2005.