UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
ÉMILIE BOURASSA

MESURE DE PERFORMANCE DE CODE MIGRÉ

JANVIER 2013

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

Remerciements

Merci aux spécialistes de la plate-forme MVS au sein de la compagnie où je travaille.

Merci également à mes directeurs de m'avoir permis d'étudier tout en travaillant.

Merci à mon conjoint de m'avoir aidé à réviser mes chapitres et de m'avoir laissé beaucoup de temps pour travailler sur ce mémoire.

Merci à ma mère d'avoir gardé ma fille pour m'aider à travailler sur ce mémoire.

Merci à mon directeur de recherche de m'avoir guidé tout au long de ce mémoire.

Abstract

Today, we cannot do without technology. It is present everywhere in our lives and we always ask more out of it. All systems that we use need to be more efficient. However, we have to keep in mind that some of them have started a few years ago and they have to update themselves with current technologies.

The service offered by the company I work for is based on a software that was written several years ago and that has not stopped evolving since its conception. We call this type of system: a legacy system. To qualify for the benefits of new technologies, this system has been migrated to a new platform. The original programming language was COBOL, after the migration, it is now COBOL.Net. Following migration, quality tests were conducted and they revealed that the performance has deteriorated a lot.

In this paper, we will look at several ways to improve performance. We will also explore the work that has already been done by researchers on the subject. Subsequently, I will explain some of the work that has been done by the company I work for to improve performance of the migrated code. Finally, I will take, as an example, a migrated program of the company I work for and I will try to improve its performance.

Résumé

Aujourd'hui, nous ne pouvons plus nous passer de la technologie. Elle est présente partout dans nos vies et nous lui en demandons toujours plus. Tous les systèmes que nous utilisons se doivent d'être de plus en plus performants. Cependant, il faut garder en tête que certains d'entre eux ont démarrés, il y a quelques années déjà, et qu'ils doivent se tenir à jour avec la technologie actuelle.

Le service offert par l'entreprise où je travaille est basé sur un logiciel qui a été écrit il y a plusieurs années et qui n'a pas cessé d'évoluer depuis sa création. On appelle ce type de système : un système légataire. Pour pouvoir bénéficier des avantages des nouvelles technologies, ce système a été migré vers une nouvelle plate-forme. Le langage original était du COBOL, suite à la migration, c'est maintenant devenu du COBOL.Net. Par la suite, des tests de qualité ont été effectués et ils ont révélés que la performance s'est grandement détériorée.

Dans ce mémoire, nous regarderons plusieurs façons d'améliorer la performance. Nous explorerons également le travail qui a déjà été effectué par des chercheurs sur le sujet. Par la suite, je vous expliquerai quelques points du travail qui a été fait pour améliorer la performance du code migré au sein de l'entreprise où je travaille. Pour terminer, je prendrai en exemple un programme migré de l'entreprise où je travaille et je tenterai d'en améliorer les performances.

Table des Matières

Liste des tableaux

Liste des figures

Chapitre 1 : Introduction

De nos jours, la majorité des services offerts en entreprise incluent quelque part un système informatique. Il y a même plusieurs entreprises qui dépendent de leur système. Par exemple, si le système ne fonctionne pas, des personnes ne peuvent plus travailler. La même chose s'applique si le système prend du temps à faire son traitement d'informations, alors le temps de réponse des personnes dépendantes du système sera long aussi. Du point de vue client, les problèmes du système d'une compagnie peuvent paraître. Prenons, par exemple, une compagnie de service à la clientèle avec des agents au téléphone. Les clients veulent être servis avec un service rapide et les agents veulent donner aux clients le meilleur service qui soit. Si le système est lent, les clients vont le ressentir, car les agents ne peuvent pas aller plus vite que leur système. Tous les employés d'entreprise dont leurs outils de travail est principalement un ordinateur dépendent du bon fonctionnement des systèmes et s'y fient de plus en plus. Les plus grandes qualités que les gens s'attendent d'un système sont qu'il soit fiable, intègre, facile d'utilisation, disponible, sécuritaire et surtout performant.

La performance est un standard de qualité que plusieurs entreprises visent pour leurs systèmes et applications. C'est aussi un sujet qui a fait l'objet de recherche chez plusieurs concepteurs de logiciels, d'outils informatique et de système d'exploitation. Plusieurs méthodes et techniques ont été développées pour « calculer » la performance. Soit en calculant le nombre d'instructions effectuées par secondes ou encore en comparant des différences de temps de différents traitements. Ces méthodes sont appelées « benchmark ».

Bien des caractéristiques d'un système sont sujettes à être performantes. C'est pourquoi il existe plusieurs types de tests pour en mesurer la performance : le test de tenue en charge [1] qui simule plusieurs utilisateurs en même temps pour une grande période de temps, le test de capacité qui consiste à connecter un nombre grandissant d'utilisateurs pour voir où est la limite du système, le test en stress [1] qui simule les périodes les plus achalandées du système, le test aux limites [1] qui simule une activité bien supérieure à la normale et, finalement, le test d'endurance [1] qui simule un comportement normal sur une longue période de temps. Il en existe aussi plusieurs autres qui sont plus ciblés sur des objectifs à atteindre : test de non-régression [1] qui s'assure que les résultats de performance obtenu en premier lieu ne sont pas dégradés par une modification quelconque, test de composants physiques des machines [1], test

de volumétrie des données [1] qui permet de voir la limite de donnés qu'une méthode peut traiter, etc. Tout dépend des besoins de l'entreprise.

Ainsi, dans un projet, il est important de bien définir les objectifs que l'on veut atteindre au début et de contrôler la performance tout au long du processus de développement et de déploiement. Plus les questions de performance sont abordées tôt dans le projet, moins les problèmes qui y sont reliés sont coûteux à rectifier. Les critères de performances établies au début du projet doivent servir de repère tout au long du développement et aider à définir les efforts d'optimisation que l'on doit apporter.

Pour que les tests soient efficaces, il faut tester d'abord de façon large, puis de manière plus pointue. Il faut également effectuer des tests dans un environnement contrôlé le plus identique possible à l'environnement de production. « Dans une architecture orientée services, il faut tester chacun des composants ainsi que leurs interactions réciproques » [2]. Il faut se définir des scénarios de tests qui permettent de simuler au plus juste, la réalité. Une fois les scénarios établis, il faut tester sur un échantillon du système, faire les analyses de ces premiers résultats, ajuster le système au besoin, exécuter les tests à grande échelle, analyser ces seconds résultats, puis optimiser le système. Comme les tests doivent être le plus près possible de la réalité, il existe des outils capables d'exécuter des scénarios d'essai. Ces scénarios sont automatisés et ils simuleront les actions de plusieurs utilisateurs sur le système. Ce qui évite de mettre plusieurs ressources de l'entreprise à tester.

Les différents tests effectués nous permettent de cibler des traitements longs qui sortent des limites acceptables définies en début de projet. Après analyse, si ces traitements sont jugés critiques, ils doivent être examinés en détail pour améliorer la performance en allant même jusqu'à l'optimisation du code ou la réécriture complète du traitement.

Dans de grands systèmes, il est impossible de trouver tous les problèmes reliés à la performance, c'est pourquoi il est important de superviser les performances en production pour ainsi détecter ces problèmes qui ne sont pas ressorti en phase de test. La performance du service de production peut, par le fait même, être améliorée en même temps que les livraisons d'améliorations logicielles.

Le plus grand problème des projets d'envergures est que plusieurs personnes y travaillent sur une grande période de temps. Prenons, par exemple, un logiciel qui a été créé dans les années 70 et qui continue d'être utilisé aujourd'hui, communément appelé système légataire. Certes, cela fait une grande période de temps et les technologies ont eu le temps d'évoluer beaucoup ainsi que les besoins de l'entreprise. Donc, le logiciel

n'a pas eu d'autre choix que d'évoluer, lui aussi, pour suivre ces besoins. En conséquence, des modifications ont été apportées au logiciel et il peut même y avoir eu des fusions avec d'autres logiciels. Il se peut également qu'il y ait eu des projets d'adaptation avec la venue d'une nouvelle plate forme ou encore une migration complète d'une plate-forme vers une autre. Tous ces changements apportent des problèmes potentiels où la performance d'un système peut être affectée. Ces problèmes peuvent ressortir dû au fait que l'application n'était pas conçue en fonction d'ajouter des fonctionnalités nouvelles, l'architecture qui était définie au début ne correspond plus aux besoins d'aujourd'hui ou encore, en cas de migration, que la logique qui servait sur de vieux systèmes ne soit pas applicable sur de nouvelles plate-forme sans processus d'adaptation majeur.

Il est clair, qu'après un certain laps de temps où le logiciel se métamorphose pour convenir aux nouveaux besoins, qu'il faudra mettre des efforts pour uniformiser le tout et ainsi gagner en qualité logiciel et, par le fait même, en performance.

Si les délais de réalisation d'un projet semblable sont courts, une solution à court terme serait de cibler les traitements les plus critiques pour un remaniement organique ou encore simplement de programmation.

1.1 Techniques d'amélioration de performance

Il existe plusieurs techniques pour améliorer les temps de traitement des différentes fonctions d'un logiciel. Cela peut se faire en minimisant les traitements couteux en temps, comme par exemple, un accès à une base de données. En effet des tests de temps ont démontré qu'il est plus avantageux de travailler en mémoire plutôt que de faire des allés-retours consécutifs à une base de données ou encore à un serveur distinct.

Il est possible d'améliorer les performances de codes en séparant le traitement en fonction du nombre de processeurs présents sur la machine qui exécute le code. On doit alors utiliser la méthode du multithreading. Il faut faire par contre très attention car un programme mal organisé en multithreading peut prendre plus de temps d'exécution qu'un programme bien ordonnée à simple thread. Pour se faire, il faut savoir comment la gestion de la mémoire se fait sur la machine où le code sera exécuté et ainsi éviter de recharger les mêmes données en répétition dans la mémoire. Ce qui veut dire d'éviter de partitionner les données en mémoire pour que le gestionnaire de cache soit efficace et charge les données relatives au traitement en une seule fois (si ceux-ci sont

regroupés en mémoire) au lieu de plusieurs fois (si ceux-ci ne sont pas regroupés en mémoire).

Il existe des outils qui aident à la gestion du multithreading à l'intérieur même de notre code. OpenMP [7], par exemple, est un de ces outils. Il s'agit d'un ensemble de directives de compilation et d'appel de routines de librairies pour le Fortran, C et C++ qui gèrent le partage de mémoire parallèle. Cet ensemble commence par la gestion de l'application comme un simple thread (master) qui s'exécute de façon séquentiel jusqu'à ce qu'il rencontre la première commande pour avoir un traitement parallèle. Il se divise par la suite en plusieurs threads (incluant le master) pour exécuter les prochains traitements jusqu'à ce qu'il rencontre une commande qui signifie que le traitement en parallèle n'est plus nécessaire.

1.2 Optimisation de code

Nous pouvons également descendre jusqu'aux fonctions utilisées et aussi voir les façons de programmer. Bien que la différence entre 2 traitements programmés de différentes façons donne presque les même temps, si ces traitements sont utilisés souvent, il est possible de voir une différence au niveau du temps d'exécution. Prenons un exemple VB.NET :

```
For i = 0 as integer to arrayList.item.count() do

Traitement

Next
```

On remarque ici qu'à chaque itération de la boucle, le nombre d'items de la liste sera recalculé. Donc, si cette boucle se fait plusieurs fois, on remarquera une certaine lenteur. Par contre, si le nombre d'éléments est calculé seulement une fois au départ, l'instruction de la boucle sera plus rapide. Exemple :

```
Dim nbItem as integer

nbItem = arrayList.item.count()

For i = 0 as integer to nbItem do

        Traitement
```

Next

Il y a bien d'autres « trucs » qui nous permettent d'optimiser du code. Comme par exemple garder dans une variable le résultat d'une opération qui sert à plusieurs endroit au lieu de refaire cet opération à chaque fois, mettre les boucles les plus actives à l'intérieur dans le cas de deux boucles imbriquées, sortir le plus de traitement possible des boucles, faire un prétraitement des données avant d'entreprendre un traitement long, prendre le bon type de variables (de préférence les moins gourmandes en mémoire) pour effectuer les traitements... Le but de ces optimisations de code est de faire le moins de traitements, de calculs et de prendre le moins d'espace mémoire possible tout en arrivant au même résultat. On appelle cette façon de faire « code-tuning ».

Il est important de vérifier les structures de données utilisées dans le code. Certaines sont plus « gourmandes » en mémoire que d'autre. Il est essentiel de faire quelques tests pour choisir la structure (que ce soit tableau, array, dataset ...) la mieux adapté aux traitements que l'on veut faire, car une structure adaptée aux besoins vaut mieux qu'une structure non-adaptée et optimisée.

Bien souvent, il y a des règles de bases que les programmeurs oublient qui dégradent la performance. Comme par exemple, enlever le code de débogage lorsque cette opération est terminée et libérer la mémoire lorsque les objets ne sont plus utilisés. Dans le passage de paramètre d'une fonction à l'autre, il est important de réfléchir avant de passer des tableaux ou arrays. En effet, la performance mémoire sera meilleure si les objets volumineux sont passés à une fonction « par référence » car le passage de paramètre « par valeur » nécessite une copie en mémoire. Donc on se retrouve avec de la mémoire vive de moins pour effectuer le travail mémoire. Pour de petites variables, l'espace mémoire est négligeable à moins qu'il y en ait un nombre considérable. L'important est simplement de ne pas faire d'abus sur le passage « par valeur ».

Les grandes étapes pour savoir si le code a besoin d'être optimisé sont bien simples. Il s'agit de développer du code maintenable et facile à comprendre puis, en cas de problèmes de performance, il faut vérifier de quelle portion de code vient ce manque. Ensuite, il faut déterminer pourquoi il y a un problème et vérifier si le code peut être optimisé. Si la réponse est oui alors il faut, optimiser le code. Une fois l'optimisation faite, il faut mesurer s'il y a des changements, si oui, alors on garde le nouveau code, si non, alors on retourne au code de départ. Pour rencontrer les exigences de départ, il faut refaire ce cycle jusqu'à ce que les spécifications définies en début de projet soient atteintes.

1.3 Optimisation extérieur au logiciel

Il n'y a pas que le code qui peut être optimisé mais tous ces outils qui gravitent autour. Une base de données peut, par exemple, être façonnée de manière à sortir les données les plus utilisées à l'aide d'indexation sur les tables ou encore par l'optimisation des requêtes SQL et des procédures stockées qui y sont exécutées. Il est également possible de faire des vues pour regrouper l'information dans un schéma de données qui est souvent utilisé. On sait bien que l'opération la plus coûteuse en temps est la connexion à une base de données. C'est pourquoi il existe des techniques comme la gestion d'un pool de connexions. Un pool de connexions est en ensemble de connexions prêtes à être utilisées. Donc, quand l'application a besoin de se connecter à la base de données, elle va chercher une connexion dans le pool, elle fait ces traitements et la redonne au pool ensuite. Cette même connexion pourra servir pour d'autres traitements plus tard. Ici, on sauve sur le temps qu'une connexion prend pour s'établir entre la base de données et l'application. On peut aussi réduire les allés-retours à la base de données en envoyant une batch d'instructions sur la même commande au lieu d'une seule. Dans chaque logiciel de gestion de base de données, une partie de la performance peut aussi se faire via les options paramétrables. Ce qui permet d'optimiser ce logiciel de gestion en fonction de son utilisation.

Les services et composants externes qu'utilise le logiciel doivent également être optimisés. Il est évident que, si l'on veut garder une constance dans les performances d'un logiciel, les services appelés doivent fournir une qualité de performance équivalente. Ces services et composants peuvent être testé de la même manière et avec les mêmes méthodes que celle pour l'optimisation de logiciel.

De plus, l'architecture réseau matériel peut être optimale en fonction de l'utilisation voulue. Par exemple, si une application fait appel à un service sur un serveur d'une autre ville ou province, il y a moyen d'avoir une ligne à haut débit de transfert (comme, par exemple, une fibre optique) qui permettra d'utiliser ce serveur comme si il était dans la pièce d'à côté. Le temps de réponse des demandes sur le réseau peut être aussi amélioré en apportant plusieurs serveurs qui peuvent répondre aux mêmes services avec la méthode du « load-balancing ». Cette méthode consiste à rendre disponible plusieurs serveurs qui offrent le même service. Les serveurs se partagent alors les demandes à mesure qu'elles arrivent. Il est également possible d'optimiser les protocoles de communications comme le TCP/IP. Cependant, c'est plus complexe à

réaliser, il existe des outils pour nous aider à configurer les performances optimales du protocole TCP/IP qui répondent le mieux aux besoins.

1.4 Programmation orientée aspect

Quelques autres techniques d'optimisation de la performance sont apportées avec la venue du langage orienté aspect (AOP) [11]. Ce type de programmation permet d'implémenter des fonctionnalités autres que fonctionnelles de manière regroupée au niveau du code au lieu que ces fonctionnalités soit réparties comme dans le cas de la programmation objet. Donc, ce langage permet de programmer des fonctionnalités reliées à la performance où le code qui y est associé se retrouvera tout au même endroit.

Plusieurs principes peuvent être utilisés avec ce type de programmation comme un mécanisme de pooling d'objets. Le mécanisme de pooling est une méthode qui permet de réduire considérablement les appels aux constructeurs et destructeurs des classes sur lesquelles il s'applique, en optimisant la gestion d'utilisation des objets avec un pool d'objets. Le principe consiste à recycler les objets en mémoire, une fois qu'ils ne deviennent plus utiles au logiciel. Donc, au lieu de détruire un objet, il est remis à son état initial, puis il est gardé jusqu'à ce que le constructeur de la classe (où l'aspect est appliqué) soit appelé. Alors, comme il y a un objet du type voulu en mémoire qui ne sert plus, celui-ci est retourné au lieu d'en créer un nouveau.

On retrouve aussi le mécanisme COM+ JIT (Just In Time) activation [12]. Ce mécanisme répond au fait qu'il est impossible de prévoir l'intervalle de temps entre 2 appels consécutifs par un même client à un objet. Donc, les ressources liées à cet objet sont inutilisées entre ces 2 appels et « bloque » des ressources qui pourraient servir au traitement. Le COM+ résout ce problème en désactivant l'instance inactive pendant que le client garde une référence active sur l'objet. La prochaine fois que l'objet sera nécessaire au traitement, COM+ réactivera l'objet de façon transparente au client. Quand le COM+ désactive l'objet, il laisse cependant son contexte en mémoire. Lorsque cette méthode est utilisée, il est préférable de ne pas implémenter de « finalize » dans la classe de l'objet car il y a déjà un processus en place avec le COM+ JIT activation. À la place du « finalize » utiliser la méthode « dispose ». En général, si on utilise un pool d'objets et qu'on a l'intention d'y faire un appel d'objet à la fois, il est préférable d'utiliser le JIT activation pour avoir de meilleures performances. Par contre, si on a

l'intention d'y faire de multiples appels, un pool d'objets seul aura de meilleures performances que combiner avec le JIT activation.

L'AOP peut également servir pour optimiser les accès réseaux. Supposons que l'on doit se servir d'un objet qui est distant et qui regroupe 4 propriétés qui doivent être instanciées avant d'être utilisable. Pour éviter de faire 4 accès réseau pour instancier ces propriétés, l'AOP permet d'ajouter une méthode qui prend les 4 valeurs des propriétés. (Cela n'est pas possible si la méthode n'a pas accès à la classe de l'objet.) Il est, par contre, possible de tisser du code qui stocke les valeurs des 4 propriétés et qui empêche l'envoie de leurs demande sur le réseau. Autrement dit, si la méthode intermédiaire qui est créée par l'AOP possède les valeurs des propriétés, elle les instancie directement sans passer par le réseau. Sinon, elle effectue le cheminement planifié et va les chercher sur le réseau tout en les stockant pour un futur appel. L'utilisation du réseau est alors grandement diminuée sans avoir à modifier le client ou la classe de l'objet côté serveur.

L'AOP permet aussi d'optimiser le chargement de données. En fait, cette autre méthode appelée « Lazy load » consiste à repousser le chargement de données jusqu'à ce que l'application en ai besoin. Ainsi, on évite de charger des données qui ne seront jamais utilisées. Prenons, par exemple, une arborescence qui contient plusieurs branches et sous branches. Et bien cette méthode nous permet de charger au fur et à mesure les branches qui sont demandées. Donc, à la fin d'exécution du programme, il n'aura été chargé que les informations dont l'utilisateur avait réellement besoin. Alors, selon les besoins de l'application, il est possible d'aller chercher beaucoup en performance avec cette méthode.

Il est quelque fois possible d'optimiser les performances en planifiant les opérations coûteuses en ressources. Par exemple, prenons un grand traitement de chargement, alors il est bien de penser de vider le « garbage collector » avant le traitement pour que toutes les ressources soient disponibles et que ce traitement ne soit pas déranger par cette action qui se passerait en arrière plan.

1.5 Optimisation à l'aide de la compilation

Il y a également des compilateurs qui offrent d'optimiser dans sa façon de compiler pour que le traitement devienne le plus performant possible. Cette façon de faire peut augmenter jusqu'à 40% les performances d'un traitement. Il suffit simplement de trouver le bon compilateur adapté à notre code. Si cette option est choisie, il faut

faire attention car certaines optimisations manuelles peuvent entrer en conflit avec les options du compilateur.

Une des façons de compiler qui apportera un gain significatif en performance est la compilation « just in time » (JIT). Ce type de compilation est une compilation dynamique qui, dans un premier temps, compile le code en « bytecode » et, par la suite, traduit ce code au moment de l'exécution en langage machine. Le « bytecode » n'est pas un code machine pour un système en particulier, il a l'avantage d'être portable entre différentes architectures. Vous comprendrez, bien sûr, que c'est cette partie qui sera installé sur les serveurs d'applications. Ce code doit être interprété et exécuter dans un environnement de compilation dynamique. C'est-à-dire un environnement où un compilateur peut être utilisé à l'exécution du code « bytecode ». Le but de combiner l'avantage de compilation en « bytecode » et la compilation dynamique est que l'analyse du code source original et les optimisations sont accomplies à la première transformation qu'est la compilation en « bytecode ». La compilation en code machine est beaucoup plus rapide à partir du « bytecode » qu'à partir du code source, car la majorité du travail de compilation a été effectué lors du passage du code source au code « bytecode ». Pour voir un exemple de « bytecode », référez-vous à l'annexe 1.

1.6 Conclusion

Nous venons de voir différents moyens pour améliorer la performance logicielle. Au cours de ce mémoire, nous regarderons plus en détails les façons possibles d'améliorer les performances au niveau du code. Nous ne prendrons pas du code de développeur, mais du code résultant d'un processus de migration automatique. Dans le chapitre 2, nous verrons l'état de l'art, c'est-à-dire où en sont les recherches par rapport à la performance de code. Quelques chercheurs ont trouvé des méthodes pour nous aider à améliorer l'exécution de code et nous en verrons quelques-unes en détails. Dans le chapitre 3, je vous expliquerai les différents enjeux de performances ainsi que les efforts qui ont été fait jusqu'à présent dans l'entreprise où je travaille. Dans le chapitre 4, nous regarderons un processus d'optimisation de performance et nous l'appliquerons sur un programme de l'entreprise où je travaille qui a été migré. Pour finir, nous tirerons nos conclusions sur les façons de faire qui peuvent nous aider à l'amélioration des performances de code migré.

Chapitre 2 : État de l'art

Comme nous venons de voir dans le chapitre précédent, il existe bien des façons pour améliorer la performance d'une application. Plusieurs chercheurs y travaillent sans cesse pour repousser les limites des systèmes.

Certaines recherches traitent d'optimisation de composantes matérielles, qui introduisent la notion de traitement en parallèle au niveau du CPU [7]. D'autres recherches sont plus axées sur le comportement des instructions en mémoire [12], la génération du code machine, la compilation et la prédiction des instructions suivantes [20]. Certains chercheurs s'intéressent même à développer des outils pour aider les développeurs à l'optimisation de leur code. L'optimisation des performances va même jusqu'à optimiser le code au niveau du code machine.

Une grande partie des recherches effectuées sur l'amélioration des performances se concentrent sur la transformation du code lors de la compilation. Les compilateurs usuels ne font que transformer le code en langage machine. Comme cette étape est nécessaire pour pouvoir lancer les applications développées, des chercheurs y ont vu là une opportunité de transformer le code de manière performante et ensuite en langage machine.

Il existe des méthodes qui aident à prédire les performances des nouveaux systèmes avant même que le processus de migration ne soit complété [22]. Ces méthodes ont aussi fait des sujets de recherches pour tenter d'être les plus précises possibles lors de ces prédictions [19].

Dans ce chapitre, nous verrons différentes recherches effectuées sur la performance de code et quelques méthodes pour nous aider avec la performance.

2.1 Projet CAPS

L'équipe de chercheurs du projet CAPS (*Compilation, architecture des processeurs superscalaires et spécialisés*) [19] de l'université de Rennes dirigé par André Seznec se sont intéressés à quelques champs de recherche notamment l'architecture des processeurs, les interactions entre les compilateurs et l'architecture, ainsi que les compilateurs et les plateformes logicielles pour l'optimisation des performances.

L'objectif de leurs recherches est d'aider les usagers à utiliser le plein potentiel de leur machine tout en leur masquant la complexité des composantes matériels.

Un des volets de leurs recherches les a menés à développer un logiciel pour aider les développeurs à la dernière étape de la programmation, c'est-à-dire l'optimisation du code. Ils ont nommé cet outil CAHT (*Computer Aided Hand Tuning*). Ce logiciel s'aide de l'intelligence artificielle pour proposer du code optimal à la place du code existant en utilisant le principe de raisonnement par cas de base.

Une autre section de leur projet cible la grosseur du code. Ils en sont venus à la conclusion que, pour avoir un bon compromis entre la grosseur du code et la performance, il fallait réduire le plus possible les lignes de code utilisées rarement et garder le gros du code dans les sections les plus souvent utilisées. Leur raisonnement est que le code souvent utilisé est chargé en mémoire et peut être utilisé plusieurs fois. Donc, le temps de chargement du code est effectué une fois et le temps d'exécution est effectué autant de fois que nécessaire. Comme le temps de chargement du code en mémoire est le plus long, il est plus performant de privilégier le code pour les traitements utilisés souvent. En minimisant le code des traitements utilisés plus rarement, on évite de trop charger les données en mémoire et on sauve ainsi sur le temps de chargement.

2.2 Migration de code par transformation

Des chercheurs des universités de Waterloo, Victoria et Toronto ont quant à eux fait des recherches sur la migration par la transformation de code. Leur rapport de recherche fait état d'une expérience de développement d'un outil de migration [21]. Cet outil rencontre des spécifications non-fonctionnelles et permet un processus de migration incrémental. Les spécifications non-fonctionnelles visées par l'outil sont que le code résultant de la migration soit au moins aussi performant que l'ancien code et que le code résultant soit facile à maintenir. Leurs recherches ont été faites avec un système légataire de moyenne envergure. La stratégie de migration qu'ils ont utilisée a été de la « traduction » de code car elle ne requiert pas un expert pour comprendre l'ancien code et parce que c'est une façon de faire qui est facilement automatisable. Le code de départ est du PL/IX (un dialecte du PL/I langage développé par IBM dans les années 1970) et il sera migré vers du C++.

La migration de code à l'aide de leur outil passe par différentes étapes. Premièrement, les structures de données sont analysés au niveau du langage de départ

et un équivalent est trouvé dans le langage cible. Comme toutes les structures de données d'un langage ne se retrouvent pas nécessairement dans un autre, il faut générer des artifices pour les simuler dans le nouveau langage. Pour faciliter la maintenance du code migré, la « traduction » du code se fait en gardant les structures d'algorithmes, les noms des variables, les commentaires et l'indentation du code. Ainsi, le nouveau code ressemblera le plus possible à l'ancien code et un programmeur qui est habitué dans l'ancien langage se retrouvera dans le nouveau.

Comme ils développent un outil qui permet la migration incrémentale d'un système, ils ont dû se baser sur certains points pour bien choisir par quelles composantes du système il fallait commencer. Une analyse complète du système doit d'abord être faite pour identifier les composantes les plus importantes ainsi que les sous-systèmes et les interfaces correspondantes. Chaque composante est analysée pour en ressortir leurs interfaces avec le système, le nombre de données qu'elle traite, les algorithmes utilisés ainsi que les structures de données utilisées. Une fois l'analyse complétée, on peut choisir les candidats idéaux pour commencer la migration. On peut se baser sur la grosseur des composantes; Une petite composante sera facile à examiner et à comparer manuellement avec le code de départ. On peut également choisir une composante par rapport à son interface, plus l'interface est simple et avec le moins d'effets secondaires possibles, plus cette composante sera facile à migrer. Les composantes qui affectent la performance à cause d'un manque de ressource ou à cause qu'elles sont un goulot d'étranglement peuvent aussi être choisies en premier.

L'outil de migration du code va tout d'abord analyser le code et le transformer en arbre syntaxique abstrait [21]. Un arbre syntaxique abstrait est un arbre dont les nœuds sont des opérations sur le code (exemple : « *if* ») et les feuilles sont des variables (exemple : un *integer*). (Voir un exemple en annexe 2) Ces arbres représentent bien tous les détails du code, ce qui facilitera la transformation de la migration. Des patrons sont alors identifiés dans l'ancien langage et leurs équivalents sont trouvés dans le langage cible. Le patron cible est alors transformé en code selon une routine de transformation préétabli pour ce patron. Dans cette routine, les structures de données sont transformées dans leur équivalent ou bien dans un artifice qui simule cette même structure.

Voici un exemple de migration de code avec leur outil de migration. L'exemple démontre une instruction « *if* ». Le code de départ est du PL/IX qui sera transformé en C++.

If ^kill then cannot_kill = true;

L'arbre syntaxique est illustré à la figure 1 :

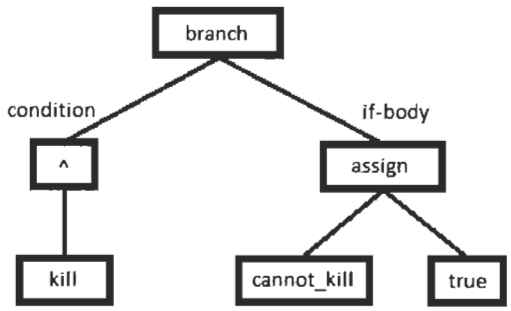

Figure 1 : Arbre syntaxique d'une condition « if ».

Le patron correspondant à cette expression est le suivant :

StatementIf
condition : Expression
then-clause : Statement

La routine de transformation est la suivante :

Transform Statement
.. Transform StatementIf
--> «if (»
....Transform Expression
.....Transform LogicalNot
--> «(!»
..... Transform IdentifierReference
--> «kill»
--> «)»
--> «)»
.... Transform Statement
..... Transform StatementAssignment
--> .....
--> .....
--> «cannot_kill = true;»

Ce qui donne comme code résultant en C++ :

```
if ( (!kill) ) {
        cannot_kill = true; }
```

Une fois la composante migrée, il faut l'intégrer dans le système et supprimer l'ancienne. Il faut la compiler et corriger le code si nécessaire, il faut écrire des classes pour simuler l'ancien système et il faut écrire une interface pour que la composante interagisse de la même façon que le faisait l'ancienne. Bien sûr, plus il y a de composantes migrées, moins il y aura d'interfaces à faire car des composantes différentes peuvent utiliser la même interface.

L'évaluation qu'ils ont faite sur leur outil de migration se base sur certains points dont voici ceux qui nous intéressent particulièrement. Ils vont vérifier les performances du nouveau code, ils vont comparer les performances de l'ancien et du nouveau code, puis finalement, ils vont regarder les efforts humains requis pour faire des corrections au code migré.

Le système légataire qu'ils transforment est un système permettant de faire de l'optimisation de compilation (avec compilateur gcc). Pour pouvoir comparer le système non-migré avec le nouveau système (partiellement migré), ils ont décidé de lancer leurs mesures sur 4 sous système différents. Premièrement, ils ont mesuré une partie de leur système qui contient 13000 lignes de code. Ensuite, ils ont mesuré le module servant à faire de la compilation gcc, ils ont également prit des mesures sur leur librairie g++, puis ils ont mesuré un module distinct que leur système légataire utilise (JPEG). Ils ont lancé leurs tests de mesures un certain nombre de fois et ont prit une moyenne des temps d'exécution. Ils ont par la suite migré les sous systèmes du système légataire et ont refait les mêmes tests avec le nouveau code, voici leurs résultats [21]:

Subsystem	PL/IX Time (*in minutes*)	C++ Time (*in minutes*)	# of Runs
Test13k	11.96	12.19	956
gcc	70.52	70.30	121
lib g++	0.916	0.8626	1065
JPEG	9.4257	9.106	764

Tableau 1 : Statistique de temps de compilation avec le vieux (PL/IX) et le nouveau (C++) compilateur. Les résultats ont étés arrondi sur le nombre de fois que la compilation a fonctionné. L'expérience a été faite sur 4 modèles différents de machine RISC/6000.

Leur système a optimisé 4 différentes sources en C. On peut voir, qu'en général, le temps de traitement s'est amélioré, et ce, pour seulement 3 composantes migrées dans le système.

Leur outil de migration les a aidés à sauver du temps au niveau des ressources humaines nécessaires pour transformer les composantes. Par exemple, pour une composante qu'ils ont migré, si la migration se serait faite manuellement du début à la fin, elle aurait demandé 50 jours/personnes, ainsi qu'un 10 jours/personnes pour l'optimisation de cette dernière. À l'aide de leur outil, l'effort demandé est de 1 jour/personne pour faire l'adaptation de la migration. On peut donc dire que leur processus de migration permet de gagner énormément de temps.

Ils ont également comparé la grosseur des composantes avant et après la migration. Ils ont constaté qu'en moyenne le code est 5% plus important une fois migré. Ceci est dû au fait qu'il a été nécessaire d'écrire des artifices pour simuler l'ancien comportement avec le nouveau code.

Si on revient sur leurs objectifs de recherche, on peut dire qu'ils ont atteints leur but qui était de développer un outil de migration de code qui rencontrerait des spécifications non-fonctionnelles. Ces spécifications étaient que le code résultant de la migration soit maintenable et au minimum aussi performant que l'ancien code.

2.3 Méthode d'évaluation de performance pour système migré

Des chercheurs de l'Université d'Hamilton en Ontario se sont intéressés à une méthode d'évaluation de performance de systèmes migrés avant même que la migration soit terminée. Ils ont étudié la méthode CAPPLES (*CApacity Planning and Performance analysis method for the migration of LEgacy System* [22]) et en sont venu à la conclusion que cette méthode avait des limitations. Ils ont donc proposé une deuxième méthode d'évaluation de la performance pour combler les manques de la méthode précédente : PELE (*Performance Evaluation method for LEgacy system* [22]). Pour bien comprendre leur recherche, regardons d'abord la méthode CAPPLES.

2.3.1 CAPPLES

Cette méthode comporte 9 étapes.

Étape 1 : Spécification de la fenêtre de mesure d'activité. Cette fenêtre de mesure correspond au temps où le système est utilisé, comme par exemple, de 8h à 17h du lundi au vendredi.

Étape 2 : Mesurer le système légataire et les spécifications de la fenêtre de mesure. Dans cette étape, des outils surveillent et compilent l'activité du système légataire pendant sa fenêtre d'activité. Les transactions demandées au système sont comptabilisées en nombre de transactions/heure. Ceci nous permettra de trouver le moment de la journée où le système est le plus sollicité.

Étape 3 : Identification des services concernés dans le système légataire. Ici, on identifie les services qui sont associés aux transactions les plus demandées lorsque le système est le plus sollicité. (Transactions trouvées à l'étape 2)

Étape 4 : Faire le lien des services concernés du système légataire avec le système cible. Chaque service du système légataire doit être offert dans le système cible. Donc, il faut faire un lien entre les transactions du système légataire et les transactions du système cible. Dû au manque de documentation, les transactions sont difficilement identifiables dans le système légataire, elles seront donc liées entre-elles en passant par des services. Ceci est possible car les services ont un grand degré d'abstraction par rapport aux transactions, ce principe est illustré dans la figure 2 [22] :

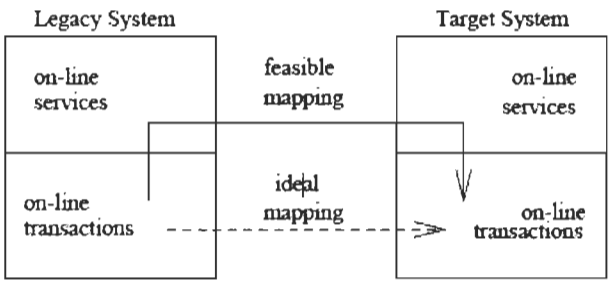

Figure 2 : Liaison des transactions en ligne.

Étape 5 : Génération d'une charge de travail synthétique dans le système cible. Bien que des mesures aient déjà été prises dans les étapes 2 et 3, on complète ici la prise de mesure de toutes les activités qui surviennent dans le système cible. On établi ainsi la charge de travail du nouveau système.

Étape 6 : Modélisation du système cible. Cette étape sert à modéliser le plus fidèlement possible ce à quoi le système cible ressemblera. Toutes les activités trouvées dans les étapes précédentes doivent y être incluses. Il ne faut pas s'arrêter aux

transactions du système uniquement, toutes les activités qui peuvent potentiellement affecter les transactions doivent être prises en compte.

Étape 7 : Calibration et validation du modèle du système cible. Le modèle de simulation du système cible créé à l'étape précédente est considéré valide si le temps de réponse de ces services est équivalent aux services du système cible. Le modèle du système doit être aussi complexe que le système cible et avoir une charge de travail équivalente pour avoir de bonnes mesures dans les étapes suivantes. Le modèle n'a par contre pas besoin d'être aussi grand que le système légataire.

Étape 8 : Prédiction de la charge de travail. Dans cette étape, la charge de travail synthétique établie à l'étape 5 doit être réglée pour simuler le système cible lorsque tous les services du nouveau système seront en fonction.

Étape 9 : Prédiction des performances du système cible. Cette dernière étape consiste à prendre le modèle de simulation créé à l'étape 6 et de lui appliquer la charge de travail prédite à l'étape 8. On peut alors prédire le comportement du système cible.

Selon ces chercheurs, la méthode CAPPLES comporte trois limitations importantes.

Premièrement, il est nécessaire d'aller prendre des mesures dans l'environnement cible pour bien établir la charge de travail à l'étape 5. Cela veut dire que l'environnement cible doit déjà être développé. Il serait préférable de faire la mesure sur le système légataire et de faire le lien sur le système cible. Ceci nous permettrait de prédire la mesure sans que le système cible ne soit prêt.

Deuxièmement, cette méthode ne modélise pas le système légataire. Un modèle du système à migrer pourrait grandement aider lors de l'étape de la validation du système cible. Ceci permettrait aussi de faire une comparaison des modèles entre l'ancien et le nouveau système. Dans la méthode CAPPLES, aucune comparaison n'est faite entre les systèmes.

Pour finir, la méthode utilise des modèles de simulation qui sont complexe à développer et qui demandent beaucoup de temps. Quand on a des échéanciers et des budgets à respecter, les modèles de simulation sont à éviter car ils demandent énormément de détails et ils sont coûteux à l'entreprise.

2.3.2 PELE

Voyons maintenant la méthode PELE [22] qui comporte 8 étapes.

Étape 1 : Détermination de la charge de travail de l'ancien système. La charge de travail est regardée dans un intervalle de temps donné et le nombre de transactions y est répertorié. Cette charge de travail doit être également représentative au système cible.

Étape 2 : Prise de mesure du système légataire. Il y a deux mesures importantes à prendre lors de cette étape : l'intensité et la demande de ressource. L'intensité est la charge appliquée au système, la demande de ressource est le temps que les services prennent pour effectuer une transaction.

Étape 3 : Identification des services concernés dans le système légataire. Cette étape est la même que la méthode CAPPLES.

Étape 4 : Création d'un modèle pour le système légataire. À cette étape, on se sert d'un modèle de file d'attente pour créer un modèle de performance pour le système légataire. Le modèle de file d'attente ressemble simplement aux étapes que doit franchir une transaction pour s'effectuer. Le modèle de performance est alors utilisé pour trouver le temps d'exécution d'une transaction. Il sera alors validé en comparant le temps d'exécution d'une transaction dans le système légataire. Le modèle de performance est considéré acceptable si les temps de réponse ont une différence de moins de 20%.

Étape 5 : Répertorier les demandes de ressource pour paramétrer le modèle du système cible. Dans cette étape, on regarde les services que l'on veut avoir au niveau du système cible en se basant sur le système légataire. Par exemple, si on veut migrer une base de données sous format « flat system » en un format relationnel, il faut prendre en considération cette nouvelle façon de faire pour paramétrer le modèle. Si la nouvelle base de données est opérationnelle sur le système cible, on peut s'en servir pour calculer les besoins en ressources. Sinon, ces besoins devront être estimés en se basant sur le système légataire.

Étape 6 : Analyse et validation du modèle du système cible. Des techniques spécifiques sont utilisées pour trouver le temps de réponse d'une opération à l'aide du modèle bâti à l'étape précédente. Ensuite, on peut comparer ces temps avec le système cible si, bien sûr, il est fonctionnel.

Étape 7 : Comparaison des performances du système légataire et du système cible. Ici, on utilise le modèle du système légataire pour calculer le temps de réponse d'une opération et on la compare avec le temps trouvé à l'étape précédente qui était basé sur le modèle du système cible. Les résultats attendus de cette comparaison devraient être meilleurs sur la cible.

Étape 8 : Prédiction de la charge de travail et de la performance du système cible. Cette étape est la même que les étapes 8 et 9 de la méthode CAPPLES.

Ce qui est bien dans cette méthode c'est que l'on peut l'adapter avec la condition du système cible. Si le système cible est fonctionnel, la méthode va valider ces évaluations de temps sur le système cible. Si le système cible n'est pas fonctionnel, la méthode se sert de modèles et d'évaluation de temps pour évaluer les performances.

La figure 3 présente des principales étapes de la méthode PELE si le système est fonctionnel [22] :

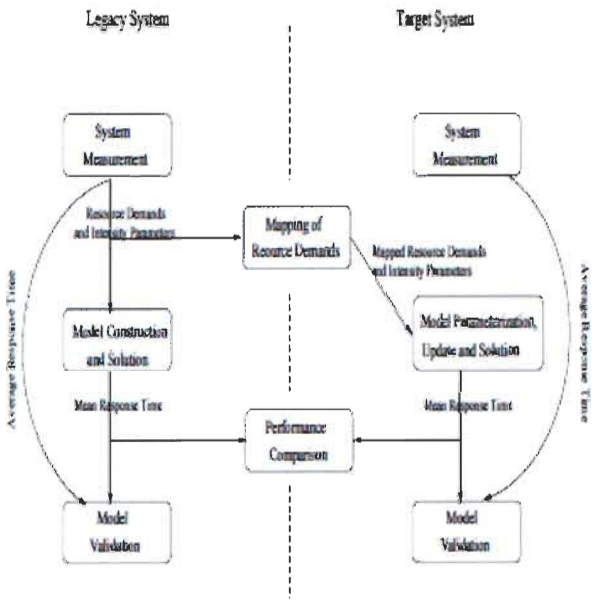

Figure 3 : Étapes de PELE pour un système cible opérationnel.

La figure 4 présente les principales étapes de la méthode PELE pour un système cible qui n'est pas fonctionnel [22] :

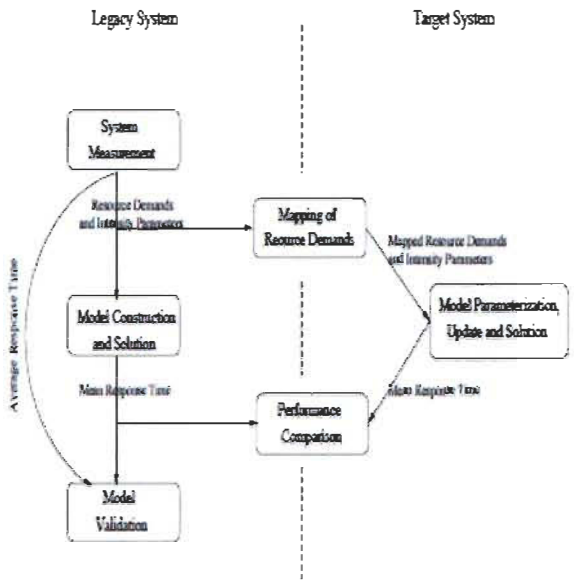

Figure 4 : Étapes de PELE pour un système cible non-opérationnel.

Maintenant que nous venons de voir les deux méthodes, regardons comment la méthode PELE répond aux limitations de CAPPLES.

Dans CAPPLES, il est nécessaire d'aller prendre des mesures dans l'environnement cible pour établir la charge de travail. Avec PELE, le modèle du système légataire est utilisé pour prendre ces mesures. Donc, pas besoin du système cible.

Dans CAPPLES, aucune modélisation du système légataire n'était faite. Avec PELE, un modèle est fait pour le système légataire et cible. Cette méthode permet donc des comparaisons entre les systèmes.

CAPPLES utilise des modèles de simulations détaillés qui sont complexe à développer et qui demandent beaucoup de temps. Dans PELE, les modèles de simulations utilisés peuvent être simple, car les techniques utilisés pour évaluer les performances n'ont pas besoin de détails. Donc, on sauve du temps en utilisant la méthode PELE plutôt que CAPPLES.

Il faut cependant faire attention lors de l'utilisation de la méthode PELE car elle ne se base uniquement que sur le temps moyen des transactions évaluées. Elle ne permet pas d'identifier une dégradation de la performance à long terme causé, par exemple, par une fuite de mémoire. Cette méthode est par contre un bon début d'évaluation. Selon ces chercheurs, un bon complément à cette méthode serait de compléter son utilisation en ajoutant d'autres méthodes pour détecter des problèmes de performance à long terme.

2.4 Étude de cas

Les chercheurs [22] se sont intéressés à un cas de migration de version de MySQL pour valider leur méthode PELE. Ils ont fait la comparaison entre trois versions soit : MySQL 4.1.22, MySQL 5.0.90 et MySQL 5.5.5. Ils ont utilisé le logiciel « TPC Benchmark W » (TPC-W) pour simuler des usagers qui naviguent sur leur application web et ainsi créer une certaine charge de travail. Ce logiciel peut être paramétré de différentes façons de manière à tester différents aspects du système. Comme par exemple, on peut paramétrer le temps entre les transactions ainsi que le nombre d'usagers et d'items présent dans la base de données. Le nombre d'usagers naviguant sur le système sera fixe tout au long de leurs essais.

Pour faire leurs tests, ils se sont fait un environnement isolé sur un portable avec un serveur web et les 3 versions de MySQL à tester. Ainsi, aucun facteur externe ne peut venir influencer leurs résultats. Leur modèle du système est d'ailleurs un réseau fermé contenant uniquement deux ressources : ressource disque et ressource CPU.

Pour commencer, le modèle du système est validé en développant un environnement expérimental en paramétrisant TPC-W avec 150 usagers, un temps entre les transactions à 0 et une base de données contenant 28800 clients et 1000 items. Avant de prendre les mesures qui serviront à comparer les différentes versions, ils laissent toujours à TPC-W 120 secondes pour que l'application ne soit pas en mode « départ » (c'est-à-dire que les temps de chargement en mémoire des principales transactions soient déjà effectués) ensuite les 600 secondes suivantes sont comptées.

La méthode PELE est alors appliquée comme suit :

- Ils exécutent TPC-W avec la version MySQL 4.1.22 pour avoir le temps moyen de réponse d'une transaction et ils gardent l'utilisation du disque et du CPU. (Tableau 2)

- Le modèle du système est utilisé pour calculer le temps de réponse à l'aide de technique « MVA » *(mean value analysis)*. Le temps trouvé est comparé avec le temps mesuré précédemment. (Tableau 3)

La même chose est également faite avec les versions MySQL 5.0.90 et MySQL 5.5.5.

En plus de la mesure MVA et de la mesure sur 600 secondes d'exécution, ils ont prit une mesure de plus en calculant le temps moyen d'une transaction sur 6000 secondes d'exécution.

Voici leurs résultats :

Tableau 2 : Degré d'utilisation pour la collecte de données.

U pour utilisation

Migration Steps	U (%CPU)	U (%Disk)
MySQL 4.1.22	6.859	99.998
MySQL 5.0.90	7.328	99.998
MySQL 5.5.5	6.976	99.989

Tableau 3 : Prédictions et mesures pour la collecte de données.

R pour temps de réponse en secondes

Migration Steps	R (MVA)	R (600 seconds)	R (6000 seconds)
MySQL 4.1.22	66.8145	68.8672	58.4084
MySQL 5.0.90	49.668	51.4355	41.0262
MySQL 5.5.5	74.9925	76.8291	55.5194

Ils en viennent à la conclusion que la méthode PELE est validée car les temps de réponses « MVA » et « 600 secondes » se ressemblent beaucoup. Également, selon la méthode PELE, la migration de la version MySQL 4.1.22 vers MySQL 5.0.90 est bénéfique car les temps sont plus performants une fois la migration complétée. Cependant, la migration va dégrader le temps de réponse si l'on passe de la version MySQL 5.0.90 à MySQL 5.5.5.

Quelques facteurs peuvent venir dégrader la performance quand l'on décide de changer de version de logiciel. Notamment, si la version précédente ne fait pas exactement les choses comme la nouvelle version. Par exemple, si la nouvelle version utilise plus de mémoire pour un traitement donné et que la machine n'a pas toute cette mémoire à offrir au traitement, il y aura plus d'accès disque pour compenser. Comme les accès disques sont très lent comparés à un accès mémoire, la dégradation dû au changement de version pourrait s'expliquer ainsi.

Les mesures calculées avec l'intervalle de 6000 secondes démontrent que la version MySQL 5.0.90 est la plus performante. Ceci vient valider la méthode d'évaluation de performance PELE.

2.5 Migration en service web

Des chercheurs du « *Centre for Advanced Studies* » [23] des laboratoires d'IBM à Toronto se sont intéressés à la migration d'application légataire en service web. Leur but étant de migrer des applications existantes en service web tout en gardant ou en améliorant le niveau de performance qui existait sur le système légataire.

Un service web est une méthode de communication entre deux machines dans un environnement distribué. C'est un programme qui offre un service à travers un réseau. L'interaction avec un service web se fait au moyen d'un protocole indépendant du langage de programmation : « Simple Object Access Protocol » (SOAP). Les services offerts par le service web sont décrits par le langage « Web Service Description Language » (WSDL). Son fonctionnement est simple : le service web est d'abord installé sur un serveur, puis sa description est publiée. Ensuite, un programmeur va déterminer si le service web répond à ces besoins avec le WSDL et il va l'utiliser dans sa logique de programmation avec SOAP.

Un service web est une option intéressante à être utilisé lors d'une migration de système car il est indépendant du langage utilisé par le développeur qui l'utilise, il améliore l'interopérabilité avec son langage de description WSDL et il ne surcharge pas un serveur car il est distribué.

Leurs recherches se concentrent sur la performance de services web migré et plus particulièrement à l'implication du protocole SOAP pour la communication entre le client et le serveur. Les principaux points qui seront évalués sont la latence, l'évolutivité et le débit. La latence est le temps que le service prend pour répondre du côté client. L'évolutivité est la capacité que le service web a à fournir un service à un nombre croissant de demandes sans dégradation de performance. Le débit est le nombre de transactions par unité de temps.

2.5.1 Explication par l'exemple

Pour bien comprendre les différents tests effectués par les chercheurs, ils se sont basés sur un exemple d'application d'enchère qui supporte trois transactions différentes soit : *createBid()* qui permet de publier un item avec un prix de départ, *find()* pour trouver un item spécifique et *makeBid()* qui permet de faire une offre sur un item. Leur exemple est une application EJB (Enterprise JavaBeans). La figure 5 montre un diagramme de classe UML simplifié de l'ancienne application (partie a) et de l'application migrée (partie b). [23]

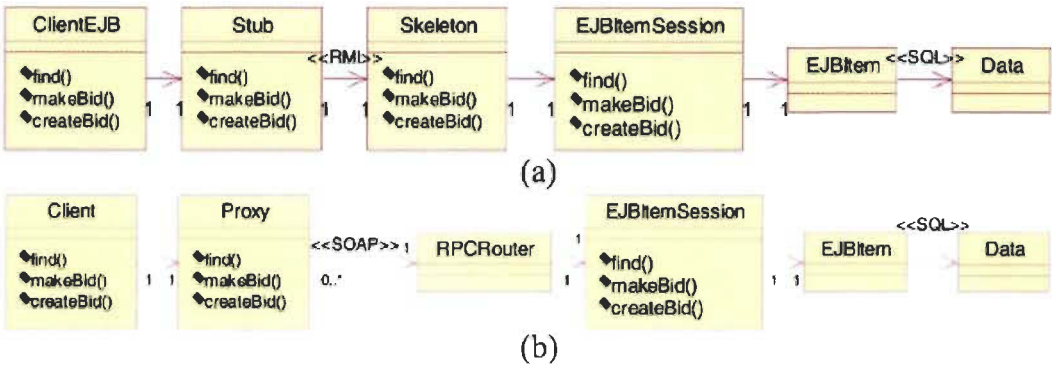

Figure 5 : Diagramme de classe d'une application d'enchère. (a) Application initiale; (b) application migrée : SOAP+EJB.

Pour faire leurs tests de mesure de la latence et de l'évolutivité, ils ont utilisé la méthode *find()* de l'application. Ils vont mesurer l'intervalle de temps entre le début et la fin de cette méthode. Pour prendre leurs mesures de latence, ils ont mesuré l'intervalle de temps de la méthode *find()* avec un client seulement sur une période de 10 minutes et ils ont gardé le temps de réponse le plus court. Le test sur l'évolutivité par rapport au temps de réponse a été fait avec une simulation de 1 à 300 clients avec un temps de 3 secondes entre chaque appel à la méthode *find()*. Pour pouvoir comparer les résultats, les paramètres de la méthode *find()* étaient les mêmes pour chaque tests.

Ils ont utilisé trois programmes différents pour prendre leurs mesures : ApacheSOAPClient qui utilise le service proxy et le client SOAP, OptimisedClient qui utilise la classe HttpURLConnection avec le paramètre de connexion « keep-alive » à vrai et le EJB qui est l'application original. Voici leurs résultats en figure 6 :

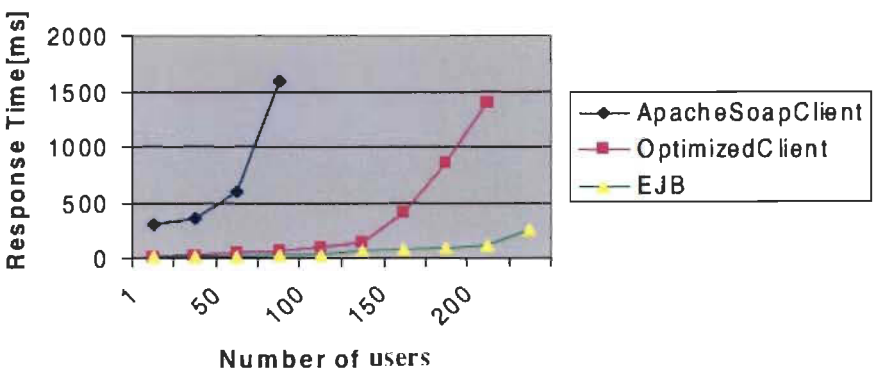

Figure 6 : Mesures d'évolutivité.

On peut voir que l'application migrée a des temps de réponse moins bons que l'application originale. Pour l'application migrée non-optimisée, le serveur devient saturé avec 100 clients alors que le temps de réponse original pour 100 clients est de moins de 0,5 secondes. On peut voir aussi que le fait d'avoir le paramètre de connexion « keep-alive » à vrai, aide au temps réponse et à l'évolutivité.

Ces grandes différences s'expliquent avec l'utilisation inappropriée du protocole HTTP et TCP/IP. Dans le but d'optimiser le transfert sur le réseau, le protocole TCP attend à l'aide de l'algorithme de Nagle [26] que le client du service web remplisse un paquet IP avant d'envoyer les informations. Cependant, ceci occasionne des délais de transmission pour le cas d'un petit message. Le protocole TCP a également un délai sur la réponse du paquet de demande pour laisser le temps au client de récupérer le plus de données possible pour son prochain paquet. Le client SOAP utilisé dans leur test écrit deux fois sur la connexion HTTP. Premièrement, la tête du message « *header* » ensuite, les informations à transmettre « *payload* ». Cette façon de faire occasionne les délais mentionnés plus haut pour chaque envoi d'information « *payload* ».

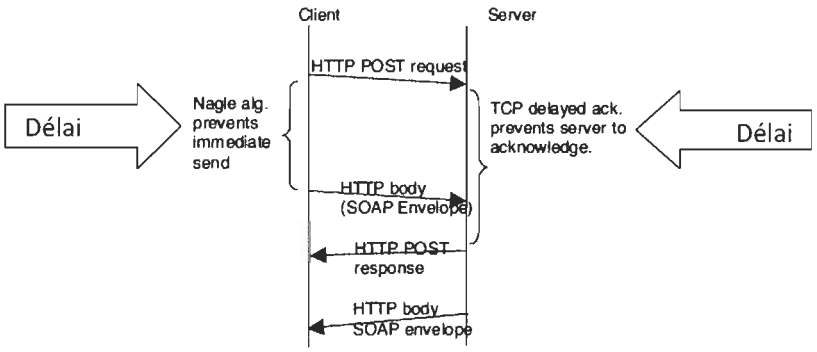

Figure 7 : Algorithme de Nagle et le délai ACK TCP.

La performance pourrait être améliorée en envoyant dans le même paquet la tête du message et l'information.

Avec le protocole SOAP, toutes les requêtes faites sur le réseau vont ouvrir une nouvelle connexion. Avec le paramètre de connexion « keep-alive » à vrai, la connexion est gardée pour les prochains appels. Ainsi, on ne vient pas surcharger avec plusieurs connexions pour un client mais bien une seule par client. Ceci permet donc au service web une meilleure évolutivité.

En plus de ce problème, ils ont découvert que l'analyseur XML apportait une grande dégradation de la performance. L'analyseur XML est utilisé pour convertir les données à transmettre sur le réseau en format XML. Plus il y aura des données à

transmettre, moins bonne sera la performance. Les chercheurs ont testé 2 analyseurs XML pour voir quel était le plus performant : DOM et SAX. Ils en ont convenu que l'analyseur DOM prenait plus de temps et que son utilisation de la mémoire était grandissante de façon exponentielle et beaucoup plus importante que SAX. Pour SAX, l'utilisation mémoire était constante avec le nombre de données à analyser. En plus de transformer les données, les analyseurs peuvent avoir une étape de validation qui vient vérifier les données transformées. Cette étape vient doubler le temps de traitement.

2.5.2 Modèle de performance pour service web

Selon les chercheurs du laboratoire d'IBM, tous les tests de performance sont mieux réalisés quand un modèle d'analyse de performance est utilisé. Un modèle de performance est bâti en incluant toute les composantes et toutes les interactions système qui sont utilisées avec un utilisateur. Une fois le modèle fait, on peut estimer la performance du système lorsqu'un nombre X d'utilisateurs s'y trouve. Le modèle nous permet, par la suite, d'estimer la performance qu'une modification quelconque apporterait sans être obligé de la tester dans le vrai système.

Pour le service web, le modèle de performance utilisé est le LQM (*Layered Queuing Model*). Ce modèle représente toutes les interactions entre le client et le serveur sous forme de file d'attente. Il peut également représenter des parties plus pointues comme un CPU ou un disque. Tout doit être prit en considération lors de la création du modèle, sinon les évaluations de performance vont être faussées.

Pour évaluer des temps d'exécution, toute l'activité d'une transaction va être transformée en équation. Une fois l'équation faite, si l'on veut avoir des temps pour un nombre précis d'usagers, il suffit de remplacer les variables qui gardaient le nombre d'usagers et on obtient notre évaluation. Il est clair que ces équations sont complexes et difficiles à réaliser. C'est pourquoi des techniques telles que MVA (*Mean Value Analysis*) [24] sont utilisées pour résoudre ces modèles de performance.

La figure 8 décrit un exemple de modèle de performance pour un appel de méthode. [23]

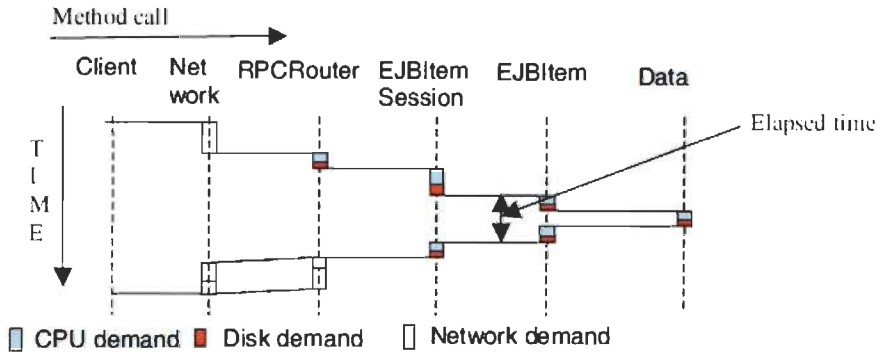

Figure 8 : Mesure du temps d'exécution pour l'appel *find()*.

Dans ce modèle, on peut voir qu'il tient compte du temps réseau, du temps CPU et du temps d'accès au disque.

Les chercheurs ont également comparé les résultats obtenus avec le modèle et de vraies mesures pour valider leur modèle. La figure 9 démontre ce qu'ils ont obtenu :

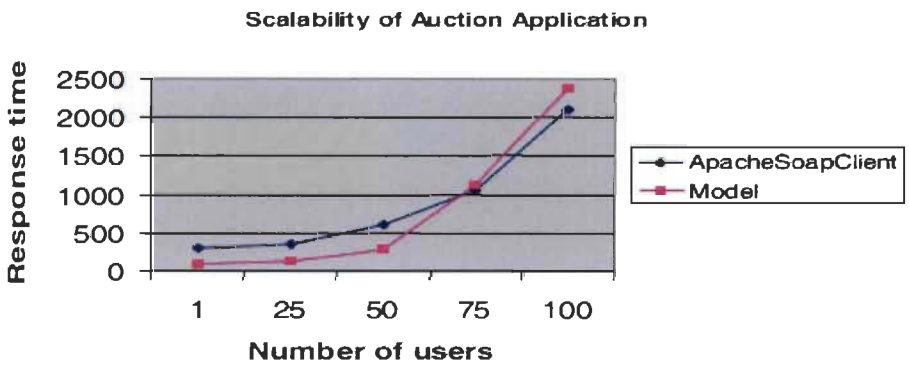

Figure 9 : Évolutivité prédite versus mesuré pour l'appel *find()*.

Un modèle de performance ne peut se faire en une fois, c'est un processus itératif. Lorsque le modèle est validé, on peut s'en servir pour changer une variable et estimer ces impacts sans le faire pour vrai. Comme par exemple, changer un CPU pour un plus puissant et évaluer les gains de performance.

2.6 Conclusion

Nous venons de voir différentes méthodes pour nous aider à prédire les mesures de performance sans les mesurer. Elles nous permettent d'évaluer les gains ou les

pertes de performance suite aux modifications que nous voudrions apporter. Dans le chapitre suivant, nous allons voir ce qui a été fait au sein de l'entreprise où je travaille et les différents besoins de performance suite à la migration de son système.

Chapitre 3 : Mise en contexte

La performance est d'actualité partout où que l'on soit. Ce chapitre traitera des opportunités d'amélioration de la performance que rencontre la compagnie où je travaille. Pour des raisons de confidentialités, je ne la nommerai pas.

Les opportunités d'améliorations sont nombreuses, mais il faut savoir choisir les plus critiques pour des fins d'amélioration. Par exemple, si un traitement peut s'exécuter une fois dans la journée et qu'il a une plage de temps amplement suffisante pour s'exécuter, et bien il ne faut pas y considérer d'efforts pour le rendre plus performant. Par contre, un traitement qui est utilisé plusieurs fois par jours se doit d'être rapide. Le gain de performance qu'on va chercher dans ce cas est significatif.

Dans la compagnie, il y a présentement un projet de migration des systèmes qui a pour but de se départir des vieilles infrastructures technologiques qui sont de plus en plus coûteuses à entretenir et à supporter. On passe d'une plate-forme terminal (MVS) vers une plate-forme d'architecture répartie (Windows). Plusieurs facteurs permettent de dire que ce ne sont pas deux environnements comparables. MVS est un environnement où tout est présent à la même place, du stockage des données jusqu'à l'exécution du code utilisateur. Windows, lui, est réparti sur plusieurs serveurs différents. Il peut y en avoir un ou plusieurs pour les données, différentes couches de communications, différents traitements ... Bref, bien que le système doit rester le plus identique possible pour l'usager, il ne fonctionnera pas de la même manière et il y aura beaucoup d'adaptations nécessaires pour « simuler » l'ancien comportement.

3.1 Projet Migration

Le projet se veut « as is » c'est-à-dire que l'on migre le fonctionnement tel quel sans apporter de modifications à la logique d'affaire au niveau des traitements. Cependant, cela peut apporter des gains de performances aussi bien que des manques de performance des traitements. Il y a deux parties de traitement pour les systèmes de cette compagnie. Une partie interactive, celle que les employés se servent à tout les jours, et une partie lot qui est un traitement de nuit. Les objectifs qui ont été fixées pour la partie interactive est que la perception de l'utilisateur face au temps réponse des traitements soit équivalente ou plus performante que l'ancien système pour les traitements les plus utilisés. Pour la partie lot, l'objectif est d'entrer dans la plage

horaire d'exécution de nuit (soit de 22h à 6h) ou, si le temps venait à dépassé cette plage, il ne doit pas nuire aux traitements interactifs.

Pour respecter les budgets et le temps alloué au projet, les efforts d'optimisation de la performance ne doivent pas viser l'amélioration de la performance face à l'ancien système, mais simplement améliorer les grands manques de performance découlant de la migration. La stratégie de performance retenue vise le minimum d'intervention dans le code sans pour autant exclure la nécessité de modifier ou de réécrire certains traitements ou éléments de code.

L'optimisation de la performance se fera en deux étapes : optimisation unitaire et en charge. Les transactions et programmes exécutés unitairement doivent avoir un niveau de performance acceptable comparé à l'ancien système.

Domaine d'optimisation

Il a été défini 4 domaines de performances au niveau de l'entreprise.

1. Structurel
    a. Serveurs
    b. Réseau
    c. Couche OS
    d. Couche de service logiciel
2. Infrastructure
    a. Travail des données en mémoire
    b. Sérialisation
    c. Etc
3. Code
    a. Otol
    b. ItoO
    c. Etc
4. Applicatif
    a. Chemin d'accès BD
    b. Logique
    c. Etc

Parmis ces domaines, 5 grands axes de performances ont découlés.

1. Performance infrastructure

   a. Matériel
   b. Logiciel
2. Vitesse du code Cobol
3. Latence accès BD
4. Optimisation de la cédule du lot
5. Optimisation applicative
   a. Optimisation pattern d'accès BD
   b. Optimisation BD
   c. Optimisation du code
   d. Ré-écriture

3.1.1 Optimisations applicatives critiques

Prenons par exemple l'ancien système, les données étant situées directement sur les mêmes machines pouvaient être accéder rapidement et plusieurs fois dans un même traitement. En migrant le tout, les données se trouvent séparées du traitement sur des machines distinctes. Donc le temps d'accès aux données doit prendre en considération l'accès réseau qui se fait pour chaque demande. Ce point a d'ailleurs grandement diminué les performances des traitements. La solution qui a été retenu est de charger le plus possible de données en mémoire au début (à l'ouverture de l'application) pour éviter le trafic réseau. Quand il y a une demande d'un dossier client, toutes les informations relatives à ce client sont chargées en mémoire. Et lorsqu'on veut enregistrer ces données, et bien on le fait une fois que toutes les modifications ont été apportées. Donc, on a un accès aux bases de données au début et un autre à la fin des transactions.

Cela a l'air simple à première vue, mais prenons ici en considération que le code de départ est du Cobol et que le code sur Windows est du Cobol.Net. Prenons aussi en considération que le code a été migré par une firme externe qui avait pour mandat de reproduire le comportement le plus efficacement possible sur la nouvelle plate-forme. Donc, on ne retrouve pas simplement du Cobol.Net migré comme si on change de version de « framework » mais bien du Cobol avec des artifices ajoutés pour simuler l'ancien comportement. Les traitements jugés les plus critiques sont les traitements de lecture et d'écriture des données. Ce sont deux traitements qui s'exécutent à chaque transaction et dont le temps réponse est important. Ils ont tous les deux étés réécrits et j'ai grandement été impliquée dans cette tâche.

Le traitement de lecture se faisait sur l'ancienne plate-forme, avec un accès par table pour les données requises à l'application. Sur la nouvelle plate-forme, comme il fallait à tout prix limiter les accès, il a été décidé de réécrire carrément cette partie du traitement pour les diminuer au maximum. Sur Windows, les données sont supportées par une base de données Oracle. Les données ont été migrés telles quelles depuis la base de données IDMS (MVS) vers Oracle (Windows). Donc une table IDMS équivaut à une table Windows. Le problème avec une base de données IDMS est que l'on ne peut pas exécuter une requête avec une clause « where ». Il faut scanner la table pour trouver son enregistrement et ainsi de suite avec les tables enfants. Avec Oracle, il est possible de faire ces requêtes avec des clauses « where », des procédures stockées, des triggers, des vues ... bref, nous avons l'avantage de ce type de SGBD. Nous avons donc rassemblé plusieurs tables ensembles sous forme de vues. De cette façon, il est possible de faire une demande pour plusieurs tables à la fois. La partie réécriture a donc fait appel à ces vues pour le chargement des données en mémoire.

Pour ce qui est du traitement contraire, il a été convenu qu'il ne fallait pas non plus faire plusieurs accès pour mettre à jour les tables de données, mais bien, le moins possible. La solution qui a été retenu est la mise à jour en « bulk » c'est-à-dire plusieurs requêtes « update » à la fois dans une transaction SQL. Par contre, il faut faire attention car cette façon de faire apporte son lot de problèmes. Si la mise à jour échoue à un endroit, c'est toutes les requêtes que l'on a essayé de mettre à jour en même temps qui échouent. Les traitements de mise à jour de la partie interactive du système sont effectués dans le jour par des personnes et ils concernent un dossier à la fois. Donc, si la mise à jour échoue, c'est simplement un dossier qu'il faut reprendre. Par contre, il y a des traitements automatisés qui s'exécutent de nuit sur plusieurs heures pour faire des modifications sur les données. Ici, nous ne pouvons pas nous permettre de faire une seule mise à jour quand toutes les modifications sont effectuées. Nous ne pouvons pas non plus faire une mise à jour par dossier traité, car il n'y aurait pas assez de temps dans la plage prévue pour faire tout ces traitements. Le temps réseau diminue de grandement la performance et le volume de données traitées est important. Alors il faut trouver un juste milieu. Donc, il a été décidé de cumuler un certain nombre de dossiers et de faire la mise à jour en bulk à intervalles de données réguliers. En conséquent, s'il arrive un problème à la mise à jour, c'est simplement un nombre de dossiers précis qui ne sera pas mis à jour contrairement à l'ensemble des dossiers si on avait gardé le principe de la mise à jour à la fin complètement.

Suite à des mesures de performance réalisées après les réécritures, nous pouvons conclure que cette façon de faire remplit ici les conditions de performance que nous voulions atteindre. Cette façon de traiter les données permet d'obtenir un temps

de traitement satisfaisant par rapport aux attentes fixées. Quelques résultats seront présentés dans la section 3.1.6.

3.1.2 Performance matériel

Plusieurs autres travaux ont été entreprit en parallèle avec la réécriture de ces deux derniers traitements. Notamment du côté de l'infrastructure matérielle. Les efforts ont été vers de nouveaux serveurs avec processeurs quadcore, avec de nouveaux routeurs et cartes réseau pour améliorer la vitesse du protocole de communication réseau et la mise en place de serveurs en parallèles pour faire un load balancing.

3.1.3 Optimisation de la base de données

L'équipe des DBA a quand à elle tenté de trouver les paramètres de configuration de base de données optimaux. Elle a également travaillé à la simplification des requêtes Oracle et à la mise en place de vues pour rassembler les données et ainsi éviter les accès multiples.

3.1.4 Tests de charge

Différents tests de charges sont prévus dans une prochaine phase du projet. Ces tests permettront de voir le comportement général du système une fois plusieurs agents connectés à faire des transactions. Les différents points évalués par ces tests sont les suivants :

1. Le comportement de l'infrastructure générale
2. Le temps réponse des transactions
3. Le temps d'exécution pour le lot
4. La stabilité et robustesse
5. La variabilité de l'expérience utilisateur

3.1.5 Optimisations applicatives sur le traitement en lot

Contrairement au traitement interactif où nous pouvons rencontrer un traitement servant à plusieurs endroits, le traitement en lot est séparé en plusieurs « jobs » qui n'ont aucune interactivité entre elles. Donc, l'optimisation se fait « jobs » par « jobs ». Ces travaux de nuits peuvent être aussi bien des impressions de rapports pour les employés, des impressions de lettres pour les clients, des traitements de données sur les banques de données ... Comme il y a plusieurs composantes qui s'enchaînent dans ces traitements de nuit, il faut identifier les plus long et travailler à l'optimisation sur ces derniers.

Pour pouvoir trouver quelles-sont les traitements critiques, la compagnie a trouvé un logiciel d'analyse de performance pour l'aider à la tâche : AQTime [16]. Ce logiciel permet de voir les temps que chaque traitement prend pour s'exécuter. Il permet également de cibler les goulots d'étranglement et les pertes de mémoire. Il va même jusqu'à pointer quel module, classe, fonction ou ligne de code cause le problème. Suite à ces résultats, les traitements sont envoyés à l'équipe de performance pour optimisation.

La plupart du temps, les traitements peuvent être raccourcit en temps en apportant une précision dans la requête SQL qui sert à l'extraction des données. C'est que, sur les anciens systèmes, comme les données étaient sur le même serveur que le traitement, le temps d'extraction des données n'était pas pris en considération. Par exemple, les données d'une table étaient rapportées en totalité au traitement et un tri était fait par la suite. Donc, en apportant une précision à ces requêtes SQL, le tri est effectué au départ par le SGBD. Il est donc plus performant car tout y est déjà indexé.

Des requêtes peuvent aussi être simplement effacées et précisées dans la requête précédente. Je m'explique, dans l'ancien système, les requêtes de jointure entre une table enfant et une table parent ne se faisait tout simplement pas. Il fallait commencer par se positionner sur la table parent et ensuite passer tous les enregistrements de la table enfant pour trouver celui qui faisait référence au parent. Comme aujourd'hui, il est possible de faire référence à l'enfant depuis le parent par une jointure SQL, cela permet de sauver du temps en diminuant grandement le nombre de données traitées.

Il peut arriver que les « jobs » aient une quantité incroyable de données à traiter. Parmi ces traitements, quelques-uns reçoivent un fichier de données en entrée plutôt que de faire une requête SQL. Pour gagner du temps au niveau de ces gros traitements,

il a été décidé de séparer le fichier de données d'entrés en plusieurs petits fichiers et de faire fonctionner cette « job » en parallèle avec les différentes entrées. Ainsi, si on divise le fichier d'entrée en 4, le temps est presque divisé par 4.

3.1.6 Quelques résultats

Prenons maintenant quelques résultats concrets. Dans le traitement de nuit qui roule en lot, un très grand nombre de dossiers clients est traité. Ce traitement doit obligatoirement s'effectuer entre le moment où l'entreprise ferme son service à la clientèle le soir et le moment où il recommence le matin. Donc, le temps de traitement de ces dossiers ne doit pas déborder de cette plage. Une fois ce traitement migré, il a été analysé et comparé avec le même traitement non-migré. Le résultat qui s'en est suivi a été désastreux pour plusieurs raisons : changement d'environnement, code migré... Le temps de traitement d'un dossier client est passé de 1,6sec à 5,63sec. Ça n'a pas l'air beaucoup quand on regarde ça pour un dossier, mais quand il y en a plusieurs, ces secondes se transforment en heures.

Ce problème de dégradation de la performance a été un des traitements qui a eu le plus d'efforts car il se trouvait sur le chemin critique dans l'exécution de la plage de nuit.

Comme ce traitement faisait plusieurs accès à la base de données pour un même dossier, il y a eu réécriture pour concentrer le chargement au début du traitement et non au fur et à mesure que le traitement le demandait. À l'aide de ceci, le temps de 5,63sec est passé à 4,63sec par dossier. Tout un exploit! Cependant, on est encore loin des temps sur l'ancienne plate-forme.

Pour faire la réécriture, les décisions d'orientations de l'entreprise ont été vers les nouvelles technologies : VB.Net, Cobol.Net et Oracle. En plus de ces nouvelles technologies, les réécritures sont passées de la façon procédurale à l'orienté objet. Le tout découpé en architecture n-tiers. Plusieurs couches de programmation ont été développées pour structurer le code et ainsi, faciliter la réutilisation. Parmi ces couches, on y retrouve une couche d'accès aux données, une couche pour la logique d'affaire, une couche d'infrastructure et une couche de présentation pour l'interactif. Il va s'en dire que s'il est possible d'optimiser la performance dans ces couches, elle va être optimisée pour tout les traitements les utilisant. La couche qui est le plus utilisée est celle de l'infrastructure. Cette couche a fait l'objet d'analyse pour optimisation.

Parallèlement à ces travaux, une équipe s'est concentré à trouver d'autres solutions pour optimiser les protocoles réseaux. Elle s'est arrêté sur le principe d'un réseau TOE (TCP Offload Engine) [25] qui permet de décharger le protocole TCP/IP et ainsi améliore la vitesse des échanges.

Suite à des modifications de code dans la couche de programmation d'infrastructure et les travaux de réseau TOE, le temps de traitement des dossiers dans le lot est passé à 4,25sec.

Avec le souci de se rapprocher le plus possible des temps de l'ancienne plate-forme, l'équipe a fait des tests avec une nouvelle version de Windows Serveur. On est passé d'un système Windows Serveur 2003 à 2008. Les tests de performances ont alors passé à 3,72sec. Les serveurs de base de données ont été remplacés par des serveurs plus performants, ce qui a changé les résultats de performances pour 3,53sec.

Pendant ce temps, l'équipe qui s'occupe de l'infrastructure a réussi à faire d'autres optimisations dans leurs codes. (Au niveau de la gestion de la mémoire surtout) Ce qui apporte maintenant le temps de traitement d'un dossier à 2,43sec.

Plusieurs autres actions ont été apportées sur ce traitement jusqu'à maintenant et il est maintenant rendu à 1,99sec.

L'équipe de performance a par la suite tentée de trouver où était le goulot d'étranglement qui ralenti le plus le traitement. À l'aide de logiciels d'analyse de performance, il a été possible de voir que l'écriture sur les disques au niveau des serveurs pouvait être améliorée.

Les disques durs ont une grande limitation : le temps de déplacement de la tête de lecture. Selon les tests effectués, la meilleure performance enregistrée pour un I/O de disque dur est de 7 millisecondes. Si on multiplie ce temps par les milliers de transactions par jours, on peut s'attendre à des problèmes de lenteurs éventuels.

Sur le marché, il y a quelques produits qui permettent de stocker les données sur un support solide et ainsi enrayer le problème de déplacement de composant physique. Notamment le RAMSAN [15] qui est une unité de stockage solide qui permet une vitesse moyenne d'I/O de 0.33 millisecondes. C'est 21 fois plus rapide! Le principe d'un RAMSAN est que la tête de lecture conventionnelle ainsi que le disque sont remplacés par des circuits hautes vitesses et des « chips » de mémoire. Ainsi, on diminue de beaucoup le temps d'accès aux données. Par conséquent, le processeur peut traiter plus de demande en même temps.

La seule crainte par rapport à ce type de stockage, c'est que c'est un type de stockage RAM. Il est nécessaire que le stockage soit sous une source constante d'électricité pour conserver ces données. La solution à ce problème est simple : avoir des batteries de secours et de prendre des backups flash ou des backups sur disque dur en cas de coupure du courant externe.

Suite à ce changement majeur, les tests de performance globaux sur les systèmes ont permis de conclure que les traitements sont passés de 1,2 à 0,76 par rapport à 1 qui est la référence du temps de l'ancien système. Cela veut dire que le lot qui roule quotidiennement prend maintenant 24% moins de temps que l'ancien système!

3.2 Conclusion

Il y a déjà beaucoup de travail qui a été fait et les résultats obtenus jusqu'à présent sont très positif. Cependant, il y a encore du travail qui pourrait être apporté pour profiter pleinement de la nouvelle plate-forme. Dans le chapitre suivant, je prendrai un programme lot migré et je tenterai d'améliorer ces performances à partir du code.

Chapitre 4 : Processus d'optimisation de performance

Le processus d'optimisation de la performance est un processus itératif. Il a pour but d'identifier et d'éliminer les goulots d'étranglement jusqu'à ce que les objectifs de performance soient atteints. Dans ce chapitre, nous regarderons un processus d'optimisation de performance et nous l'appliquerons sur un programme de l'entreprise qui a été migré.

Premièrement, il faut se définir un point de départ. Ensuite, on entreprend des mesures de performance (Telles que décrites dans les chapitres précédents). Suite à ceci, on fait une collecte de ces données, puis on analyse les résultats. Selon les résultats obtenus, on fait les ajustements nécessaires et on reprend des mesures pour réévaluer la performance. Le processus recommence alors du point de départ jusqu'à ce que les exigences de performance fixées au départ soient respectées. Le schéma de la figure 10 démontre bien ce processus.

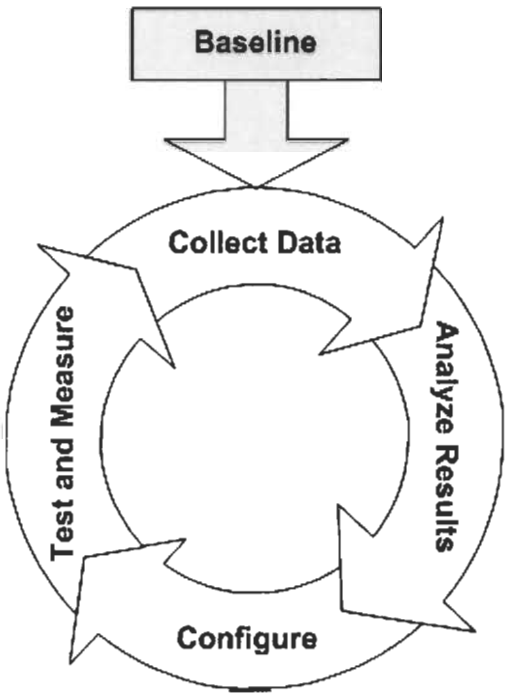

Figure 10 : Schéma du processus d'optimisation de la performance.

Il est bien important de prendre des mesures de performance après chaque modification effectuée dans le but de l'améliorer. Ainsi, on peut déterminer si la modification est bénéfique ou si l'on revient en arrière.

Voyons maintenant ces étapes plus en détail.

4.1 Définir des références

Il faut se définir des objectifs de performance. Ces objectifs peuvent être mesurés en temps réponse, en opérations par secondes, en niveau d'utilisation de ressource sur un serveur, en nombre d'accès disque, en nombre d'accès réseau. Etc. Il faut également avoir un plan de tests et utiliser ce même plan à chaque prise de mesures tout au long du processus d'optimisation. Il faut s'assurer que le système soit identique à chaque prise de mesures pour être certain que les résultats ne soient pas faussés par une donnée externe à notre optimisation.

4.2 Collecte d'information

Pour tester, nous pouvons utiliser des outils tels que Microsoft Operations Manager ou autre pour calculer des compteurs de performance. Lors des tests, il faut s'assurer que la charge que l'on applique lors de la prise de mesures soit constante. Il est d'une bonne pratique de ne pas prendre de résultat avec la première exécution car il peut être faussé dû à la compilation « just-in-time » (JIT), à la mise en cache ou autre. Le temps d'exécution du test doit être un temps réaliste même lors de cette prise de mesure.

Lorsqu'une composante est testée, il faut s'assurer de prendre tous les résultats possibles qui découlent de son exécution. Si la composante fait appel à une banque de données, il faut être en mesure d'aller chercher les logs de performance sur ce serveur de même que toutes les interactions possibles. Ainsi, il sera plus facile de voir où la modification que l'on vient d'apporté sera effective et si cette modification a vraiment changé la performance où nous voulions la changer. En effet, une modification de paramètre peut avoir un effet de cascade, par exemple, si on décide de changer des paramètres de « thread pool », on peut venir affecter l'utilisation du CPU ou même le nombre d'écritures sur disque.

Un test de performance génère beaucoup de résultats sur des machines différentes et dans différents formats. Pour pouvoir les interpréter correctement et

éventuellement comparer nos résultats, il faut rassembler l'information ensemble et la formater.

4.3 Analyse des résultats

Dans cette étape, on analyse les données qui sont ressorties de l'étape précédente pour identifier les points critiques, ces données sont uniquement des indicateurs qui nous aident à orienter nos observations. Ainsi, ils nous permettent d'identifier des zones en problèmes que l'on peut ensuite retravailler.

Si on retrouve des pics intermittents dans l'analyse, il ne faut pas s'en faire et les ignorer car ils ne font pas partis des vrais résultats. Par contre, il faut en tenir compte s'ils reviennent à chaque analyse.

Si le premier traitement semble plus long, on l'ignore. Ceci est sûrement dû au JIT ou à la mise en cache. Il est recommandé de lancer plusieurs fois le traitement analysé pour éviter ces derniers et ainsi avoir des mesures plus justes.

Si vous êtes déjà dans la boucle d'analyse, on peut comparer nos nouvelles données avec les anciennes pour vérifier que l'on s'approche des références établies au point de départ.

Si l'analyse démontre plusieurs goulots d'étranglement, il faut les prioriser et régler celui qui aura un plus grand impact en premier. Une fois la correction faite, refaire les tests et l'analyse. Seulement après avoir refait un tour dans le processus, passer au goulot suivant.

Dans l'étape d'analyse, il est conseillé de la documenter. Écrire nos recommandations en incluant ce qu'on a observé lors des tests et ce qu'on a apporté comme configurations ou modifications pour résoudre le problème.

4.4 Configuration

Faire des changements soit à l'aide d'un nouveau système, une nouvelle plate-forme ou des modifications de configurations. L'analyse faite à l'étape précédente peut contenir plusieurs recommandations, il est fortement conseiller de les suivre.

Lors de cette étape, ne faire qu'une seule modification. Ainsi, nous pouvons déterminer lors de notre étape de test si la modification effectuée est efficace.

Lorsqu'il y a plusieurs solutions à un problème, il faut prioriser celle qui, selon son expérience, aura le plus d'impact sur la performance. Ainsi, si le niveau de performance devient suffisant simplement après une étape, on se sauve des tests!

4.5 Tests et prise de mesures

Une fois que nous avons apporté des modifications, on teste et on mesure pour voir si nous avons une amélioration ou non. Si on a amélioration, on vérifie si cette amélioration est suffisante ou si on continue. Il se peut que l'on ait besoin de faire d'autres configurations, que l'on décide d'apporter des modifications au code ou que l'on veuille changer le design du programme que l'on teste.

Si on n'est pas satisfait de nos résultats de test, le processus de performance recommence jusqu'à ce qu'on aille des résultats satisfaisants.

4.6 Expérimentation

Ce processus d'optimisation de performance s'applique pour plusieurs ajustements tant au niveau machine qu'au niveau du code. Je me concentrerai sur ce qui est modifiable au niveau du code, car c'est de cette partie précisément que ce mémoire fait sujet.

Mes références d'optimisation seront mesurées en temps CPU et le but sera d'améliorer le temps de traitement le plus possible.

Ma collecte d'information se fera via le logiciel NeoBatch , un logiciel d'exécution de JCL. Un JCL est un type de script qui appellera l'exécution du programme que je veux mesurer, il peut recueillir le temps d'exécution réel et le temps d'exécution CPU. Pas besoin de récupérer les données de performance du côté de la base de données car ceux-ci ne feront pas partie de l'analyse et ils demeureront constant.

Cela sera suivi d'une analyse des données recueillit précédemment avec quelques recommandations.

Par la suite, j'appliquerai une recommandation formulée dans l'étape précédente (Une modification au niveau du code).

Finalement, une autre prise de mesure viendra nous dire si nous avons eu amélioration ou non.

4.7 Analyse d'un traitement en lot qui génère un rapport

Ici, j'ai choisi un traitement qui roule en lot et qui produit un rapport, ce traitement sera assez simple à suivre. Il sera également facile de vérifier si les modifications de performance qui seront apportées au code, vont modifier le rapport en sortie. Il me suffira de comparer les deux rapports (avant/après).

Avant de commencer les tests de performance, laissez-moi vous expliquer pourquoi j'ai retenu ce traitement. Premièrement, il faut comprendre certaines différences qui existent entre IDMS (MVS) et Oracle (Windows).

La base de données IDMS doit toujours chercher ces données à partir d'une table parente même lorsqu'on a besoin de chercher un enfant. Cela veut dire qu'il faut parcourir la table parente pour se positionner sur l'enregistrement que l'on a besoin (commande FIND CURRENT). Ensuite, à partir d'un lien qui existe entre le parent et les enfants, il nous sera possible de parcourir les enfants.

Pour Oracle c'est différent. Les tables sont liées entre elles avec des clés primaires et lointaines. Donc, quand nous avons besoin d'un enfant, nous pouvons simplement faire une requête « select » sur la table enfant avec la clé du parent (clé lointaine sur la table enfant) dans la clause « where » du langage SQL.

Dans la conversion du code, tout ce qui était « FIND CURRENT » a été converti en un « select * from table where cleParent = :cleParent ». Ensuite le code continu normalement en allant chercher l'enfant avec la clé du parent dans sa clause « where ». Ici, il est évident que cette première requête de positionnement sur le parent n'a plus lieu d'être.

Le programme que j'ai choisi d'analyser comporte une requête du style « FIND CURRENT » qui a été convertie. Je vais faire des mesures de performance avec et sans cette requête pour voir si nous pouvons avoir un gain. Comme je sais que cette requête est exécutée souvent au cours du traitement, je crois que la performance sera améliorée si on enlève cette requête de positionnement.

4.7.1 Collecte d'information

Pour avoir une bonne idée de combien de temps mon traitement a besoin pour s'exécuter, je vais l'exécuter 100 fois et faire une moyenne de mes temps d'exécution. Pour être capable de lancer 100 fois mon JCL, je me suis faite un script qui l'appelle 100 fois.

Quand un JCL est terminé, NeoBatch crée un fichier de log (SYSLOG) sur la machine. Pour pouvoir cumuler tout ces fichiers de log, je me suis créé un petit analyseur de fichiers qui cherche le résultat dans les fichiers de log et en fait une moyenne.

Une exécution réelle de ce JCL prend entre 40 et 60 secondes. Lors de ma première mesure avec le programme non-modifié, j'en suis venu à une moyenne d'exécution réelle de 39,04643 secondes et un temps CPU de 14,72058 secondes.

4.7.2 Analyse de résultat

J'ai ressorti un graphique (figure 11) démontrant les temps d'exécution. Les temps sont dans l'axe des Y et sont en secondes, dans l'axe des X, on retrouve le numéro de l'exécution. Voici ce que ça donne :

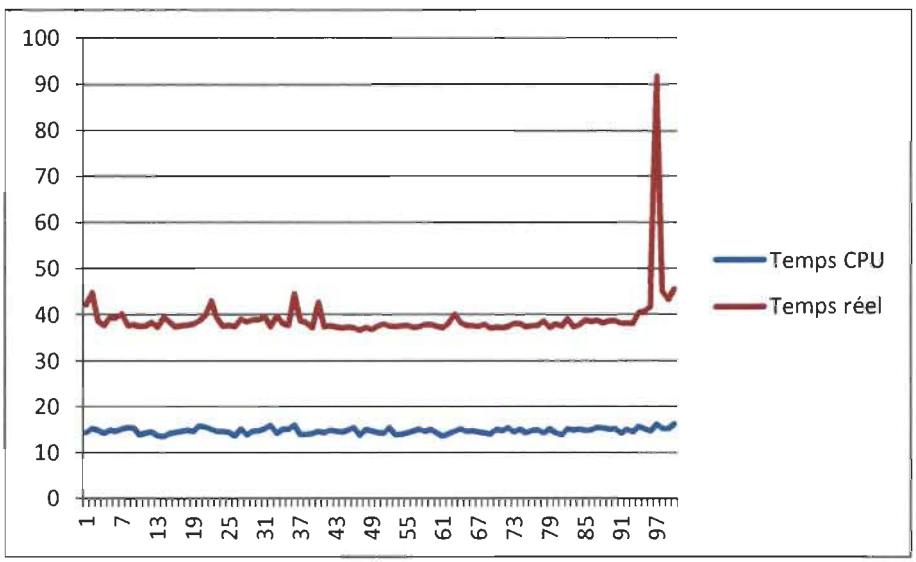

Figure 11 : Temps d'un traitement qui s'exécute en lot.

On peut y voir un pic vers les dernières exécutions. Comme ce pic n'arrive qu'une fois, je vais retirer ce temps de mon calcul de moyenne et refaire un graphique. (figure 12)

Mes nouveaux temps deviennent alors 14,70611 CPU et 38,51404 secondes réelle. Le graphique est également plus stable :

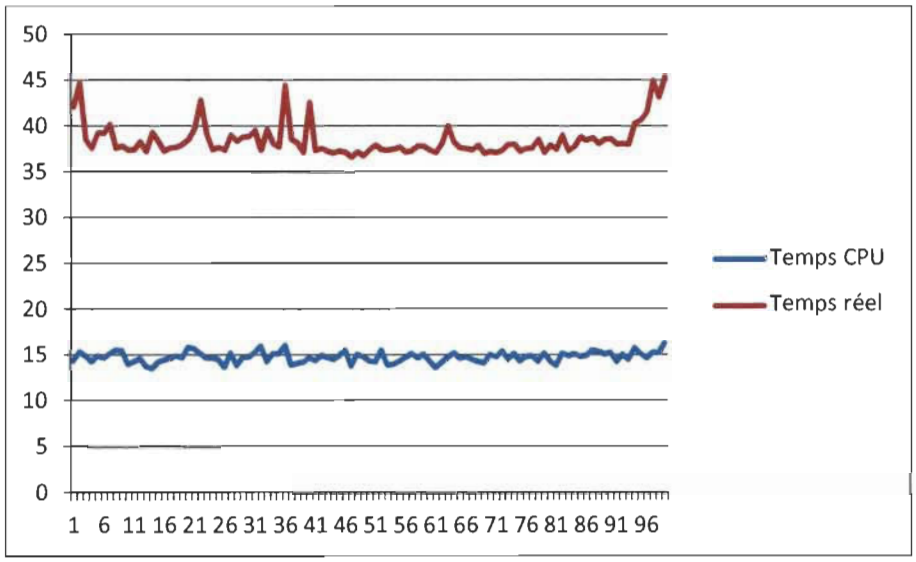

Figure 12 : Temps d'un traitement qui s'exécute en lot sans pic.

Les pics restants ne sont pas assez important pour être enlevés. Ils sont dus aux autres processus qui s'exécutent en même temps que mes mesures sur ma machine de test. La preuve en est que ces pics se retrouvent uniquement sur la courbe des résultats de temps réel à la figure 11. Ces temps ne sont pas si mal pour le volume de données généré par ce rapport. (On retrouve 1794 enregistrements dans le fichier en sortie et en tout 120 729 données traités par le programme) Ici, ce que je veux valider c'est si les temps vont changer si on enlève le select de positionnement inutile.

4.7.3 Configuration

Pour modification, je vais mettre en commentaire la ligne d'appel au « select » de « positionnement » (le FIND CURRENT migré) et je vais modifier la ligne suivante pour que le traitement continu.

4.7.4 Test et prise de mesures

La modification n'a rien changé dans le fichier de sortie. Donc, on peut affirmer que ce positionnement est complètement inutile.

Pour ce qui est des nouveaux temps, on obtient 13.14931 en temps CPU et 36,97928 en temps réel. On peut voir à la figure 13 qu'il y a eu une légère amélioration du temps CPU et du temps réel.

4.7.5 Analyse de résultat

Voici le graphique de ces derniers résultats :

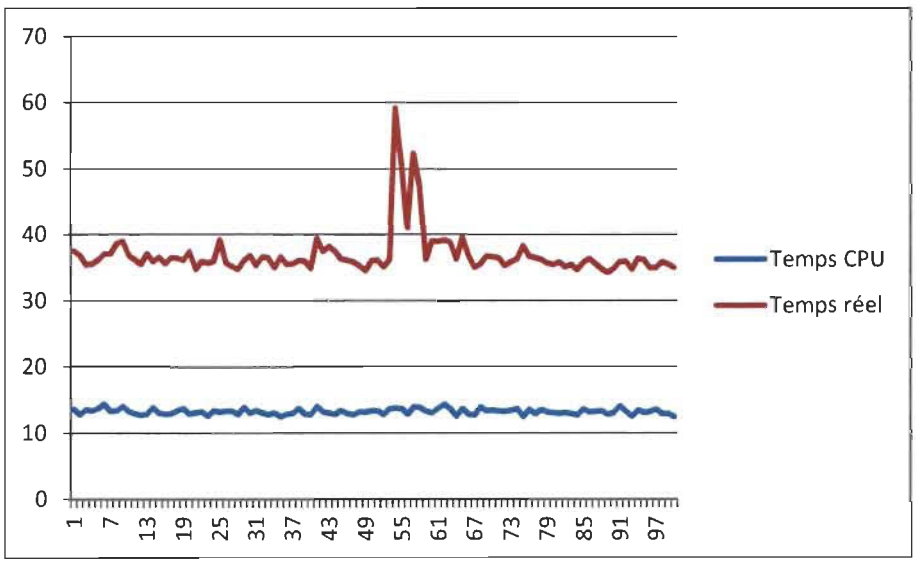

Figure 13 : Temps d'un traitement qui s'exécute en lot sans « select » de positionnement.

Si nous enlevons le pic de temps réels qui se retrouve vers le milieu du graphique et que nous recalculons nos moyennes, nous obtenons le graphique de la figure 14.

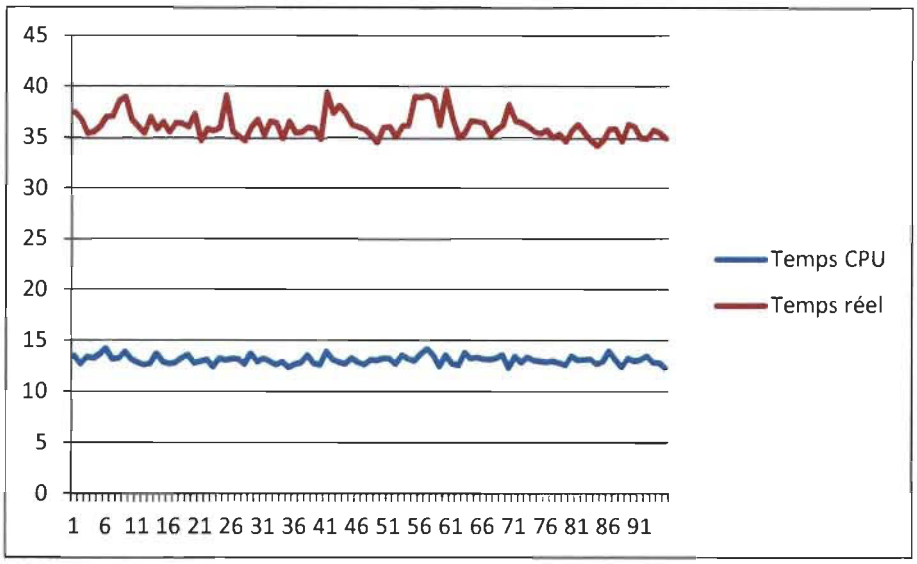

Figure 14 : Temps d'un traitement qui s'exécute en lot sans « select » de positionnement et sans pic.

Avec 13,12839 sec en temps CPU et 36,28247 sec en temps réel. Par rapport à nos mesures de l'étape précédente, nous avons une amélioration d'environ 1,5 secondes en temps CPU et un peu plus de 2 secondes en temps réel.

Le code migré ne pourra plus être amélioré de façon significative à moins de lui faire subir de gros changements. En effet, la logique des traitements sur MVS ne peut être appliquée tel quel sur Windows en espérant avoir de bonnes performances car ces deux plates-formes ne sont pas pensées pour travailler de la même manière.

Si nous prenons juste l'aspect base de données, sur MVS, lorsqu'on a besoin de travailler sur des données, on se positionne sur une donnée parente. Les informations relatives à cette donnée sont alors chargées en mémoire pour permettre le travail direct avec elle et ses enfants. Sur Windows, c'est la programmation qui détermine quelle donnée est chargée en mémoire par le lien de requête précise. Ainsi, un positionnement sur MVS charge en mémoire les données relatives à cette donnée, mais ne sert à rien sur Windows. Une requête précise est permise directement sur les enregistrements enfants sur Windows alors que cela n'est pas possible sur MVS.

J'ai décidé de refaire un tour de roue pour voir si la réécriture de ce traitement nous donnerais un meilleur résultat en se servant le plus possible des performances que nous offre la base de données. La plate-forme IDMS ne nous permettait pas de faire de requêtes complexes dû à son mode de fonctionnement. (Balayage de table, positionnement de curseur pour aller fouiller dans l'enregistrement enfant...) Donc, cela force le programme migré à faire plusieurs accès BD, traiter des données, refaire des

accès BD ... Oracle nous offre l'opportunité de faire des requêtes complexes et de faire un « traitement » de données à même une requête. Il nous offre aussi la possibilité de faire des jointures entre plusieurs tables, ce qui nous permet d'aller chercher plusieurs informations dans des tables différentes. Je vais faire cette requête et je vais l'inclure dans un petit exécutable qui sert uniquement à exécuter la requête et écrire les enregistrements résultants dans un fichier de sortie. Ainsi, je pourrai lancer le nouveau traitement exactement comme l'ancien programme et comparer mon fichier de sortie avec l'ancien.

Le programme que j'optimise appelle deux sous-programmes. Ces deux sous-programmes se transforment aisément en une requête assez simpliste que je pourrai inclure dans ma requête principale comme des conditions dans ma clause « where ».

Les conditions appliquées sur les données au niveau du programme principal se transcrivent également comme des conditions de plus dans ma clause « where ».

Le programme allait chercher des données sur 5 tables différentes (une table à la fois et même plusieurs fois par table). Dans ma nouvelle requête, je peux faire des jointures avec toutes ces tables à la fois.

4.7.6 Collecte d'information

Maintenant que ma requête est terminée et vérifiée, je la donne en entrée à un exécutable qui sert uniquement à l'exécuter et écrire le résultat dans un fichier, ce qui donne le même format de rapport que ce qu'on retrouvait avec le programme initial. Je peux ainsi lancer mon traitement de la même manière que l'ancien avec NeoBatch et comparer l'ancien avec le nouveau rapport. Je vais lancer le traitement 100 fois et je ferai une moyenne des temps d'exécution à l'aide de mon analyseur de SYSLOG.

Les résultats que j'obtiens suite à cette réécriture sont de 3,17279 secondes d'exécution en temps réel et 0,9522 secondes en temps CPU.

4.7.7 Analyse des résultats

Voici le graphique des exécutions (figure 15 et 16) :

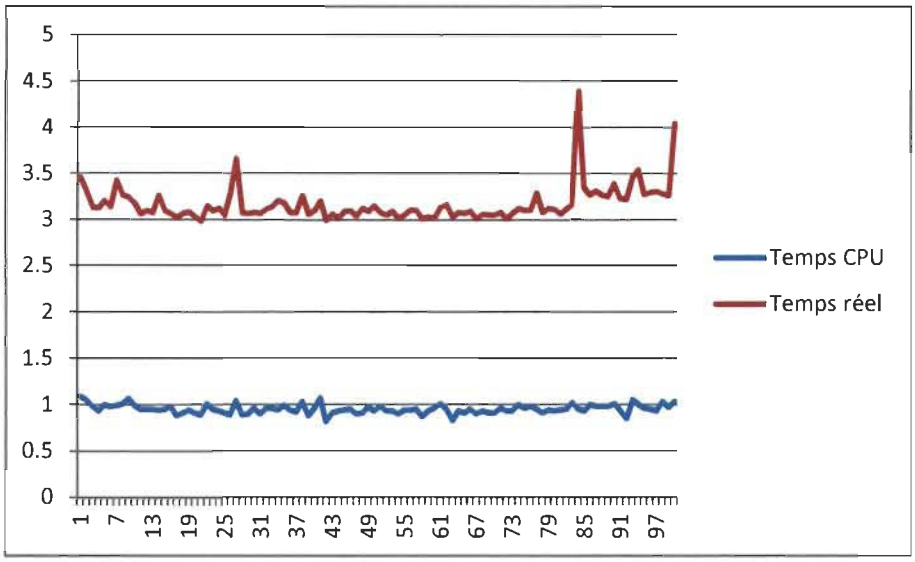

Figure 15 : Temps d'un traitement qui s'exécute en lot suite à une réécriture.

J'enlève ici les deux plus hauts pics de temps puis je recalcule la moyenne. Les pics qui restent ne sont pas assez significatifs pour être enlevé des calculs. Ils sont dus à d'autres processus qui se sont exécutés en même temps que les prises de mesures. (Ils influencent juste la courbe de temps réel.)

Mes nouveaux temps sont alors : 0,951408 secondes temps CPU et 3,151378 seconds temps réels.

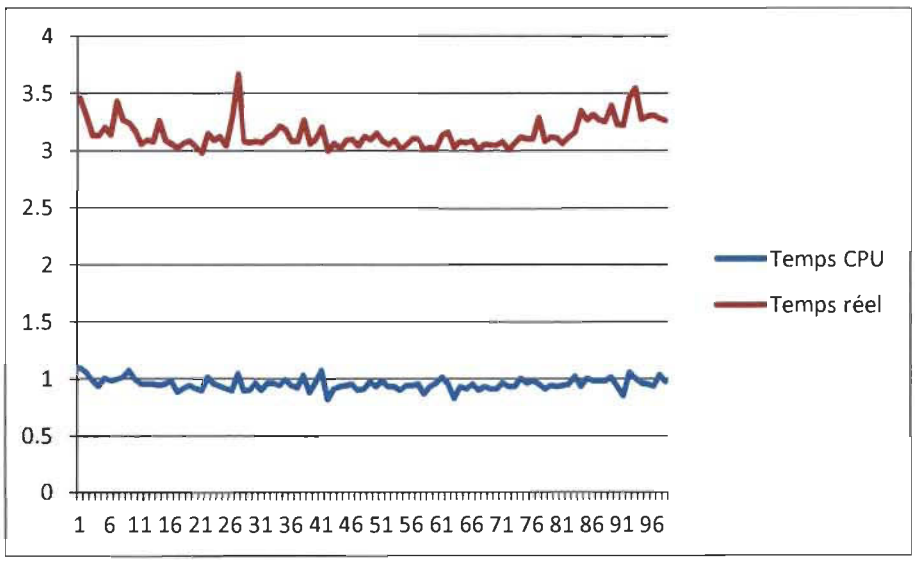

Figure 16 : Temps d'un traitement qui s'exécute en lot suite à une réécriture sans pic.

Je m'attendais à voir une bonne amélioration, mais je ne pensais pas que se serait aussi rapide! C'est sûr qu'il faut prendre en compte qu'ici, un seul accès BD est réalisé pour le traitement tandis qu'avant c'était plusieurs accès BD. Le traitement est d'ailleurs transféré à Oracle où les données sont indexés au lieu qu'ils soient gérées par programmation.

Bref, nous pouvons très bien voir que réduire les accès BD et faire le plus de traitement possible au niveau des requêtes SQL aide grandement la performance.

J'arrête ici mon analyse de ce traitement car les résultats rejoignent l'objectif qui était d'améliorer le temps de traitement le plus possible.

4.8 Conclusion

Nous avons vu ici que le processus de migration migre avec le comportement de l'ancienne plate-forme et ce n'est pas toujours nécessaire de tout garder pour le système cible. Même si le code est fonctionnel une fois migré, il y a place à amélioration. Nous nous sommes également rendu compte que la réécriture de code vient grandement aider le temps d'exécution. Le seul problème à la réécriture, c'est qu'elle demande énormément de temps et nécessite des ressources humaines. Par contre, la migration automatique de code se fait en demandant majoritairement du temps machine. Un autre avantage de la migration automatique est que si l'on retrouve une erreur dans un modèle de migration, on peut corriger ce modèle et relancer la migration de code. On vient ici corriger plusieurs programmes en relativement peu de temps. La réécriture ne permet pas de corriger une erreur de migration rapidement.

Chapitre 5 - Conslusion

Nous avons vu que la performance est un standard de qualité envers les entreprises. C'est pourquoi il est important que ce standard soit présent partout. Il existe plusieurs façons d'optimiser la performance. Les composants matériels peuvent aider de beaucoup à améliorer les performances au niveau du trafic réseau, au niveau de la vitesse d'exécution du code, au niveau de la mémoire vive disponible au traitement... Plusieurs outils peuvent aussi aider à la performance, comme des outils aidant à faire du multi-threading, des outils de gestion de pool de connexions à une base de données, des outils de gestion de pool d'objets, des outils de compilation « just in time » ...

Bien sûr, toutes ces méthodes sont des façons qui peuvent aider grandement. Par contre, il ne faut pas oublier que l'optimisation au niveau du code est la base d'une bonne performance d'exécution. Nous avons vu plusieurs « trucs » pour aider le code, comme par exemple sortir le plus de traitement possible des boucles, ne pas refaire les mêmes calculs plusieurs fois, passer les variables volumineuses par référence plutôt que par valeur pour éviter la copie en mémoire...

Dans le cas de code migré, il est très important d'adapter le code le plus possible à la technologie qu'il utilise. Nous avons vu dans la partie « Expérimentation » (4.6) de ce mémoire que la façon dont le code est pensé influence beaucoup les performances. Il faut savoir adapter la logique entre les différentes plateformes. Quand le code est migré, il est difficile de changer la logique, elle est alors transférée telle quelle en ajoutant des artifices pour simuler le comportement de la plateforme d'origine. Les programmes migrés vont fonctionner, mais la performance ne sera pas au rendez-vous. Dans l'entreprise où je suis, le code a été migré mais pas nécessairement tel quel. Cela est dû aux manques de performance résultant de la migration. Il y a alors eu des analyses qui ont fait que certains « patterns » de code ont été modifiés pour s'adapter à la nouvelle plateforme. Certains traitements critiques ont même été ciblés pour être réécris car la migration de ces traitements n'a pas donné de bons temps.

D'un autre point de vu, la migration de code peut faire sauver énormément de temps au niveau du développement. Oui, les programmes seront transférés plus rapidement sur la nouvelle plateforme et ils seront fonctionnels. Par contre, ces programmes contiennent du code migré qui est composé de code présent pour simuler l'ancien comportement. Le code se trouve surchargé de code de « simulation ». Ceci rend plus difficile la maintenance à long terme.

Nous avons vu que la performance de code migré est loin d'être optimale. Il faut savoir faire des adaptations au niveau du code pour qu'il s'adapte le plus possible à la logique de la nouvelle plate-forme. Ainsi, il est possible d'identifier des « patterns » que nous n'avons plus besoin ou qui sont moins performant et de modifier notre outil de migration de code pour éliminer ou modifier ces « patterns ». Par exemple, enlever des accès inutiles à la base de données comme nous avons vu précédemment. Lorsque la performance devient critique pour des programmes précis, la meilleure solution est la réécriture. Par contre, il faut avoir le temps de réécrire.

L'expérimentation que j'ai effectuée dans la dernière section de ce travail est basée sur le processus d'optimisation de la performance illustré à la figure 9. En résumé, j'ai commencé en fixant des objectifs de performance puis j'ai recueillit des donnés d'exécution. Par la suite, on analyse les possibilités d'amélioration, on fait ces modifications et on regarde si on a rejoint ces objectifs du départ. Après quelques analyses, j'ai pu démontrer que le processus d'optimisation automatique pouvait être amélioré en ajoutant un pattern de code à optimiser. Cependant, c'est la réécriture qui a eu le plus de succès avec l'amélioration de la performance.

Bref, oui la performance est un standard de qualité, mais il faut savoir faire des compromis. La migration de code permet de sauver beaucoup de temps au niveau du changement de plate-forme et l'adaptation des programmes utilisés. On peut donc dire que la migration est performante sur le temps de développement. Lorsqu'on arrive à l'exécution, la migration fait le travail mais pas de manière optimale. Il est donc normal d'ajuster certain programmes soit en modifiant un peu ou allant jusqu'à une réécriture. Cela dépend du temps et du budget que l'on dispose.

Il faut aussi penser à la performance au niveau de la maintenance. Plus il y a de code migré qui simule le comportement de l'ancienne plate-forme, plus il sera dur de faire des modifications au code. Donc, la performance de maintenance d'un programme sera plus difficile que si le programme avait été réécrit. Il faut savoir faire des compromis.

Références :

[1] http://fr.wikipedia.org/wiki/Test_de_performance

[2] http://www.ideotechnologies.com/Documents/Press_book_2/2006/LMI.171106.pdf

[3] http://msdn.microsoft.com/fr-fr/magazine/cc850829.aspx

[4] http://cobweb.ecn.purdue.edu/~eigenman/reports/wompat01spec.pdf

[5] http://www1.euro.dell.com/content/topics/global.aspx/vectors/en/2004_benchmarks?c=eu&l=en&s=gen

[6] http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf

[7] http://openmp.org/wp/

[8] http://www.aivosto.com/vbtips/vbtips.html#optimize

[9] http://www.delphifr.com/tutoriaux/TACTIQUES-OPTIMISATION-VITESSE-EXECUTION-CODE_755.aspx

[10] http://www.info2.uqam.ca/~makarenv/INF3135/Performances.ppt

[11] http://www.dotnetguru.org/articles/dossiers/aop/quid/AOP15.htm#_Toc47186519

[12] http://msdn.microsoft.com/en-us/library/he59ebt3(VS.71).aspx

[13] http://msdn.microsoft.com/en-us/library/ff7105zk(VS.71).aspx

[14] http://en.wikipedia.org/wiki/Just-in-time_compilation

[15] http://www.ramsan.com/whatisassd.htm

[16] Documentation du logiciel AQTime

[17] Différents projets de l'entreprise où je travaille. (confidentiels)

[18] http://msdn.microsoft.com/fr-ca/library/ff647813.aspx

[19] http://www.ens-lyon.fr/LIP/COMPSYS/evalComD/teams/CapsSynthese.pdf

[20] http://spiral.net/related.html

[21] http://www.google.ca/url?sa=t&rct=j&q=one%20approach%20to%20dealing%20with%20spiraling%20maintenance%20costs%2C%20manpower%20shortages%20and%20f

request%20breakdowns%20for%20legacy%20code%20is%20to%20%22migrate%22&source=web&cd=5&ved=0CD4QFjAE&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.50.4373%26rep%3Drep1%26type%3Dpdf&ei=a3E0T7w886uwAoqJ5aQC&usg=AFQjCNH4Y2RofqeiPAWOh_inCX7uJSgrpA&cad=rja

[22] http://www.ksl.stanford.edu/people/pp/papers/PinheirodaSilva_REIS_1999.pdf

[23] ftp://ftp.sce.carleton.ca/pub/cmw/lq-library/marin-migrate-web-04.pdf

[24] http://en.wikipedia.org/wiki/Mean_value_analysis

[25] http://en.wikipedia.org/wiki/TCP_offload_engine

[26] http://searchnetworking.techtarget.com/definition/Nagles-algorithm

Annexe 1 – Exemple de bytecode java

```
  0:   iconst_2
  1:   istore_1
  2:   iload_1
  3:   sipush  1000
  6:   if_icmpge       44
  9:   iconst_2
  10:  istore_2
  11:  iload_2
  12:  iload_1
  13:  if_icmpge       31
  16:  iload_1
  17:  iload_2
  18:  irem
  19:  ifne    25
  22:  goto    38
  25:  iinc    2, 1
  28:  goto    11
  31:  getstatic       #84; //Field
java/lang/System.out:Ljava/io/PrintStream;
  34:  iload_1
  35:  invokevirtual   #85; //Method java/io/PrintStream.println:(I)V
  38:  iinc    1, 1
  41:  goto    2
  44:  return
```

Voici le code java qu'il représente :

```
  outer:
  for (int i = 2; i < 1000; i++) {
      for (int j = 2; j < i; j++) {
          if (i % j == 0)
              continue outer;
      }
      System.out.println (i);
  }
```

Annexe 2 – Exemple d'arbre syntaxique abstrait

Pour l'opération suivante 4+5*10 nous avons l'arbre syntaxique abstrait suivant :