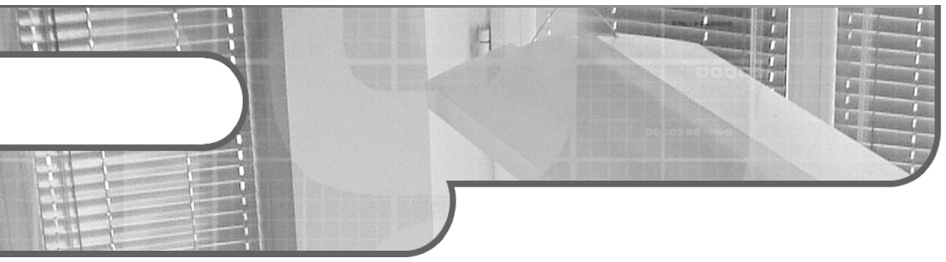


Chapitre 2
Principes du profilage

1. Une activité strictement régulée

Les trois étapes de l'amélioration de performance dans un logiciel sont les suivantes :

- Le profilage : il s'agit, grâce à un outillage spécialisé, de trouver tous les points de contention simples dans du code et de les résoudre. C'est cette étape que nous allons détailler principalement dans ce livre, car elle est souvent sous-estimée alors qu'elle permet presque toujours de résoudre les problèmes de lenteur d'une solution logicielle. À ce niveau d'analyse, toutes les améliorations se cumulent et il est extrêmement rare qu'une modification de code pour la performance supprime les effets de la correction précédente.
- Le tuning : cette deuxième étape correspond au moment où le profilage ne montre plus de goulets d'étranglement. L'application est donc considérée comme correcte du point de vue du code. Toutefois, la performance peut ne pas être suffisante pour les usages, et il faut jouer sur la configuration finement pour améliorer le temps de déroulement des scénarios. Il s'agit de réglages dépendant fortement du contexte, et qui du coup ont de nombreuses interférences les uns avec les autres. Une amélioration de l'occupation mémoire pourra se solder par plus de CPU consommé, ou une modification permettant d'exécuter un scénario plus rapidement fera peut-être ralentir un autre scénario. Le tuning de perfor-

mance est donc un art très difficile, et c'est là l'origine de la réputation de domaine complexe que celui des améliorations de performance, alors que le stade de profilage montre un rendement excellent par rapport à la difficulté de réalisation.

- La réarchitecture : arrivé à ce stade, les efforts de tuning n'ayant pas donné de résultats suffisants, il convient de repasser à l'étape d'architecture, matérielle comme logicielle, de la solution. Bref, nous remettons sur l'ouvrage l'application au sens large (logiciel et infrastructure d'exécution) et rebouclons sur le cycle de conceptualisation, tests, développement et validation, de préférence avec une méthode Agile et de nombreux tests intégrés pour accélérer le retour sur investissement.

Ce livre s'intéresse surtout à la première de ces trois étapes. Le profilage est une activité informatique consistant à recueillir des métriques sur l'exécution d'une application cible, et à analyser celles-ci au regard du scénario joué pour trouver des améliorations de performance possibles.

Tous les mots sont importants dans cette dernière phrase et ils doivent être expliqués minutieusement.

Tout d'abord, le profilage est une méthode informatique : nous nous appuyons sur des outils logiciels dédiés à la recherche de problèmes de performance dans les applications. Dans notre cas, ces outils seront adaptés au profilage d'une application .NET, et seront présentés dans un prochain chapitre.

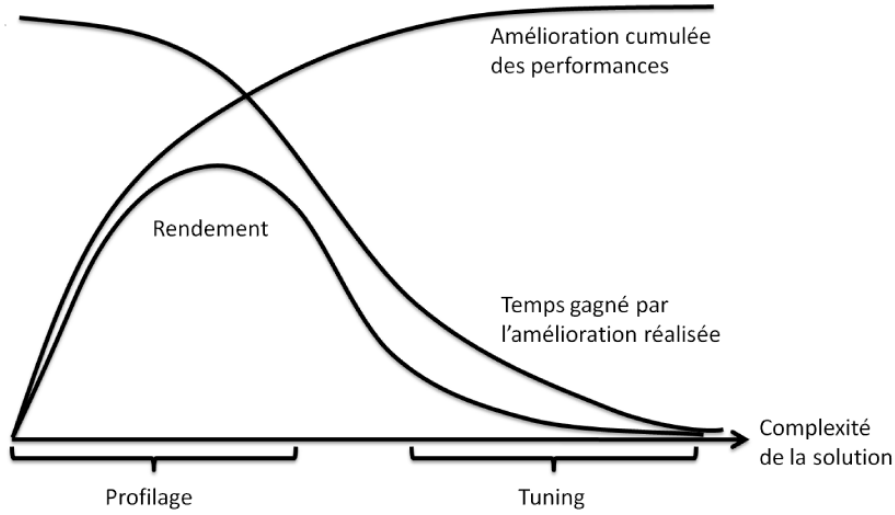
Plus loin, nous parlons de métriques, car le profilage consiste principalement à mesurer. Que ce soit des temps, des nombres de passages dans des fonctions, des pourcentages de temps passé dans chaque méthode, les profileurs servent surtout à produire des valeurs. Dans certains cas, ils peuvent apporter une légère analyse sur ces métriques, et proposer des pistes d'amélioration à l'analyste, mais il ne faut pas attendre beaucoup de ces outils pour une aide active à l'optimisation. Ceci reste le travail d'un humain, et ce livre propose justement des outils intellectuels en plus de ces outils informatiques pour aider le développeur à diagnostiquer au plus vite le pourquoi d'un goulet d'étranglement.

Ensuite, le terme de scénario est employé à dessein pour mettre en évidence une caractéristique fondamentale du profilage : il se réalise sur des scénarios d'utilisation d'une application, et non sur l'application elle-même. Les profileurs n'analysent pas le code d'un logiciel pour trouver les problèmes de performance : ils se contentent de suivre le comportement interne de l'application lors de l'exécution d'un enchaînement donné d'interactions. En ce sens, c'est un abus de parler d'optimisation des performances d'une application, dans le sens où nous n'optimisons que des scénarios donnés. Au final, c'est encore au développeur de faire en sorte que ces scénarios couvrent au mieux les principales fonctionnalités du logiciel.

Enfin, le but du profilage est une amélioration des performances de l'application. Mais de quelles performances parlons-nous ? Légéreté en mémoire ? Robustesse de l'application ? Ces buts sont évidemment intéressants, mais il faut se rendre à l'évidence : un client porte principalement son attention, en plus de l'aspect graphique d'une application, sur sa réactivité (on parle de temps de latence). Sur une application serveur, en plus de la latence, le point principal de mesure sera le throughput (le nombre de transactions traitées dans un temps donné). Bref, la sobriété d'une application reste une qualité désirable, mais sa rapidité est une qualité nécessaire. Malgré qu'il existe des outils de profilage de la mémoire (et cette activité remplirait à elle seule un livre complet), nous parlerons donc ici de rapidité uniquement.

D'ailleurs, quelle doit être cette rapidité ? Quels sont nos buts, et sont-ils chiffrés ? La réponse est tellement dépendante du contexte que la définition ne peut être technique : une application est assez rapide lorsque nos clients estiment qu'une amélioration de performance ne lui apportera pas une valeur additionnelle. Cette définition peut paraître très vague, mais l'auteur souhaite insister sur le pragmatisme nécessaire à l'activité de profilage : il ne s'agit pas de tirer la quintessence d'un algorithme, mais de maximiser le rendement entre le temps passé et l'augmentation d'utilisabilité pour le client. Attention : le profilage est une activité où, encore plus que lors du développement, il est aisé de se perdre dans un perfectionnisme sans rapport avec l'intérêt de nos clients, et donc le nôtre.

Le graphique ci-dessous montre clairement cette réalité : à part dans les cas particuliers d'une application de qualité extrême, les premières modifications seront relativement simples et elles apporteront de grandes améliorations de performance. Bref, le rendement sera très bon. Au fur et à mesure, les solutions pour améliorer la performance seront plus complexes, et n'auront plus le même rendement. En fin de cycle, nous entrerons dans une phase dite de "tuning" où les efforts devront être conséquents pour gagner encore quelques millisecondes.



Sans remettre en cause l'utilité de la seconde phase dans certains cas précis, le présent ouvrage se consacre principalement à la première, dite de "profilage". L'auteur souhaite montrer que cette étape est à la portée de tout développeur, à l'inverse du tuning qui est réservé aux experts. C'est certainement un défaut d'image qui explique que le simple profilage est souvent négligé. C'est pourtant lui qui a le meilleur rendement entre le temps investi et le gain en performance.

Dans la suite de ce livre, une application exemple nous servira de référence pour deux scénarios que nous amènerons pas à pas au niveau de performance voulu. Ceci nous conduira à montrer les différentes facettes du processus de profilage, ainsi qu'à approfondir la façon dont il peut être mené, et surtout avec quels acteurs. L'activité est traditionnellement réservée dans les équipes informatiques à quelques personnes expertes. Pourtant, il est généralement constaté en pratique qu'à condition de suivre quelques règles fondamentales, un simple développeur sans connaissance particulière du fonctionnement interne de .NET peut tout à fait réaliser des opérations de profilage d'une grande utilité.

Ceci est dû à deux facteurs :

- Tout d'abord, de nombreuses améliorations de performance peuvent être réalisées en supprimant des erreurs de programmation, et il n'est absolument pas nécessaire de se pencher sur des solutions très complexes techniquement pour y parvenir. Typiquement, le fait de lancer en boucle une requête SQL au lieu de réaliser une seule requête et de boucler sur le résultat est une erreur de programmation, et l'analyse des traces peut assez facilement faire remonter cette erreur. N'importe quel développeur est capable d'analyser cette condition de contention et de la résoudre.
- En plus de ceci, l'expérience montre que les optimisations les plus simples à réaliser sont celles qui apportent le plus de gain de performance. Il n'est pas rare que la suppression des trois points de contention les plus saillants dans un scénario permette de réduire le temps d'exécution de celui-ci par un ou deux ordres de grandeur (soit une division par 10 ou 100). À l'inverse, si nous souhaitons réduire encore le temps passé, il est alors nécessaire d'entrer dans des phases d'optimisation plus complexes (on parle de "tuning"). Celles-ci mettent en jeu le fonctionnement fin du moteur d'exécution de .NET (la CLR, pour *Common Language Runtime*). Par exemple, le positionnement dans le code de l'instanciation des variables par rapport à leur utilisation influe directement sur le ramasse-miettes, et peut poser un problème de performance. Ce genre de profilage nécessite alors un expert, mais n'intervient qu'après une première phase plus simple, dans les rares cas où celle-ci n'a pas suffi.

Tout le principe du présent livre est justement de montrer que l'amélioration des performances d'une application .NET est du ressort de chaque développeur, et surtout pas le seul fait d'un expert technique au sein de l'équipe de développement.

Ceci n'est possible que si ce développeur connaît et respecte quelques règles de base, sans lesquelles le profilage peut se révéler être une simple perte de temps, voire dans certains cas contre-productif. Ces règles sont au nombre de cinq, à savoir :

- Stabilité de la plate-forme
- Neutralité de l'environnement
- Objectifs fixés au préalable
- Améliorations mesurables
- Granularité descendante

Ces cinq règles sont reprises et détaillées dans les cinq sections ci-dessous.

2. Stabilité de la plate-forme

2.1 Pourquoi cette règle ?

Il est essentiel de bien préciser sur quelle plate-forme nous travaillons, et ce, de manière détaillée : version du logiciel ciblé, type de base de données, système d'exploitation, type de machine, etc.

Les raisons sont avant tout de nature quantitative. Pour comparer des analyses dans le temps, et en particulier savoir si un logiciel s'améliore ou voit ses performances se dégrader au fur et à mesure des versions, il faut pouvoir comparer sur une référence stable.

Mais c'est surtout à l'intérieur d'une campagne d'optimisation de la performance que cette stabilité est nécessaire. Si un banc de test doit évoluer lors du passage à une nouvelle version, il est tout à fait possible de tester celle-ci sur l'ancien banc de test en même temps, et ainsi établir des correspondances de performances entre les anciens temps et les nouveaux.