# Généralisation

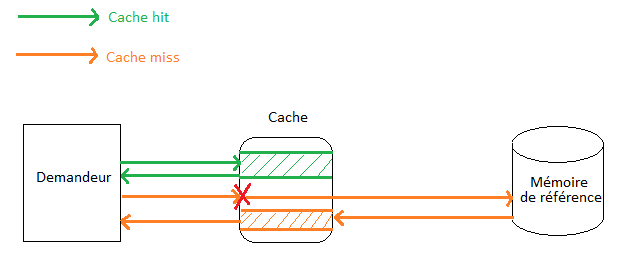
De manière générale, la mémoire cache (ou l’antémémoire en Français), permet d’économiser et de réduire le temps d’accès aux ressources. Par exemple le CPU qui souhaite accéder aux données contenues en mémoire centrale.

# Principe de fonctionnement

L’antémémoire se positionne entre un élément « demandeur » et une mémoire de références contenant les ressources (données). Elle est de très petite capacité et très onéreuse. Par définition l’accès à l’antémémoire est beaucoup plus rapide.

Lorsque le demandeur à besoin d’accéder à une ressource en écriture ou en lecture, le principe de fonctionnement est le suivant :

1. L’antémémoire vérifie qu’elle ne contient pas une copie de la ressource réclamée. Si la ressource est contenue, l’antémémoire la fournie directement au demandeur. On parle alors de ***succès de cache*** (Cache hit en anglais) (voir flèche verte sur le schéma plus bas).  
   En revanche si la ressource n’y est pas, on parle alors de ***défaut de cache*** (Cache miss) (voir flèche orange sur le schéma plus bas). Le cas échéant, on procède aux étapes suivantes…
2. La demande se propage au niveau de mémoire supérieur. Cette dernière renvoie la ressource souhaitée.
3. L’antémémoire stocke la ressource demandée pour une utilisation ultérieur puis la retransmet à l’élément demandeur.



Ce qui détermine la performance d’une antémémoire est ce que l’on appelle le taux de succès (Hit ratio) qui est le nombre de succès de cache par le nombre d’accès mémoire.  
Il est évident que plus celui-ci est élevé, plus efficace sera l’antémémoire, et plus l’on aura économisé du temps d’accès aux ressources.

Les défauts de cache se distinguent en 4 types :

* les défauts de cache obligatoires : ils correspondent à la première demande du processeur pour une donnée/instruction spécifique et ne peuvent être évités ;
* les défauts de cache capacitifs : l'ensemble des données nécessaires au programme excèdent la taille du cache, qui ne peut donc pas contenir toutes les données nécessaires ;
* les défauts de cache conflictuels : deux adresses distinctes de la mémoire de niveau supérieur sont enregistrées au même endroit dans le cache et s'évincent mutuellement, créant ainsi des défauts de cache (voir les caches directement adressé plus bas) ;
* les défauts de [cohérence](https://fr.wikipedia.org/wiki/Coh%C3%A9rence_(donn%C3%A9es)) : ils sont dus à l'invalidation de lignes de la mémoire cache afin de conserver la cohérence entre les différents caches des processeurs d'un système multi-processeurs.

# Les types de cache

On distingue deux types de mémoire cache. Le cache matériel essentiellement (ou totalement) implémentée de manière électronique, ou bien le cache logiciel, implémenté directement par un programme informatique.

## Les caches matériels

Une mémoire cache matériel est très performante mais aussi très onéreuse. Un exemple de cache matériel très utilisé est la SRAM (Static Random Access Memory). La SRAM est un type de mémoire vive utilisant des bascules pour mémoriser l’information.

Une SRAM de 1999 :



## Les caches logiciels

L’un des caches logiciel le plus connus est le cache implémenté par les navigateurs WEB. Lorsque qu’une page web est chargée, il est fréquent que celle-ci requière un certain nombre de fichiers annexes (images, javascript, css, etc.). Afin de limiter le nombre de requête et d’améliorer la vitesse de chargement des pages ultérieurement, les fichiers annexes sont stockés sur la mémoire de masse de l’ordinateur et seront réutilisé à la demande.

Un autre exemple concerne le SGF d’un système d’exploitation. Lorsqu’un fichier est ouvert pour lecture et/ou écriture, celui-ci est en réalité mis dans un cache logiciel sur la mémoire centrale afin de rendre son accès plus rapide.

# Problématiques d’implémentation

La conception des caches implique de prendre en compte plusieurs problématiques listée ci-après :

## Notion d’optimisation

La raison d’être de l’antémémoire est de permettre l’accès aux ressources plus rapidement afin d’améliorer la vitesse d’exécution des programmes.

Deux grands principes ont pu être mis en lumière lors d’étude comportementale des programmes informatiques :

* La localité spatiale, qui indique que l’accès à une ressource située à l’adresse X, sera probablement suivi d’un accès à la ressource située à une adresse X+1 ;
* La localité temporelle, qui indique que l’accès à une certaine zone mémoire à un instant T, a de forte chance de se reproduire dans un lapse de temps très cours (principe des traitements itératifs).

Ainsi, la conception des mémoires cache va tendre à tirer un maximum profit des ses principes. Certaine implémentation cherche même à améliorer le taux de succès (hit ratio) en prédisant à l’avance (1 ou 2 cycles CPU) la donnée à charger en cache. Pour des programmes correctement optimisés, la plupart des prédictions seront correctes, ce qui rend l’implémentation pertinente.

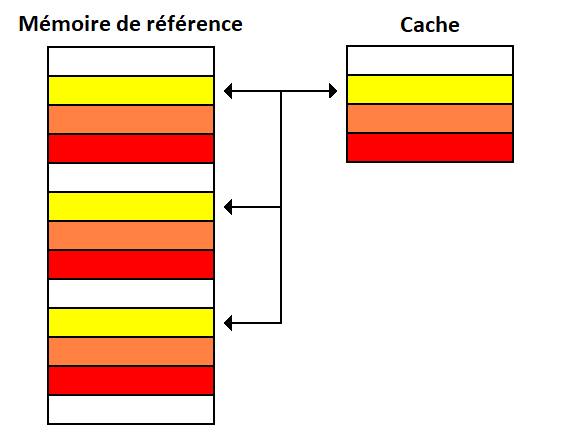
## Placement de l’information

Le choix de la façon d’associer le contenue du cache avec le fournisseur est primordiale, car les performances en dépendent.

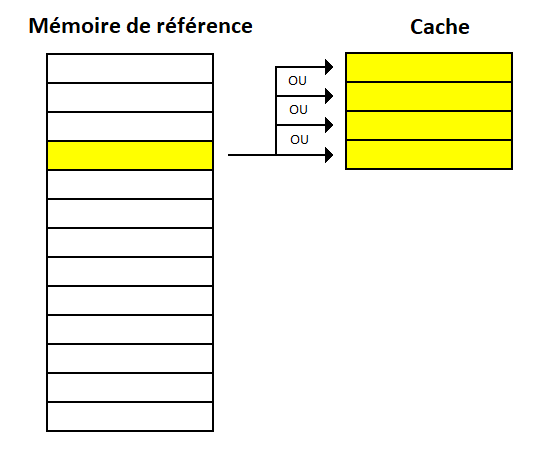
Pour l’heure il existe 4 types de caches :

* Directement adressés (direct mapped)
* Totalement associatifs (fully associative)
* Associatifs par voie (N-Way associative)
* Pseudo-associatifs

### Cache directement adressé



### Cache totalement associatif



### Cache associatif par voie

### Cache pseudo-associatif

## Politique de remplacement

Par définition une antémémoire est de petite taille. Par conséquent il va finir par arriver une situation où lors d’un cache miss il n’y ai plus de place pour accueillir la nouvelle ressource. L’antémémoire devra se débarrasser d’une des ressources préalablement mises en cache afin de la remplacer.

En fonction du contexte d’utilisation du cache le choix va se porter sur un algorithme ou un autre. Il en existe un certain nombre (FIFO, LRU, etc…), mais ce document ne traite pas ce sujet.

Il est à noter qu’on retrouve ces algorithmes dans d’autre utilisation que pour l’antémémoire, ils sont utilisés par le système d’exploitation pour gérer la pagination de la mémoire centrale par exemple.

## Gestion des mises à jour des données

## Dimensionnement

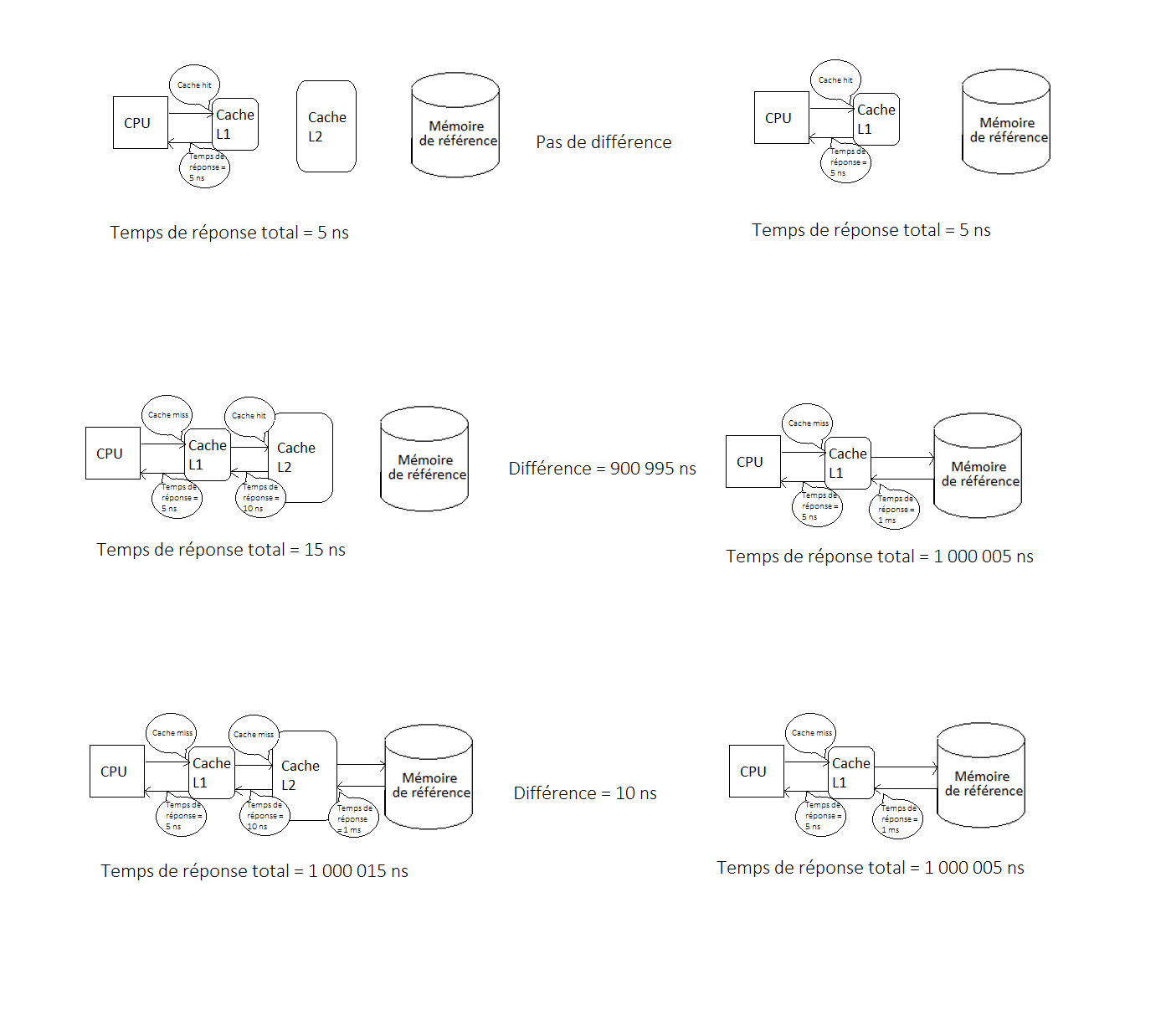
De manière très théorique et à relativement petite échelle, on peu dire que plus on augmente la taille d’une mémoire cache, plus l’accès aux ressources sera performant.

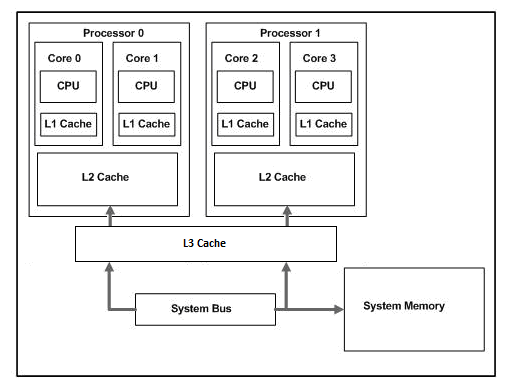
Toutefois cela devient faux à partir d’un certain point. Il existe une taille à partir de laquelle l’augmentation de la capacité de la mémoire cache deviens inutile. Cela réside dans le fait que la structure d’un programme est constituée de plusieurs embranchements qui ne sont pas prédictible pas le CPU. Chaque embranchement pointe sur une zone mémoire différente. Ainsi, l’optimisation par anticipation d’accès aux ressources (voir 4.1) deviens obsolètes.

## Hiérarchie des caches

Il est courant d’avoir plusieurs niveaux de caches.

En effet, la principale raison d’utiliser plusieurs niveaux réside dans la volonté d’augmenter la valeur moyenne du taux de succès des caches. Lorsque le la ressource n’est pas présente en cache, il faut la réclamer à la mémoire centrale qui prend plus de temps à répondre. Si l’on place un second niveaux, on se donne alors une secondes chance de pouvoir accéder à la ressource directement depuis l’antémémoire. Voir schéma comparatif ci-dessous :





# Références

<https://fr.wikipedia.org/wiki/Static_Random_Access_Memory>

<https://fr.wikipedia.org/wiki/M%C3%A9moire_cache>

<https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_m%C3%A9moires_cache>