UNIVERSITÉ DE MONTRÉAL

STOCKAGE SUR DISQUE POUR ACCÈS RAPIDE D'ATTRIBUTS AVEC INTERVALLES DE TEMPS

ALEXANDRE MONTPLAISIR-GONÇALVES
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2011

© Alexandre Montplaisir-Gonçalves, 2011.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

STOCKAGE SUR DISQUE POUR ACCÈS RAPIDE D'ATTRIBUTS AVEC INTERVALLES DE TEMPS

présenté par : MONTPLAISIR-GONÇALVES Alexandre
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. OZELL Benoit, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. QUINTERO Alejandro, Doct., membre.

À ma tante Carmen,
qui m'a toujours supporté et encouragé
et qui le fait encore aujourd'hui.

Remerciements

Le travail présenté dans ce mémoire n'aurait pu être rendu possible sans l'aide et le soutien de plusieurs personnes et organisations. J'aimerais donc prendre quelques moments pour les remercier.

Tout d'abord, merci à Michel Dagenais, mon directeur de recherche, non seulement pour le soutien et la guidance qu'il m'a offerts tout au long de mon projet, mais aussi pour m'avoir donné l'opportunité de le faire. Michel m'a épaulé et a pu me guider à travers la montagne de nouvelle information, pour cibler ce qui était nécessaire et pertinent pour le projet. Si j'en ai appris autant durant ces deux années, c'est avant tout grâce à lui.

Ensuite, merci à Ericsson pour le financement de ce projet de recherche. Au-delà du financement, j'ai pu constaté un intérêt évident pour les résultats de mon travail. La grande proximité entre mon projet et les logiciels qu'ils mettent eux-mêmes directement en production a permis de voir un effet très rapide et très concret de son utilité. Cela était particulièrement motivant, d'autant plus que ce ne sont pas tous les projets de recherche qui bénéficient d'une telle situation.

De plus, merci à tous mes collègues du laboratoire DORSAL, autant à mes “seniors”, Mathieu, Yannick et Matthew, qui m'ont apporté une aide précieuse à plusieurs reprise, mais également à mes pairs, qui m'ont eux aussi beaucoup aidé et beaucoup appris, et grâce à qui l'ambiance était très agréable. Merci à David, Julien, Michael, et tous les autres. Je ne serai pas près d'oublier les meetings stratégiques au DDC !

Enfin, merci à vous membres du jury et autres lecteurs, pour l'intérêt porté à mon projet et à ce présent mémoire. J'espère pouvoir le rendre aussi intéressant pour vous qu'il l'a été pour moi !

Résumé

En informatique, le traçage est une méthode permettant de recueillir de l'information sur le comportement d'un système ou d'un programme donné. De manière très générale, le concept est d'ajouter des points de trace dans le code du programme qui, lorsqu'ils seront exécutés, notifieront le traceur pour dire “je suis rendu à X” ou “je viens de terminer Y”, etc.

Le traceur LTTng (*Linux Tracing Toolkit Next Generation*) a été developpé au laboratoire DORSAL de l'École Polytechnique de Montréal. Ce traceur s'intègre au noyau Linux et permet d'en extraire des “traces noyau”. Une variante du traceur appelée UST (*UserSpace Tracer*) a également été développée pour s'intégrer plutôt à des applications utilisateur. On parle dans ce cas de “traces utilisateur”.

Comme on peut s'en douter, la quantité d'information provenant d'un système tracé peut être très importante, surtout dans le cas de traces noyau. C'est tout un défi de simplement extraire cette information sans affecter la performance du système, mais c'est également un défi que d'analyser toute cette information.

Il existe plusieurs outils d'analyse de traces. Parmi ceux-ci on retrouve les visualiseurs, qui permettent de donner une représentation graphique de l'information contenue dans une traces à l'aide de différentes vues. Un des besoins des visualiseurs de traces noyau est de pouvoir recréer l'état complet du système (table des processus, table des fichiers ouverts, etc.) tel qu'il était à n'importe quel point dans la trace.

Pour ce faire, les visualiseurs de traces LTTV (*Linux Tracing Toolkit Viewer*) et TMF (*Tracing and Monitoring Framework*), développés par le laboratoire DORSAL et ses partenaires, utilisent une méthode de clichés de l'état. Cette méthode consiste à sauvegarder en mémoire plusieurs clichés, à intervalles réguliers, qui contiennent l'état complet du système à ces moments donnés. Lorsqu'une requête est faite pour connaître l'état à un temps t quelconque dans la trace, le cliché précédent le plus proche est rechargé. On relit ensuite les événements de le trace en mettant à jour l'état, jusqu'au moment t, obtenant ainsi l'état voulu.

Cette méthode fonctionne bien pour les traces de petites tailles, mais rencontre certains problèmes à plus grande échelle. Par exemple, sur des gros systèmes avec beaucoup de processus actifs, plus d'événements seront générés par seconde, donc des clichés auront à être faits plus souvent (pour conserver le nombre d'événements entre les points, et par le fait même les temps de requêtes, constants). Chacun de ces clichés contiendra également plus

d'information, ce qui cause un double problème d'espace.

Il existe donc un besoin de concevoir une nouvelle méthode de stockage de cette information d'état, qui puisse être plus efficace en terme d'espace utilisé, sans trop sacrifier de performance. De plus, contrairement à la méthode des clichés, le stockage devrait être fait sur disque de manière à pouvoir supporter des traces de très grande taille.

Dans ce mémoire, nous proposons un nouveau système d'état qui permet de définir les changements d'états de manière générique, stocke son information sur disque et conserve un temps de réponse $\mathcal{O}(\log n)$ pour les requêtes. De plus, nous ne stockons pas d'information redondante.

Pour ce faire, chaque état à être enregistré dans l'historique est représenté sous forme d'un intervalle, avec ses bornes égales aux temps de début et de fin de l'état. Les intervalles sont ensuite stockés sur disque, dans une structure spécialement conçue pour cet usage, que nous avons nommé *arbre à historique*.

L'arbre à historique est une structure de données pour intervalles, optimisée pour l'accès sur disque. Les intervalles sont regroupés par noeuds, qui sont de taille constante sur le disque, et ce sont ces noeuds qui sont agencés en arbre.

Le fait que les événements dans une trace soient automatiquement ordonnés chronologiquement garantit que les intervalles d'état seront triés par ordre croissant de temps de fin. Grâce à cette hypothèse, nous pouvons construire un arbre à historique de manière incrémentale. Ceci permet à la construction d'être arrêtée et reprise à tout moment, ce qui augmente la flexibilité : nous pouvons par exemple procéder à cette construction seulement lorsque le système est peu utilisé.

Il est même possible de lancer des requêtes à l'arbre pendant que celui-ci est en construction. Cette propriété serait très utile pour l'intégration aux visualiseurs de traces, puisqu'elle permettrait à un utilisateur de commencer à utiliser l'information d'état qui a déjà été traitée, sans avoir à attendre que l'historique complet ne soit terminé.

Enfin, nous avons testé les performances de l'arbre à historique comparées à celles du travail équivalent effectué par LTTV. Nous avons également testé le comportement du système en utilisant des méthodes de stockages plus générales et mieux connues, comme les *R-Trees* et les bases de données spatiales.

Le système d'état de LTTV est très performant, par contre, tel que mentionné plus tôt, il stocke de l'information redondante, est limité par l'espace mémoire, et est difficile à modifier

ou étendre. Le nouveau système d'état générique est habituellement plus lent, mais capable de rester à l'intérieur d'un facteur de deux par rapport à LTTV.

L'arbre à historique présente cependant un grand avantage par rapport aux structures commes les *R-Trees* et les bases de données, principalement dû au fait que celui-ci n'a pas à être constamment rebalancé. Les temps de construction d'un historique basé sur ces dernières sont beaucoup trop grands pour être utilisable dans un visualiseur de traces.

Il semble donc que l'arbre à historique remplit bien sa fonction de structure de stockage d'information d'état venant de traces, et offre un avantage apparent par rapport aux structures de données similaires existantes. En plus, la généricité du système proposé permettra de facilement créer de nouveaux système d'états ou d'en étendre des déjà existants. Cette flexibilité sera très utile pour définir les systèmes d'états d'applications utilisateur, provenant de traces provenant de programmes utilisateurs par exemple.

Abstract

The tracing of computer systems allows programmers and users to extract precious behavioral data about their systems or applications. From a very high-level point of view, the concept of tracing relies on inserting trace points, or probes, at specific places in a program's code. Whenever execution reaches those points, the application will notify the tracer, to say "I just got to point X", "I just finished Y" or anything similar.

The LTTng tracer (Linux Tracing Toolkit Next Generation) was developped by the DORSAL lab in the École Polytechnique de Montréal. This tracer integrates with the Linux kernel and allows taking kernel traces. A variant called UST (UserSpace tracer) allows instrumenting and tracing userspace applications. Traces coming from userspace applications are called, simply, userspace traces.

As one could expect, the amount of information coming from a running system can be quite big, especially in the case of kernel traces. Extracting and saving this information without affecting the running system is already quite a challenge, but analyzing this huge amount of information is also a challenge by itself.

There are many tools available to analyse such traces. Among those are trace viewers, which allow seeing a graphical representation of the different elements in a trace. In the case of kernel trace viewers, it is useful to be able to reproduce the state of the traced system, like the process table, the open files, etc. at any point in the trace.

To do so, viewers like LTTV (Linux Tracing Toolkit Viewer) or TMF (Tracing and Monitoring Framework, an Eclipse plugin part of Linux Tools), use a checkpoint method. This method consists in saving, in memory and at regular intervals, complete dumps of the system state. We call those dumps the "checkpoints". Whenever a request to regenerate the state at an arbitrary time t is sent, the state system will reload the previous checkpoint, seek to the corresponding location in the trace, then replay the events and update its temporary state until we reach time t. Using such a method avoids having to start from the beginning of the trace every time.

The checkpoint method works well enough for small traces, but is problematic at larger scales. The number of events between checkpoints is kept constant, to keep the average time for requests constant too. This means that the number of checkpoints will increase with the number of events in the trace. In addition, the size of each checkpoint will increase if there are more processes running and more stuff going on at the same time. And since these checkpoints are stored in RAM, there is a hard limit on the size of the trace that can be

analyzed. We could, up to a certain point, reduce the amount of checkpoints, but that would degrade the performance of the state system.

There is clearly a need for a new method of state storage, which should allow opening and analyzing traces up to the terabyte range. It is obvious that whatever information we store about the state will have to be written to disk (since the amount of state information increases linearly with the size of the trace but we can't, unfortunately, expect RAM sizes to increase linearly with disk space).

In this thesis, we propose a new state system for trace viewers, which diminishes the weaknesses presented previously. This new system also provides a generic way of defining state changes (the current kernel state systems are hard-coded into the viewers). It stores its information on disk but maintains a reasonable response time, which scales with $\mathcal{O}(\log n)$. It also avoids storing any redundant information.

To do so, every state is saved as an interval of time. The start and end of the interval correspond to the start time and end time of the state. These intervals are then stored on disk, inside a container specifically designed for this task. We have called this new container the History Tree.

The History Tree is a data structure for intervals optimized for disk access. The intervals are stored in nodes, which occupy a pre-defined constant amount of space on disk. It's those nodes which are arranged in a tree structure.

Since the events in the trace are sorted chronologically, the corresponding states are created and inserted in the History Tree by ascending order of end times. The History Tree has been specifically tailored to receive intervals ordered in such a way. For example, it does not need rebalancing: when a node is full, it is marked as done and a sibling node is created. This allows for incremental construction, which can even be stopped and resumed at any time.

It is also very possible to query the Tree while it's being built. This feature will be very interesting for the integration in trace viewers : the users could start working with the data that has already been parsed, without having to wait for the entire history to be built.

We've then compared the performance of the History Tree to the one of LTTV. We have also tested the new state system using more generic data structures, like R-Trees or spatial databases, as a backend instead of the History Tree.

It was hard to beat the performance of LTTV's hard-coded state system. However, the new generic state system using the History Tree could usually remain within a factor of two of LTTV's performance when comparing history construction times.

The History Tree however proved much better suited at handling our state intervals than the preexisting data structures like the R-Tree and the database. While the History Tree was designed around the fact that the intervals are received by ascending order of end times, the R-Trees had to be constantly rebalanced to cope with it. This resulted in very long history construction times, which made them inappropriate for use in trace viewers.

Finally, we can say that the History Tree is an efficient data structure for storing state intervals coming from kernel traces. It also offers obvious advantages over well-known preexisting data structures like R-Trees, at least in our particular use cases. The genericity of the new proposed framework will also allow users to easily extend existing state systems, as well as create new ones. This will be useful when defining custom state systems for userspace applications, for example.

Table des matières

Dédicace iii

Remerciements iv

Résumé v

Abstract viii

Table des matières xi

Liste des tableaux xiv

Liste des figures xv

Liste des annexes xviii

Liste des sigles et abréviations xix

Chapitre 1 INTRODUCTION 1
- 1.1 Définitions et concepts de base 1
  - 1.1.1 Événement 1
  - 1.1.2 Intervalle d'état et Valeur d'état 2
  - 1.1.3 Attribut 2
  - 1.1.4 Changement d'état 2
  - 1.1.5 État du système ou État courant 3
- 1.2 Éléments de la problématique 4
  - 1.2.1 Définir les systèmes d'état de manière générique 4
  - 1.2.2 Stocker l'information d'état sur disque de manière efficace 5
- 1.3 Objectifs de recherche 6
- 1.4 Plan du mémoire 6

Chapitre 2 REVUE DE LITTÉRATURE 7
- 2.1 Le traçage sur Linux 8
  - 2.1.1 Le traçage au niveau noyau 8
  - 2.1.2 Le traçage d'applications utilisateur 9

2.2 Modélisation de l'état d'un système 10
2.2.1 Méthode des clichés d'état 10
2.2.2 Stocker l'information d'état dans la trace 12
2.3 Le stockage d'intervalles 14
2.3.1 Le *segment tree* 14
2.3.2 La famille des R-Trees 15
2.3.3 La famille des B-Trees 18
2.3.4 Structure hybrides 20

Chapitre 3 DÉTAILS DE LA SOLUTION 23
3.1 Arbre à historique 23
3.1.1 Intervalles 24
3.1.2 Noeuds de l'arbre 24
3.1.3 Constantes de configuration 25
3.1.4 Construction de l'arbre 26
3.1.5 Fermeture de l'arbre 29
3.1.6 Requêtes 29
3.2 Système d'état en général 32
3.2.1 Gestionnaire d'événements 33
3.2.2 Arbre d'attributs 34
3.2.3 État transitoire 37
3.2.4 Interface de stockage 38
3.3 Historique partiel 40

Chapitre 4 RÉSULTATS THÉORIQUES ET EXPÉRIMENTAUX 45
4.1 Traces de références 46
4.2 Description et méthodologie des tests 47
4.2.1 Environnement de test 47
4.2.2 Construction 47
4.2.3 Ouverture d'un historique existant 48
4.2.4 Requêtes complètes 48
4.2.5 Requêtes ponctuelles 49
4.2.6 Taille du fichier sur disque 49
4.2.7 Profondeur de l'arbre 50
4.3 Variantes de configuration 50
4.3.1 Taille des blocs sur disque 50
4.3.2 Nombre d'enfants maximal par noeud 54

4.4 Optimisations . . . 57
4.4.1 Placement des attributs en arbre . . . 57
4.4.2 Éviter la création d'objets Java . . . 59
4.4.3 Éviter les attributs redondants . . . 60
4.4.4 Séparer le système en différents fils d'exécution . . . 63
4.4.5 Lire les événements directement sans JNI (format VSTF) . . . 64
4.4.6 Conserver des références aux attributs entre les événements . . . 67
4.4.7 Trier les intervalles dans les noeuds . . . 69
4.4.8 Trier les noeuds de l'arbre dans le fichier d'historique . . . 71
4.4.9 Taille de blocs non-multiple de 4KB . . . 74
4.5 Historique partiel . . . 83
4.6 Comparaisons avec d'autres méthodes de stockage . . . 88
4.6.1 *R-Tree* en mémoire . . . 89
4.6.2 Base de données spatiale (PostgreSQL avec PostGIS) . . . 91
4.7 Tests à grande échelle avec le format CTF . . . 95

Chapitre 5 CONCLUSION . . . 102
5.1 Synthèse des travaux . . . 102
5.2 Limitations de la solution proposée . . . 103
5.3 Améliorations futures . . . 104
5.3.1 Noeuds étendus . . . 104
5.3.2 Stockage hybride . . . 106

Références . . . 108

Annexes . . . 110
A.1 En-tête du fichier d'historique . . . 110
A.2 Blocs . . . 111
A.2.1 En-tête d'un bloc . . . 111
A.2.2 Section “Data” . . . 112
A.2.3 Section “Strings” . . . 113
A.3 Arbre d'attributs . . . 114
B.1 En-tête du fichier de trace . . . 116
B.2 Événements . . . 116

Liste des tableaux

Tableau 3.1 Méthodes pour décrire un changement d'état 33
Tableau 3.2 Exemples d'attributs 35
Tableau 4.1 Traces de référence LTTng 46
Tableau 4.2 Spécifications de la machine de test 47
Tableau 4.3 Traces CTF 96

Liste des figures

FIGURE 1.1 Relation entre événements, changements d'état et états 3
FIGURE 2.1 Les points de sauvegarde de l'état dans une trace 11
FIGURE 2.2 Une requête au système d'état 12
FIGURE 2.3 Un exemple de *segment tree* 14
FIGURE 2.4 Division des noeuds dans un R-Tree 16
FIGURE 2.5 Division des noeuds dans un R*Tree 17
FIGURE 2.6 Insertions successives dans un B-Tree 19
FIGURE 3.1 Premier noeud vide 26
FIGURE 3.2 Premier noeud rempli 27
FIGURE 3.3 Ajout d'une nouvelle racine et d'un noeud frère 27
FIGURE 3.4 Premier frère est rempli, ajout d'un autre 27
FIGURE 3.5 Noeud racine rempli, ajout d'une nouvelle racine et nouvelle branche 28
FIGURE 3.6 Historique complété, requête au temps $t = 300$ 30
FIGURE 3.7 Système d'état lors de la construction de l'historique 32
FIGURE 3.8 Des intervalles composant un historique 41
FIGURE 3.9 Retrait des intervalles qui n'intersectent pas les points de sauvegarde 41
FIGURE 3.10 Requête sur un historique partiel 43
FIGURE 4.1 Comparaison des tailles de blocs, temps de création de l'historique . . 51
FIGURE 4.2 Comparaison des tailles de blocs, taille de l'historique résultant . . . 52
FIGURE 4.3 Comparaison des tailles de blocs, requêtes complètes 52
FIGURE 4.4 Comparaison des tailles de blocs, requêtes ponctuelles 53
FIGURE 4.5 Comparaison du nombre max. d'enfants par noeud,
temps de création de l'historique 54
FIGURE 4.6 Comparaison du nombre max. d'enfants par noeud,
taille de l'historique résultant 55
FIGURE 4.7 Comparaison du nombre max. d'enfants par noeud,
requêtes complètes 56
FIGURE 4.8 Comparaison du nombre max. d'enfants par noeud,
requêtes ponctuelles 56
FIGURE 4.9 Organisation des attributs en arbre, construction de l'historique d'état 58
FIGURE 4.10 Organisation des attributs en arbre, système d'état sans historique . . 59
FIGURE 4.11 Impact d'allouer ou non des objets Java à chaque insertion 60

FIGURE 4.12 Retrait des attributs redondants, temps de construction 62
FIGURE 4.13 Retrait des attributs redondants, taille de l'historique résultant 62
FIGURE 4.14 Traitement et insertion des intervalles dans un fil d'exécution séparé . 64
FIGURE 4.15 Utilisation du lecteur VSTF en Java vs. lecteur LTTng via JNI Construction de l'historique d'état 65
FIGURE 4.16 Utilisation du lecteur VSTF en Java vs. lecteur LTTng via JNI État transitoire sans historique 66
FIGURE 4.17 Comparaison du lecteur VSTF vs. LTTV 67
FIGURE 4.18 Conservation de références aux attributs entre les événements (méthode JNI) 68
FIGURE 4.19 Impact de trier les intervalles dans les noeuds, construction de l'historique 69
FIGURE 4.20 Impact de trier les intervalles dans les noeuds, requêtes complètes . . 70
FIGURE 4.21 Impact de trier les intervalles dans les noeuds, requêtes ponctuelles . 71
FIGURE 4.22 Tri des blocs (noeuds) dans le fichier, construction de l'historique . . 72
FIGURE 4.23 Tri des blocs (noeuds) dans le fichier, requêtes complètes 73
FIGURE 4.24 Tri des blocs (noeuds) dans le fichier, requêtes ponctuelles 73
FIGURE 4.25 Blocs désalignés, construction de l'historique 74
FIGURE 4.26 Blocs désalignés, requêtes complètes 75
FIGURE 4.27 Blocs désalignés, requêtes ponctuelles 76
FIGURE 4.28 Blocs désalignés, profondeur de l'arbre à historique 76
FIGURE 4.29 Blocs désalignés, construction de l'historique 77
FIGURE 4.30 Blocs désalignés, requêtes complètes 78
FIGURE 4.31 Blocs désalignés, requêtes ponctuelles 78
FIGURE 4.32 Blocs désalignés, profondeur de l'arbre à historique 79
FIGURE 4.33 Blocs désalignés, construction de l'historique 80
FIGURE 4.34 Blocs désalignés, requêtes complètes 81
FIGURE 4.35 Blocs désalignés, requêtes ponctuelles 82
FIGURE 4.36 Blocs désalignés, profondeur de l'arbre à historique 82
FIGURE 4.37 Historique partiel, temps de construction 84
FIGURE 4.38 Historique partiel, taille du fichier d'historique 85
FIGURE 4.39 Historique partiel, ouverture d'un arbre déjà construit 86
FIGURE 4.40 Historique partiel, requêtes complètes 87
FIGURE 4.41 Historique partiel, profondeur de l'arbre à historique 88
FIGURE 4.42 Historique dans un R-Tree en mémoire vive, construction de l'historique 89
FIGURE 4.43 Historique dans un R-Tree en mémoire vive, requêtes complètes . . . 90
FIGURE 4.44 Historique dans un R-Tree en mémoire vive, requêtes ponctuelles . . 91

FIGURE 4.45 Historique dans PostgreSQL/PostGIS, construction de l'historique . . 92
FIGURE 4.46 Historique dans PostgreSQL/PostGIS, taille de l'historique sur disque 93
FIGURE 4.47 Historique dans PostgreSQL/PostGIS, requêtes complètes 94
FIGURE 4.48 Historique dans PostgreSQL/PostGIS, requêtes ponctuelles 95
FIGURE 4.49 Traces CTF, temps de construction 97
FIGURE 4.50 Traces CTF, taille du fichier d'historique 98
FIGURE 4.51 Traces CTF, requêtes complètes 99
FIGURE 4.52 Traces CTF, requêtes ponctuelles 99
FIGURE 4.53 Traces CTF, profondeur de l'arbre à historique 100
FIGURE 5.1 Comportement normal 105
FIGURE 5.2 Utilisation de noeuds étendus 105

Liste des annexes

Annexe A Format de fichier sur disque de l'arbre à historique 110

Annexe B Format de trace VSTF 116

Liste des sigles et abréviations

COW	Copy-on-Write
CTF	Common Trace Format
JNI	Java Native Interface
KVM	Kernel-based Virtual Machine
LTTng	Linux Tracing Toolkit Next Generation
LTTV	Linux Tracing Toolkit Viewer
MPI	Message Passing Interface
PID	Process ID
PPID	Parent's PID
RDBMS	Relational Database Management System
TGID	Thread Group ID
TMF	Tracing and Monitoring Framework
UST	Userspace Tracer
VSTF	Very Simple Trace Format

Chapitre 1

INTRODUCTION

Les travaux de recherche présentés dans ce mémoire se penchent sur la question du stockage de l'information d'état dans les visualiseurs de traces de systèmes informatiques.

Présentement, les visualiseurs de traces conservent habituellement leur information d'état en mémoire. Ceci limite donc la taille des traces pouvant être ouvertes et analysées. De plus, les systèmes d'états sont définis à même le code du programme, ce qui rend les modifications difficiles.

Nous présentons ici un nouveau système d'état pour les visualiseurs de traces, qui tente de pallier au problème d'espace mémoire en stockant son information sur disque. De plus, puisque nous étions à revoir le système en entier, nous nous sommes assurés que le nouveau système soit générique pour pouvoir facilement le modifier et l'étendre par la suite.

1.1 Définitions et concepts de base

Cette section présente différents concepts et définitions qui seront utilisés tout au long du mémoire. Des définitions additionnelles qui seront apportées dans le cadre du présent travail seront présentées au chapitre 3.

1.1.1 Événement

Dans une trace, un événement est l'enregistrement d'une action qui a eu lieu dans le système ou l'application, à un instant donné. Un événement est *ponctuel*, ce qui veut dire qu'il n'a aucune durée.

Dans les traces noyau, les événements sont habituellement définis par une estampille temporelle, un type d'événement (comme "appel système" ou "faute de page", etc.) et enfin une charge utile. Cette charge utile contient de l'information additionnelle par rapport à l'événement et sa structure va varier en fonction du type d'événement.

Les traces que nous utiliserons contiennent seulement des événements ; d'autres types de traces peuvent directement contenir des états par exemple, mais l'étude ici se concentre sur des traces ne contenant que des événements ponctuels. Les traces sont donc vues exclusivement comme des sources d'événements.

1.1.2 Intervalle d'état et Valeur d'état

Contrairement à un événement, un *état* a une durée dans le temps. Il possède donc deux estampilles temporelles, un temps de début et un temps de fin. Puisque le terme "état" revient assez souvent, on peut parler d'un *intervalle d'état* pour être plus précis.

1.1.3 Attribut

Dans un modèle de l'état d'un système, un attribut est la plus petite unité du modèle qui puisse être dans un état indépendant.

Cela signifie qu'un attribut ne peut être que dans un seul état à la fois. Si dans un modèle, un attribut pouvait avoir deux valeurs d'état différentes en même temps, cela signifie qu'il s'agit en fait de deux attributs différents, et le modèle devrait être révisé.

1.1.4 Changement d'état

Le *changement d'état* est le lien entre un événement et un état. Un changement d'état est défini par trois composantes : un temps, un attribut et une valeur d'état. Nous séparons la définition de changement d'état à celle d'un simple événement, puisqu'il est possible pour un seul événement de causer plusieurs changements d'état au modèle. Similairement, il est possible qu'un événement n'affecte pas l'état du tout, il ne causera donc aucun changement d'état.

L'estampille temporelle du changement d'état est normalement égale à l'estampille de l'événement qui l'a causé. L'attribut du changement d'état indique quel attribut doit être modifié dans le modèle. Habituellement, cet attribut sera fonction de l'information contenue dans l'événement.

Enfin la valeur d'état du changement indique la *nouvelle* valeur que l'attribut devra prendre dans notre modèle. Encore une fois, nous nous basons habituellement sur l'information contenue dans l'événement.

La figure 1.1 montre comment certains événements seulement vont causer des changements d'état, et comment l'attribut affecté va passer d'un état à un autre.

Par exemple, un traceur nous envoie un événement lors des appels systèmes, comme `exec()`. Nous voulons associer à l'événement `exec()` un changement d'état pour mettre à jour un attribut représentant le nom d'éxécutable du processus en question. Nous irons lire dans la charge utile de l'événement l'argument du `exec()` correspondant au nouvel

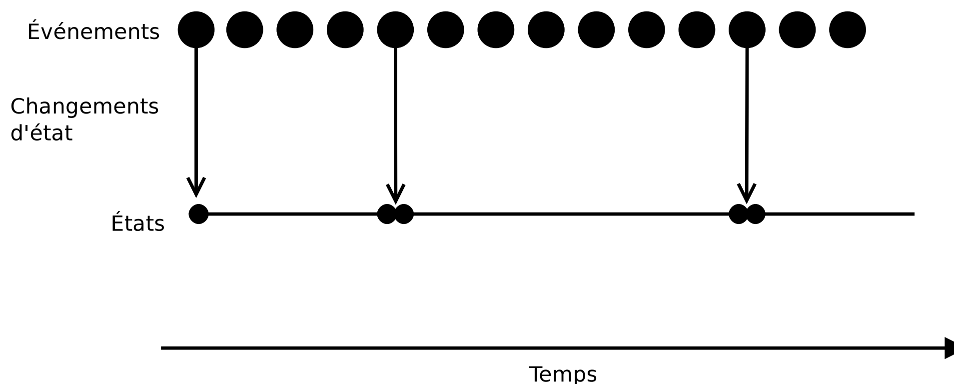

FIGURE 1.1: Relation entre événements, changements d'état et états

exécutable que le processus va rouler. Nous assignerons ensuite ce nom à la valeur d'état du changement, et ce changement sera enregistré dans le modèle.

1.1.5 État du système ou État courant

Lorsqu'on parle d'intervalle d'état, on parle de l'état unitaire d'un seul attribut particulier, à un moment dans le temps. On peut aussi plutôt parler de *l'état du système*, ou l'état courant (du système). Ceci réfère à l'état complet du modèle que nous nous sommes fait de notre système, ou de notre application, tracé *à un moment précis dans le temps*.

Plus spécifiquement, l'état courant correspond à l'ensemble des valeurs d'état de tous les attributs du modèle, à un même temps donné.

Si, par exemple, un utilisateur clique dans une vue sur l'instant t = 10 s, le visualiseur demandera à son système d'état de recharger "l'état courant" du système tel qu'il était à t = 10 s pour pouvoir présenter à l'utilisateur les informations pertinentes.

L'état courant est donc le résultat des requêtes faites au système d'état.

1.2 Éléments de la problématique

Dans cette section nous identifierons les lacunes du système d'état des visualiseurs de traces actuels. Il faudra s'assurer que la solution proposée au chapitre 3 pallie à chacune de ces lacunes, ou au moins en réduise les effets.

1.2.1 Définir les systèmes d'état de manière générique

Présentement, les systèmes d'état des visualiseurs de traces étudiés sont définis à même le code du programme. Les attributs et changements d'états sont définis, par exemple, par des `struct` en C. De plus, ces systèmes sont peu flexibles : ils ne supportent habituellement que le noyau (ou l'application) pour lequel ils ont été créés.

Avec la popularité grandissante des traceurs d'applications utilisateurs, il deviendra bientôt très souhaitable de pouvoir créer facilement des systèmes d'état spécifiques à certaines applications. Bien entendu, le visualiseur de traces ne connaîtra pas le fonctionnement interne de chaque application tracée, et il serait illogique de coder le support pour chaque application instrumentée dans le visualiseur lui-même !

Au-delà du support pour des applications arbitraires, la simple maintenance d'un visualiseur ou lecteur de traces serait rendue plus simple à l'aide d'un système d'état générique. Si, par exemple, un nouvel appel système était ajouté (ou encore un nouveau point de trace à un appel existant), il ne serait pas nécessaire de modifier plusieurs structures dans le code du programme.

Il existe donc un besoin apparent de découpler le "stockage de l'information d'état" de la "définition du système d'état". Le premier point est évidemment la responsabilité du visualiseur ou du lecteur de traces. Cependant, le deuxième point relève plutôt du programmeur qui va instrumenter son application. Très souvent c'est le même programmeur qui a codé l'application qui va l'instrumenter : qui d'autre connaît mieux les changements et états internes à son programme ?

Pour pouvoir découpler le stockage de la définition du système d'état, il faudra pouvoir définir les attributs et changements d'états de manière générique. De cette façon, le système d'état du visualiseur pourra utiliser les changements d'états conçus par quelqu'un connaissant très bien son application. Cette personne n'aura jamais à regarder le code source du visualiseur comme tel.

Il serait même théoriquement possible de définir le système d'état à part, dans un fichier XML ou autre, ce qui permettrait de faire des modifications au gestionnaire d'état sans même avoir à recompiler le visualiseur.

1.2.2 Stocker l'information d'état sur disque de manière efficace

Le point précédent discutait de la séparation du stockage et de la définition du système d'état. Cependant, l'étape de stockage pourrait elle aussi bénéficier de quelques améliorations.

Comme nous le verrons au chapitre suivant, il est monnaie courante pour les visualiseurs de traces de conserver en mémoire leur information d'état. Lorsqu'ils lisent des événements d'une trace, ils mettent à jour cet état. Ils doivent également pouvoir répondre aux requêtes des différentes vues, donc pouvoir reconstruire le modèle de l'état à n'importe quel point de la trace.

Comme la quantité d'information d'état augmente proportionnellement avec la taille de la trace, le fait de stocker cette information en mémoire peut limiter assez sévèrement la taille des traces qui peuvent être analysées. L'idéal serait de stocker cette information sur disque. Le stockage sur disque donne bien entendu des temps de réponse et des débits d'information beaucoup plus lents que du stockage en mémoire. Il faudra voir comment agencer l'information de manière à ce que les requêtes restent efficaces.

1.3 Objectifs de recherche

Il est maintenant possible de préciser les objectifs de rercherche :

1. Proposer et concevoir une organisation pour le stockage de l'état sur disque.
2. Implémenter un système d'état générique utilisant cette structure. Connecter cette implémentation à des librairies de lecture de traces connues.
3. Comparer les performances (vitesse, utilisation d'espace) avec la méthode précédente. Comparer entre elles les différentes variantes du nouveau système d'état.
4. Comparer la structure de stockage proposée avec des structures génériques déjà existantes, pour démontrer l'intérêt d'utiliser cette nouvelle structure dans notre cas d'utilisation particulier.

Il faudra également générer des traces, idéalement représentatives, et de différentes tailles, avec lesquelles nous pourrons exécuter les tests de performance.

1.4 Plan du mémoire

Au chapitre 2, nous présenterons une revue de littérature faisant le point sur l'état de l'art dans le domaine du traçage, de la modélisation d'états de systèmes, ainsi que du stockage d'intervalles dans des structures de données variées.

Ensuite, le chapitre 3 présentera en détails la solution proposée aux problèmes identifiés précédemment avec les systèmes d'état des visualiseurs de traces.

Le chapitre 4 présentera les paramètres et les résultats des divers tests de performances effectués. Ceux-ci sont classés par catégorie (tests de variantes du système de base entres elles, puis tests par rapport à d'autres structures de données existantes).

Suivra enfin une conclusion, qui fera la synthèse des travaux, et qui identifiera quelques possibilités d'améliorations futures.

Chapitre 2

REVUE DE LITTÉRATURE

Dans ce chapitre, nous ferons une revue de littérature sur trois sujets reliés à la problématique qui a été identifiée précédemment.

Le premier sujet sera le traçage, à savoir ce qui se fait dans le domaine de la prise de traces et de la génération d'événements.

Le second sujet sera la modélisation de l'état dans les sytèmes informatiques. Nous nous intéresserons principalement au fonctionnement d'autres logiciels developpés au laboratoire DORSAL, puisque ce sera principalement à partir de ceux-ci que le reste de la solution sera développée. Nous irons aussi explorer ce qui est fait ailleurs dans ce domaine, et verrons des exemples d'utilisation pratique des modèles d'état.

Enfin, nous nous intéresserons à des méthodes de stockage qui pourraient nous aider à enregistrer un historique d'état sur disque de manière efficace. Nous verrons comment l'utilisation d'intervalles de temps semble être une bonne approche, et comment il est possible de stocker de tels intervalles soit sur disque ou en mémoire.

Nous tenons également à préciser que tout le travail présenté ici, autant au niveau de la revue de littérature que pour la solution proposée, n'utilise que des algorithmes publiés et des logiciels libres. Le fait que de tels logiciels soient ouverts permet d'étudier le code source sans aucune restriction et de bien en comprendre le fonctionnement.

Nous sommes également d'avis qu'une implémentation libre de la solution que nous proposerons est la meilleure manière de garantir que celle-ci puisse être utilisée et intégrée à des programmes existants.

Il existe sans doute des solutions propriétaires non-publiées utilisant des concepts plus avancés ou plus performants que ceux que nous explorerons ici. Mais comme ils ne sont pas accessibles à la communauté scientifique en général, ils ne seront pas couverts par les présents travaux.

2.1 Le traçage sur Linux

En premier lieu, nous présenterons une revue des solutions de traçage existantes sur la plateforme Linux. L'ouverture de la plateforme, ainsi que les nombreuses avancées technologiques qui y sont constamment intégrées en font une cible de choix pour étudier les méthodes de traçage.

2.1.1 Le traçage au niveau noyau

En 2008, Jonathan Corbet a fait une revue des différentes solutions de traçage présentement offertes sur Linux, sur le site LWN dont il est l'éditeur en chef. Il présente SystemTap et Ftrace, ainsi que les API *Tracepoints* et *Tracehook*.

SystemTap est une solution développée et supportée principalement par Red Hat. À ce jour, cette solution est toujours activement développée et très utilisée, mais n'a jamais été intégrée officiellement à Linux.

Ftrace, pour *function tracer*, est une solution plus légère qui elle a été intégré officiellement. Jake Edge, un autre éditeur sur le même site, a fait une analyse plus détaillée de ftrace en 2009.

Les *tracehook* quant à eux n'ont pas retenu l'attention très longtemps. Il ont finalement été retirés du noyau en 2011 [1].

L'API des *tracepoints* a été utilisé comme remplacement aux anciens *kernel markers*. Ils avaient l'avantage d'être beaucoup plus performants. Ils ont éventuellement été remplacés à leur tour par la macro `TRACE_EVENT`. En 2010, Steven Rostedt a fait une analyse très appronfondie sur l'utilisation des `TRACE_EVENT`.

Un nouvel arrivant est apparu dans le noyau Linux en 2009. Il s'agit des compteurs de performances (*perfcounters*) et de l'outil *perf*. Edge en a fait une bonne explication, encore une fois. Cet outil est toujours supporté à ce jour.

En parallèle, Desnoyers et Dagenais ont présenté en 2006 le traceur noyau LTTng, suivi par des mises à jour sur son développement en 2008 et 2009. Une nouvelle version, LTTng 2.0, devrait être présentée en fin 2011.

Étant donné la nature ouverte de la plateforme Linux, il est très normal est même souhaitable que plusieurs solutions différentes soient offertes, même si elles peuvent sembler

1. `https://lwn.net/Articles/448136/`

mutuellement exclusives. Il est rare, en général, qu'une solution soit meilleure que toutes les autres pour tous les cas d'utilisation.

Par exemple, plusieurs systèmes de fichiers différents sont offerts. Dans certains cas, il peut exister plusieurs pilotes différents pour le même matériel. Si c'est l'espace occupé qui est un problème (par exemple, sur les plateformes embarquées), il est toujours possible de compiler son noyau seulement avec les options qui seront utilisées.

Dans le cas des solutions de traçage, elles peuvent habituellement utiliser une instrumentation commune du noyau (comme les *tracepoints* ou les `TRACE_EVENT`), ce qui évite d'alourdir le code et d'en compliquer la maintenance.

2.1.2 Le traçage d'applications utilisateur

Jusqu'à maintenant, nous avons vu des solutions de traçage pour le noyau Linux comme tel. Le besoin d'un traceur dans le noyau est plus apparent, puisqu'il est très difficile de déboguer celui-ci (on ne peut pas vraiment l'arrêter et l'exécuter pas-à-pas).

Par contre le traçage d'applications utilisateur est aussi très utile, et offre une alternative intéressante au déboguage et aux primitifs `printf()`.

Le système DTrace, présenté en 2004 par Cantrill *et al.*, est le traceur de référence sur le système d'exploitation Solaris. Celui-ci est reconnu pour ne poser pratiquement aucun impact de performance si le traçage est désactivé. Il offre également une interface unifiée pour le traçage du noyau et des applications utilisateur.

SystemTap supporte également le traçage d'applications. Ce dernier est d'ailleurs très similaire à DTrace, si bien que son interface d'instrumentation est compatible avec celle de DTrace. Cela offre un grand avantage côté interopérabilité. Par contre, tous deux utilisent le traceur noyau de leur système d'exploitation respectif.

Il est possible de tracer des applications utilisateur via un traceur noyau, à l'aide de points d'interruption et d'appels au code noyau. Cependant, cela a un impact non-négligeable sur la performance de l'application.

Une solution à ce problème a été proposée avec UST, le *Userspace Tracer* (Fournier *et al.*, 2009). Il s'agit d'un port de LTTng, qui s'exécute complètement en espace utilisateur. Une nouvelle version de ce traceur, qui inclut entre autres le support des points de trace SystemTap, est prévue avec la sortie imminente de LTTng 2.0.

2.2 Modélisation de l'état d'un système

Pour complémenter toutes les solutions de traçages identifiées à la section précédente, il existe une panoplie de visualiseurs de traces. Puisque les présents travaux de recherche s'intègrent au projet LTTng, nous nous intéresserons principalement aux visualiseurs qui supportent ce traceur.

Il en existe à ce jour deux, soit LTTV [2] (*Linux Trace Toolkit Viewer*) et TMF [3] (*Tracing and Monitoring Framework*, le plugin d'intégration de LTTng à Eclipse.)

Dans un premier temps, nous verrons comment ces deux visualiseurs génèrent et conservent leur information d'état. Nous explorerons ensuite une alternative, qui est de stocker l'information d'état directement dans la trace.

2.2.1 Méthode des clichés d'état

Comme nous l'avons vu dans l'introduction, il est utile pour les visualiseurs de traces de pouvoir modéliser l'état du système tracé. Certaines vues peuvent afficher de l'information provenant de ce modèle de l'état. Il est donc nécessaire pour le visualiseur de pouvoir regénerer l'état du système en tout point de la trace. Des collègues ont récemment montré (Giraldeau *et al.*, 2011) des exemples d'informations qui peuvent être tirées des traces noyau et du système d'état associé.

Les visualiseurs comme TMF et LTTV procurent un modèle d'état pour les traces noyau ayant été prises avec LTTng. La définition de ce modèle est faite à même le code du programme. Les attributs sont donc tous définis d'avance, et l'état courant modélisé est mis à jour par les événements de la trace via les changements d'états pré-programmés.

À partir d'un état initial vide, les visualiseurs peuvent donc lire les événements de la trace, en commençant au début, et mettre à jour leur état courant en mémoire. Il faut maintenant pouvoir recharger cet état, mais pour des points arbitraires de la trace, de manière à répondre aux requêtes demandées par les vues.

Une méthode inefficace serait de relire la trace depuis le début, jusqu'au temps cible de la requête, et de remettre à jour l'état en fonction des événements de la trace. Bien entendu, cette méthode serait beaucoup trop lente, et les requêtes plus près du temps de fin de la trace seraient injustement désavantagées par rapport aux requêtes près du début.

2. `http://lttng.org/lttv`
3. `http://www.eclipse.org/linuxtools/projectPages/lttng/`

Pour uniformiser et améliorer en général les temps d'accès, ces visualiseurs utilisent une méthode de clichés d'état, ou *snapshots*.

Lorsqu'une trace est ouverte, le système d'état passe en revue tous les événements de la trace, et met à jour son état (on parlera ici d'un *état transitoire*). À intervalles réguliers [4], une copie de l'état transitoire est faite et conservée en mémoire, avec la position dans la trace qui lui est associée. Ces copies ces sont les *clichés* de l'état. Ceci est fait jusqu'à ce qu'on arrive à la fin de la trace. Le visualiseur est ensuite prêt à être utilisé.

Les endroits où des clichés sont pris sont appelés *points de sauvegarde*. La différence entre "cliché" et "point de sauvegarde" est subtile [5], mais deviendra importante plus tard, lors de la présentation de la solution.

Lors d'une requête pour reconstruire l'état à un temps t arbitraire de la trace (il est bien rare que les requêtes tombent pile sur les points de sauvegarde !), le point de sauvegarde précédent le plus proche est chargé et le descripteur de lecture de la trace est positionné à l'endroit correspondant à ce dernier. Puis, les événements sont rejoués dans l'ordre en mettant à jour l'état transitoire, jusqu'à ce que nous atteignons un événement dont la valeur de temps est plus grand ou égal à t. Le contenu actuel de l'état transitoire est ensuite retourné comme réponse à la requête.

La figure 2.1 représente une trace et ses différents points de sauvegarde. La figure 2.2 présente les opérations qui sont effectuées pour répondre à une requête pour le temps cible indiqué par un X.

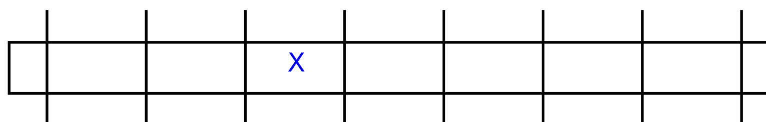

FIGURE 2.1: Les points de sauvegarde de l'état dans une trace

La méthode des clichés fonctionne bien avec les traces de petite taille (quelques dizaines ou centaines de MB), qui sont souvent utilisées. Cependant, le fait que toute l'information est stockée en mémoire peut limiter assez rapidement la taille des traces pouvant être analysées. Bien entendu, la quantité d'information d'état générée dépend beaucoup du contenu de la trace. Des traces plus "denses", contenant plus d'événements par seconde, requerront des points de sauvegarde plus fréquents. Similairement, des traces possédant beaucoup

4. Pour donner un ordre de grandeur, la valeur par défaut est de 50 000 événements.
5. Un cliché est une copie de l'état complet faite en mémoire. Un point de sauvegarde est simplement un endroit dans la trace où on peut recharger rapidement l'état.

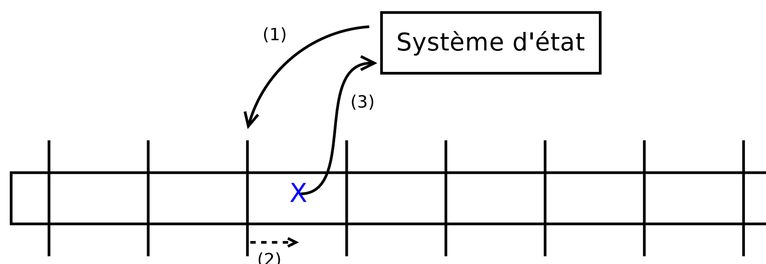

Figure 2.2: Une requête au système d'état

(1) Charger le cliché précédent le plus proche
(2) Rejouer les événements de la trace et mettre à jour l'état transitoire
(3) Retourner l'état actuel comme réponse à la requête

d'information (beaucoup de processus actifs, etc.) généreront des états courants plus gros, augmentant ainsi l'espace utilisé par chaque cliché.

Il devient vite apparent que le stockage en mémoire est un problème. Une solution court-terme pourrait être de diminuer le nombre de points de sauvegarde (donc augmenter le nombre d'événements entre ceux-ci). Mais une diminution linéaire du nombre de points de sauvegarde entraînerait une dégradation aussi linéaire de la performance (puisqu'on augmenterait en moyenne le nombre d'événements à relire à chaque requête), cette solution ne serait donc pas souhaitable.

Il serait possible de conserver la même méthode, mais en enregistrant les clichés directement sur disque. Cependant, un autre problème existe, et c'est le fait que de l'information redondante peut être enregistrée. En effet, si un état est constant pour toute la durée d'une trace, il sera enregistré dans *chacun* des clichés, même si c'est exactement la même valeur qui est écrite à chaque fois.

Si nous pouvions stocker l'information d'état sur disque, et garantir qu'il n'y ait aucune redondance dans ce qui est enregistré, cela assurerait une utilisation optimale de l'espace occupé par le système d'état.

La solution qui semble la plus prometteuse est de stocker l'information d'état sur disque, mais sous forme d'intervalles de temps. À la section 2.3, nous explorerons diverses techniques de stockage d'intervalles, et tenterons de trouver un moyen de les appliquer pour une structure sur disque.

2.2.2 Stocker l'information d'état dans la trace

Une autre alternative possible serait de stocker l'information d'état directement dans la trace. Cela ne remplacerait pas le besoin de conserver un état en mémoire (et, optionellement, d'en prendre des clichés), mais éviterait au visualiseur de devoir définir lui-même les

changements d'états associés aux événements.

C'est cette approche qui est utilisée par Chan *et al.* (2008). Leurs traces contiennent à la fois des événements ponctuels, et des états avec une certaine durée. Ils affichent celles-ci avec l'outil Jumpshot [6], qui montre très clairement les communications MPI entre les processus.

Une différence avec l'approche LTTng/LTTV est que leur information d'état est très étroitement liée à l'information d'affichage. Par exemple les rectangles représentant les états seront affichés tels quel dans la vue. Ceux-ci sont stockés en mémoire, dans des R-Trees.

Cet étroit mariage entre le contenu de la trace et l'affichage à l'écran simplifie énormément la logique au niveau du visualiseur. Par contre, cela limite la flexibilité, puisque ces outils doivent toujours être utilisés ensemble.

Enfin, bien que le titre de leur article laisse sous-entendre un *Nearly Constant-Time Access to [...] Intervals*, nous avons été quelque peu déçu de comprendre que le temps est constant par rapport à l'*endroit* de la requête dans une même trace, et non pas par rapport à la taille de la trace.

Il aurait été intéressant de voir les temps moyens d'accès, et ce pour des traces de différentes grandeurs.

Il serait techniquement possible avec LTTng de générer le modèle de l'état au moment même du traçage, et de sauvegarder l'information qui en résulte dans la même trace que les événements. Par contre cela causerait deux problèmes.

D'abord, LTTng est présenté comme un traceur haute-performance, qui affecte très peu le comportement du système même lorsque le traçage est activé. Le maintien de l'état ainsi que les écritures additionnelles risqueraient de défaire cet équilibre.

Ensuite, si nous ajoutons des processus actifs pendant que le traceur logiciel fonctionne, celui-ci captera les actions de ces processus additionnels, et ces événements seront enregistrés dans la trace ! Cela ajouterait du “bruit” indésirable dans les traces résultantes.

Bref, nous concentrerons pour l'instant sur la gestion de l'information d'état au niveau du visualiseur. La génération et le traitement de cette information, au moment du traçage, pourront toutefois être explorés dans des travaux futurs.

6. `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm`

2.3 Le stockage d'intervalles

Comme il a été mentionné plus tôt, nous proposons de stocker l'information d'état à l'aide d'intervalles de temps. Ceci permettra de garantir qu'aucune information redondante n'est gardée. Nous étudierons dans cette section les différentes structures de données existantes qui puissent stocker de tels intervalles. Nous explorerons également les structures optimisées pour le stockage sur disque, comme les B-Trees.

2.3.1 Le *segment tree*

L'arbre à segments (ou *segment tree*), tel que décrit dans De Berg *et al.* (2008), est une structure bien connue en science informatique pour stocker des intervalles à une dimension. Il s'agit d'un arbre binaire, balancé, dans lequel on ajoute un branchement pour chaque intervalle inséré dans la structure.

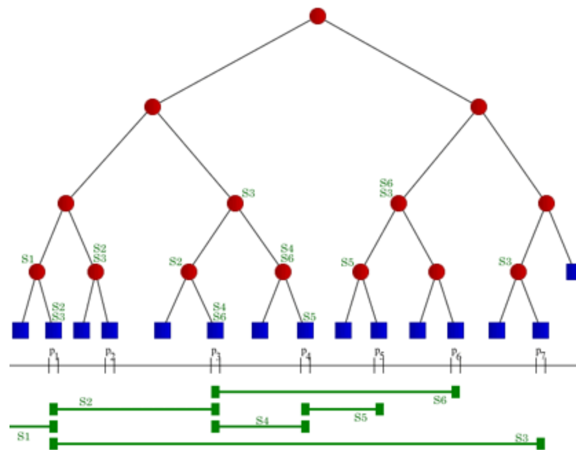

FIGURE 2.3: Un exemple de *segment tree*

(source : wikimedia.org)

La figure 2.3 montre un *segment tree* contenant six intervalles. Chaque noeud de l'arbre est défini par sa position dans l'axe de la dimension (habituellement le temps). Chaque noeud contient également la position de ses deux enfants.

Les noeuds à tous les niveaux peuvent contenir des intervalles. Cependant, le même intervalle peut être situé dans plusieurs noeuds (habituellement, on ne stockera qu'une référence commune dans les noeuds). Lors d'une requête pour récupérer tous les intervalles qui croisent un temps t, on ne fait que suivre la branche se rapprochant le plus du temps t, tout en conservant les intervalles situés dans les noeuds visités.

Côté implémentation pratique, le *segment tree* est une structure idéale pour le stockage en mémoire. Le fait que l'arbre soit binaire (deux enfants par noeud) implique que l'arbre sera en général assez profond. De plus, si on ne stocke que des références dans l'arbre, il faut conserver un index de tous les intervalles existants dans la structure (ou utiliser des pointeurs).

Si on utilise un disque rotatif comme médium de stockage, nous préférerons autant que possible ne pas utiliser d'arbres binaires. Pour des lectures non-séquentielles (ce qui est habituellement le cas avec les noeuds d'un arbre), la tête du disque devra être repositonnée avant chaque lecture, ajoutant ces temps de positionnement aux temps de lecture comme tels. Un arbre binaire sera relativement plus profond qu'un arbre possédant plus d'enfants, ce qui augmente le nombre de noeuds à être lus à chaque requête, donc le nombre de repositionnements.

De plus, comme les intervalles d'un *segment tree* peuvent être contenus par plus d'un noeud, deux solutions sont possibles. D'abord, nous pouvons dupliquer toute l'information de chaque intervalle dans chaque noeud, mais cela engendrerait un grand gaspillage d'espace.

Une autre solution serait de conserver un index de tous les intervalles (qui pourraient être placés séquentiellement dans un autre fichier par exemple), et de ne stocker que ces index dans l'arbre. Cette méthode serait plus économe en termes d'espace occupé, mais il faudrait rajouter une lecture, habituellement non-séquentielle, pour *chaque* intervalle qui est rencontré, ce qui ralentirait considérablement les requêtes à cause des nombreux repositionnements de la tête du disque.

Bref, le segment tree est beaucoup plus attrayant pour du stockage en mémoire. Si nous utilisions une solution qui conserve des intervalles d'état en mémoire, c'est sans doute la structure que nous choisirions. Comme nous voulons proposer une méthode qui puisse rester sur disque, nous étudierons d'autres structures qui se prêtent mieux à cette approche.

2.3.2 La famille des R-Trees

Nous regarderons maintenant le *R-Tree* et ses variantes. Ceux-ci sont beaucoup utilisés pour enregistrer de l'information spatiale, habituellement à 2 ou 3 dimensions, mais ils peuvent aussi être utilisés à une seule dimension.

Toute les données enregistrées dans l'arbre sont contenues complètement à chaque niveau. Chaque niveau additionnel ne fait que raffiner le "maillage". Il faut donc encore utiliser des références aux objets réels, mais comme l'arbre n'est pas binaire, ils se prêtent un peu mieux que les *segment trees* au stockage sur disque.

Le R-Tree original a été présenté la première fois par Guttman en 1984. Plusieurs variantes ont ensuite été proposées par la suite. Les plus connues sont sans doute le *R+ Tree* (Sellis *et al.*, 1987) et le *R* Tree* (Beckmann *et al.*, 1990).

Par rapport au R-Tree original, le R+ Tree modifie certaines contraintes, notamment que chaque objet puisse être contenus dans plus d'un noeud, mais on force les noeuds à être disjoints. Pour les requêtes en un point (par rapport aux requêtes par intervalle ou par rectangle), cela garantit que nous n'aurons qu'une branche de l'arbre à explorer. Cette particularité est intéressante pour notre cas d'utilisation, puisque ce sont principalement les requêtes en un point qui nous intéressent.

Le R* Tree quant à lui, est plus près du R-Tree original, mais propose une méthode d'insertion plus complexe, qui utilise des heuristiques pour s'assurer que chaque noeud feuille reste le plus équilibré possible. Les figures 2.4 et 2.5 montrent la différence entre l'agencement d'un R-Tree et d'un R* Tree pour un même jeu de données.

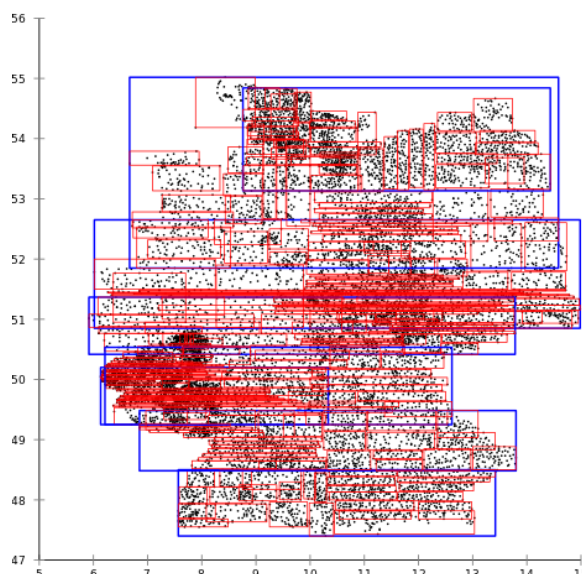

FIGURE 2.4: Division des noeuds dans un R-Tree

(source : wikimedia.org)

Les requêtes sur un R* Tree sont en général plus rapides, grâce à un meilleur agencement des noeuds, mais le prix à payer est une implémentation plus complexe et des insertions légèrement plus longues. En pratique, ce sont habituellement les temps de requêtes qui nous intéressent. Le R* Tree est donc une variante qui remplace presque complètement le R-Tree de base. La plupart des implémentations dénommées "R-Tree" dans les librairies communes sont très souvent des R* Tree en réalité.

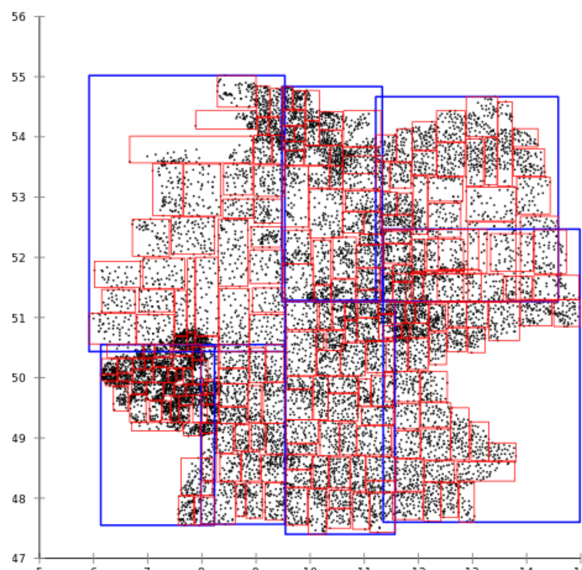

FIGURE 2.5: Division des noeuds dans un R*Tree

(source : wikimedia.org)

Une autre variante du R-Tree est le *Hilbert R-Tree* (Kamel et Faloutsos, 1994). Le principe derrière celui-ci consiste à utiliser la courbe de Hilbert [7] sur les valeurs à stocker pour délimiter les noeuds de l'arbre. Ceci résulte en un arbre très compact, qui se prête bien aux requêtes d'intersection de régions.

Le *Hilbert R-Tree* se prête mieux aux jeux de données hautement dimensionnelles. Ce n'est donc probablement pas la structure idéale pour stocker de l'information temporelle à une seule dimension.

Enfin, une autre variante possiblement intéressante est le *Historical R-Tree* (Nascimento et Silva, 1998). Il permet de stocker de l'information qui varie dans le temps. La méthode utilisée est de conserver un R-Tree *par unité de temps*. Les noeuds identiques d'un arbre à l'autre ne seront pas copiés, nous ne copierons qu'une référence au premier noeud de la série.

Néanmoins, nous pouvons nous attendre à une certaine surutilisation d'espace, puisque si *une* seule valeur change dans un noeud, le noeud au complet doit être recopié dans le prochain arbre. De plus, la création d'un arbre par unité de temps (ou *pas* d'une simulation par exemple) risque de devenir très lourde pour l'analyse de traces, où l'unité de base est habituellement la nanoseconde ! Par contre, ce concept d'ombrage (*shadowing*) des noeuds est potentiellement intéressant, et sera exploré de nouveau à la prochaine section.

7. `https://fr.wikipedia.org/wiki/Courbe_de_Hilbert`

En résumé, les R-Trees sont en général plus propices à des jeux de données à 2 dimensions ou plus (habituellement 2 ou 3). Ils peuvent être utilisés à une dimension (comme pour de l'information temporelle), mais ils sont dans ce cas moins efficaces. Au chapitre 4, nous ferons une comparaison de la méthode de stockage qui sera choisie avec un R-Tree générique, pour vérifier cette affirmation.

Comme les requêtes que nous utiliserons seront habituellement du type “point” (pour recharger un état à *un* moment de la trace à la fois), il serait intéressant d'intégrer la particularité des R+ Tree qui stipule que que les noeuds doivent rester disjoints. Ceci garantit qu'une seule branche de l'arbre doive être explorée lors d'une requête, ce qui limite le nombre de noeuds à être ouverts (donc, le travail à effectuer).

2.3.3 La famille des B-Trees

Dans cette section, nous nous intéresserons aux structures de la famille des B-Trees. Les B-Trees sont connus depuis bien longtemps, ayant été présentés la première fois par Comer en 1979.

Comme nous l'avons mentionné plus tôt, les arbres binaires causent des problèmes lorsqu'enregistrés sur un disque rotatif. La longueur des branches cause beaucoup de repositionnements de la tête du disque, qui sont très coûteux en termes de temps.

Le B-Tree tente de pallier à ce problème en suggérant une organisation qui sera propice au stockage sur disque. Les noeuds seront habituellement gros, et possèderont un grand nombre d'enfants. Ceci mènera à des arbres peu profonds, donc peu de noeuds différents devront être lus lors de l'exploration d'une seule branche.

Les B-Trees sont des structures balancées, où les données sont triées. Lors de l'utilisation, il faut spécifier une taille de noeud (en nombre d'entrées). Chaque lien vers un noeud enfant est toujours contenu “entre” deux entrées existantes ; toutes les valeurs contenues dans cet enfant (et ses enfants récursifs) seront situées numériquement entre les deux valeurs du noeud d'origine.

L'arbre peut être rebalancé, en ajoutant ou retirant des noeuds et en réagenceant les valeurs affectées. Les critères pour ajouter ou retirer un noeud sont configurables. Par exemple, on peut décider de n'ajouter des noeuds que lorsqu'un noeud devient plein, mais de retirer tout noeud qui n'est utilisé qu'à 25% ou moins.

La figure 2.6 montre un exemple d'insertions de valeurs dans un B-Tree. Les noeuds ne contiennent ici que deux entrées, et peuvent donc avoir jusqu'à trois enfants (en pratique,

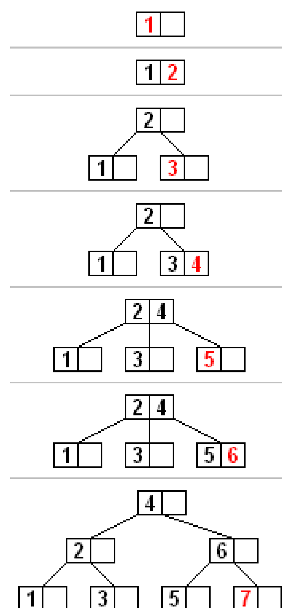

Figure 2.6: Insertions successives dans un B-Tree

(source : wikimedia.org)

des limites beaucoup plus élevées sont utilisées, comme 50 ou 100).

Comme ils sont efficaces pour le stockage sur disque, les B-Trees sont à la base de plusieurs bases de données et systèmes de fichiers.

Par exemple, le système de fichiers btrfs, sur Linux, est basé sur les B-Trees (c'est d'ailleurs de là que lui vient son nom). Une excellente description en a été faite par Aurora en 2009.

Btrfs est une implémentation de concepts présentés l'année précédente (Rodeh, 2008). L'utilisation de B-Trees pour un système de fichiers est intéressante, mais c'est surtout l'utilisation du COW (*Copy-on-Write*) qui le rend plus exotique.

Le COW permet de faire des "copies" de l'arbre complet en quelques instants, en ne copiant que des références et en indiquant seulement que les prochaines écritures seront faites dans de nouveaux noeuds et pas dans les noeuds originaux. Ce concept est aussi utilisé par le noyau Linux lors de l'appel `fork()`, qui crée un nouveau processus et "copie", sans vraiment le faire, tout l'espace mémoire du processus initial.

Ce principe est appelé *ombrage*, où le nouveau noeud est d'abord une "ombre" du noeud original, jusqu'à ce qu'on le modifie. À ce moment une véritable copie est faite et est utilisée par la suite. De cette façon il est possible de stocker plusieurs copies de l'arbre, représentant plusieurs moments dans le temps par exemple, mais tout en garantissant que les noeuds

identiques ne seront stockés qu'une seule fois.

Un des grands avantages de btrfs est de pouvoir faire des clichés du système de fichiers, à moindre coût, utilisant le COW.

Il va sans dire que cette approche ressemble grandement au concept de clichés d'état, qui a été présenté à la section précédente. Initialement, nous avons tenté de voir s'il était possible d'utiliser une méthode de stockage d'état utilisant du COW pour être plus efficace. Nous avons éventuellement réalisé que pour être efficace sur disque, la méthode devrait regrouper les intervalles d'état en noeuds assez volumineux. Mais si un seul des états contenus dans un noeud allait changer entre deux points de sauvegarde, le noeud en entier aurait à être recopié, perdant l'avantage du *Copy-on-Write*.

Il aurait été possible d'utiliser du COW pour améliorer la méthode de clichés de l'état, ce qui aurait notamment réduit l'espace mémoire utilisé par les clichés. Malheureusement, cette méthode semblait mal s'appliquer au stockage d'intervalles sur disque. Le concept reste toute de même très intéressant, peut-être pourra-t-il être appliqué à d'autres solutions.

2.3.4 Structure hybrides

Dans cette dernière section, nous présenterons des structures plus spécifiques à certaines applications. Il serait surprenant de trouver une solution pouvant s'appliquer telle quelle à notre problème, mais au moins nous pourrons peut-être découvrir certains aspects ou certaines particularités qui pourraient s'intégrer à notre solution.

Nous avons d'abord étudié le *Interval B-Tree*, de Ang et Tan (1995). Celui-ci propose l'utilisation de trois structures de données, toute contenues en mémoire, pour stocker des intervalles. Les trois composantes sont un *B+ Tree*, deux listes chaînées et un arbre de recherche binaire. Les deux premières contiennent les informations par rapport aux intervalles eux-mêmes, et la troisième sert d'index permettant d'accélérer les requêtes.

Pour augmenter la capacité de la structure, le B+ Tree pourrait probablement être enregistré sur disque et demeurer relativement efficace. Par contre ce n'est pas autant le cas pour les listes chaînées ou l'arbre binaire. Et comme le contenu de ces structures grandira avec le nombre d'intervalles insérés, nous serions tout de même limités par l'espace mémoire éventuellement.

Nous nous sommes ensuite penchés sur le *Relational Interval Tree*, ou RI-Tree (Kriegel *et al.*, 2000). Leur solution est en fait un ajout externe à une base de données relationnelle. Ils

utilisent une structure pour intervalles en mémoire, qui permettent de faire de l'optimisation des requêtes qui seront ultimement envoyées à la base de données.

Leur travail est ingénieux, et permet ainsi d'étendre les fonctionalités d'une RDBMS standard pour supporter efficacement des données sous formes d'intervalles. Ceux-ci peuvent ensuite être utilisés pour des requêtes d'intersection d'étendues, ou simplement de points.

Il s'agit ici d'une bonne preuve de concept. Ce qui serait très intéressant serait d'intégrer ce genre d'optimiseur de requête dans une base de données connue, pour que les utilisateurs puissent en profiter directement.

Enfin, l'utilisation d'une base de données externe n'est pas préférable à notre solution, puisque nous ne voulons pas ajouter une telle dépendance à un simple visualiseur de traces ! Par contre, nous ferons des tests avec une base de données de manière à comparer nos résultats. Nous nous limiterons cependant à une solution offrant déjà le support de données sous forme d'intervalles, à des fins de simplicité.

Ensuite, c'est le *MV3R-Tree* (Tao et Papadias, 2001) qui a retenu notre attention. Le *Multi-Version 3D R-Tree* est en fait un MVR-Tree (*Mutli-Version R-Tree*) couplé avec des R-Trees à 3 dimensions (mais comme on peut s'y attendre, ils utilisent l'algorithme des R* Tree pour la séparation des noeuds, puisque celui-ci donne en général des arbres plus performants).

Les auteurs affirments qu'en utilisant ces deux structures en parallèle, ils peuvent répondre aux deux types de requêtes courantes sur les structures de données spatiales : le MVR-Tree sera idéal pour les requêtes en un point, mais les R-Tree 3D seront meilleurs pour les requêtes d'intersection d'intervalles. Dans notre cas, ce sera principalement les requêtes en un point qui nous intéressent, donc nous chercherons à en comprendre davantage sur les MVR-Tree.

Un MVR-Tree est une alternative au *Historical Tree*, vu plus tôt, qui permet de stocker l'évolution dans le temps des données d'un R-Tree ordinaire. Au lieu de créer un nouvel arbre pour chaque pas de temps comme le Historical Tree, le MVR-Tree est en fait un R-Tree à 3 dimensions, dont la 3e dimension est le temps. Il permet donc des économies d'espace considérables par rapport au Historical Tree. Cela en fait une excellente structure pour, par exemple, enregistrer l'évolution de la position 2D d'objets à travers le temps.

Quoique très intéressante, ce genre de structure ne se prêtera sans doute pas très bien au stockage d'intervalles d'états, où notre seule dimension est le temps. Il serait également erroné de considérer par exemple les attributs comme une autre dimension, puisque nous n'avons aucune garantie qu'il y ait des liens logiques entre eux (ce n'est pas parce que l'attribut voisin a changé que nous aurons plus de chances de changer similairement). De plus, les valeurs d'états sont le plus souvent qualitatives, donc on ne peut habituellement même pas parler

de valeurs plus “près” des unes que des autres.

Enfin, nous avons étudié le *External Interval Tree* (Arge et Vitter, 2003). De toutes les solutions proposées jusqu'à présent, celle-ci semblait se rapprocher le plus de ce que tentons ici de faire, soit le stockage d'intervalles en mémoire externe (sur disque).

Les auteurs définissent le *Weight-balanced B-tree*, ou B-Tree balancé par rapport au poids. Dans cette variante, ce n'est pas le nombre d'enfants qui est le critère de rebalancement, mais plutôt le poids affecté à chaque sous-branche. Ce poids est déterminé par le nombre d'*éléments* au total situé dans la sous-branche, et non pas au nombre de noeuds du prochain niveau.

L'utilisation d'une telle structure permet de mieux gérer les étapes de rebalancement de la structure sur disque. En effet, puisque nous sommes sur disque, nous ne voulons pas subir de rebalancements trop souvent, et il faut s'assurer au minimum que chaque rebalancement en “vaut la peine”.

En fait, la majeure partie de l'article décrit les manières d'effectuer ces rebalancements sans trop sacrifier de performance. Les techniques sont très ingénieuses mais malheureusement, ce n'est pas quelque chose qui va nous aider pour le stockage d'intervalles d'état. Comme nous le verrons à la prochaine section, nos intervalles seront déjà triés lorsqu'ils seront insérés, il est donc possiblement plus simple de concevoir une structure qui n'a jamais besoin d'être rebalancée.

Par contre, la mention de conserver une profondeur d'arbre constante pour toutes les branches a retenu notre attention. Effectivement, si l'arbre contenant les intervalles (ou les références vers ceux-ci) possède le même nombre de noeuds dans toutes ses branches, les temps de requêtes seront sensiblement constants, peu importe l'endroit ciblé par celles-ci. Cela permet aussi de contrôler le comportement en pire cas, c'est donc une particularité que nous désirerons intégrer dans notre solution.

Chapitre 3

DÉTAILS DE LA SOLUTION

Ce chapitre décrira en détail le nouveau système d'état qui est proposé pour pallier aux lacunes du système actuel, qui ont été énoncées précédemment. Nous présenterons d'abord l'arbre à historique indépendemment. C'est une structure générique pour intervalles, qui pourrait même être utilisée dans d'autres contextes que le traçage. Nous verrons ensuite le reste du système d'état, et comment celui-ci utilise l'arbre à historique pour stocker ses intervalles d'état.

3.1 Arbre à historique

L'arbre à historique est une structure pour intervalles, optimisée pour enregistrer son contenu sur disque. Comme nous l'avons vu au chapitre 2, il y a eu jusqu'à maintenant quelques recherches dans le stockage efficace d'intervalles (à une dimension) sur disque, mais aucune solution ne semblait pouvoir directement résoudre notre problème.

Nous ne proposons pas ici de régler totalement le problème de stockage des intervalles sur disque, qui demeure encore un défi très complexe. Cependant, il nous est possible de faire quelques simplifications pour notre cas d'utilisation particulier. Plus spécifiquement, nous pouvons faire les hypothèses suivantes sur les intervalles qui seront générés par le système d'état :

- Les intervalles de temps sont courts relativement à la durée totale de la structure (dans notre cas, de la trace).
- Les intervalles seront insérés en étant triés par ordre de temps de fin.

Nous verrons plus clairement à la section 3.2 comment ces hypothèses sont respectées.

La seconde hypothèse est particulièrement importante dans la conception de l'arbre à historique. Si les intervalles sont strictement insérés par ordre chronologique, sans jamais rien insérer "dans le passé", cela signifie que la structure n'aura jamais à être rebalancée. Elle pourra également être construite de manière incrémentale (où la portion "terminée" de la structure pourra être figée et ne plus avoir à être modifiée).

Nous verrons maintenant les différentes composantes de l'arbre à historique. Aux sections 3.1.4 à 3.1.6, nous comprendrons comment l'arbre est construit et comment il peut utilisé par la suite.

3.1.1 Intervalles

La composante de base de l'arbre à historique est l'intervalle, appelé aussi "intervalle d'état". Chaque intervalle représente un état qui existe dans l'historique, et est composé de :

- Un temps de début
- Un temps de fin
- Un *quark* d'attribut
- Une valeur d'état

Les temps de début et de fin représentent bien entendu le commencement et la fin de l'état. Il s'agit donc des bornes de l'intervalle. Ces temps doivent être des valeurs entières. Il est à noter que la structure elle-même ne spécifie pas d'unités, cela doit être fait à un autre niveau du système. Les microsecondes et les nanosecondes seront sans doute les unités les plus communément utilisées.

Le *quark* est le nombre entier représentant un attribut du modèle de l'état. Nous verrons plus loin comment ceux-ci sont attribués et comment nous pouvons récupérer un attribut à partir de son quark.

La valeur d'état est la "charge utile" de l'intervalle. Elle indique dans quel état l'attribut concerné se trouvait, aux valeurs de temps comprises entre les bornes. Dans le modèle proposé ici, les valeurs d'état peuvent être soit des nombres entiers, soit des chaînes de caractères de taille variable.

3.1.2 Noeuds de l'arbre

La prochaine composante est le noeud de l'arbre. Les noeuds sont les conteneurs pour les intervalles, et sont agencés en arbre les uns par rapport aux autres, d'où le nom arbre à historique. Ils comprennent :

- Un temps de début
- Un temps de fin
- Lien vers le noeud parent
- Liens vers les noeuds enfants
- Liste des intervalles contenus

Les bornes, donc les temps de début et de fin, du noeud englobent TOUS les intervalles se trouvant dans ce noeud. Cela veut dire que les temps de début des intervalles contenus sont tous plus grands ou égaux au temps de début du noeud. Similairement, les temps de fin des intervalles dans ce noeud seront tous plus petits ou égaux au temps de fin du noeud.

Les noeuds et leur contenu sont écrits sur disque sous forme de "blocs". La taille de chaque bloc est fixée d'avance (voir la prochaine section). Les noeuds ont donc une limite quant au nombre d'intervalles qu'ils peuvent contenir. Cette limite dépend aussi des intervalles contenus, puisque des valeurs d'état comportant des chaînes de caractères peuvent être de taille variable. Lorsqu'un noeud est "plein", c'est-à-dire qu'il a atteint ou est sur le point de dépasser sa taille maximale permise, plus aucun intervalle ne pourra y être inséré.

Sur disque, chaque bloc est composé d'un en-tête de taille fixe, d'une section d'entrées, où nous utilisons une entrée par intervalle contenu, et d'une section pour les chaînes de caratères de taille variable. Les entrées de la deuxième section sont toutes de la même taille, ce qui permet d'itérer sur celles-ci sans avoir à les lire complètement.

Chaque bloc est aussi identifié par un "numéro de séquence", qui représente son index dans le fichier (le premier bloc après l'en-tête de fichier a le numéro de séquence 0, le suivant 1, et ainsi de suite). Comme tous les blocs ont la même taille, ceci permet d'aller rapidement se positionner à n'importe quel d'entres eux lorsqu'on en connait le numéro. Les "pointeurs" vers les parents et enfants sont enregistrés en termes de numéro de séquence.

Le format des blocs sur disque, et de tout le reste du fichier d'historique, est expliqué plus en détails à l'annexe A.

3.1.3 Constantes de configuration

Lorsque nous intancions un arbre à historique, il faut spécifier deux constantes de configuration. Ces valeurs resteront les mêmes pour tous les noeuds de cet arbre, mais elles pourraient varier d'un arbre à l'autre.

La première constante est la *taille des blocs*, donnée en octets. Tel que mentionné précédemment, la taille des blocs sur disque est fixe, ce qui impose une limite sur l'espace occupé par chaque noeud. De plus, si cette taille est un multiple de 4096 octets, ils seront alignés sur les blocs du système de fichiers et sur les pages mémoires, permettant une utilisation optimale de ces supports. Nous vérifierons au prochain chapitre si cette précaution en vaut la peine.

La deuxième constante de configuration est le *nombre maximal d'enfants par noeuds*. Dans l'en-tête de chaque bloc, nous stockons un tableau comportant les numéros de séquence des enfants du présent noeud. Comme on veut un en-tête de taille fixe, ces tableaux doivent être alloués statiquement. Le paramètre du nombre maximal d'enfants sera donc la taille à allouer pour ces tableaux.

Bien entendu, cet espace pourrait être utilisé par des intervalles si le tableau était plus petit. Pour justifier le nombre maximal choisi, il faut que ce nombre d'enfants soit atteint autant que possible, sinon nous serions en situation de gaspillage d'espace.

Au moment du design initial, nous n'avions pas vraiment d'idée quant à quelles valeurs utiliser pour ces deux paramètres. Nous savions seulement que la taille des blocs devrait être multiple de 4096, et que le nombre d'enfants devrait être relativement grand (donc 10 ou plus), de manière à obtenir un arbre très large mais pas trop profond. Ce sera le premier test à être effectué lors des tests expérimentaux.

3.1.4 Construction de l'arbre

Jusqu'à présent, nous avons vu les différentes composantes de l'arbre à historique. Voyons maintenant comment celles-ci s'agencent entre elles pour former un arbre.

À l'aide de l'hypothèse stipulant que les intervalles arrivent par ordre croissant de temps de fin, il nous est possible de déterminer le "temps courant" de l'historique, c'est-à-dire le temps de fin du dernier intervalle qui a été inséré. Comme il est garanti qu'aucun intervalle ne se terminera avant ce temps courant, nous pouvons donc isoler les parties "complétées" de l'arbre, les écrire sur disque et ne plus jamais les modifier. Ces parties complétées deviennent donc l'historique lui même.

Les figures 3.1 à 3.5 montrent le processus de construction de l'arbre, à mesure que les intervalles sont insérés.

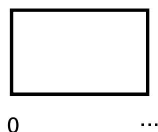

FIGURE 3.1: Premier noeud vide

Nous commençons avec un seul noeud, vide, dont la borne initiale est la valeur de début donnée à l'historique (nous utiliserons ici 0). Tous les intervalles insérés seront placés dans

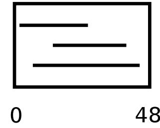

Figure 3.2: Premier noeud rempli

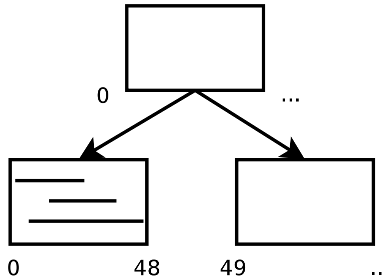

Figure 3.3: Ajout d'une nouvelle racine et d'un noeud frère

ce noeud. Une fois le noeud rempli, nous marquons celui-ci comme "plein" et le fermons, c'est-à-dire que lui assignons sa borne finale. Celle-ci est égale au temps de fin le plus élevé parmi les intervalles existants.

Le premier noeud étant plein, nous devons maintenant lui ajouter un frère dans lequel nous pourrons continuer à insérer les intervalles suivants. Comme le premier noeud est présentement la racine de l'arbre, nous devrons d'abord ajouter une nouvelle racine, qui deviendra le parent du premier noeud et de son nouveau frère (Figure 3.3).

Le noeud racine conserve la borne initiale de l'historique d'état. Le nouveau noeud frère aura une borne initiale égale à 1 unité de plus que la borne finale du premier. Comme les temps sont représentés par des entiers, ceci permet de s'assurer que les noeuds soient juxtaposés, tout en restant disjoints.

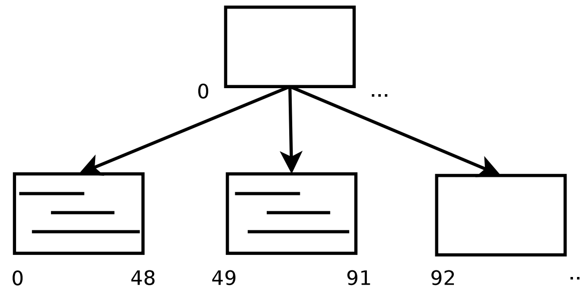

Figure 3.4: Premier frère est rempli, ajout d'un autre

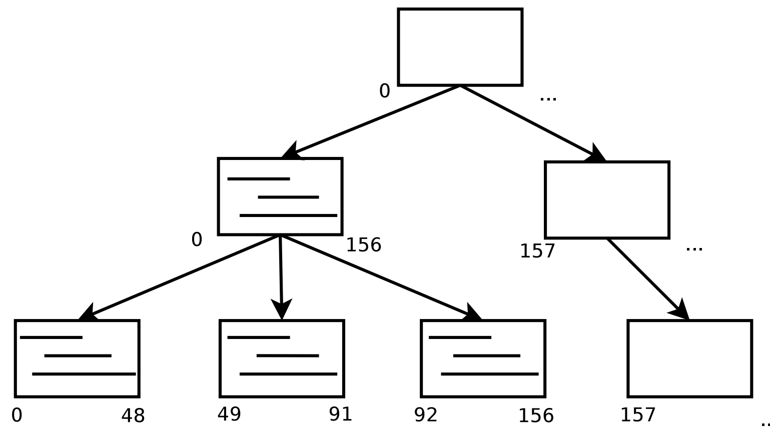

Figure 3.5: Noeud racine rempli, ajout d'une nouvelle racine et nouvelle branche

Pour les prochaines insertions, un choix devra être fait entre les deux noeuds de la branche de droite. Nous voudrons autant que possible insérer les intervalles dans le noeud le plus bas de la branche. Ceci ne sera permis que si le temps de début de l'intervalle est plus grand ou égal à la borne initiale de ce noeud.

Dans l'exemple de la figure 3.3, si un intervalle possédant un temps de début de 50 demande à être inséré dans l'historique, nous pourrons le placer dans le noeud feuille. Par contre, un intervalle avec un temps de début de 20 ne pourra pas y être, nous le placerons donc dans ce cas dans le noeud racine.

Nous constatons donc que le noeud racine permet de stocker les intervalles qui chevauchent les deux autres. C'est ici que la première hypothèse mentionnée à la section 3.1 entre en jeu. Puisque les intervalles sont habituellement courts, la majorité des intervalles entreront dans le noeud feuille, et celui-ci sera rempli avant le noeud racine. Cette hypothèse est nécessaire pour pouvoir obtenir un nombre de noeuds enfants raisonnable.

Si, comme nous l'espérons, le noeud feuille est rempli en premier, un nouveau noeud feuille devra être ajouté. Comme prédémment, nous fermerons le noeud précédent avec le temps de fin le plus élevé qu'il contient, et assignerons la borne initale du nouveau noeud (figure 3.4). Pour les insertions suivantes, nous choisirons entre le nouveau noeud feuille et le noeud racine.

Les insertions continueront ainsi, jusqu'à ce que l'une des deux situations suivantes se produisent : soit le noeud racine deviendra lui-même "plein" en termes d'intervalles, soit il atteindra son nombre maximal d'enfants permis (qui, rappelons-nous, est configurable pour chaque arbre à historique). Dans les deux cas, nous devrons fermer le noeud racine.

Si cela n'est pas déjà fait, nous fermerons également le noeud feuille le plus récent, avec la même borne de fin que son parent. La prochaine étape serait d'ajouter un nouveau "frère" au noeud racine. Avant de pouvoir ajouter des noeuds à ce niveau, il faudra créer une nouvelle racine.

À chaque fois que nous créons une nouveau noeud racine dans l'arbre, nous créerons tout de suite une branche descendant jusqu'au niveau feuille. Les temps de début de ces nouveaux noeuds seront tous les mêmes. Nous nous retrouvons alors avec une situation comme celle de la figure 3.5.

Les bornes des noeuds seront donc toujours égales aux bornes les plus distantes de leurs enfants. Les bornes du noeud racine seront égales aux bornes actuelles de l'historique en entier.

3.1.5 Fermeture de l'arbre

Lorsque nous n'avons plus aucun intervalle à insérer, nous indiquons simplement à l'historique qu'il est terminé. Le temps de fin le plus élevé qui se trouve dans l'arbre est utilisé comme borne finale pour tous les noeuds de la branche de droite, et l'historique au complet est marqué comme "fermé".

Lors de l'utilisation de l'arbre dans le contexte du système d'état, cette étape sera normalement précédée de l'insertion de plusieurs intervalles possédant tous le même temps de fin, de manière à vider l'état transitoire qui est maintenu en mémoire. Nous comprendrons d'où ces intervalles viennent à la section 3.2.3.

3.1.6 Requêtes

Une fois fermé, l'arbre à historique pourra être la cible de requêtes visant à reconstruire un état à un temps quelconque. Nous avons vu à la section 3.1.4 que lors de sa construction, seule la branche la plus à droite peut recevoir des intervalles et que tous les autres noeuds sont fixés et ne seront plus modifiés.

Il serait techniquement possible de pouvoir lancer des requêtes sur un arbre qui n'est pas fermé, tant que nous nous limitons à la partie fixée de l'historique. Cependant, dans le travail qui est présenté ici, nous faisions seulement l'analyse de traces post-mortem, nous utiliserons donc seulement des requêtes sur des historiques fermés.

La figure 3.6 présente un arbre à historique complété et fermé, avec les bornes de début et de fin de chaque noeud. En pratique, le nombre d'enfants par noeud est beaucoup plus élevé, nous le limitons ici à deux ou trois pour simplifier la représentation.

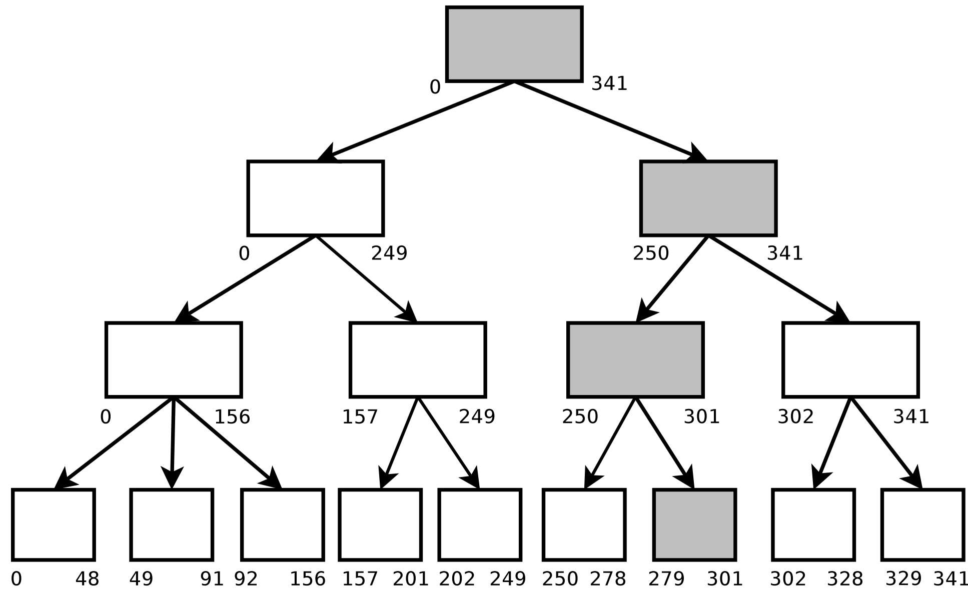

FIGURE 3.6: Historique complété, requête au temps $t = 300$

Supposons que l'arbre reçoit une requête pour reconstruire l'état du système étudié au temps $t = 300$. L'arbre devra explorer les noeuds indiqués en gris sur la figure pour trouver l'information nécessaire.

"Explorer un noeud" consiste à recharger celui-ci en mémoire depuis le disque et inspecter les intervalles pour déterminer lesquels croisent le temps de requête demandé. Pour chaque intervalle retenu, nous lirons l'attribut et la valeur d'état qu'il contient. Cette valeur sera assignée à cet attribut particulier dans la réponse qui sera retournée.

Nous verrons à la section 3.2 comment les intervalles sont créés. Si l'historique est bien construit, il ne devrait y avoir qu'une seule valeur d'état pour chaque attribut à chacun des temps compris dans l'historique.

Dans l'exemple précédent, nous voulions reconstruire l'état complet du système au temps $t = 300$. Ceci veut dire que nous irons chercher les valeurs de tous les attributs existants. Nous devrons donc explorer tous les noeuds de la branche, jusqu'au noeud feuille. Nous appelons ce type de requête une *requête complète*.

Il est aussi possible d'effectuer un autre type de requêtes sur l'arbre à historique, appelée *requête ponctuelle*. Celles-ci vont recharger l'état d'un seul attribut à un temps donné, et non pas de tous les attributs existants. Une requête ponctuelle fonctionne de la même manière qu'une requête complète, sauf que nous pouvons nous arrêter dès que nous rencontrons l'intervalle correspondant à l'attribut qui nous intéresse. Les requêtes ponctuelles vont chercher beaucoup moins d'information, mais sont beaucoup plus rapides en moyenne que les requêtes

complètes. Nous verrons au chapitre suivant les différences de performance entre les deux types de requêtes.

3.2 Système d'état en général

Nous verrons maintenant le fonctionnement du système d'état pour visualiseurs de traces qui est suggéré pour remplacer la méthode des clichés. De manière générale, nous créons des intervalles d'états pour représenter tous les états de notre modèle à travers le temps. Nous stockons ensuite ces intervalles dans une structure comme l'arbre à historique.

Les sections suivantes décrirons les différentes composantes du système d'état. Nous verrons comment il est possible de construire un historique d'état, de manière générique, à partir d'événements de traces seulement.

Un schéma du système d'état complet est présenté à la figure 3.7. Cette figure montre l'interaction des différentes composantes lors de la construction de l'historique de l'état.

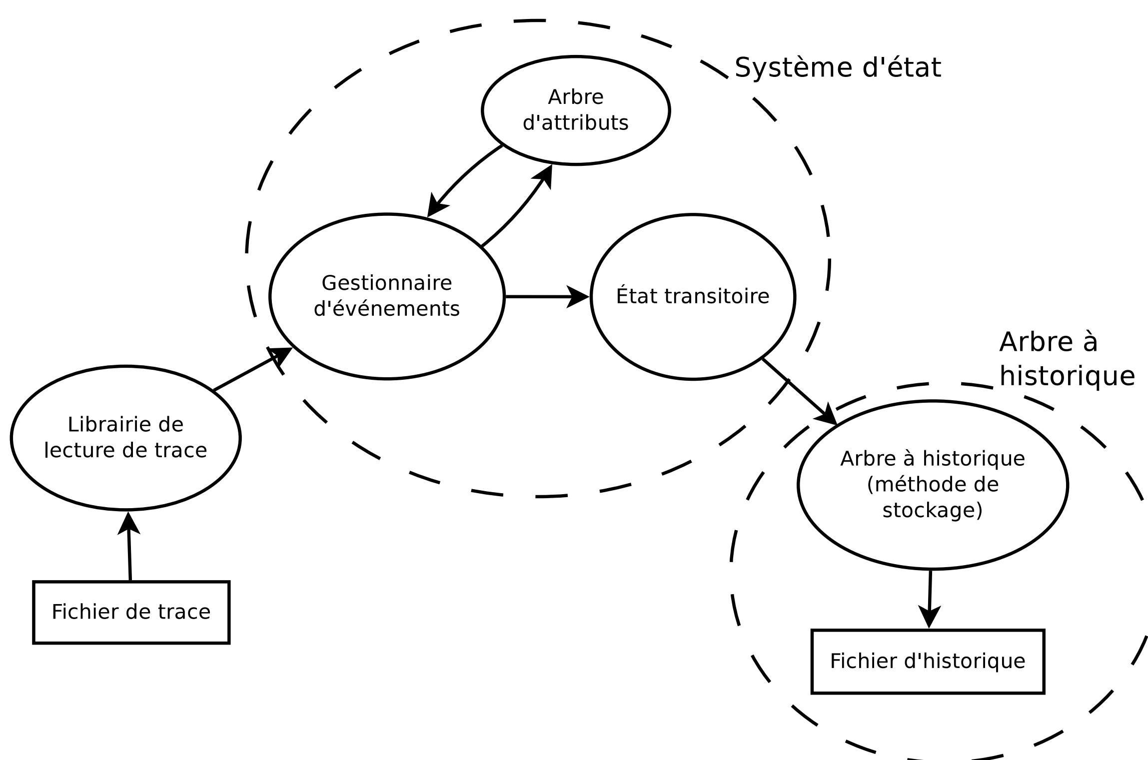

FIGURE 3.7: Système d'état lors de la construction de l'historique

Le groupement marqué "Arbre à historique" était le sujet de la section précédente. Dans cette section nous discuterons du groupement "Système d'état".

3.2.1 Gestionnaire d'événements

La première composante du système d'état est le *gestionnaire d'événements*. Le but de ce gestionnaire est de convertir les événements de trace en des *changements d'état*.

Un changement d'état est décrit par trois critères :
- Une valeur de temps
- Un attribut
- Une valeur d'état

Ce sont ces changements d'état qui seront utilisées par les autres parties en aval du sytème.

Le gestionnaire d'état doit être défini pour chaque type de trace que nous voulons utiliser. Il faut, par exemple, connaître les types d'événements qui seront présents dans la trace, leur nom ainsi que leur contenu.

Dans le système proposé, cinq méthodes génériques sont fournies pour pouvoir décrire ces changements d'état. Elles sont présentées au tableau 3.1.

Tableau 3.1: Méthodes pour décrire un changement d'état

Méthode	Paramètres requis
`modifier`	temps, attribut, valeur
`effacer`	temps, attribut
`empiler`	temps, attribut, valeur
`dépiler`	temps, attribut
`incrémenter`	temps, attribut

Dans tous les cas, nous devons spécifier un temps, ainsi que l'attribut qui sera affecté par le changement d'état. Les temps sont encore une fois donnés en nombres entiers. Les valeurs d'état présentement supportées par le système peuvent être soit des entiers, soit des chaînes de caractères.

Les attributs, quant à eux, peuvent être spécifiés soit par leur *quark*, soit par leur chemin d'accès. Le chemin d'accès peut être absolu, ou relatif à un autre attribut dont on connaît déjà le quark. Si l'attribut spécifié n'existe pas présentement dans le système, il sera créé (nous n'avons donc pas à spécifier tous les attributs au préalable).

La méthode `modifier` est habituellement la plus courante, et stipule de simplement remplacer la valeur d'état existante pour cet attribut (s'il y en a une) par la nouvelle qui est passée en paramètre.

La méthode `effacer` est similaire à `modifier` utilisée avec une valeur d'état "nulle" (qui est aussi une valeur d'état valide). Cependant, elle va aussi "nullifier" tous les attributs enfants de l'attribut spécifié, sans avoir à tous les énumérer. Nous verrons à la section 3.2.2 les relations entre les attributs.

Les méthodes `empiler` et `dépiler` permettent d'insérer des nouvelles valeurs dans l'historique de l'état mais sans écraser la valeur précédente. Elles sont utiles pour modéliser des états sous forme de piles (comme, par exemple, une pile d'appels de fonction).

Enfin, il est possible d'`incrémenter` la valeur d'un attribut. Cette méthode ne fonctionne qu'avec les attributs de type nombre entier. Elle est équivalente à un `modifier` qui spécifierait une nombre égal à 1 de plus que la valeur présente. Elle évite cependant de devoir aller chercher la valeur courante, ce qui simplifie légèrement l'utilisation (et qui est aussi un peu plus performant). Cette méthode est utile pour générer des statistiques, par exemple.

En réalité, toutes ces méthodes accomplissent la même chose : modifier un attribut donné, à un temps donné, pour lui assigner une nouvelle valeur d'état. Ces différentes méthodes ont été implémentées pour simplifier les cas d'utilisation rencontrés, mais ultimement, c'est un simple changement d'état sous la forme {temps, quark, valeur d'état} qui sera envoyé au reste du système.

La manière suggérée de construire un gestionnaire d'état est d'observer le type de tous les événements de la trace un à un (à l'aide d'un `switch/case` par exemple), et d'assigner les changements d'état pour chaque type d'événement. Il est très possible pour des événements de causer plusieurs changements d'état, ou encore aucun changement du tout (on dit dans ce cas que cet événement ne modifie pas l'état).

Le gestionnaire d'état est une pièce critique du système. Comme il est appelé pour chaque événement de la trace, il est important qu'il soit le plus optimisé possible. Par exemple, il est idéal de réutiliser les quarks autant que possible pour spécifier les attributs. Il faut également éviter les changements d'états redondants ou inutiles, puisque cela peut affecter grandement la quantité d'information qui sera enregistrée. Nous verrons au chapitre suivant l'impact que le gestionnaire d'état peut avoir sur la performance du système, ainsi que des optimisations qui peuvent être utilisées.

3.2.2 Arbre d'attributs

Nous avons parlé à plusieurs reprises des attributs du système d'état. De manière générale, un attribut est une unité de base du modèle, qui ne peut contenir qu'une seule valeur d'état à la fois.

Cependant, les attributs sont souvent reliés les uns aux autres. Par exemple, plusieurs attributs peuvent être utilisés pour décrire l'état d'un processus sur le système. Il serait souhaitable de pouvoir regrouper ensemble des attributs similaires. Ceci aiderait à l'organisation, mais aussi à la performance comme nous le verrons plus loin.

TABLEAU 3.2: Exemples d'attributs

Quark	Nom de l'attribut
(1)	Process/1500
(2)	Process/1500/PPID
(3)	Process/1500/TGID
(7)	Process/1500/Exec_name
(4)	Process/1501
(5)	Process/1501/PPID
(6)	Process/1501/TGID
(8)	Process/1501/Exec_name

Le tableau 3.2 montre des attributs qui pourraient être utilisés dans un système d'état modélisant l'état d'un système informatique tracé.

La solution proposée est d'organiser les attributs en un arbre, que nous appellerons simplement *arbre d'attributs*. Attention, cet arbre est indépendant de l'arbre à historique dont nous avons parlé précédemment. Il s'agit simplement de l'organisation en mémoire des différents attributs du modèle d'état.

Nous pouvons comparer cet agencement à celui des fichiers et des répertoires dans un système de fichiers. Le nom du fichier représente le nom de l'attribut. Le contenu du fichier, variable dans le temps, représente la valeur d'état actuellement associée à cet attribut. De plus, chaque noeud est à la fois et un fichier et un répertoire : un attribut peut posséder et une valeur d'état et des attributs enfants. Ce comportement se rapproche de celui de système de fichiers comme ZFS [1].

Chaque noeud de l'arbre d'attributs représente, bien entendu, un attribut dans le modèle. Un noeud-attribut contient :

- Son nom (chaîne de caractères)
- Son quark (valeur entière unique le représentant)
- Un lien vers son parent (le quark du parent)
- Des liens vers ses attributs enfants

1. http://en.wikipedia.org/wiki/ZFS

Les quarks sont assignés aux attributs dans l'ordre, à mesure que ceux-ci sont créés. Comme nous l'avons vu à la section précédente, les attributs sont créés automatiquement lorsqu'on les accède pour la première fois. Les quarks sont donc indépendants de la "proximité" des attributs eux-mêmes. Dans l'exemple du tableau 3.2, on peut supposer que l'information par rapport aux attributs *Exec_name* n'est arrivée que plus tard, donc ces attributs ont été créés en dernier.

Nous remarquons également qu'aucune valeur d'état n'est stockée dans l'arbre d'attribut. Le but de l'arbre d'historique est simplement de pouvoir convertir les chemins d'accès et les noms des attributs en leurs quarks. Ce sont les structures comme l'état transitoire, vu à la prochaine section, qui stockeront les valeurs d'état.

Dans chaque noeud-attribut, les liens vers les attributs enfants son gardés sous forme d'une table de hachage. Cela permet de convertir le nom de chaque enfant en son quark correspondant. De cette façon, il est possible d'accéder à n'importe quel attribut de l'arbre en spécifiant son chemin d'accès sous forme d'un tableau de chaînes de caractères.

En parallèle, nous conservons des références à tous les noeuds de l'arbre à l'aide d'un tableau. L'index dans ce tableau correspond au quark de chaque attribut. Ceci permet d'accéder à n'importe quel attribut du modèle en temps $\mathcal{O}(1)$ lorsque nous connaissons son quark.

En revanche, l'utilisation du chemin d'accès serait de l'ordre de $\mathcal{O}(n \times m)$, où n est la longueur du chemin d'accès, et m le temps de hachage moyen d'un nom d'attribut. C'est pourquoi nous préférerons utiliser les quarks autant que possible.

Par exemple, si nous voulons accéder à l'attribut "Processes/1500/TGID" du tableau 3.2, il faudra hacher les chaînes "Processes", "1500" et "TGID" (même les nombres sont stockés sous forme de chaînes de caractères à des fins de flexibilité). Par contre, si nous connaissons déjà le quark (3), l'accès sera beaucoup plus rapide.

Tel que mentionné plus tôt, il est aussi possible de spécifier des chemins d'accès relatifs. Si, par exemple, nous voulons accéder à l'attribut "Processes/1501/Exec_mode/2" (qui représente le deuxième élément de la pile d'exécution du processus #1501) mais que nous connaissons déjà le quark de "Processes/1501" (qui est 4), nous pouvons appeler une fonction du type :

```
int quark = AccesRelatif(4, [‘‘Exec_mode’’, ‘‘2’’]);
```

Ce qui est plus rapide que de rehacher le chemin en entier.

Comme un arbre d'attributs est spécifique à un historique en particulier, il a été décidé de sauvegarder celui-ci directement dans le fichier de l'arbre à historique. Pendant l'utilisation, toute l'information est conservée en mémoire, mais lorsque l'historique est complété, nous écrivons l'information sur les attributs à la fin du fichier (puisque ceux-ci ne devraient maintenant plus changer). Voir l'annexe A pour plus d'informations.

3.2.3 État transitoire

Jusqu'à maintenant, nous avons vu comment le gestionnaire d'état permet de convertir les événements de la trace en changements d'état qui nous intéressent, et comment l'arbre d'attribut permet de convertir un chemin d'accès en un quark d'attribut et vice-versa.

L'*état transitoire* contient principalement les deux tableaux suivants :

- Tableau de valeurs d'état
- Tableau de valeurs de temps

Les index des deux tableaux correspondent aux quarks des attributs du modèle. Leur taille sera donc toujours la même que celle du tableau d'attributs. Des entrées seront ajoutées lorsque de nouveaux attributs sont créés.

La tableau de valeurs d'état représente l'*état courant* du système, au moment où le descripteur de lecture de la trace est présentement situé. Le tableau des valeurs de temps quant à lui sert à enregistrer *depuis quel moment* la valeur d'état est valide pour cet attribut.

Lorsqu'un changement d'état est envoyé au système par le gestionnaire d'état, l'attribut demandé est converti en quark par l'arbre d'attributs. L'état transitoire se retrouve donc avec un quark, un temps et une valeur d'état.

Lorsque le quark est utilisé pour la première fois, les entrées correspondantes à celui-ci dans les deux tableaux seront bien entendu vides. Nous assignerons simplement la valeur d'état et la valeur de temps à ce quark dans le tableau respectif.

Si, par contre, il existe déjà une entrée, donc une valeur d'état, pour cet attribut particulier, nous devrons *remplacer* cette valeur. Mais, il ne faut pas écraser la valeur précédente ! Nous créerons d'abord un *intervalle d'état* à partir de l'information dont on dispose :

- L'attribut
- L'ancienne valeur de temps (borne initiale de l'intervalle)
- L'ancienne valeur d'état
- La *nouvelle* valeur de temps (borne finale de l'intervalle)

Cet intervalle sera ensuite inséré dans l'historique de l'état. L'état qu'il représente n'est plus *courant* à proprement parler. Il est terminé et fait maintenant partie de l'historique. C'est l'interface de stockage, décrite à la prochaine section, qui décidera quoi faire avec cet intervalle.

Comme nous l'avons vu à la section 3.1.4 pour les noeuds de l'arbre à historique, nous souhaitons que les intervalles d'un même attribut soient juxtaposés, mais pas "embarqués" un par-dessus l'autre. En effet, si le temps de fin d'un état était égal au temps de début de l'état suivant, et qu'une requête était faite exactement à cet endroit, il y aurait ambiguïté quant à quel état retourner.

Il faut donc qu'il y ait au minimum une unité de temps entre la fin d'un état et le début du suivant. Dans le modèle utilisé ici, nous affirmons que si un changement d'état a lieu au temps t, le nouvel état sera valide à cet instant et l'intervalle du nouvel état commencera à t. L'intervalle de l'état précédent se terminera à $t - 1$.

Il aurait aussi été possible d'utiliser t et $t + 1$. Ultimement, cela n'affecte que le comportement des requêtes tombant exactement sur les temps des changements d'état.

L'état transitoire n'est nécessaire que pendant l'étape de construction de l'historique. Lorsque la lecture de la trace sera terminée, plus aucun événement ne sera envoyé au gestionnaire, donc plus aucun changement d'état ne sera envoyé au système. L'état transitoire en sera informé, et il pourra procéder à "se vider" dans l'historique.

Le temps de fin de l'historique sera identifié (habituellement égal au temps de fin de la trace d'origine), et un intervalle sera créé pour toutes les valeurs d'état qui existent présentement dans l'état transitoire. Les attributs, les temps de début et les valeurs d'état de ces intervalles proviendront des deux tableaux mentionnés plus tôt. Les valeurs de fin seront toutes égales au temps de fin de l'historique qui nous aura été envoyé.

Ces intervalles seront tous envoyés à l'historique. Le système d'état va ensuite informer l'historique qu'il peut à son tour se "fermer". L'étape de construction sera terminée, et le système sera prêt à recevoir des requêtes.

3.2.4 Interface de stockage

À la section précédente, nous avons expliqué comment les intervalles d'état étaient créés, puis avons mentionné que ceux-ci étaient simplement envoyés à l'historique d'état. En conservant une distinction entre la génération des intervalles et leur stockage, il est possible de facilement changer la méthode de stockage utilisée. Bien entendu, l'arbre à historique est le

principal intéressé. Cette interface nous a cependant permis de comparer l'arbre à d'autres méthodes de stockage, comme nous le verrons à la section 4.6.

Cette flexibilité permet non seulement d'utiliser différentes méthodes de stockage, mais aussi de chaîner plusieurs d'entres elles (pourvu que les méthodes intermédiaires implémentent elles aussi l'interface). Cette particularité permettra l'implémentation de méthodes de stockages plus complexes, comme l'*historique partiel*, qui sera présenté à la prochaine section.

Nous pouvous maintenant revenir à la figure 3.7 et comprendre comment les différentes composantes agissent entre elles.

Une fois l'historique complété, le système d'état est prêt à recevoir des requêtes. À ce moment, l'application pourra accéder à l'arbre d'attribut pour convertir les attributs en quarks, et utiliser ceux-ci pour les requêtes qui seront envoyées directement à l'arbre à historique. Le résultat sera retourné dans un tableau de valeurs d'état, où l'index correspond au quark de chaque attribut.

3.3 Historique partiel

Jusqu'à présent, nous avons décrit un système dans lequel tous les intervalles d'état créés par le système sont enregistrés dans l'arbre à historique. Par contre, comme nous le verrons très tôt au prochain chapitre, la taille des fichiers d'historique obtenus en pratique est très importante. Avec les traces et les gestionnaires utilisés, ils sont environ de une à deux fois plus grands que leurs traces d'origine.

Cela n'est pas un problème pour les traces de petite taille. Par contre, puisque le support de traces de très grande taille est l'un des objectifs même du système d'état proposé ici, il serait souhaitable de pouvoir diminuer l'espace requis par l'historique.

C'est ainsi que l'*historique partiel* a vu le jour. Nous avons vu au chapitre précédent la méthode des clichés pour recharger un état en tout point. Celle-ci utilise des points de sauvegarde pour indiquer à quels moments de la trace il est possible de recharger un état. Une relecture de la trace est ensuite effectuée pour mettre à jour l'état transitoire, jusqu'à ce que nous atteignions le point de la requête.

Le problème de l'historique complet est qu'il y a simplement trop d'information à enregistrer. Ce problème est apparent pour les statistiques : il est très intéressant de pouvoir connaître le nombre d'événements de chaque type qui ont eu lieu jusqu'à présent, mais l'insertion d'un intervalle par événement augmente la taille de l'historique beaucoup trop rapidement.

L'idée était alors d'intégrer la notion de points de sauvegarde au système, mais tout en continuant d'utiliser un arbre à historique comme méthode de stockage. Si nous conservons tous les intervalles qui croisent un temps donné, il demeure possible de faire une requête complète à ce point précis. Nous pouvons donc définir des points dans l'historique (qui seront les points de sauvegarde), et nous assurer de conserver tous les intervalles qui les intersectent. Tous les autres intervalles, qui ne croisent aucun point de sauvegarde, pourront être rejetés.

La figure 3.8 montre des intervalles d'états dans un historique, appartenant à quatre attributs différents. Normalement ces intervalles seraient tous insérés dans l'arbre à historique.

À la figure 3.9, nous définissions trois points de sauvegarde. Dans l'historique partiel qui serait construit avec ces points, les intervalles plus pâles seraient délaissés, et seul les autres se retrouveraient dans l'arbre à historique.

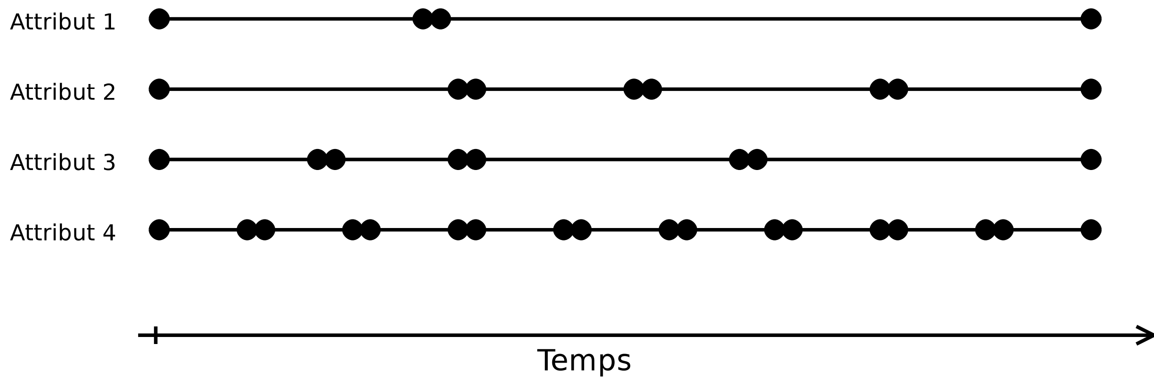

Figure 3.8: Des intervalles composant un historique

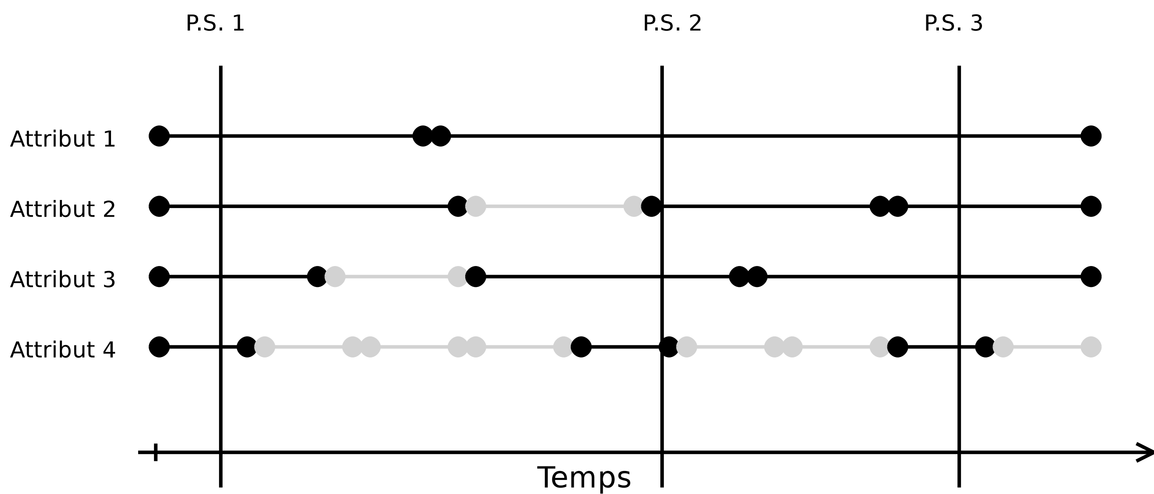

Figure 3.9: Retrait des intervalles qui n'intersectent pas les points de sauvegarde

Un intervalle est défini par ses deux bornes et son contenu, donc il utilise le même espace sur disque peu importe sa longueur. On constate que les courts intervalles sont les plus aptes à être retirés de l'historique, comme ceux de l'attribut 4 dans l'exemple.

Cependant, la méthode partielle offre aussi un avantage pour les longs intervalles. Regardons le deuxième intervalle de l'attribut 1 : il ne sera stocké qu'une seule fois dans l'historique même s'il croise deux points de sauvegarde. Ceci est un avantage par rapport à la méthode des clichés, puisque nous ne stockons maintenant aucune information redondante.

Au niveau implémentation, il suffit d'ajouter une étape après la création des intervalles d'état qui vérifie si l'intervalle à être inséré croise le point de sauvegarde précédent ou pas. Si le temps de début de l'intervalle est plus grand que le temps du point de sauvegarde précédent le plus près, l'intervalle est rejeté. S'il est égal ou inférieur, nous le conserverons. Nous ne nous préoccupons pas du fait qu'il croise un point de sauvegarde antérieur ou non.

Si l'intervalle croise un point antérieur, il croisera nécessairement le point le plus près aussi.

L'arbre à historique résultant pourra seulement recevoir des requêtes aux temps correpondant à ces points. Si une requête était faite à un autre moment, l'information retournée risquerait d'être incomplète.

Pour répondre à une requête visant un moment arbitraire de l'historique, l'idée est de recharger l'état, depuis l'arbre à historique, au point de sauvegarde précédent le plus proche. En utilisant cette réponse comme "état transitoire temporaire", nous retournerons au point correspondant dans la trace, et rejouerons les événements en utilisant le même gestionnaire d'état qui aura créé l'historique la première fois ! Lorsque nous aurons atteint le moment de la requête, nous nous arrêterons et le contenu actuel de l'état transitoire sera retourné comme réponse finale.

Pour ce qui est du choix des endroits où placer les points de sauvegarde, il existe plusieurs possiblités. La manière la plus évidente serait sans doute d'utiliser un intervalle de temps constant entre chacun (par exemple, un point à chaque seconde de l'historique).

En réalité, un nombre d'événements de trace constant entre les points de sauvegarde est un meilleur choix. Il se peut que des endroits de la trace soient plus denses en événements que d'autres (lorsque le système tracé était plus occupé). Les requêtes lancées à ces endroits prendraient plus de temps, puisqu'il faudrait relire plus d'événements depuis la trace. De plus, ce sont habituellement les moments "occupés" qui intéressent les utilisateurs !

L'utilisation d'un nombre d'événements constant entre chaque point de sauvegarde permet aussi de borner supérieurement le nombre d'événements qui devront être lus depuis la trace, donc le travail à effectuer, à chaque requête.

Il devient par contre nécessaire de conserver un mappage entre le nombre d'événements et la position dans le fichier de la trace ou l'estampille temporelle visée (tout dépendant de la librairie de lecture). Ce mappage doit également être sauvegardé quelque part si on veut réouvrir un historique existant plus tard.

Il aurait été possible de sauvegarder cette information directement dans le fichier de l'arbre à historique. Par contre, l'arbre à historique ne sait pas s'il est "partiel" ou non, ce n'est qu'une structure à intervalles. Le choix a donc été fait de laisser le fichier d'historique indépendant, et de sauvegarder l'information à propos des positions des points de sauvegarde dans un autre fichier.

La construction d'un historique partiel est très semblable à celui d'un historique complet, sauf que nous vérifions d'abord chaque intervalle pour savoir s'il est conservé ou pas. Le fonctionnement des requêtes, par contre, est plus différent. La figure 3.10 résume les étapes d'une requête sur un historique partiel.

FIGURE 3.10: Requête sur un historique partiel

Le fichier de points de sauvegarde est chargé à l'ouverture et est conservé en mémoire tant que le système d'état reste ouvert. Celui-ci permet au système de savoir en quels points il peut lancer des requêtes sur l'arbre à historique.

Lorsque'une requête est envoyée au système, l'interface intermédiaire (ici appelée "stockage partiel") va d'abord envoyer une requête à l'arbre à historique correspondant au temps du point de sauvegarde précédent le plus proche (1). Ensuite, la réponse obtenue sera utilisée comme état transitoire et sera mise à jour à l'aide de la trace et du gestionnaire d'événements

originaux (2). Une fois le temps de requête atteint, l'état transitoire sera retourné au système principal comme réponse.

Un autre point indiqué sur la figure est que les deux systèmes d'états (le "vrai" système, ainsi que le "mini-système" qui est utilisé temporairement à chaque requête) utilisent le même arbre d'atttributs. Ceci permet de n'avoir qu'une seule copie de l'arbre d'attributs en mémoire à la fois. Le mini-système n'a pas la permission d'écrire dans l'arbre, c'est-à-dire d'insérer des attributs. De toute façon, comme les événements ont déjà tous été vus lors de la construction initiale de l'historique, les attributs devraient déjà tous être présents.

En résumé, l'utilisation d'un historique partiel permet de réduire l'espace disque occupé par le fichier d'historique, mais nous devons nous attendre à des requêtes plus lentes, puisque plus de travail doit être effectué lors de celles-ci. La section 4.5 présentera les différences quantitatives de performance entre les deux variantes.

Chapitre 4

RÉSULTATS THÉORIQUES ET EXPÉRIMENTAUX

Dans ce chapitre nous présenterons les résultats des différents tests effectués lors de l'implémentation du système d'historique d'état.

À ces fins, une série de traces de référence a été prise. Ces traces seront utilisées pour tous les tests présentés dans ce chapitre, sauf indication contraire. Elles seront brièvement décrites à la section 4.1. La section qui suit décrira les différents tests ainsi que la méthodologie utilisée.

Nous avons vu au chapitre précédent que deux paramètres permettent de configurer un arbre à historique : la taille des blocs du fichier, et le nombre maximal d'enfants par noeuds permis. Bien que l'ajustement optimal de ces paramètres peut beaucoup varier en fonction du contenu, nous tenterons d'abord de trouver de bonnes valeurs par défaut que nous pourrons utiliser avec les traces de test utilisées.

Nous avons également évalué plusieurs variantes lors du design du système. Dans certains cas il s'agissait de choisir entre deux alternatives, dans d'autres il s'agissait de justifier l'ajout d'un changement ou pas. Nous verrons les résultats de ces comparaisons à la section 4.4.

La section suivante présentera l'impact de l'utilisation d'un historique partiel.

Ensuite nous comparerons les performances du système d'état utilisant l'arbre à historique par rapport au même système utilisant des structures génériques déjà existantes comme méthode de stockage à la place. Les détails seront donnés à la section 4.6.

Finalement, nous présenterons les résultats de performance du design final mais en utilisant un nouvel ensemble de traces, beaucoup plus grandes cette fois. Ceci permettra de vérifier le comportement du système à grande échelle (on parle ici de traces d'origine allant jusqu'à 500 GB et de systèmes d'états approchant le téraoctet).

Comme ces test prenaient beaucoup plus de temps en général, ils ont été réalisés séparément, et seront présentés à la section 4.7.

4.1 Traces de références

De manière à pouvoir éprouver le système d'état, il nous fallait des traces le plus possible représentatives d'un cas d'utilisation typique.

La série de traces initiale a été prise dans une machine virtuelle KVM, en utilisant LTTng 0.232 et un simple script Python qui démarrait une trace, vérifiait périodiquement la taille du répertoire et l'arrêtait lorsqu'elle dépassait la taille ciblée. Bien entendu, à cause du délai entre les vérifications et la nécessité de vider la mémoire tampon, les traces résultantes étaient toujours légèrement plus grandes. Mais cela donnait tout de même une bonne plage de grandeur.

Tableau 4.1: Traces de référence LTTng

Taille cible (MB)	Taille réelle (MB)
10	27
50	79
100	121
500	622
1000	1426
2000	2582
5000	5880
10000	11691

Pour simuler une charge sur le système, un processus `ping` roulait en continu dans la machine tracée. Celui-ci était lancé avec une commande comme :

```
ping -f -i 0.2 localhost
```

ce qui indiquait au programme de lancer des *pings* sur la machine elle-même, sans interruption (`-f`), plusieurs fois par seconde (`-i 0.2`, pour obtenir un intervalle de 0.2 s entre chaque *ping*). Ceci causait donc principalement des événements réseaux, ainsi que des appels systèmes (pour les ouvertures et fermetures des *sockets*) qui eux sont utilisés par le gestionnaire d'état.

Il fallait également faire attention à ce qu'aucun événement ne soit perdu dans les traces. Si les événements arrivent plus rapidement que le disque n'est capable de les écrire, les mémoires tampons du traceur risquent de se remplir, empêchant l'écriture des événements suivants. Comme on ne peut arrêter le noyau, le traceur doit jeter ceux-ci. Il a donc fallu bien caliber le paramètre `-i`, de manière à ce que les événements n'arrivent pas trop vite, mais que les traces soient prêtes en quelques heures et non quelques semaines !

Enfin, le gestionnaire d'état utilisé permettait d'enregistrer sensiblement le même type d'information que celle enregistrée par le système d'état de LTTV. À celle-ci nous ajoutions un changement d'état par événement indiquant le type de l'événement, ce qui permettait d'avoir des statistiques sur le nombre d'événements de chaque type en tout point.

Ce gestionnaire a légèrement varié lors du développement du système et de la réalisation des tests. Cependant, il reste identique pour chaque groupe de tests, de manière à ce que les comparaisons soient valables.

4.2 Description et méthodologie des tests

Nous décrirons maintenant l'environnement de test utilisé, ainsi que les types de tests principaux qui auront été réalisés. Pour les autres types de tests uniques ou plus spécifiques, ils seront décrits dans leur section correspondante.

4.2.1 Environnement de test

Les tests ont tous été réalisés sur une machine du laboratoire DORSAL, dont les spécifications sont présentées au tableau 4.2.

Tableau 4.2: Spécifications de la machine de test

CPU	Intel Core i7 920 @ 2.67GHz
RAM	6 GB
Disque dur	ST3500418AS de 500 GB
OS	Ubuntu 11.04 - 11.10
Version Java	OpenJDK 6
Version Eclipse	3.6 - 3.7

Ces spécifications changeront légèrement pour les tests de la section 4.7. La version d'Eclipse passe à 3.8, OpenJDK passe à 7 et un disque dur externe de 2 TB est ajouté pour contenir et les traces et les historiques d'état résultants.

4.2.2 Construction

L'étape de construction consiste à lire tous les événements de la trace, passer ceux-ci dans le gestionnaire d'état, construire les intervalles d'états correspondants et finalement écrire ceux-ci sur disque. Cette étape doit évidemment être réalisée avant de pouvoir utiliser

l'historique pour des requêtes. Par la suite, nous pouvons réutiliser le fichier d'historique existant correspondant à une trace sans avoir à le reconstruire.

Dans les tests, la construction n'est effectuée et mesurée qu'une seule fois, étant donné le temps relativement important qui est requis.

4.2.3 Ouverture d'un historique existant

Dans quelques cas, nous mesurerons le temps d'ouverture d'un historique déjà construit. Ceci consiste à simplement ouvrir le fichier d'historique et charger les structures requises en mémoire, avant de pouvoir lancer des requêtes.

Ces tests sont réalisé en mesurant l'étape de chargement seule, en moyenne sur 10 essais. Les mémoire caches du noyau sont vidées entre chaque mesure, à l'aide de la commande :

```
# sysctl vm.drop_caches=3
```

Bien entendu, l'étape de vidange elle-même n'est pas prise en compte dans la durée indiquée.

4.2.4 Requêtes complètes

Une fois l'historique construit, il est possible d'envoyer des requêtes au système d'état. Les requêtes complètes consistent à recharger l'état complet du système à un moment quelconque de l'historique. Ceci revient à récupérer la valeur d'état de *tous* les attributs de l'arbre d'attributs pour le temps ciblé.

Lorsque nous utilisons un arbre à historique comme structure de stockage (ce qui est le cas pour tous les tests, sauf ceux de la section 4.6), une requête complète consiste en une requête en un point, qui parcourt l'arbre sur toute sa profondeur.

Un test de temps de requête consiste à mesurer la durée entre le moment où le temps cible est envoyé au système et l'instant où il nous retourne le tableau d'attributs rempli avec les valeurs appropriées.

Pour les tests, nous générons d'abord 200 valeurs de temps aléatoires se retrouvant dans l'étendue de la trace. Ces valeurs sont ensuite envoyées au système, dans l'ordre où elles ont été générées, de manière à exécuter 200 requêtes consécutives.

Nous reproduisons ce cycle 10 fois. Les caches sont vidées entre chaque cycle de 200 requêtes (mais pas après chaque requête unique). Encore une fois le temps mesuré se limite au temps des requêtes, excluant les temps de vidange et de génération de nombres. Le résultat final est divisé par 2000, pour obtenir le temps moyen de chaque requête.

Initialement, nous exécutions également deux variantes de ce tests, où les 200 temps aléatoires générés étaient triés en ordre croissant et décroissant avant d'être envoyés au système. Nous pensions que ceci permettrait de modéliser différents cas d'utilisation.

Dans les résultat cependant, les temps requis était pratiquement toujours les mêmes, peu importe si les temps cibles étaient triés ou pas. Même dans les cas triés, les requêtes étaient en moyenne suffisamment éloignées pour qu'on doive explorer des branches différentes de l'arbre lors des requêtes subséquentes. Nous avons éventuellement consolidé ces tests en un seul, où nous ne trions rien.

4.2.5 Requêtes ponctuelles

Comme nous l'avons vu au chapitre précédent, il est aussi possible de réaliser des requêtes dite *ponctuelles* sur l'historique d'état. Celles-ci sont utiles lorsqu'on cherche la valeur d'état d'un ou de quelques attributs seulement, et non pas du système entier.

Une requête complète nécessite seulement un temps cible comme paramètre. Une requête ponctuelle demande un temps et un attribut (sous forme de *quark*). Dès qu'un intervalle correspondant à l'attribut demandé est rencontré, on s'arrête et la valeur est retournée. Ceci est donc normalement plus rapide qu'une requête complète.

Une méthodologie similaire à celle des requêtes complètes est utilisée pour les tests. Nous générons d'abord 200 temps et 200 entiers entre 0 et le nombre d'attributs du système (pour représenter les quarks des attributs). Ces deux séries seront les paramètres de 200 requêtes consécutives. Nous répétons le cycle 10 fois, en vidant les caches et régénérant des nouvelles valeurs entre chaque cycle. La valeur finale donnée est la moyenne pour chaque requête.

4.2.6 Taille du fichier sur disque

Comme l'arbre à historique est stocké dans un seul fichier, nous pouvons facilement en mesurer la grandeur depuis le programme avec la méthode Java `File.length()`. Celle-ci retourne le nombre d'octets occupés par le fichier, tel que rapporté par le système de fichiers.

Il est à noter que c'est aussi cette méthode qui est utilisée pour mesurer la taille des traces, sauf avis contraire. Dans le cas des traces LTTng, qui comportent plusieurs fichiers, nous itérons sur tous les fichiers du répertoire et retournons la somme des tailles de chacun.

4.2.7 Profondeur de l'arbre

La “profondeur de l'arbre” correspond au nombre de niveaux dans l'arbre à historique complété. Si celui-ci a été bien construit, cette profondeur devrait être la même pour tous les noeuds feuilles. Elle correspond au nombre de blocs qui devront être lus du disque lors de requêtes complètes.

La profondeur donne une idée générale de l'agencement et de la taille de l'arbre. Elles ne seront pas présentées ici, mais d'autre métriques comme le pourcentage d'utilisation moyen et le nombre d'enfants de chaque noeud permettaient également de s'assurer de la cohérence de l'arbre d'attributs.

4.3 Variantes de configuration

Tel que décrit au chapitre précédent, l'arbre à historique possède deux paramètres de configuration. Les premiers tests réalisés avaient pour but de trouver des valeurs raisonnables pour ces paramètres, de manière à ce que les tests subséquents soient représentatifs.

4.3.1 Taille des blocs sur disque

Nous nous sommes d'abord intéressé à la taille de blocs idéale à utiliser. Pour permettre une plus grande flexibilité, il serait souhaitable d'utiliser pour tous les tests à venir une taille de bloc identique, qui puisse bien couvrir l'étendue des traces utilisées.

Des noeuds plus petits se rempliront plus vite, donc le nombre de noeuds et le nombre de niveaux dans l'arbre sera plus grand. Cependant, chaque noeud contient moins d'intervalles à traiter et comparer lors des requêtes. Il n'était donc pas évident de voir quelle valeur serait idéale avant de faire des tests.

Les figures 4.1 à 4.4 montrent les résultats des différents tests en utilisant des tailles de noeuds de 16 KB, 64 KB, 256 KB, 1 MB et 4 MB.

Comme on peut s'y attendre, la taille des noeuds n'influence pas beaucoup le temps de création. Plus intéressant à remarquer à ce point est que le temps de construction est relativement long ($\tilde{4}5$ minutes pour construire l'historique de la trace de 11 GB). Diverses optimisations, présentées plus loin, permettront de réduire considérablement le temps requis par cette étape.

La taille du fichier d'historique est pratiquement la même, peu importe la taille de blocs utilisée, nous ne présentons donc qu'une seule courbe. On peut dire que pour l'instant l'his-

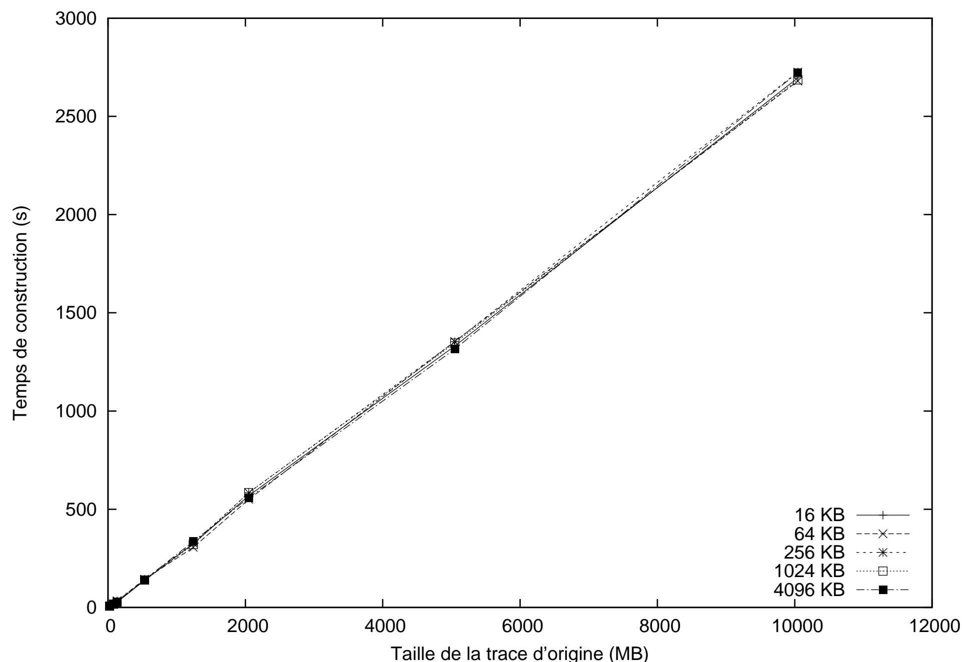

FIGURE 4.1: Comparaison des tailles de blocs, temps de création de l'historique

torique est *énorme*, plus de 3 fois la taille de la trace d'origine. Nous verrons plus loin des manières de conserver une taille plus raisonnable.

Bien entendu, les résultats des temps de requêtes deviennent plus intéressants lorsque nous atteignons des tailles d'historique qui ne tiennent plus complètement en mémoire (donc, que nous explorons les nouvelles possibilités offertes avec le nouveau système d'état).

Il ne semble pas y avoir de grand gagnant, mais les tailles de 16 KB à 256 KB semblent définitivement se comporter mieux pour l'étendue des traces qui sont utilisées.

La taille de blocs de 4 MB par exemple semble ici peu appropriée : les arbres résultants ne sont pas très profonds, donc le nombre d'intervalles à comparer est relativement plus élevé. Ceci ralentit considérablement les requêtes.

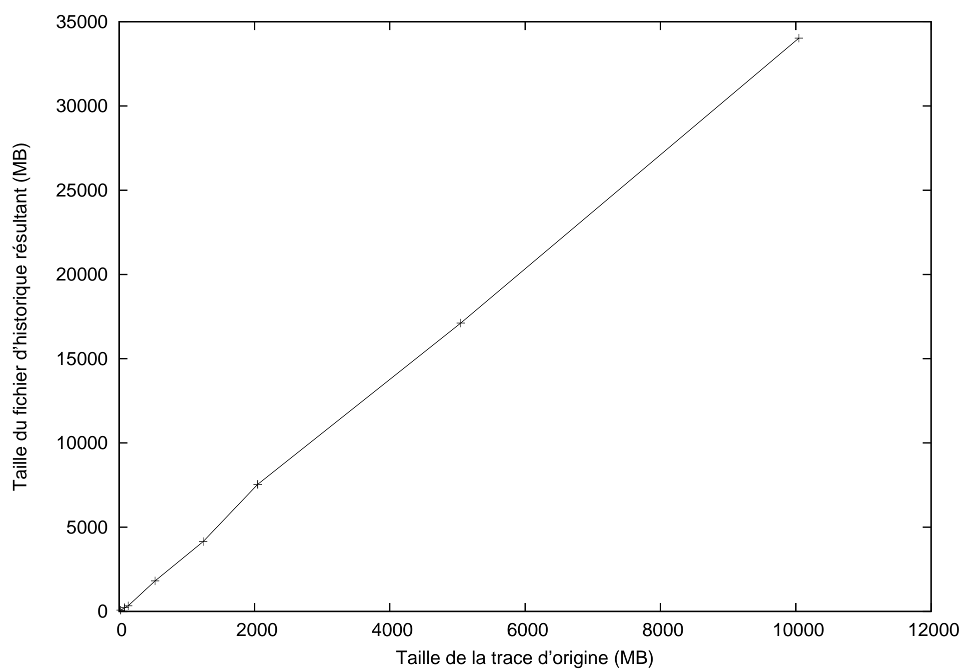

FIGURE 4.2: Comparaison des tailles de blocs, taille de l'historique résultant

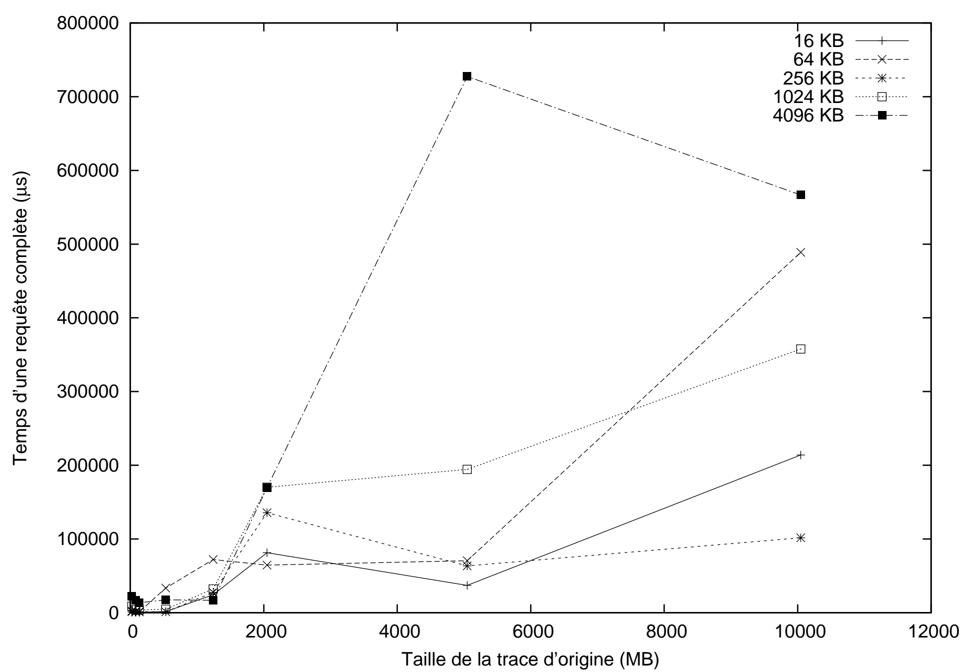

FIGURE 4.3: Comparaison des tailles de blocs, requêtes complètes

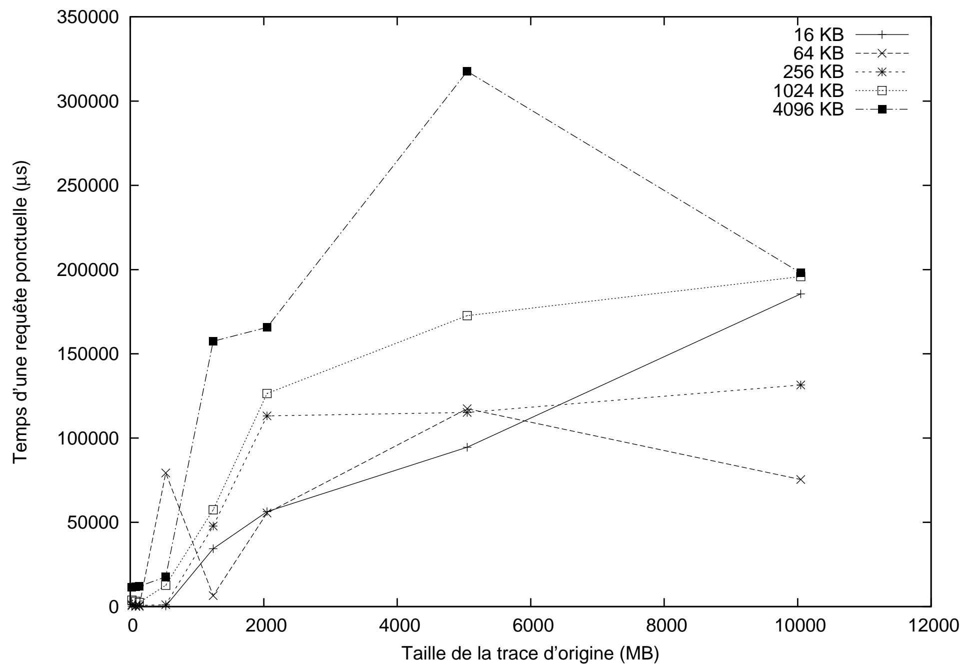

FIGURE 4.4: Comparaison des tailles de blocs, requêtes ponctuelles

4.3.2 Nombre d'enfants maximal par noeud

L'autre variable de configuration de l'arbre à historique est le nombre maximal d'enfants par noeuds. Tel que décrit précédemment, de l'espace est réservé dans l'en-tête de chaque noeud pour les "pointeurs" vers les noeuds enfants.

Un nombre d'enfants plus grand résulte en un nombre de niveaux moins élevé, donc moins de noeuds à ouvrir et traiter lors d'une requête. Cependant, un nombre d'enfants plus élevé cause plus d'espace utilisé dans l'en-tête des noeuds, qui peut même être gaspillé si ce nombre maximal n'est pas atteint.

Dans les tests qui suivent, nous avons utilisé une taille de blocs de 256 KB, qui semble s'être relativement bien comportée à la section précédente. Nous avons comparé des nombres maximum d'enfants de 10, 20, 50, 100 et 200. Les figures 4.5 à 4.8 présentent les résultats.

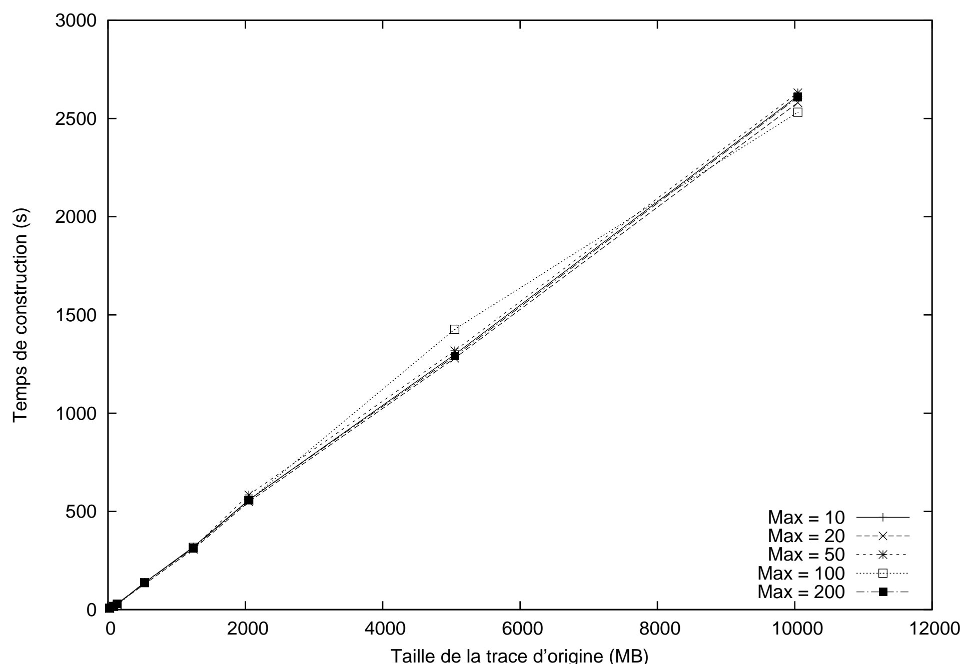

FIGURE 4.5: Comparaison du nombre max. d'enfants par noeud, temps de création de l'historique

Encore une fois, le paramètre étudié ne fait pas beaucoup varier le temps de construction de l'historique.

Cependant, la taille finale de l'historique varie légèrement. En diminuant le nombre d'enfants par noeuds, l'historique devient légèrement plus gros. Ceci est logique puisque des noeuds plus petits seront plus nombreux, donc plus d'espace sera utilisé par les en-têtes, qui ne sont pas de la charge utile.

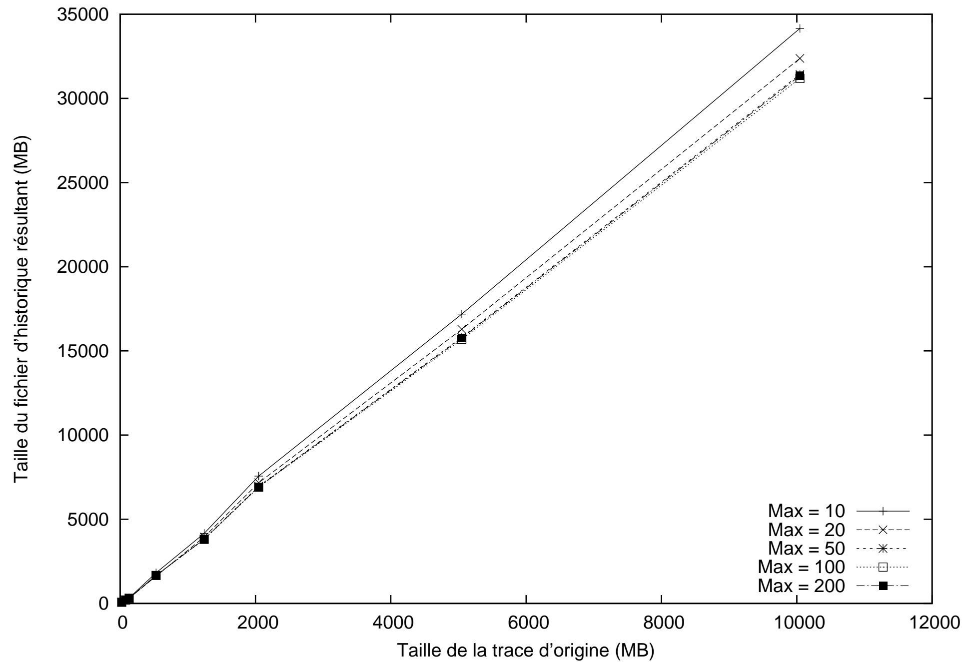

FIGURE 4.6: Comparaison du nombre max. d'enfants par noeud, taille de l'historique résultant

Pour les requêtes complètes, on remarque l'avantage d'utiliser un nombre maximal d'enfants plus élevé jusqu'à environ 50. Pour les requêtes ponctuelles, cet avantage dure jusqu'à environ 100.

On remarque également, aux figures 4.7 et 4.8, l'étrange comportement de la trace de 2 GB, particulièrement pour le nombre maximal d'enfants de 100. Ces résultats étaient répétables. Cette trace est intéressante puisque c'est habituellement autour de ce point que le fichier d'historique dépasse la taille de la mémoire vive du système (d'après la figure 4.6).

Il s'agit peut-être d'un cas où le noyau pense pouvoir faire rentrer le fichier complètement en mémoire, mais finit par en mettre des parties en *swap*, ce qui rend la situation pire. Il nous a été suggéré d'utiliser des fonctions du noyau comme `fadvise()`, mais malheureusement celles-ci ne sont pas exposées en Java.

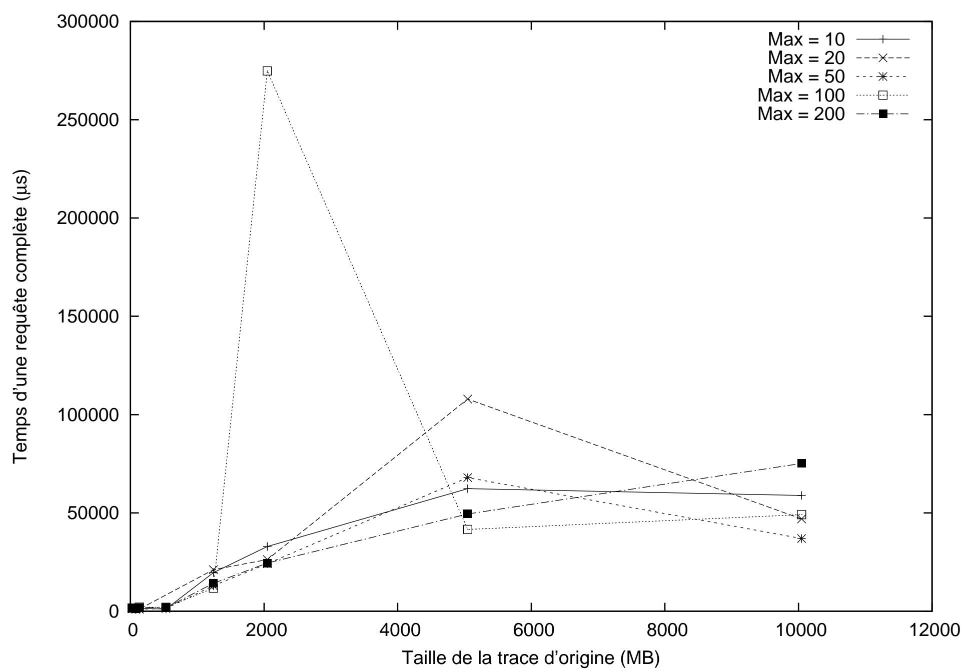

FIGURE 4.7: Comparaison du nombre max. d'enfants par noeud, requêtes complètes

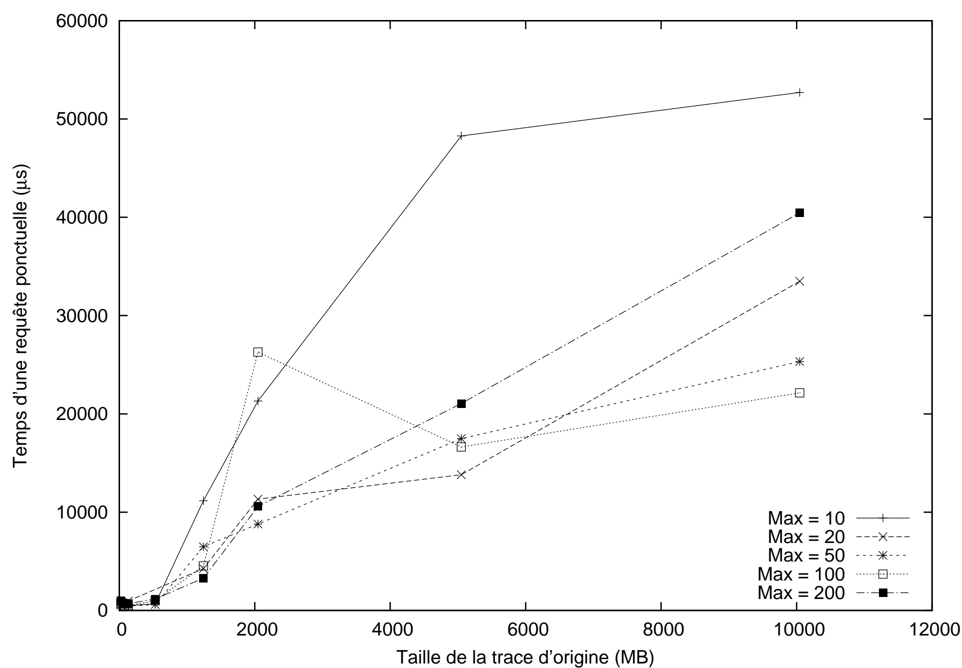

FIGURE 4.8: Comparaison du nombre max. d'enfants par noeud, requêtes ponctuelles

4.4 Optimisations

À la lumière des résultats présentés à la section précédente, nous avons décidé d'utiliser des valeurs par défaut de 64 KB et 50 pour la taille des blocs et le nombre maximal d'enfants par noeuds, respectivement.

Les prochains tests ont été réalisés au cours du développement du système d'état et de l'arbre d'historique, pour comparer différentes variantes de conceptions ou différentes options qui ont été envisagées.

4.4.1 Placement des attributs en arbre

La section 3.2.2 a décrit comment nous pouvons placer les attributs en arbre, où chaque attribut (noeud de l'arbre) contient une table de hachage permettant d'accéder à ses enfants.

Dans le design initial, les attributs étaient tous placés à un seul niveau. Pour accéder à un attribut, il fallait hacher l'entièreté de la chaîne de caractères représentant le chemin d'accès de l'attribut.

Les premières rondes de profilage ont indiqué que le plus important demandeur de temps processeur était le hachage des chemin d'accès des attributs. Nous avons donc tenté de voir s'il était possible de réduire les quantités de hachage requises.

En plaçant les attributs en un arbre, il est possible d'accéder à un attribut en particulier relativement à un autre qui lui est parent : ceci permet d'éviter de re-hacher la partie commune du chemin d'accès.

Nous avons donc réorganisé l'agencement des attributs du système d'état, et avons mis à jour le gestionnaire d'événements pour qu'il ne re-hache pas inutilement les chemins d'accès utilisés plusieurs fois. Nous pouvions maintenant utiliser le quark d'un attribut comme point de départ pour en trouver un autre plus bas dans la hiérarchie.

La figure 4.9 montre le temps de construction de l'historique complet, comparant l'ancienne et la nouvelle méthode.

La figure 4.10 quant à elle montre le temps requis pour lire une trace, mettre à jour l'état transitoire, mais sans enregistrer aucun historique sur disque. Sur la même figure, nous avons aussi mesuré le temps pris par le lecteur de traces pour seulement lire tous les événements de la trace, sans les passer dans le gestionnaire d'état. Ceci donnait une idée de la proportion du temps prise par chaque étape.

Nous avons aussi mesuré le temps pris par LTTV pour effectuer une tâche équivalente. Cette mesure se retrouve également sur la figure 4.10. Cette tâche consistait à lire les mêmes

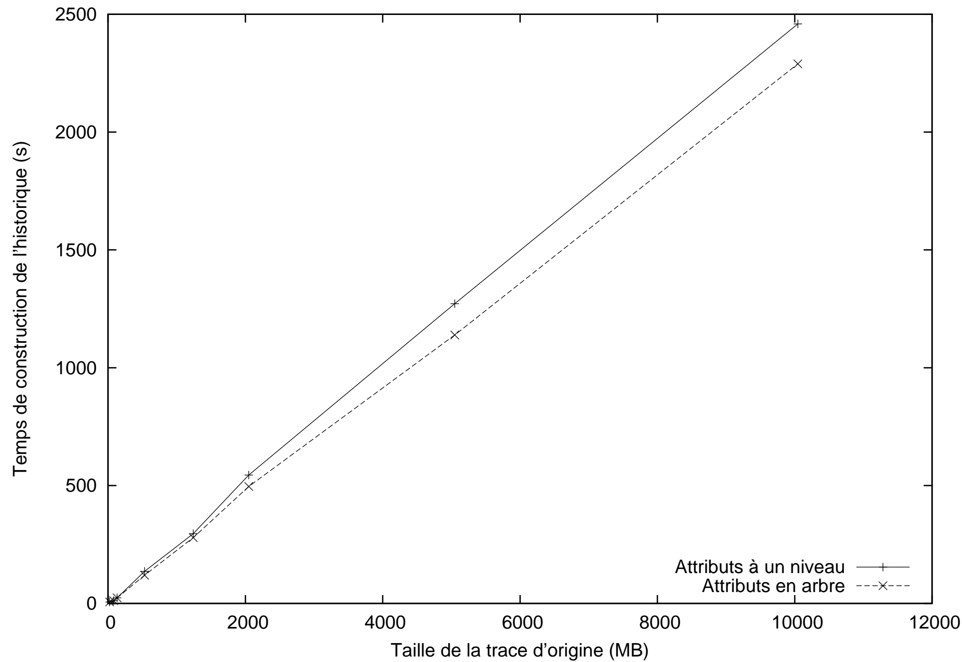

FIGURE 4.9: Organisation des attributs en arbre, construction de l'historique d'état

traces, passer à travers en mettant à jour l'état transitoire et enregistrer les clichés de l'état en mémoire.

Eh oui, il était plus rapide pour LTTV de lire la trace et générer son système d'état, que pour seulement lire la trace depuis Java à travers l'interface JNI ! Il s'agit d'un bon exemple de l'impact sur la performance causé par l'utilisation d'un pont comme JNI.

Bien entendu il est difficile de comparer les deux systèmes directement, puisque l'information enregistrée n'est pas exactement la même. Cela permettait toutefois de donner un bon ordre de grandeur du comportement, ainsi qu'une bonne cible de performance à atteindre !

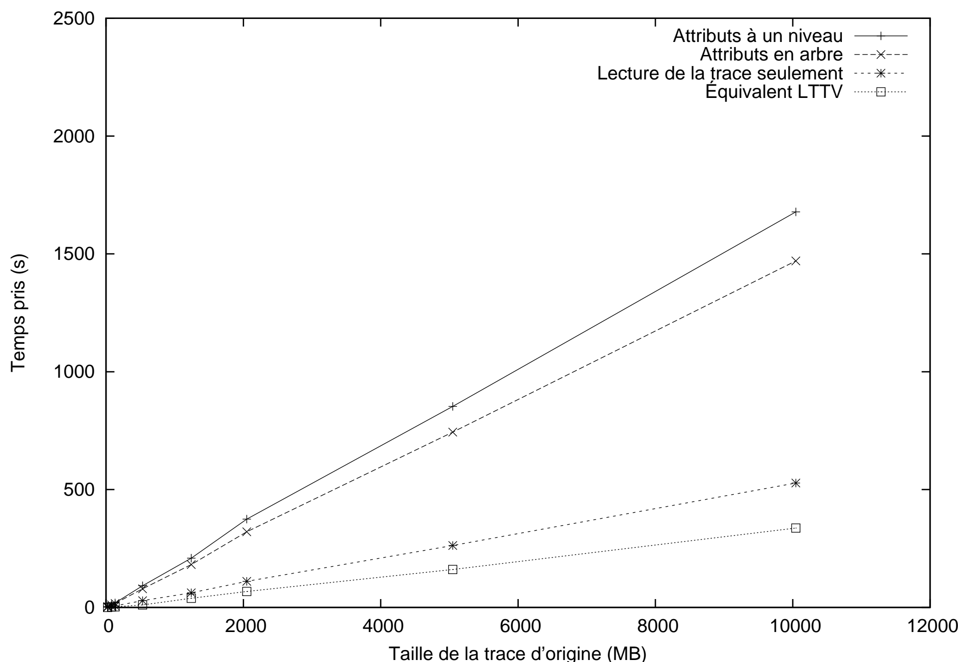

FIGURE 4.10: Organisation des attributs en arbre, système d'état sans historique

En observant les deux figures, on remarque une amélioration non-négligeable de la performance du système. La différence semble être la même dans les deux figures, cela indique que le temps sauvé est bien à l'étape de mise à jour de l'état transitoire.

Par contre, en comparant la performance avec celle de LTTV, nous constatons qu'il reste encore bien du travail à faire.

4.4.2 Éviter la création d'objets Java

Nous avons discuté plus tôt des valeurs d'état, qui peuvent dans notre modèle contenir soit une valeur entière soit une chaîne de caractères. Côté implémentation, ceci a été réalisé avec un objet *wrapper* qui spécifiait son contenu et comment celui-ci devait ultimement être écrit sur le disque.

Cependant, ceci impliquait la création d'un nouvel objet de ce genre pour chaque changement d'état inséré dans le système. Il y aurait sans doute un impact de performance dû à la création répétée de nouveaux objets. Par contre, la documentation de Java affirme que l'impact des `new` est souvent surestimé dans ce language. Il fallait faire le test.

Le code du programme a été modifié de telle sorte qu'on ne crée pas de nouvel objet à chaque changement d'état. À la place, on créerait un objet par attribut, et celui-ci serait

réutilisé une fois que son ancien contenu ait été écrit au disque.

Ceci rendait le code moins flexible et légèrement plus lourd, mais il y avait au total moins de `new`. La figure 4.11 montre la différence de performance pour la création complète de l'historique et la mise à jour de l'état transitoire sans historique, tel que nous avons montré précédemment. Le temps de lecture de la trace n'était pas touché par ce changement, donc demeurait le même.

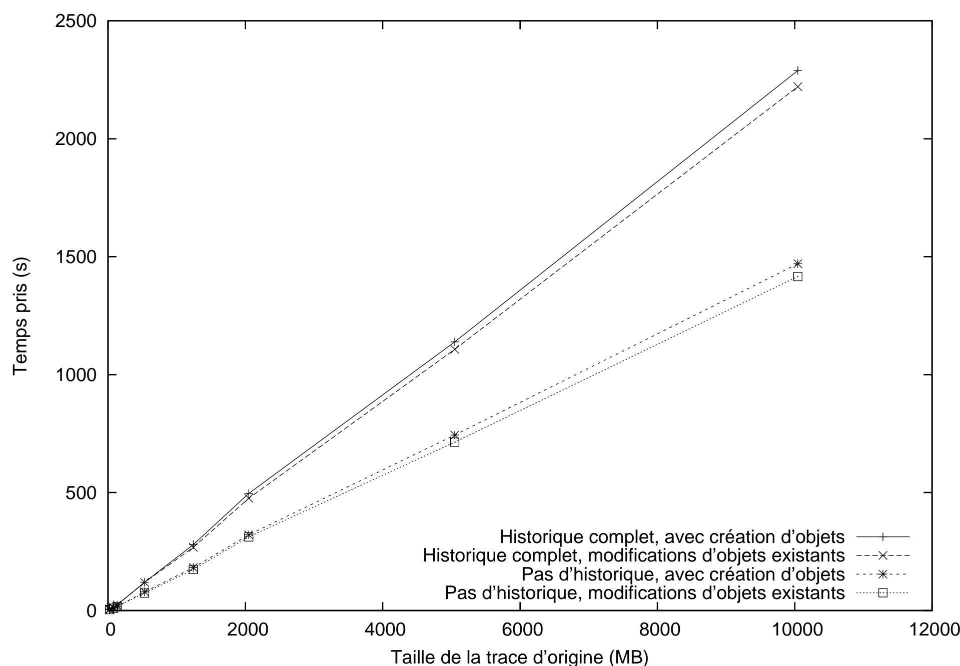

FIGURE 4.11: Impact d'allouer ou non des objets Java à chaque insertion

Il y a bel et bien une augmentation de performance au niveau de la mise à jour de l'état transitoire, mais elle est pour ainsi dire minime. Nous avons jugé que cela ne justifiait pas la complication additionnelle du code, et la modification n'a pas été conservée.

4.4.3 Éviter les attributs redondants

Ce n'est pas tout d'avoir un bon système performant, il faut aussi bien l'utiliser. En observant pas-à-pas l'exécution du programme, nous avons constaté que dans quelques cas nous insérons dans le système des changements d'états inutiles.

Par exemple, pour chaque événement de type "appel système", nous enregistrons le numéro (entier) dans un attribut, mais aussi le nom (chaîne de caractères) de l'appel système dans un autre. À chaque fois. LTTV permet d'accéder aux appels systèmes des deux manières, nous avons donc fait ainsi à l'origine pour s'approcher de ce comportement.

Mais en y songeant mieux, il est tout à fait inutile d'insérer deux (ou plus) changements d'états identiques pour un seul événement. Les intervalles créés auront les mêmes bornes, seulement des attributs et valeurs différentes. Cette information était mieux représentée sous forme d'un seul attribut.

Nous avons donc modifié le gestionnaire d'état pour que celui-ci enregistre seulement le numéro de l'appel système, et non le nom. Nous avons également modifié les IRQ et les attributs *Status* des CPU, qui fonctionnaient de la même manière.

Puisque le mapping numéro-nom (pour les appels systèms, IRQs, etc.) reste le même pour une trace, il devient possible de transférer cette méta-information une seule fois, et non pas à chaque événement. Nous avons rajouté cette information au gestionnaire d'état utilisé. Nous n'avons pas fourni de structure de référence générique pour ce type d'information, puisqu'elle peut varier grandement d'un système d'état à un autre. Le gestionnaire d'état peut la spécifier de la manière qu'il le veut. Nous ne perdons pas d'information comme tel, il y en a seulement moins qui est enregistrée dans l'historique.

Il ne s'agit donc pas ici d'une modification au système d'état comme tel, mais plutôt à l'optimisation de son utilisation. Nous avons tout de même mesuré la différence de performance, de manière à prouver qu'il est aussi très important de bien optimiser le modèle qu'on utilise.

Observons la figure 4.12 attentivement. Les deux courbes du haut représentent le temps pris pour construire l'historique au complet, avant et après la modification.

Les deux courbes suivantes représentent la différence pour l'étape de mise à jour de l'état transitoire seulement, sans historique. Nous avons également rapporté les courbes du temps de lecture seulement ainsi que du temps pris par LTTV pour son système d'état, à des fins de comparaison.

Nous constatons donc que la majorité du temps est sauvé à l'étape d'écriture de l'historique sur disque. Ceci devient évident lorsqu'on observe la figure 4.13, qui montre la différence de taille entre les fichiers d'historique produits.

La taille de l'historique est coupée pratiquement en deux ! Ceci est logique, puisque nous avons remplacé à plusieurs endroits deux attributs par un seul, et nos traces contiennent principalement des événements d'appels système.

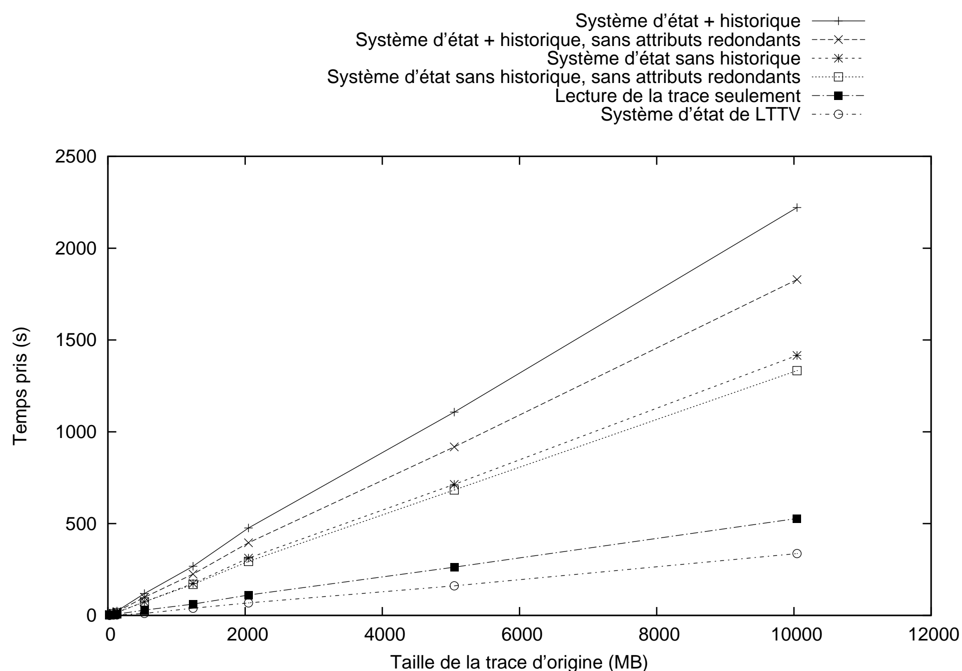

FIGURE 4.12: Retrait des attributs redondants, temps de construction

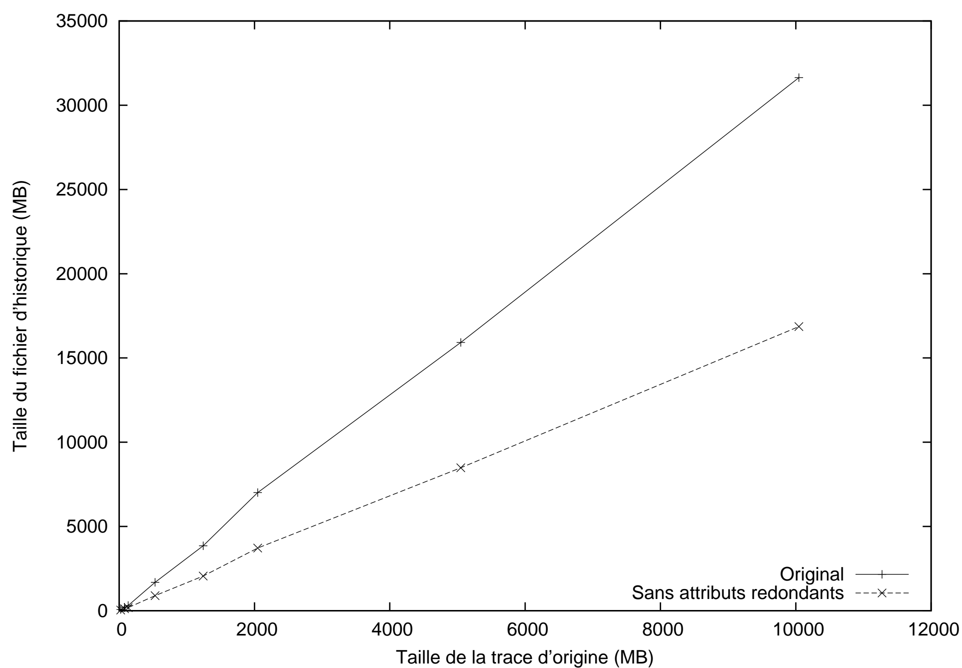

FIGURE 4.13: Retrait des attributs redondants, taille de l'historique résultant

Il va sans dire que cette modification a été conservée, et que nous serons à l'avenir prudents avant de créer de nouveaux attributs. Les fichiers d'historique sont maintenant environ une fois et demi plus grands que les traces à partir desquelles ils sont créés. Cela correspond à nos attentes, compte tenu que nous insérons un changement d'état par événement à des fins de statistiques (x1) et qu'environ la moitié des événements, en moyenne, modifient l'état comme tel (x0.5).

4.4.4 Séparer le système en différents fils d'exécution

Jusqu'à maintenant, tout le système présenté ici n'utilisait qu'un seul fil d'exécution. Comme nous avons pu le constater aux sections précédentes, un ralentissement soit à l'étape d'écriture sur le disque ou soit au traitement des changements d'états ralentissait tout le système.

Il serait sans doute souhaitable d'utiliser plusieurs fils, pour pouvoir au minimum éviter d'interrompre le traitement pendant qu'on écrit au disque. De plus, puisque la trace et le fichier d'historique seront souvent situés sur le même disque, cela veut dire que nous lisons *et* écrivons de grandes quantités d'information sur le même support en même temps !

Dans une première étape, nous mesurerons l'impact de séparer la partie “arbre à historique” du système dans un fil à part, de manière à ce que le traitement des événements ne soit pas ralenti par l'écriture au disque. Le système d'état “central” lira les événements et générera les intervalles d'états, qui seront envoyés à l'autre fil pour placement dans l'historique. Pour ce faire, nous utiliserons les `BlockingQueue` de Java.

Cette abstraction permettra plus tard de facilement intégrer différentes méthodes de stockage pour l'historique.

Une fois construit, l'historique résultant sera exactement le même. Les requêtes se comporteront également de la même façon. Nous modifions ici seulement le comportement lors de la construction de l'historique d'état.

La figure 4.14 montre la différence de performance entre les deux méthodes. Nous n'avons pas mesuré la mise à jour de l'état transitoire, puisque les deux méthodes étaient maintenant très différentes donc peu comparables. Toutefois le temps de lecture de la trace est rapporté, puisqu'il n'est pas affecté par ces changements.

Nous notons encore une fois une amélioration non-négligeable de la performance de construction de l'historique d'état. Toutefois, la séparation de la *lecture* de la trace l'améliorera encore plus. Ce changement sera présenté à la prochaine section.

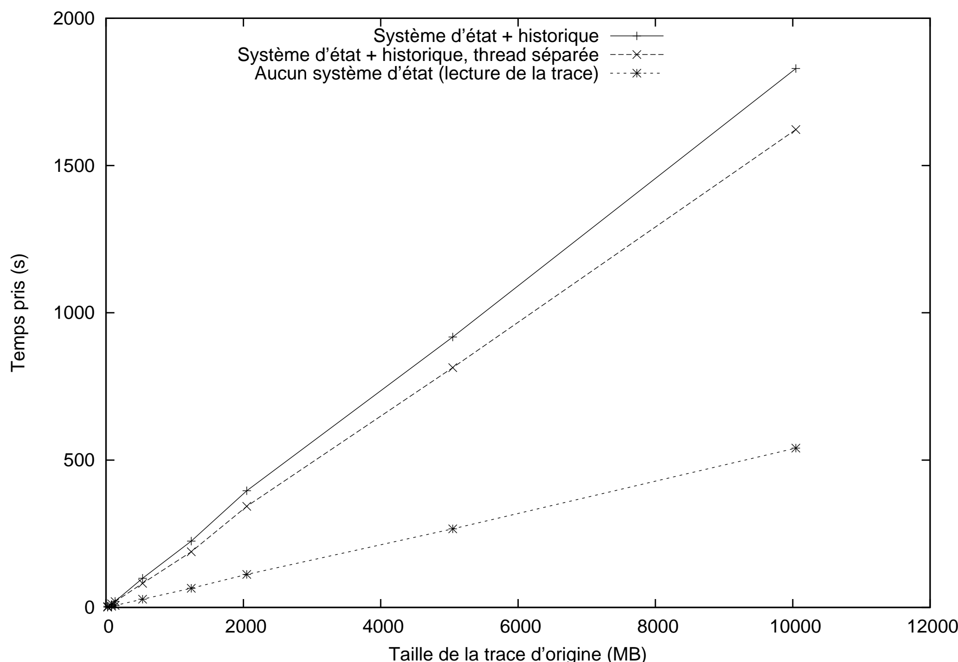

FIGURE 4.14: Traitement et insertion des intervalles dans un fil d'exécution séparé

4.4.5 Lire les événements directement sans JNI (format VSTF)

Nous venons d'évaluer la séparation de l'écriture de l'historique en un fil séparé. Il serait également souhaitable de séparer la lecture de la trace similairement, de manière à ce que le coeur du système qui effectue le traitement des événements et des attributs ne soit aucunement bloqué par le disque.

Malheureusement, le programme de lecture de TMF (qui utilise l'interface JNI de la librairie de lecture de traces) ne semble pas être facilement séparable, à moins de faire des copies en mémoire de toute l'information désirée (donc, éventuellement de toute la trace, ce qui semble être un piètre solution).

Le prochain problème sur la liste était de tester la lecture de traces directement depuis Java, sans passer par une interface JNI (qui est très taxant sur la performance pour notre cas d'utilisation). De plus, comme nous contrôlerions le lecteur de traces, nous pourrions nous assurer qu'il puisse facilement être séparé du reste. Nous règlerions donc le problème de de séparation de la lecture en même temps.

Comble de malheur, le format de trace LTTng est très complexe. Une implémentation Java du lecteur prendrait sans aucun doute plusieurs mois. Il nous est alors venu l'idée de concevoir un nouveau format de trace, qui serait *très* simple. Ceci a donné naissance au format VSTF (pour *Very Simple Trace Format*), qui est décrit à l'annexe B.

Ce format très simple a permis d'écrire très rapidement un lecteur et un écrivain implémentés aussi en Java. L'idée était ensuite de convertir les traces LTTng, en utilisant le lecteur de TMF, au format VSTF. Ceci nous a donc donné un nouveau jeu de traces, équivalentes au jeu initial mais dans un format que nous pourrions lire directement en Java.

Les temps de conversion n'ont pas été mesurés comme tel, mais ils étaient *très* longs. Beaucoup trop longs pour convertir de façon transparente lors de l'ouverture d'une trace. Cependant les traces résultantes devraient être représentatives de traces LTTng lues directement depuis Java.

Le format étant peu optimisé, les traces en VSTF étaient légèrement plus grandes que leurs équivalentes en format LTTng. Cependant dans les résultats qui suivent, nous continuerons d'utiliser les tailles en format LTTng même si ce ne sont pas ces traces qui auront été réellement lues, dans le but de garder les graphiques comparables.

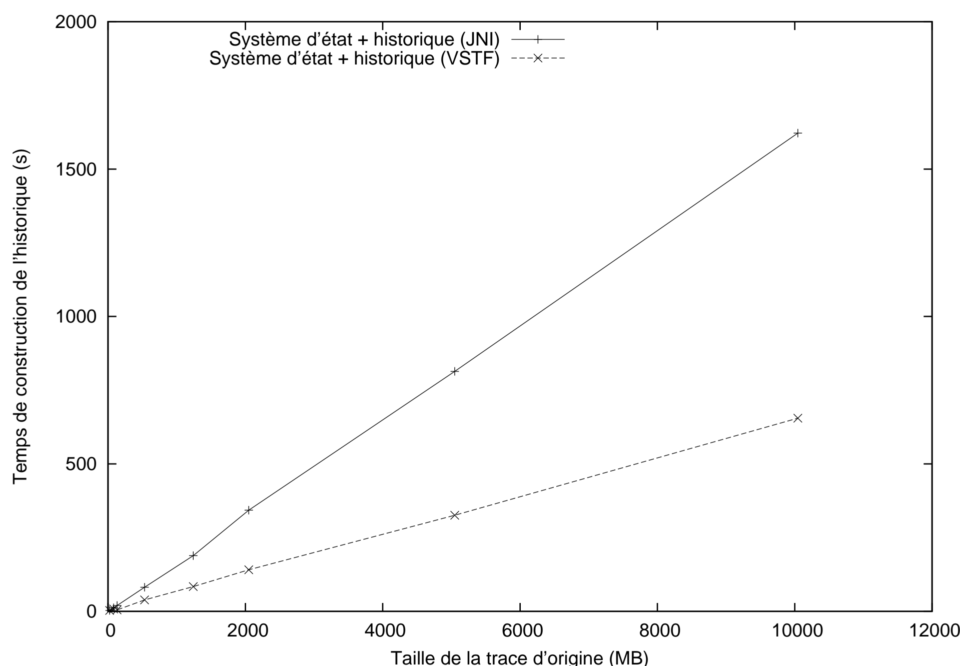

FIGURE 4.15: Utilisation du lecteur VSTF en Java vs. lecteur LTTng via JNI
Construction de l'historique d'état

Encore une fois, ce changement n'affecte que l'étape de construction de l'historique. Les requêtes n'utilisent que le fichier d'historique, pas les traces, donc ne sont pas affectées. La figure 4.15 montre la construction complète de l'historique en comparant la méthode JNI+LTTng et la méthode VSTF directe. Nous voyons tout de suite l'effet extrêmement bénéfique des deux avantages mentionnés, qui sont la lecture directe en Java et la séparation de la lecture en un fil d'exécution distinct.

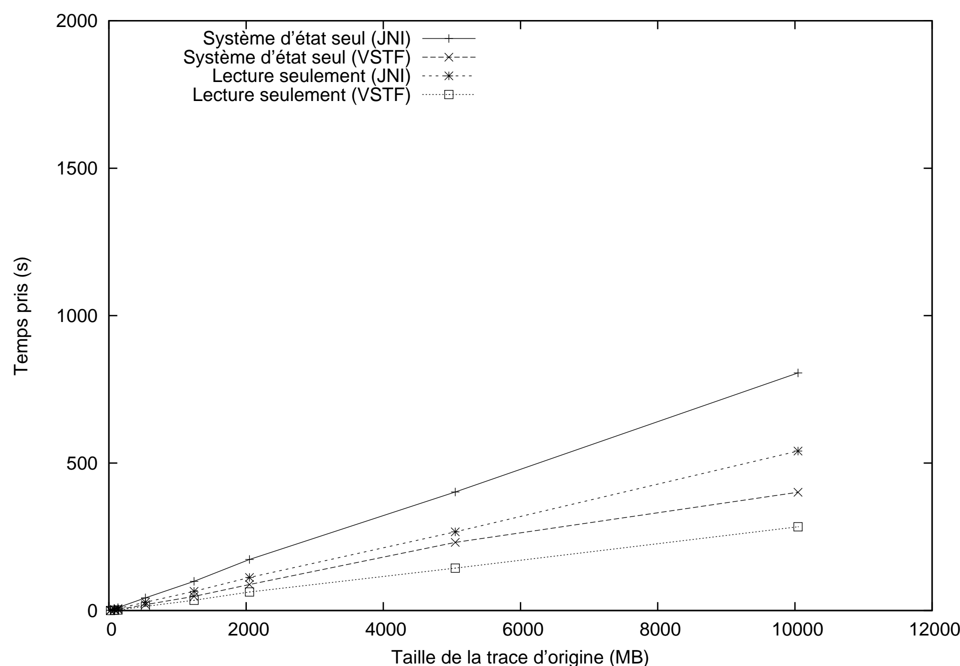

FIGURE 4.16: Utilisation du lecteur VSTF en Java vs. lecteur LTTng via JNI
État transitoire sans historique

La figure 4.16 montre les différences pour les étapes de lecture + système d'état en mémoire (sans historique), et la lecture seule. Les deux courbes du haut représentent la méthode JNI+LTTng, et les deux du bas la nouvelle méthode.

Sur la figure 4.17, nous rapportons simplement toutes les courbes de la méthode VSTF, ainsi que celle du système d'état de LTTV. Nous pouvons constater que enfin, l'étape de lecture de la trace est plus rapide que la construction du système d'état de LTTV.

Nous remarquons également que le système d'état sans historique, qui se rapproche du comportement de LTTV, est aussi très près en terme de performance.

Enfin, lorsqu'on y ajoute l'écriture de l'historique, nous restons tout de même en-deçà d'un facteur de deux par rapport à LTTV.

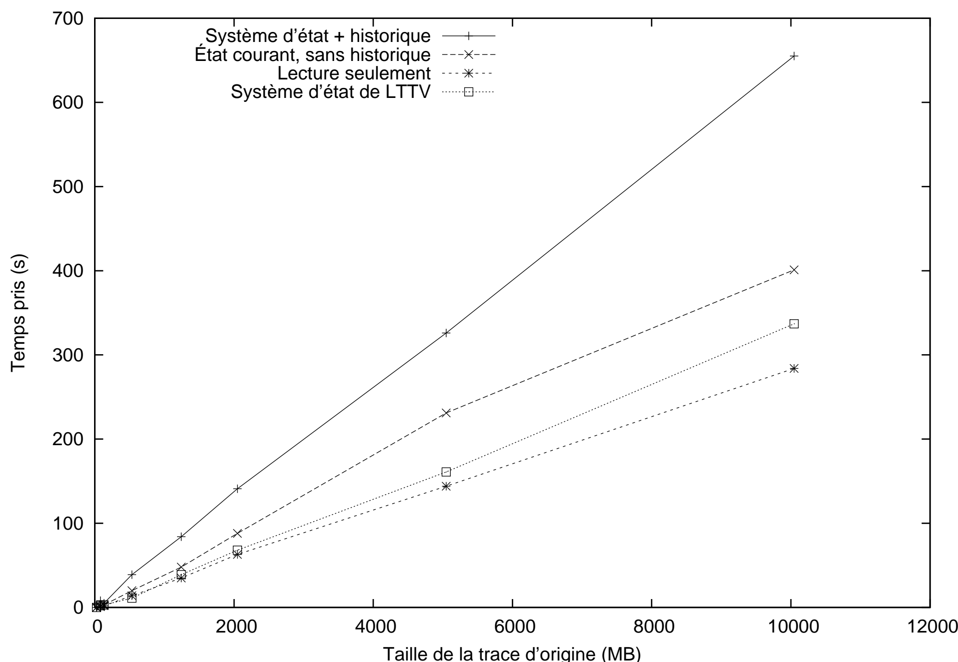

FIGURE 4.17: Comparaison du lecteur VSTF vs. LTTV

4.4.6 Conserver des références aux attributs entre les événements

De nouvelles rondes de profilage ont montré qu'une bonne quantité de temps était encore passée à faire le hachage des chemins d'accès des attributs lors de la construction de l'historique. Le changement apporté à la section 4.4.1 avait aidé la situation, mais il semblerait qu'on puisse l'améliorer davantage.

En regardant l'exécution pas-à-pas encore une fois, nous avons remarqué le phénomène suivant : ce sont souvent les mêmes attributs qui sont modifiés pour plusieurs événements d'affilée. Ceci est logique, puisqu'un seul processus est ordonnancé sur un CPU en particulier à la fois, la série d'événements provenant de ce CPU va toujours affecter le même processus, donc la même branche dans l'arbre d'attributs.

Suite au changement de la section 4.4.1, nous allions chercher des “pointeurs” vers les attributs représentant le CPU courant et le processus courant une seule fois pour chaque événement, et réutilisions ceux-ci pour chaque changement d'état.

L'amélioration proposée ici est de conserver en mémoire deux tableaux de ces pointeurs, un pour les attributs représentant les CPU et le deuxième pour les processus courants (la taille de ces tableaux étant égale au nombre de CPUs dans la trace). Ces pointeurs seront conservés d'un événement à l'autre autant que possible.

Les attributs des CPUs eux ne devraient pas changer après avoir été vus la première

fois. Ceci évitera des hachages systèmatiques à chaque événement. Les processus courants eux vont changer par contre. Il faudra donc s'assurer de mettre à jour ces pointeurs lorsque nous rencontrerons des événements d'ordonnancement (appelés *sched_schedule* dans les traces LTTng).

Le changement proposé ici est un changement au gestionnaire d'état, donc pas au système d'état en tant que tel. Cependant, ceci démontre encore une fois que l'optimisation de ce gestionnaire est tout aussi cruciale pour obtenir de bonnes performances du système.

Encore une fois, ce changement n'affecte que l'étape de construction de l'historique.

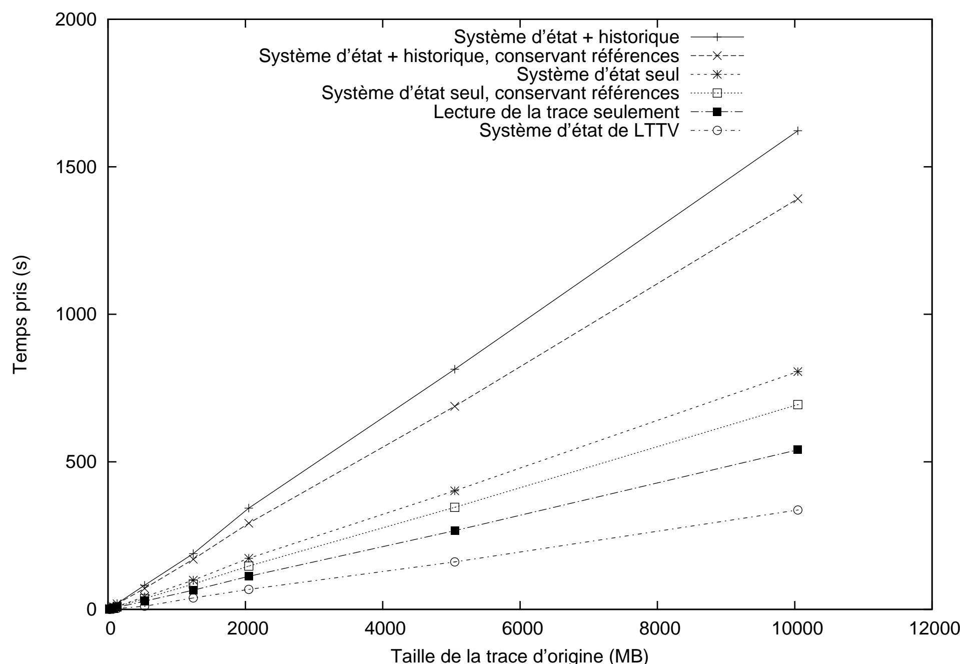

FIGURE 4.18: Conservation de références aux attributs entre les événements (méthode JNI)

La figure 4.18 montre la différence de performance lorsqu'on conserve les références aux attributs entre les événements ou pas. Les deux courbes du haut montrent le temps pris pour la construction de l'historique complet, et les deux suivantes pour l'état transitoire seulement. Nous rapportons encore une fois les temps pour la lecture de la trace et pour le système d'état de LTTV.

Cette figure montre les différences de temps pour la méthode JNI. Pour la méthode VSTF, introduite à la section précédente, ce changement n'affectait pas les résultats (qui étaient déjà meilleurs). En effet, la méthode VSTF traite les événements dans un fil d'exécution séparé de la lecture de la trace, et ce n'est pas ce fil qui limite la performance. Néanmoins, le changement sera appliqué pour les deux gestionnaires d'état.

4.4.7 Trier les intervalles dans les noeuds

Jusqu'à maintenant, nous avons pris pour acquis que les événements n'étaient pas nécessairement triés par ordre chronologique dans les traces. Ceci impliquait que les intervalles d'état créés pouvaient ne pas nécessairement être triés par temps de fin.

Après vérification avec les auteurs, il semble plutôt que la librairie de lecture de traces LTTng s'assure que les événements seront bien ordonnés. Ceci permet donc de faire certaines hypothèses autant au niveau de la construction de l'historique que lors des requêtes.

Lors de la construction, plus précisément de l'insertion des intervalles d'état dans les noeuds de l'arbre, nous nous assurons présentement que les deux bornes de l'intervalle sont bien contenues dans celles du noeud. Si nous sommes assurés que les intervalles arriveront en véritable ordre croissant de temps de fin, cette vérification n'est plus nécessaire.

Lors de la lecture d'un noeud pendant une requête, nous comparons présentement le temps cible avec les deux bornes de chaque intervalle. Avec la garantie de tri, il nous est possible de faire une recherche dichotomique pour trouver l'intervalle de départ, et ensuite de seulement comparer avec les temps de début des intervalles subséquents.

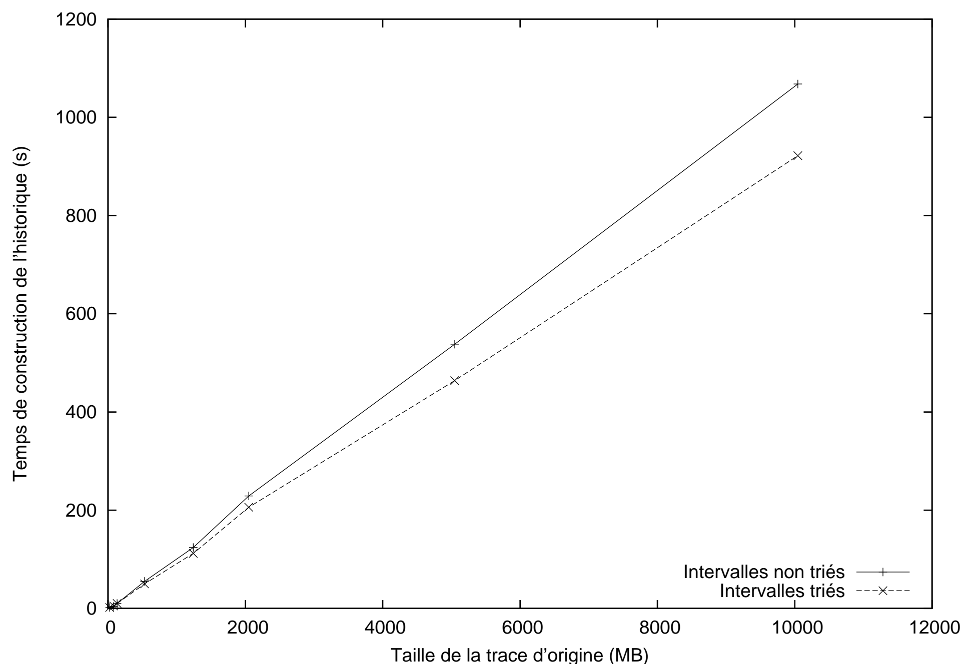

FIGURE 4.19: Impact de trier les intervalles dans les noeuds, construction de l'historique

Les figures 4.19 à 4.21 montrent comment la garantie que les intervalles soient triés affecte les temps de construction et de requêtes.

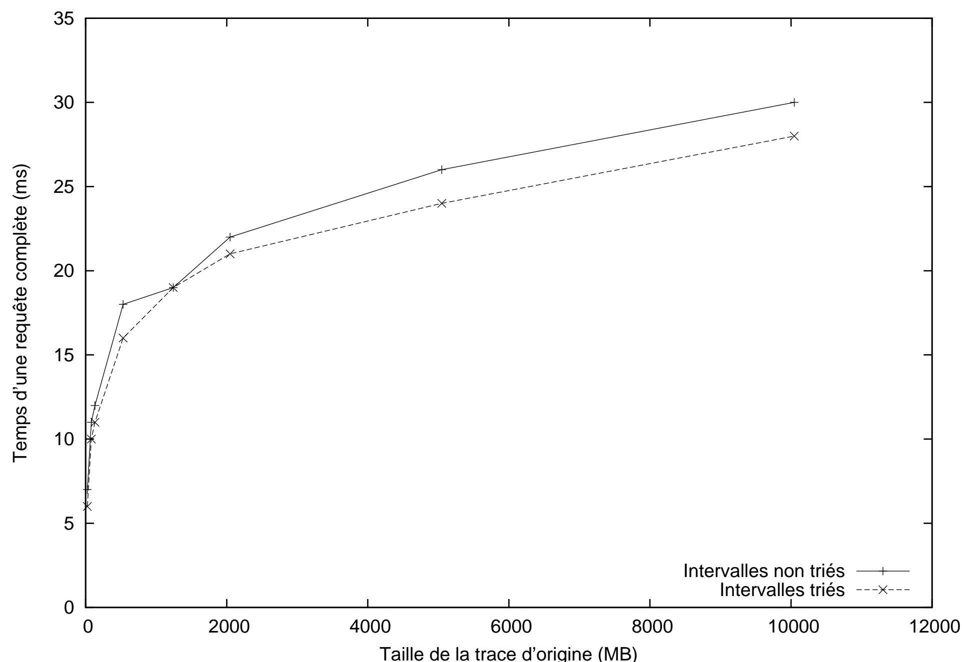

FIGURE 4.20: Impact de trier les intervalles dans les noeuds, requêtes complètes

On voit tout de suite l'effet bénéfique de cette garantie, autant au niveau de la construction de l'historique que lors des requêtes. Comme toutes les traces que nous utilisons ici sont assurément triées, nous conserverons cette modification.

Pour l'instant, une vérification a été ajoutée au niveau du gestionnaire d'événements pour s'assurer que cette garantie est vraie. Pour supporter des traces où les événements ne seraient pas exactement triés, le plus simple serait sans doute de trier les intervalles lors de la fermeture des noeuds, et d'utiliser la garantie de tri pour les requêtes puisque cela donne une amélioration non-négligeable de la performance des requêtes.

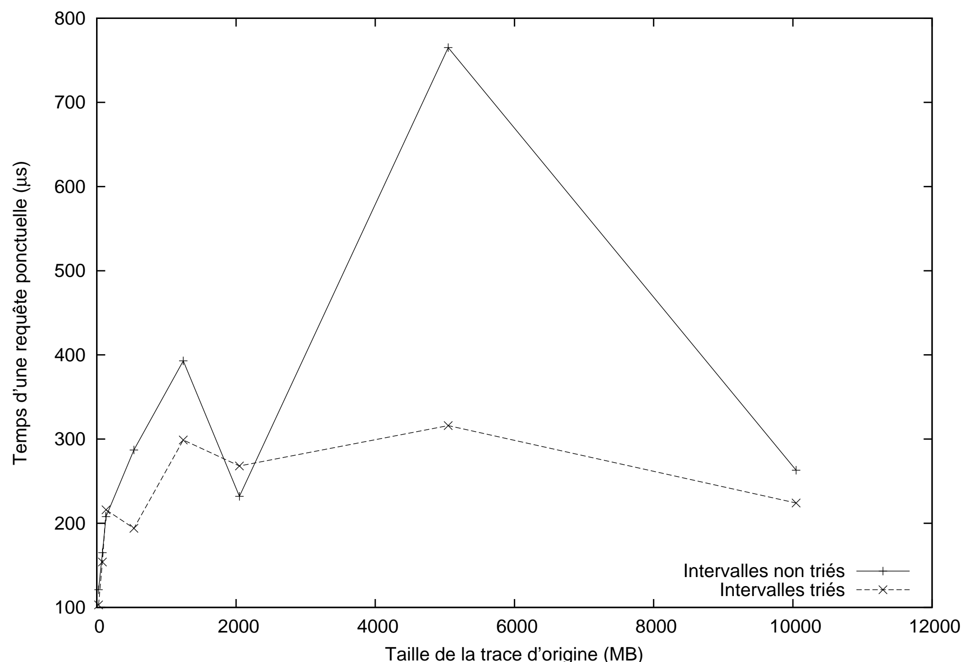

FIGURE 4.21: Impact de trier les intervalles dans les noeuds, requêtes ponctuelles

4.4.8 Trier les noeuds de l'arbre dans le fichier d'historique

La prochaine expérience consiste à trier les *noeuds* de l'arbre à historique dans le fichier. Ceci signifie que le noeud racine serait situé en premier, suivi de ses enfants directs, suivi des petits-enfants, et ainsi de suite. À chaque niveau, les noeuds seraient ordonnés de “gauche à droite”, donc en ordre chronologique.

L'idée a été inspiré par ce que fait le démon d'initialisation Upstart avec les fichiers qui doivent être lus au démarrage du système (Remnant, 2010).

En agençant les noeuds de la sorte, le disque n'aurait pas à revenir “en arrière” lors des requêtes. Comme une requête consiste à explorer une seule branche de l'arbre, en commençant par la racine, les blocs seraient lus en parcourant le fichier du début à la fin, dans une seule direction. L'hypothèse est que ce tri permettrait des requêtes plus rapides, mais qu'il y aurait un prix à payer pour le réagencement des blocs une fois la construction de l'historique terminée.

La figure 4.22 montre le temps de construction avec et sans le tri des blocs à la fin. À noter qu'on doit absolument attendre que l'historique soit complètement construit, puisque de nouveaux noeuds peuvent être ajoutés à tous les niveaux y compris à la racine, ce qui requerrait de décaler tout le fichier.

Comme on peut le constater, le tri des blocs augmente considérablement le temps de

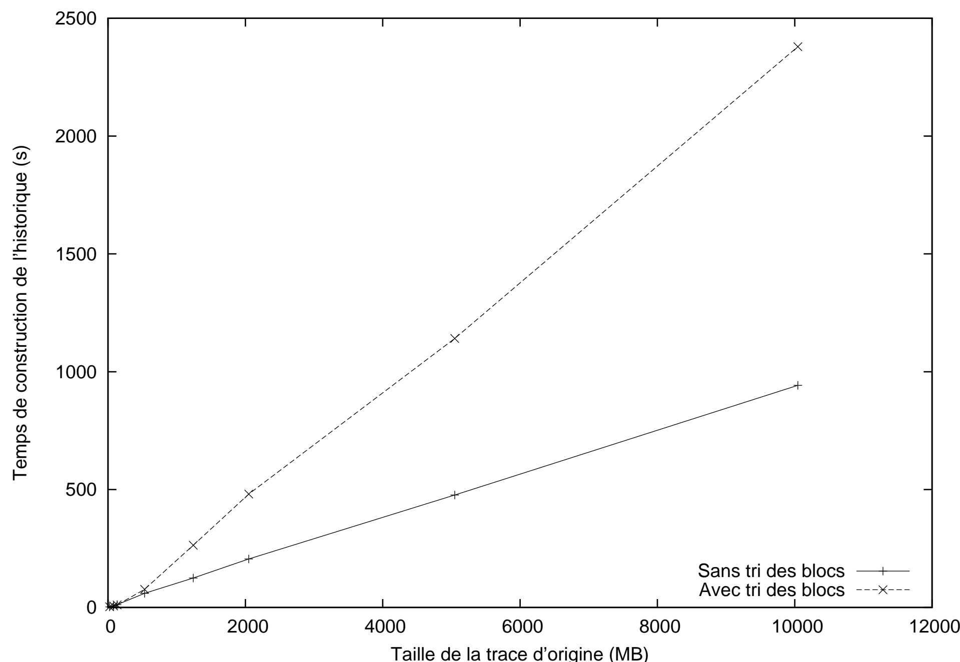

FIGURE 4.22: Tri des blocs (noeuds) dans le fichier, construction de l'historique

construction, particulièrement pour les traces plus volumineuses. Les petits historiques ne voient presque pas d'impact, puisqu'ils peuvent être totalement contenu en mémoire cache.

Les figures 4.23 et 4.24 montrent la différence pour les temps de requêtes des deux méthodes. On peut constater que le tri des blocs améliore, si ce n'est que légèrement, les temps de requêtes pour les historiques de petite taille (traces allant jusqu'à 1 GB). Pour les plus grandes traces, l'effet est inversé et les résultats sont pires.

Nous pouvons conclure que, pour les petits historiques, l'effet escompté est atteint : le fait que les blocs à lire pour les requêtes soient placés en ordre dans le fichier évite des mouvements au disque et améliore la performance.

Cependant, l'effet est totalement perdu pour les gros historiques. D'autant plus que chaque niveau additionnel de l'arbre est exponentiellement plus grand que le précédent, l'effet de trier les blocs ne fait qu'éloigner en moyenne les noeuds de niveaux subséquents. Cela empire les temps de requêtes. En plus, les temps de construction sont augmentés de manière inacceptable.

Bref, le changement proposé ici améliore les petits historiques, mais a de sérieux problèmes à grande échelle. Comme c'est le comportement à grande échelle qui nous intéresse ici, la modification ne sera pas conservée.

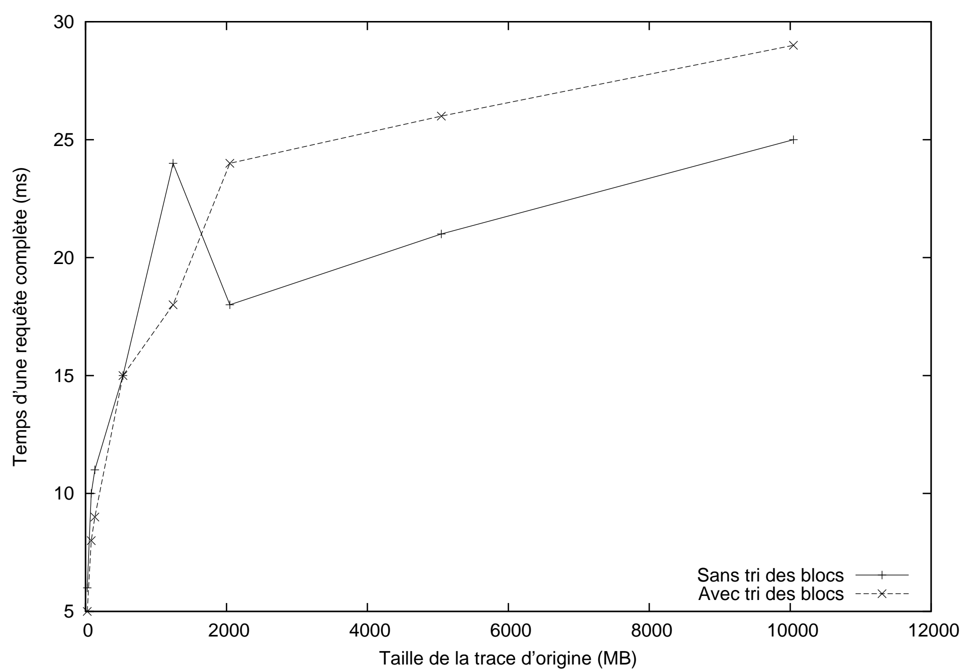

FIGURE 4.23: Tri des blocs (noeuds) dans le fichier, requêtes complètes

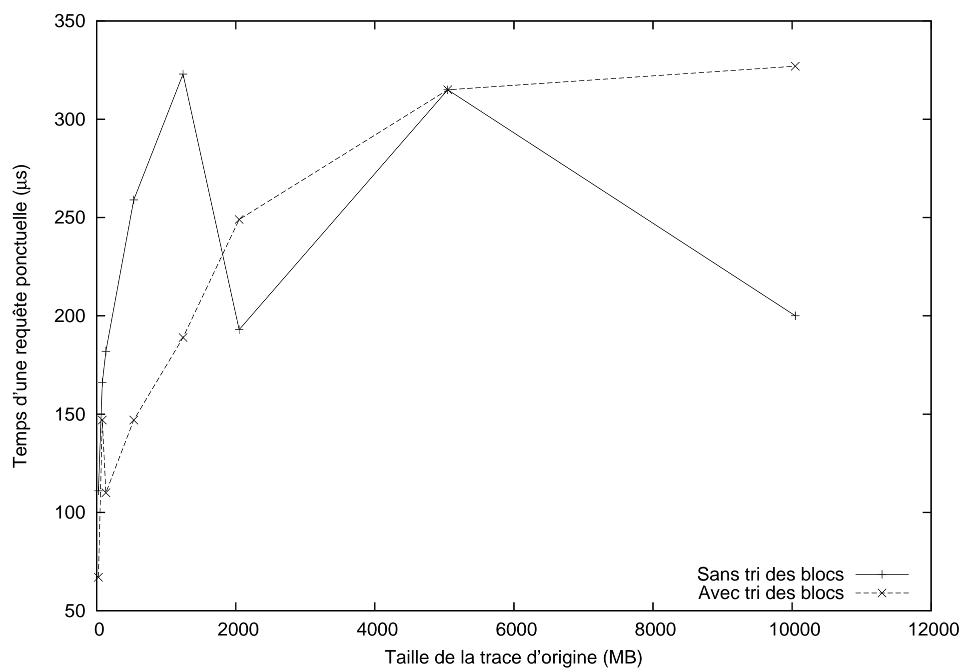

FIGURE 4.24: Tri des blocs (noeuds) dans le fichier, requêtes ponctuelles

4.4.9 Taille de blocs non-multiple de 4KB

Tel que nous l'avons vu au tout début de ce chapitre, il est possible de configurer la taille sur disque des blocs de l'arbre à historique et ainsi d'obtenir divers résultats. Jusqu'à présent, nous avons supposé que des tailles de blocs multiples de 4096 octets seraient idéales, puisqu'elles resteraient alignées avec la taille des pages en mémoire, ainsi que la taille (par défaut) des blocs des systèmes de fichiers les plus courants.

Nous étions curieux de savoir si cette hypothèse était vraiment valide, ou si nous nous étions rajouté des contraintes inutilement. Nous avons donc effectué les tests habituels (construction, requêtes complètes, requêtes ponctuelles) pour différentes tailles de blocs, alignés sur 4 KB ou non. Nous avons également rapporté le nombre de niveaux des arbres à historique résultants, de manière à s'assurer que les différences dans les temps de requêtes relèvent bien de la taille des blocs et non simplement d'une différence du nombre de niveaux.

Nous avons d'abord effectué la comparaison avec des tailles de blocs de 32 et 64 KB (alignés sur 4096 octets) et 34 000 et 62 000 octets (non-alignés). Les résultats sont présentés dans les figures 4.25 à 4.28.

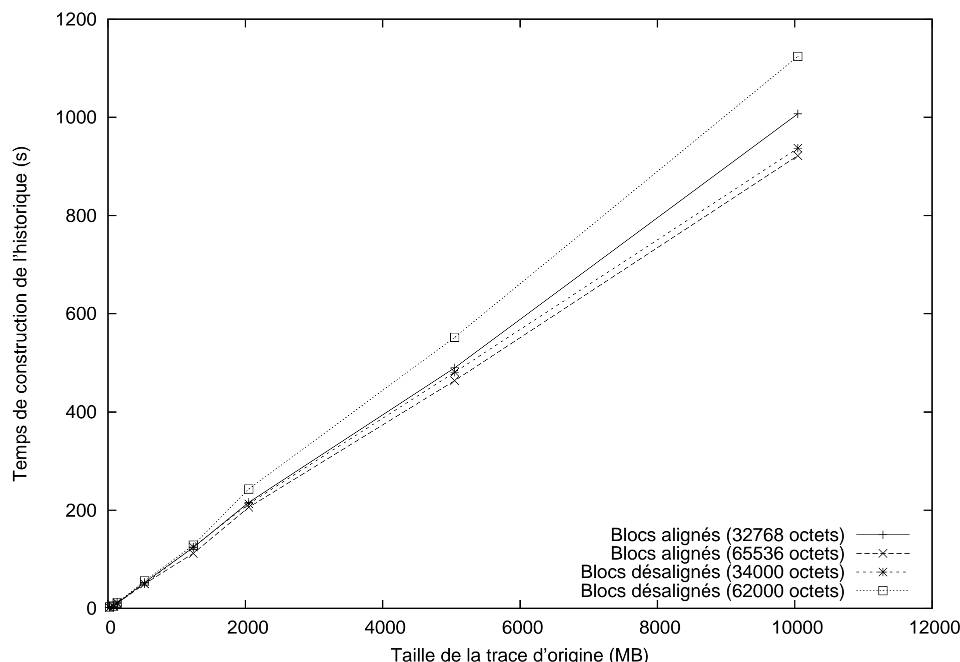

FIGURE 4.25: Blocs désalignés, construction de l'historique

L'arbre comportant des blocs de 62 000 octets présente une régression de performance notable par rapport aux autres tailles, autant au niveau de la construction que pour les

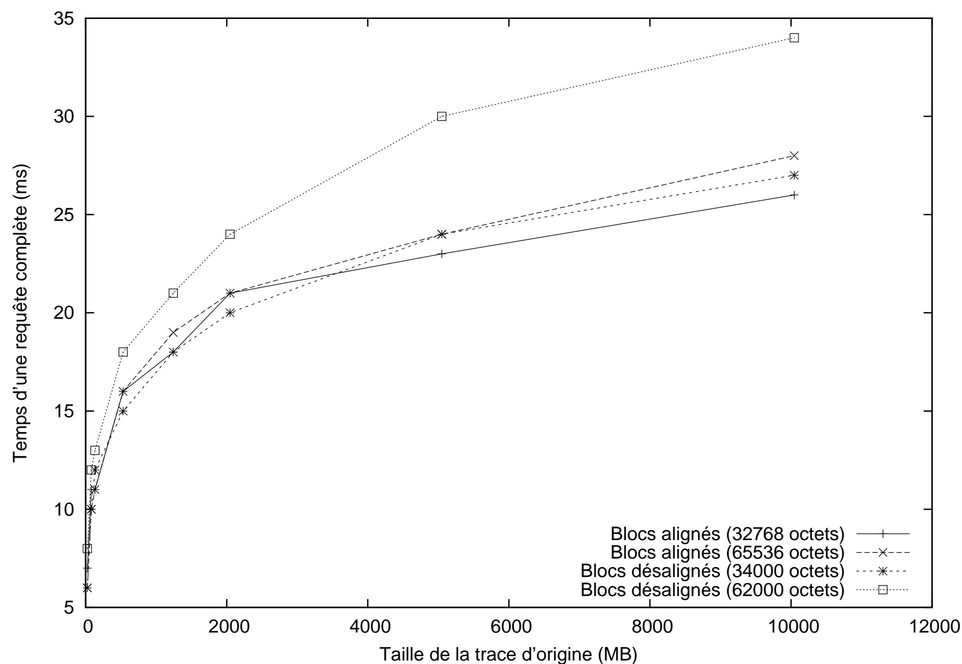

FIGURE 4.26: Blocs désalignés, requêtes complètes

requêtes complètes. La taille de 34 000 octets a un comportement plus rapproché de son homologue de 32 768 octets. La profondeur des arbres reste presque toujours la même pour chaque groupe de tailles.

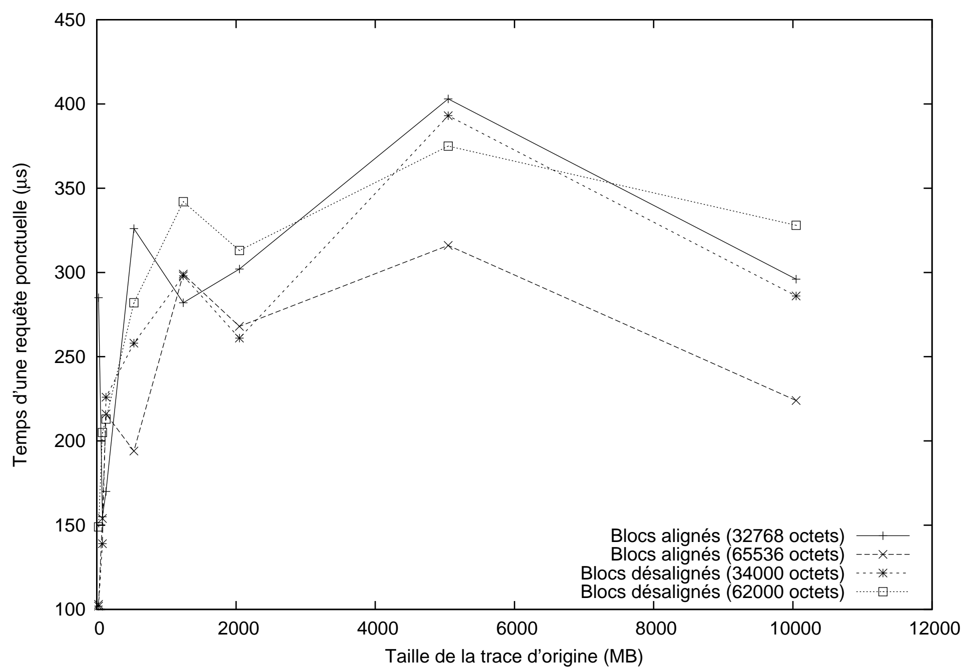

FIGURE 4.27: Blocs désalignés, requêtes ponctuelles

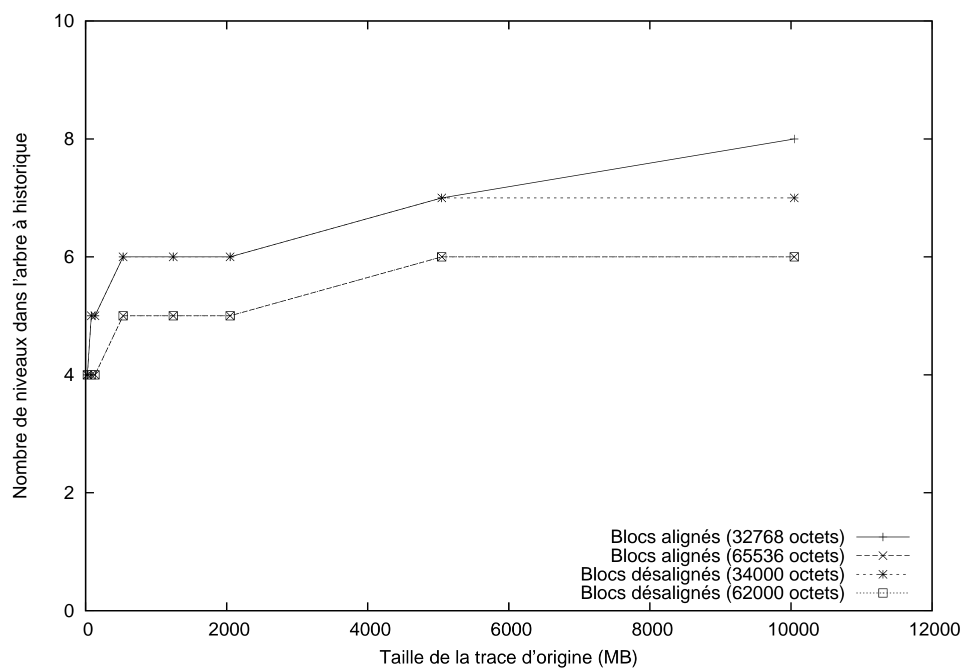

FIGURE 4.28: Blocs désalignés, profondeur de l'arbre à historique

Il serait également intéressant de comparer des tailles alignées et non-alignées, mais avec des blocs relativement plus petits. Peut-être l'impact sera-t-il plus important à plus petite échelle (où la différence du nombre de pages de mémoires à lire sera proportionnellement plus grande).

Les figures 4.29 à 4.32 montrent les résultats des mêmes tests, mais effectués avec des tailles de blocs de 4096 (aligné) et 5120 octets (= 5 KB, mais désaligné par rapport à 4 KB).

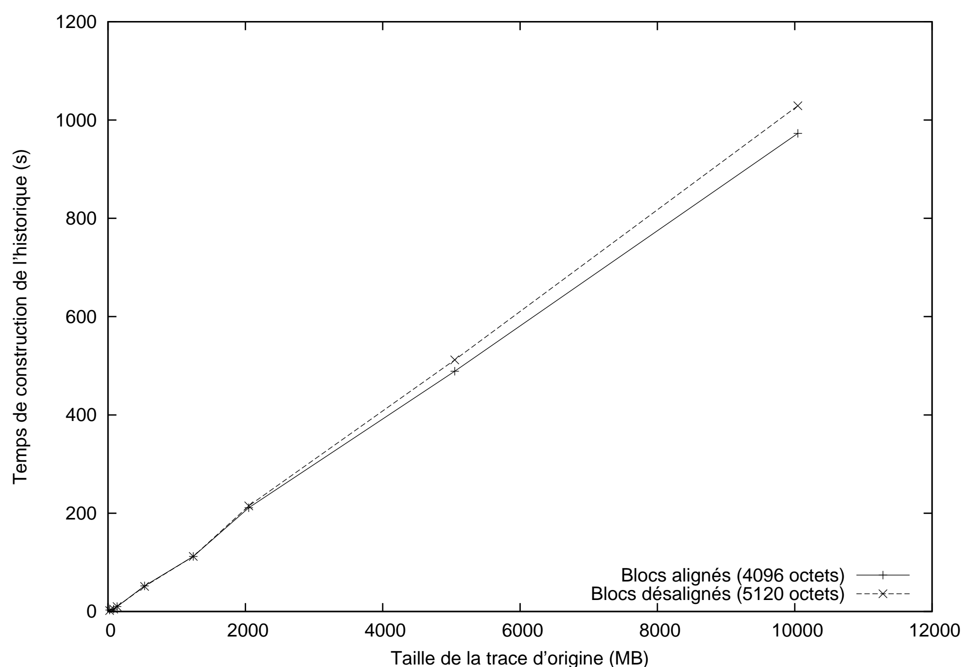

Figure 4.29: Blocs désalignés, construction de l'historique

Les différences sont ici plus importantes, et la taille de 5120 octets, qui est pourtant désalignée, obtient de meilleures performances au niveau des requêtes.

Cependant, on comprend pourquoi en observant la profondeur des arbres résultants (figure 4.32) : les blocs de 4096 octets sont organisés en un arbre beaucoup plus profond. Donc, plus de blocs doivent être explorés lors des requêtes, ce qui augmente au total le temps requis.

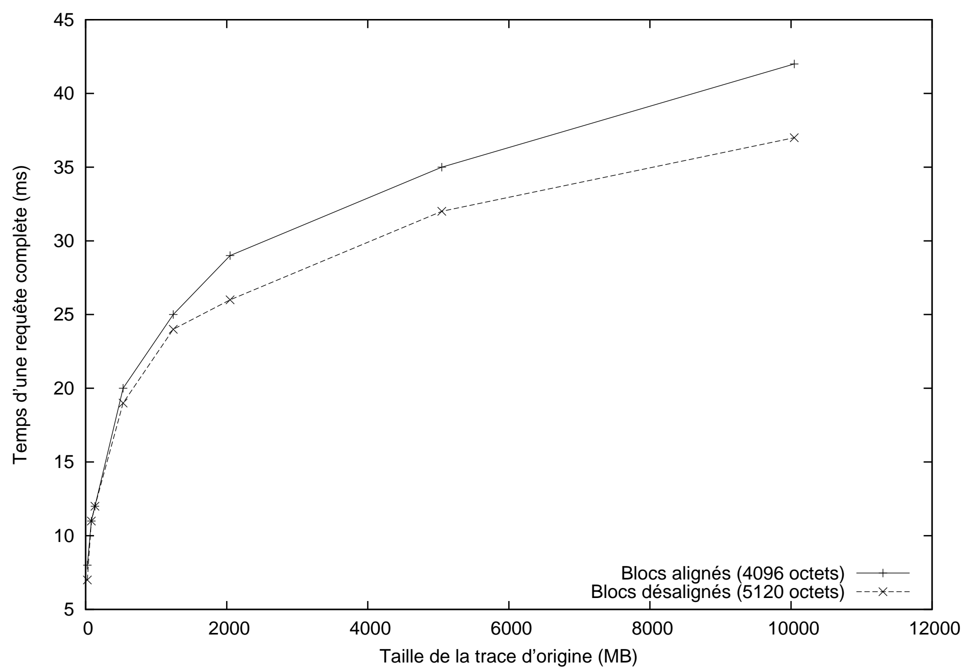

FIGURE 4.30: Blocs désalignés, requêtes complètes

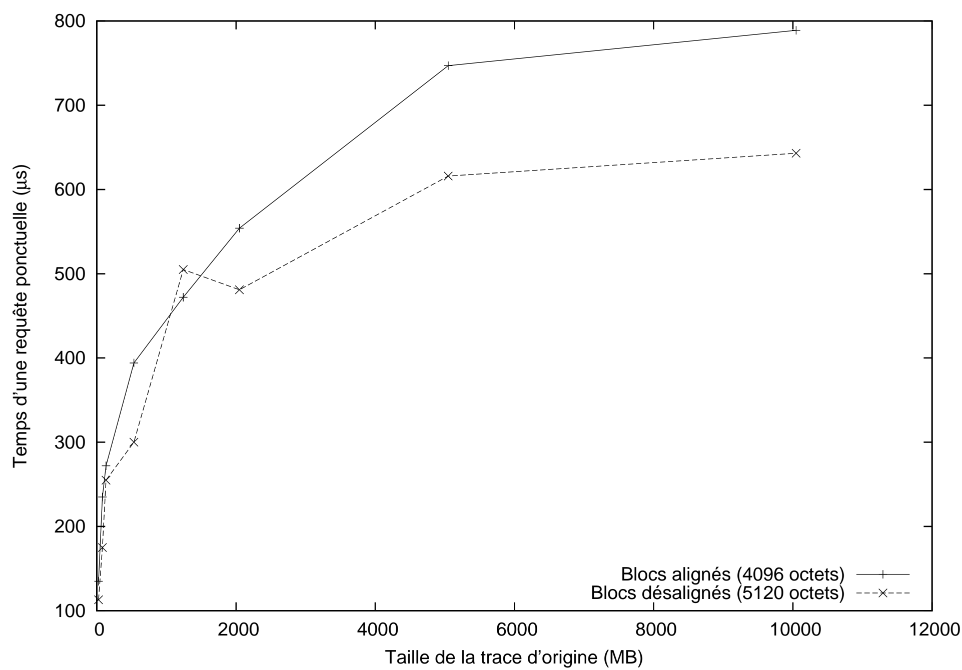

FIGURE 4.31: Blocs désalignés, requêtes ponctuelles

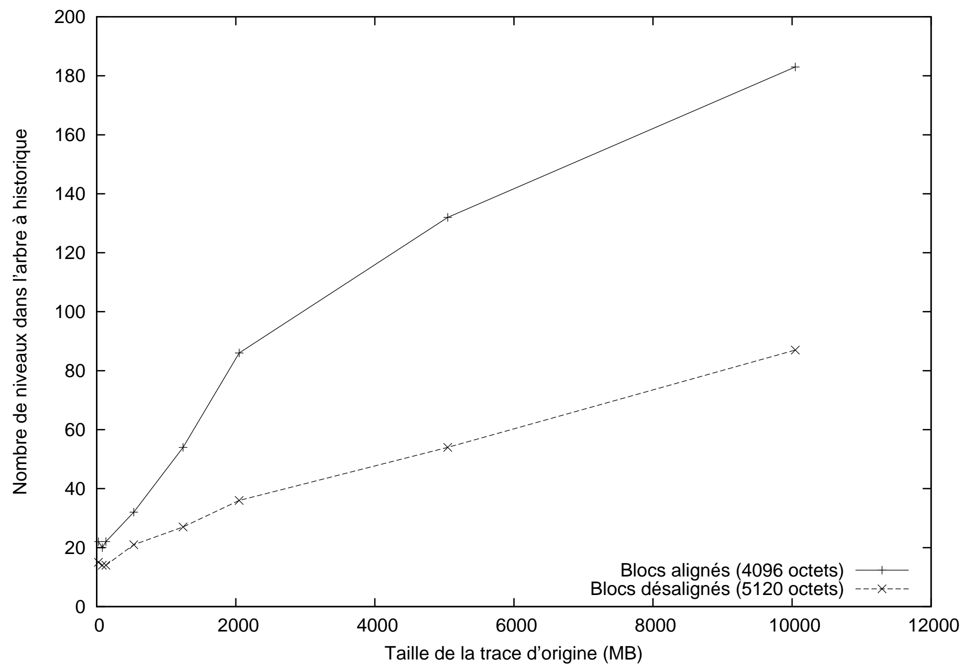

FIGURE 4.32: Blocs désalignés, profondeur de l'arbre à historique

Il semble donc que la comparaison 4096 vs. 5120 ne soit pas valide pour évaluer le bénéfice d'aligner les blocs, puisque la différence dans l'arbre résultant est trop importante. Nous ferons donc une dernière tentative, avec des blocs légèrement plus gros. Les figures 4.33 à 4.36 présentent les résultats pour des blocs de 8192 octets (alignés) vs. 9216 octets (désalignés).

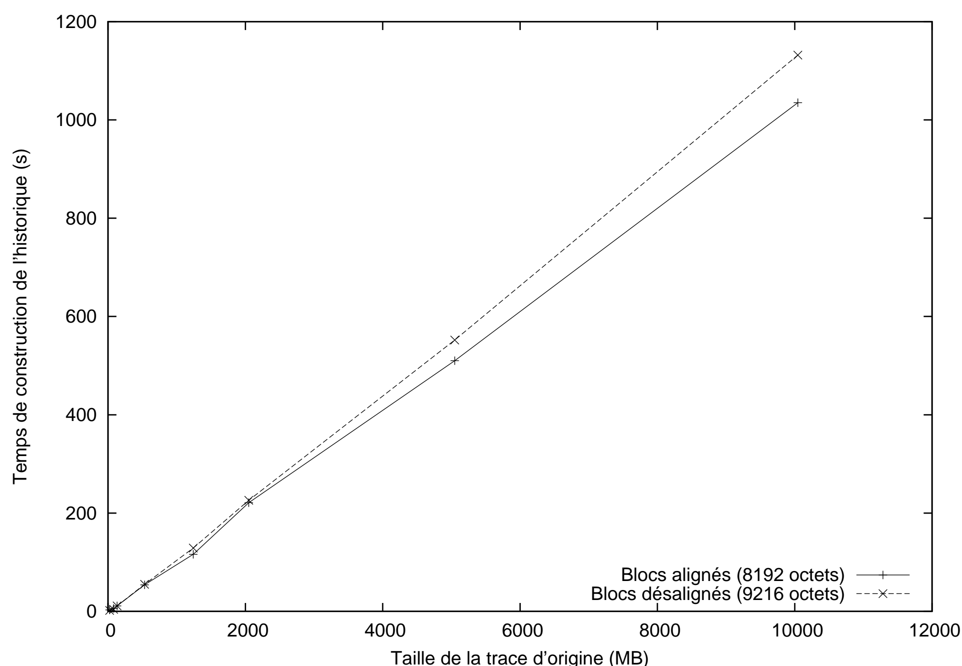

FIGURE 4.33: Blocs désalignés, construction de l'historique

On remarque encore une fois une lègère différence entre les profondeurs d'arbres, dû au fait que la différence de taille de blocs entre les deux arbres est encore assez importante proportionnellement parlant.

Par contre, la figure 4.34 montre un phénomène très intéressant : pour les traces (donc les historiques d'états) plus petites, la taille de blocs plus élevée (9 KB) présente un léger avantage. À plus grande échelle cependant, cet avantage est perdu et le fait que les blocs soient alignés devient plus important. On observe un phénomène similaire avec les requêtes ponctuelles également.

La figure 4.35 montre encore une fois des irrégularités pour la trace de 1 GB. Tel que décrit plus tôt, il s'agit probablement d'un cas où le noyau pense pouvoir faire entrer complètement le fichier d'historique en mémoire, mais finalement ne peut y arriver et utilise la *swap*. Étrangement, l'effet est plus sévère ici que pour les tailles de blocs étudiées précédemment.

Nous avons pu démontrer qu'il est en général plus prudent d'utiliser des tailles de blocs alignées sur 4 KB. Ceci devient encore plus vrai à mesure que les tailles d'historiques (donc,

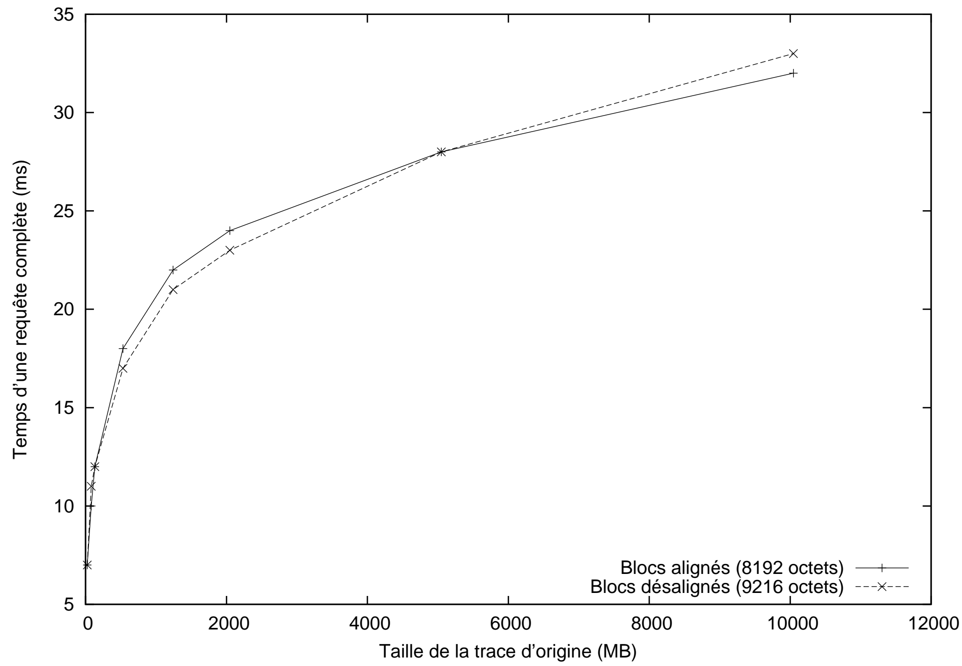

Figure 4.34: Blocs désalignés, requêtes complètes

le nombre de noeuds) augmente. Nous nous attendions à ce que l'alignement ou non des blocs influence les temps de requêtes, mais étonnamment les temps de construction bénéficient eux aussi de blocs alignés.

Nous conserverons donc la contrainte stipulant que la taille des blocs de l'arbre à historique soit un multiple de 4096 octets.

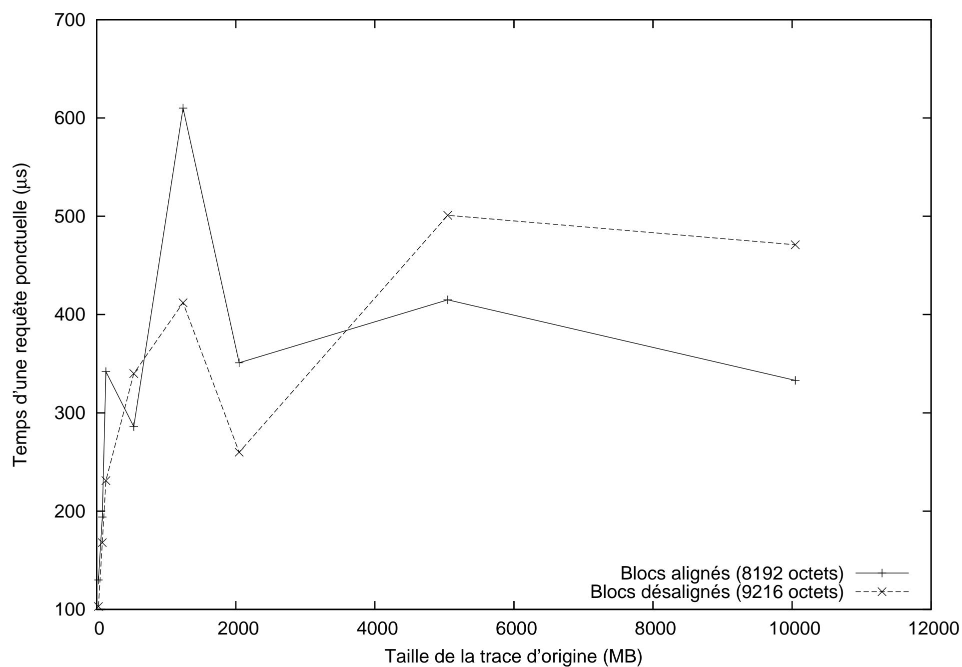

Figure 4.35: Blocs désalignés, requêtes ponctuelles

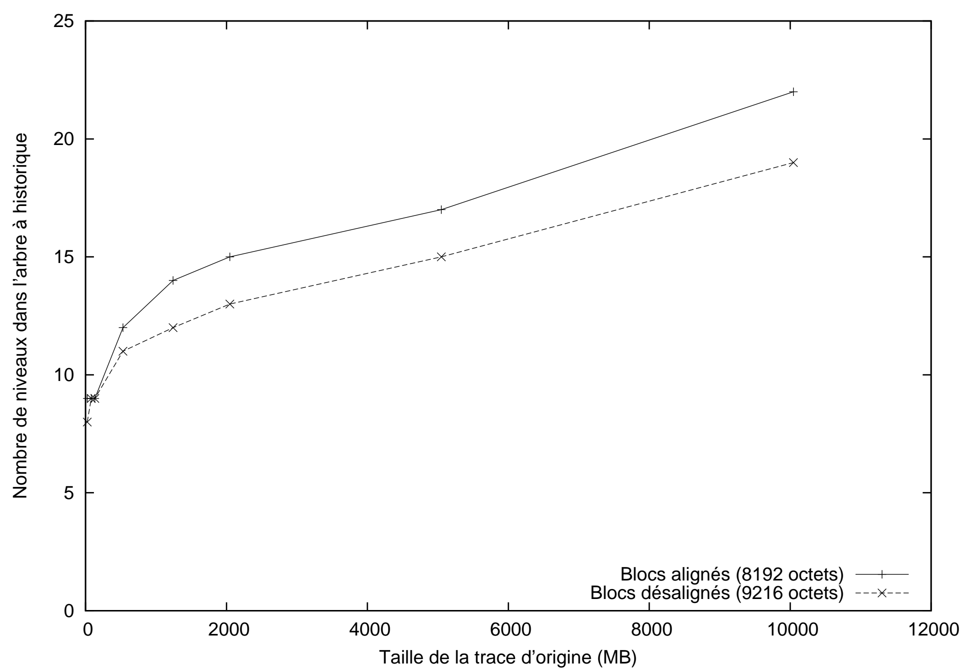

Figure 4.36: Blocs désalignés, profondeur de l'arbre à historique

4.5 Historique partiel

À la section 3.3, nous avons vu comment il est possible de conserver seulement un sous-ensemble de l'état dans l'arbre à historique et de retourner lire la trace pour régénérer les états intermédiaires.

Comme nous avons pu le constater jusqu'à présent, les historiques peuvent être très volumineux, surtout si nous y ajoutons des statistiques. Des optimisations commes celles de la section 4.4.3 peuvent aider la situation, mais il est encore très courant d'obtenir des historiques d'états plus gros que la trace qui les a généré.

L'idée proposée ici était de seulement stocker sur disque, dans l'arbre à historique, les états courants à certains points de la trace seulement (appelés points de sauvegarde), et d'utiliser la trace et le gestionnaire d'état original pour recréer l'état courant pour tout point de requête. Nous nous attendons donc à une diminution de la taille de l'historique stocké, au prix d'une augmentation des temps de requêtes.

Nous parlons dans ce cas d'un historique dit "partiel". Un historique partiel ajoute une nouvelle constante de configuration : l'espacement (en nombre d'événements) entre les points de sauvegarde. Nous appellerons ceci la *granularité* de l'historique. Une granularité plus élevée diminuera le nombre de points de sauvegarde, ce qui causera un historique plus petit mais des requêtes plus longues (puisqu'il faudra en moyenne relire plus d'événements depuis la trace).

Nous avons effectué la batterie de tests habituels, dans le but de comparer un historique complet (c'est-à-dire, où tous les intervalles d'états générés sont stockés dans l'historique, comme nous avons fait jusqu'à présent) avec des historiques partiels. Nous avons testé l'historique partiel avec des points de sauvegarde à chaque 20 000, 50 000 et 100 000 événements.

La figure 4.37 montre les temps de construction des différents historiques d'état. Nous y avons aussi rapporté le temps pris par LTTV pour générer son sytème d'état.

Nous constatons que le temps de construction diminue considérablement lorsque nous passons à un historique partiel. Ceci est dû au fait qu'il y a beaucoup moins d'information écrite sur le disque, comme nous le verrons bientôt.

Il ne semble pas y avoir beaucoup de différences si on ne fait que changer la granularité de l'historique partiel. On peut supposer que ce n'est maintenant plus l'écriture sur disque qui limite le temps de construction, mais simplement le traitement des événements.

Nous pouvons aussi constater que le temps requis est maintenant beaucoup plus près de celui pris par LTTV pour son système d'état (rappelons-nous que LTTV n'enregistre pas

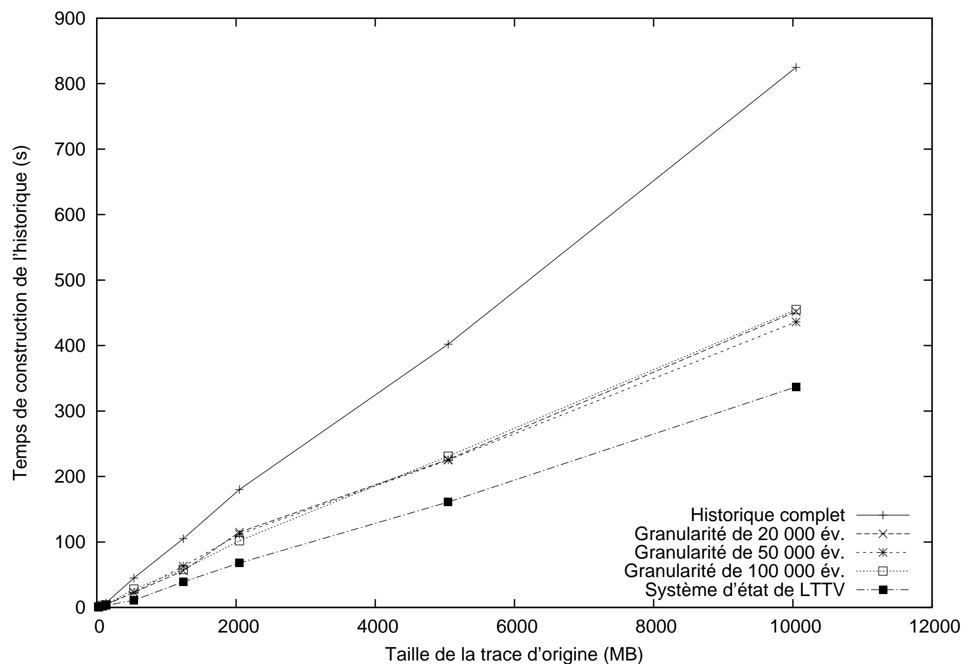

FIGURE 4.37: Historique partiel, temps de construction

d'historique sur disque).

La figure 4.38 montre la taille des fichiers d'historique résultants. Les deux graphiques sont en fait le même, seulement avec une ordonnée différente dans les deux cas. On remarque la différence phénoménale d'ordre de grandeur entre la taille d'un historique complet et partiel : l'historique partiel avec une granularité de 20 000 est pratiquement 1000 fois plus petit sur disque que l'historique complet ! Cela dépasse de loin toutes nos attentes en termes d'économies d'espace.

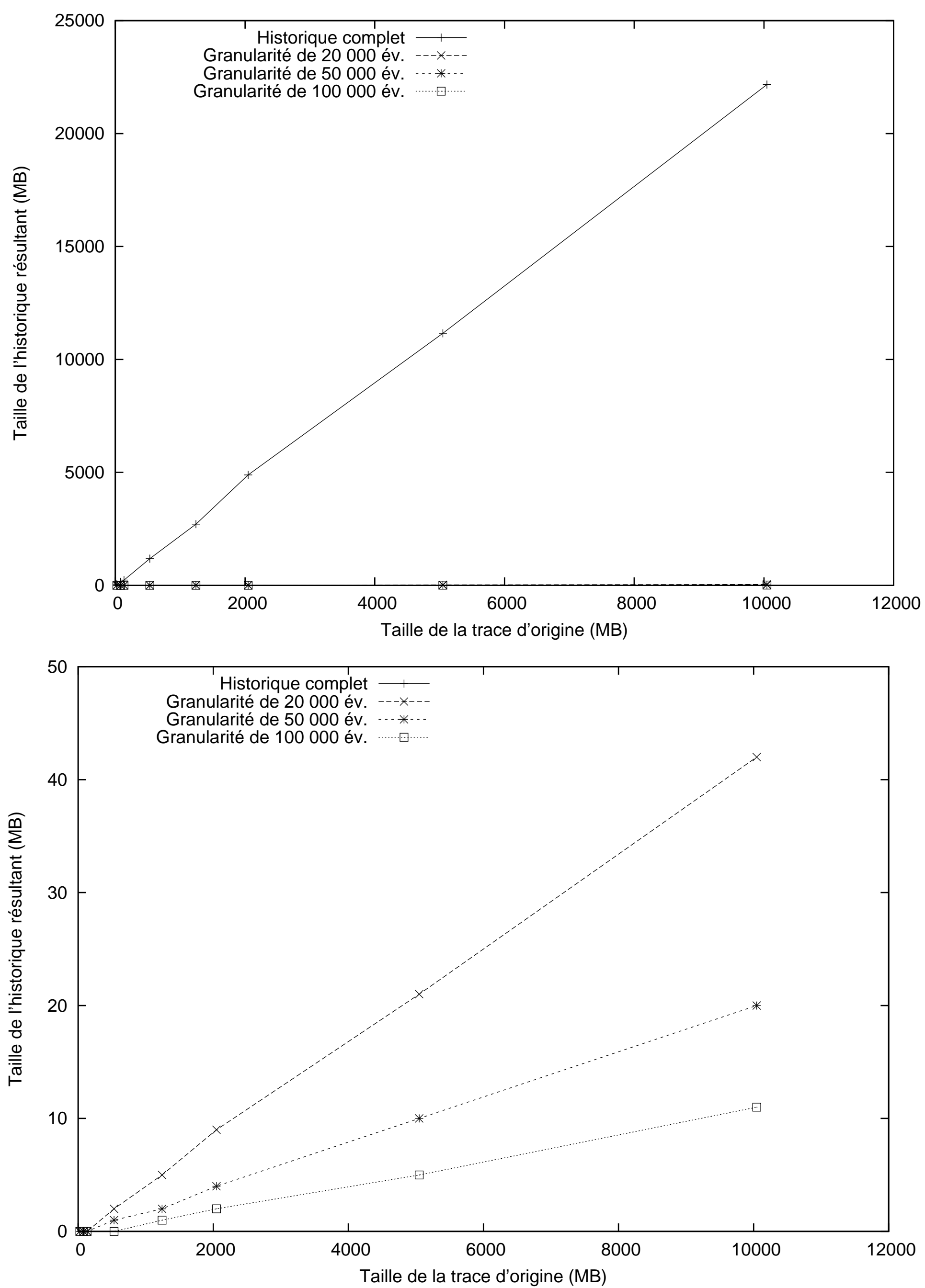

FIGURE 4.38: Historique partiel, taille du fichier d'historique

Il faut spécifier cependant que les traces utilisées ici comportaient beaucoup d'appels systèmes répétés, en plus de changements d'états à chaque événement (pour les statistiques). Ce genre d'historique d'état, où nous sommes en présence de beaucoup d'intervalles très courts dans le temps, va très bien se “compresser” avec un historique partiel.

Les résultats confirment aussi que la taille d'un historique partiel est inversément proportionnelle à sa granularité, comme on pourrait s'y attendre.

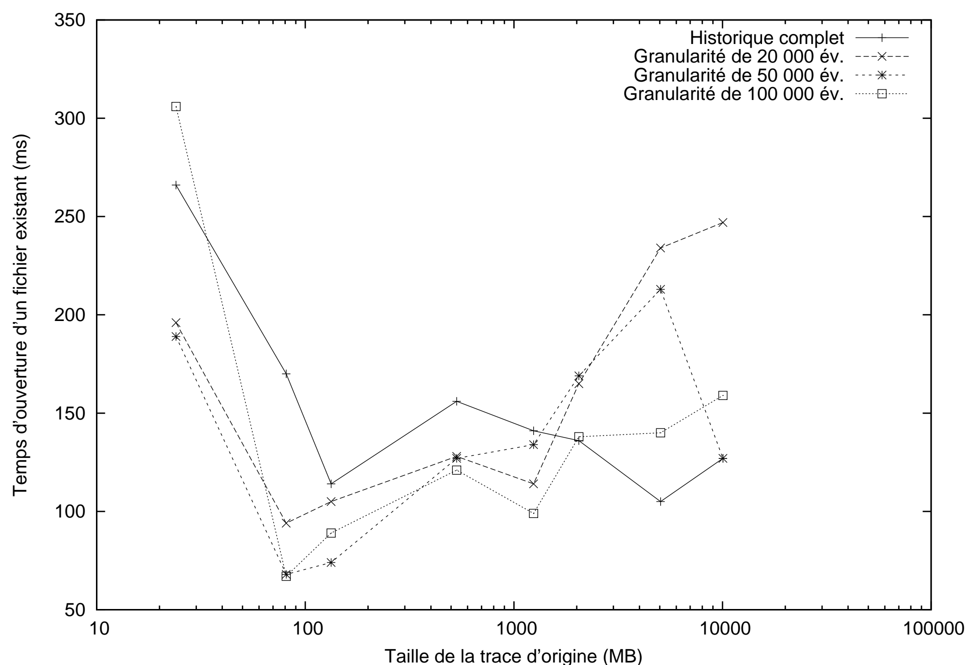

FIGURE 4.39: Historique partiel, ouverture d'un arbre déjà construit

Sur la figure 4.39, nous avons mesuré le temps d'ouverture de l'historique. Ce n'est pas une métrique qui a souvent été testée, mais nous voulions ici simplement vérifier que les étapes addtionnelles de l'ouverture d'un historique partiel (rechargement des points de sauvegarde, etc.) ne seraient pas trop perceptibles.

Le graphique est présenté avec une abcisse logarithmique, pour bien voir le comportement pour tous les types de traces. Curieusement, les historiques partiels sont plus rapides à ouvrir pour les petites traces. Ils deviennent plus lents à partir des traces d'environ 2 GB (vu le plus grand nombre de points de sauvegarde à recharger), mais deumeurent très raisonnables : dans l'ordre de quelques centaines de millisecondes.

La figure 4.40 montre les performances pour les requêtes complètes. On peut bien y voir le comportement logarithmique des temps de requêtes. L'utilisation d'historiques partiels

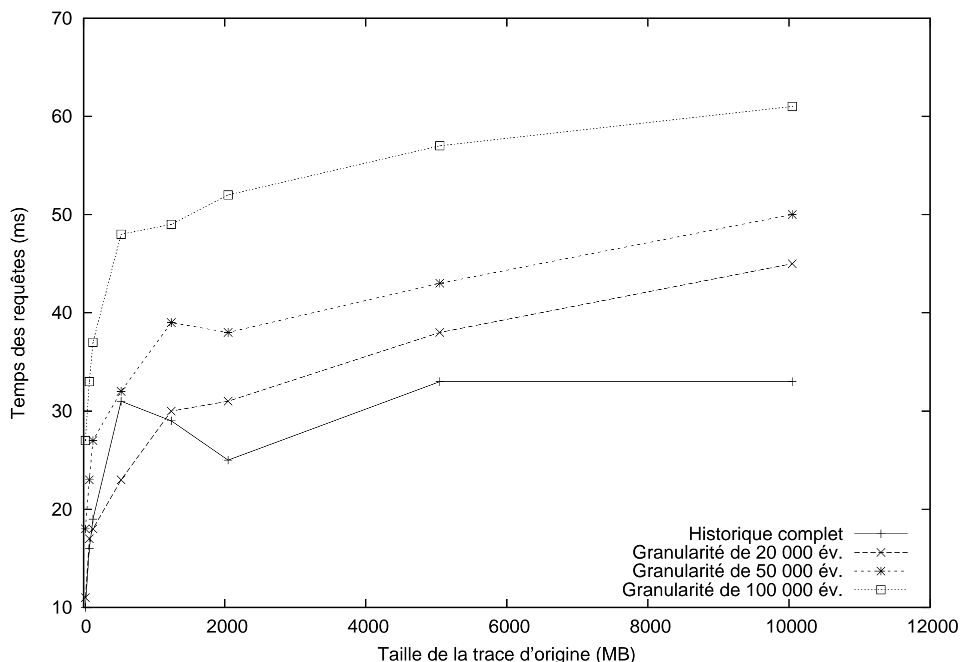

FIGURE 4.40: Historique partiel, requêtes complètes

ralentit les requêtes d'un facteur constant, et ce facteur est proportionnel à la granularité de l'historique.

Ce qui est très intéressant, c'est que cette augmentation du temps de requête est très petite, beaucoup moins que la diminuation de l'espace requis sur disque. Par contre, nous ne pouvons plus faire de requêtes ponctuelles aussi efficacement qu'avec un historique complet (nous devons faire une requête complète, puis récupérer l'information d'état pour l'attribut qui nous intéresse, ce qui est plus long).

Enfin, la figure 4.41 montre la profondeur des arbres d'historiques résultant. Les arbres partiels obtenus sont environ deux fois plus courts que l'arbre comportant un historique complet.

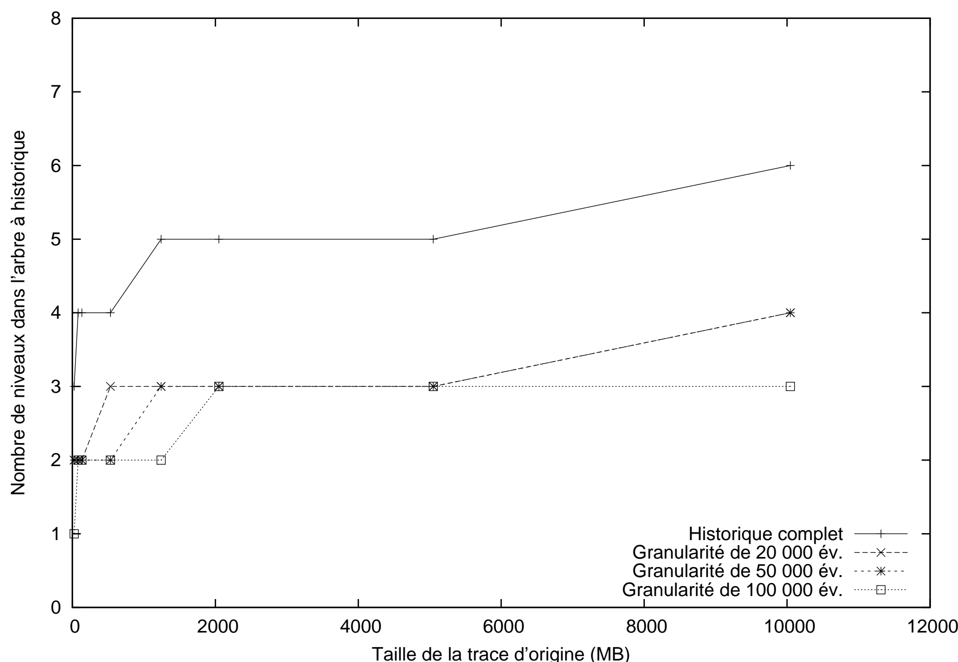

FIGURE 4.41: Historique partiel, profondeur de l'arbre à historique

En résumé, l'historique partiel permet d'énormes économies d'espace disque, au coût d'augmenter légèrement les temps de requêtes. Tel que mentionné, nous perdons la possibilité de faire des requêtes pontuelles efficaces, ainsi que de se promener "latéralement" dans l'historique de l'état.

Il serait théoriquement possible de définir un gestionnaire d'état de telle sorte que nous enregistrerions différents types d'information dans différents types d'historiques. Par exemple, il serait logique de stocker les statistiques dans un historique partiel, mais conserver les attributs "importants" comme les états des processus dans un historique complet pour permettre des analyses plus avancées. Nous reviendrons sur cette possibilité à la conclusion.

4.6 Comparaisons avec d'autres méthodes de stockage

Jusqu'à maintenant, nous avons évalué les performances du système d'état utilisant l'arbre à historique pour stocker les intervalles d'état. Il serait toutefois possible d'utiliser le même système d'état (avec l'arbre d'attributs, l'état transitoire, etc.) mais avec une méthode de stockage sur disque différente.

Nous avons vu au chapitre 2 qu'il est difficile de stocker des intervalles sur disque tout en permettant qu'ils soient accessibles efficacement. Nous tenterons tout de même d'utiliser des méthodes de stockage génériques, déjà éprouvées, et de les comparer à l'arbre à historique

dans le cadre du stockage d'un historique d'état. Ceci permettra de voir si l'utilisation d'une nouvelle structure conçue expressément pour nos besoins est justifiée.

4.6.1 *R-Tree* en mémoire

Nous avons vu au chapitre 2 qu'il est possible de stocker des intervalles dans les R-Trees. Il semble cependant que les R-Trees ne sont pas des structures très efficaces pour stocker de l'information en seulement une dimension (telle de l'information temporelle, comme dans notre cas). Nous voudrions confirmer cette assertion, du moins pour notre cas d'utilisation particulier.

Une nouvelle extension de stockage a été implémentée pour le stockage des intervalles du système d'état. Celle-ci utilise la librairie JSI, qui implémente des R-Trees en Java. Cette librairie ne fait que la représentation en mémoire de la structure. Nous serons donc limités en termes de taille d'historique. Il demeure néanmoins intéressant de faire la comparaison pour les traces de petites tailles.

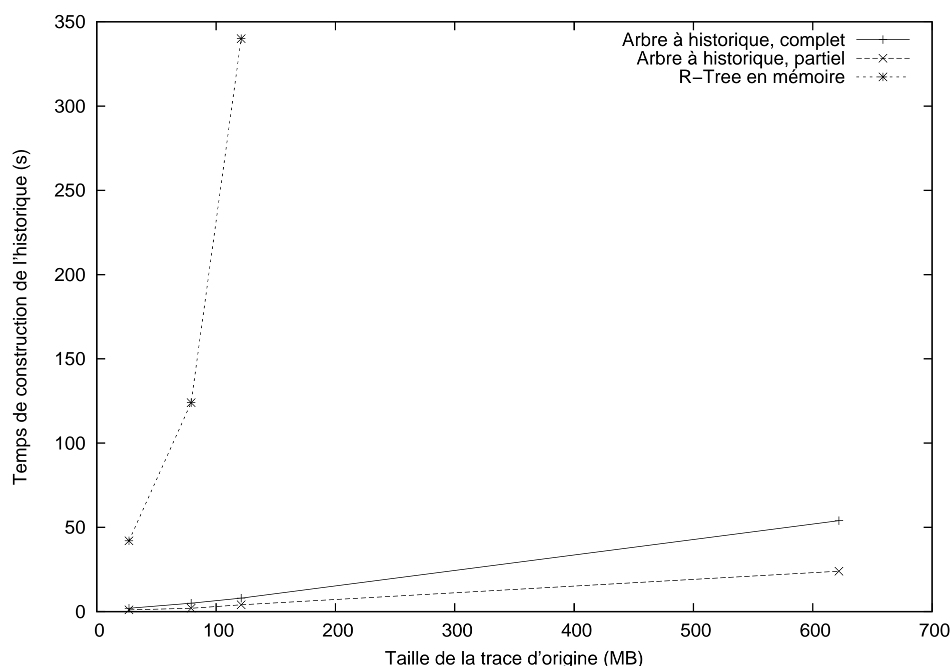

FIGURE 4.42: Historique dans un R-Tree en mémoire vive, construction de l'historique

La figure 4.42 montre la comparaison pour les temps de construction des historiques. Le temps pris par les arbres à historiques complets et partiels a été rapporté. Comme on peut

le constater, le remplissage du R-Tree en mémoire est énormément plus long que celui de l'arbre à historique.

Le profilage de l'exécution révèle que, comme on pourrait s'y attendre, c'est bien le rebalancement de l'arbre qui est le plus coûteux en termes de temps. La construction de l'arbre à historique est basée sur le fait que les intervalles arrivent triés en termes de temps. Cependant, cette particularité rend les insertions beaucoup plus longues pour des structures plus génériques et "agiles", qui doivent constamment être rebalancées.

Un autre point à noter est l'utilisation extrêmement élevée de mémoire RAM par la librairie. Pour terminer le test avec la trace de 100 MB, il a fallu modifier les paramètres d'exécution par défaut pour permettre d'allouer plus de mémoire au processus Java. Après le remplissage du R-Tree avec les intervalles provenant de cette trace, l'utilisation de mémoire du processus environnait les 4 GB ! Il a également été impossible de construire l'historique de la trace de 500 MB et des suivantes, à cause d'erreurs du type `OutOfMemory`.

Le but ici n'était pas de comprendre le fonctionnement interne de la librairie JSI, mais seulement d'expérimenter sur ce qu'il était possible de faire. Nous pouvons cependant supposer la création d'indexes de recherche très complets, ce qui expliquerait l'utilisation de RAM importante mais aussi les temps de requêtes extrêmement rapides.

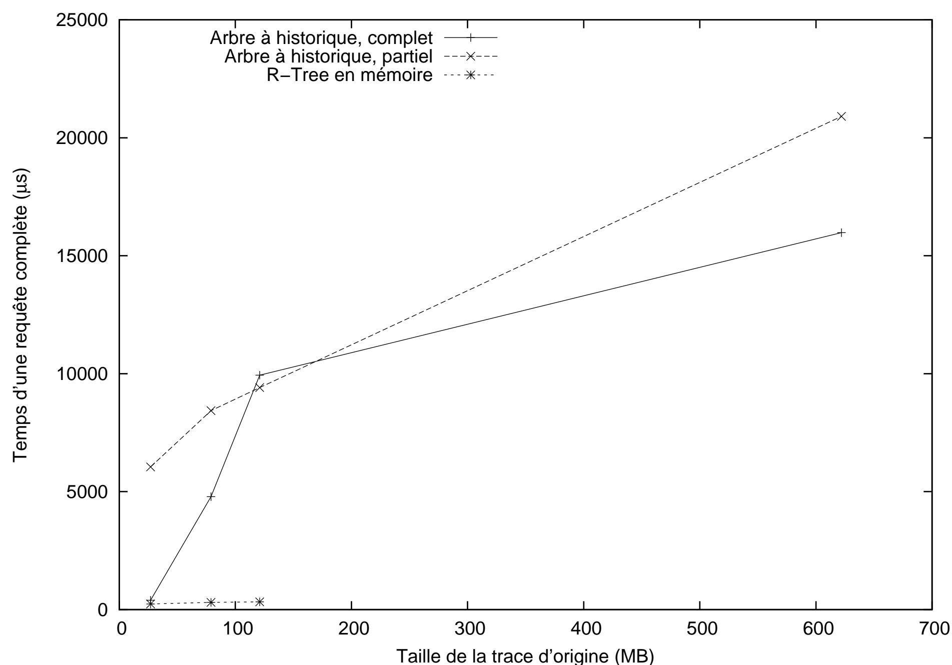

FIGURE 4.43: Historique dans un R-Tree en mémoire vive, requêtes complètes

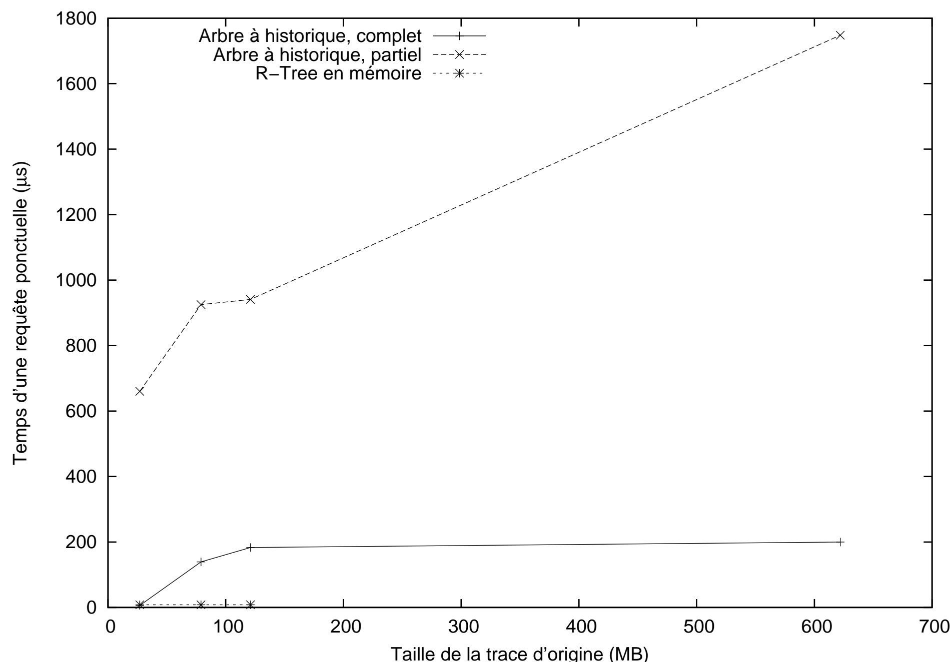

FIGURE 4.44: Historique dans un R-Tree en mémoire vive, requêtes ponctuelles

Les figures 4.43 et 4.44 montrent les résultats pour les temps de requêtes. Les requêtes aux R-Tree sont très rapides, beaucoup plus que celles des arbres à historique. On parle d'environ 300 μs pour les requêtes complètes et 8 μs pour les requêtes ponctuelles.

Cette dernière valeur se situe presque dans le même ordre de grandeur que des accès mémoire directs (comme l'accès à une valeur d'un tableau), qui ont été mesurés à environ 300 ns sur le système utilisé. On peut supposer que des index et des optimisations du compilateur Java sont à l'oeuvre.

En résumé, l'utilisation du R-Tree de JSI comme stockage des intervalles d'état permet des temps de requêtes très rapides et tout à fait inégalés. Malheureusement, ceux-ci sont possibles seulement aux dépens de temps de construction très longs, et d'une utilisation de mémoire très importante. Cette dernière limite aussi sévèrement la taille des historiques qu'il est possible de stocker.

4.6.2 Base de données spatiale (PostgreSQL avec PostGIS)

La prochaine comparaison avec une méthode de stockage générique et éprouvée consiste à utiliser une base de données pour le stockage des intervalles d'états. Les bases de données “spatiales” permettent de stocker de l'information mutli-dimensionnelle. Encore une fois, elles

se prêtent apparamment mieux à l'information à deux ou plusieurs dimensions, il est toutefois techniquement possible de les utiliser à une seule dimension. La comparaison pourrait être intéressante.

De plus, une base de données stocke ses fichiers sur disque. Nous pourrons peut-être utiliser des historiques plus volumineux que ceux qui étaient limités par l'implémentation en mémoire.

Le choix s'est arrêté sur PostGIS, une extension à PostgreSQL, qui permet d'utiliser des colonnes d'informations spatiales dans la base de données. L'existence d'un connecteur JDBC a rendu la tâche d'intégration plus facile.

Les paramètres de configuration ont été laissés à ceux par défaut autant que possible. Un changement qui a été effectué par contre est de désactiver l'*auto-commit* (qui crée une transaction pour chaque opération d'insertion), et d'appeler
`Connection.commit()` une seule fois, à la fin. Ce seul changement permettait d'insérer tous les intervalles environ 100 fois plus rapidement.

Un index de recherche est ensuite créé sur la colonne contenant les géométries (ici les bornes des intervalles de temps). Nous comptons le temps de création de cet index dans le temps de construction de l'historique.

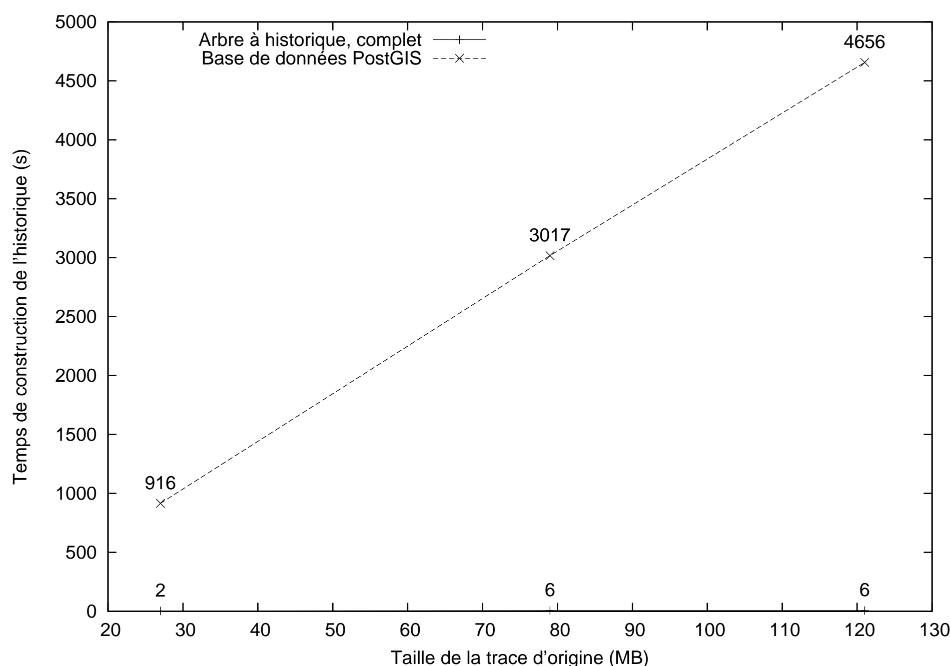

FIGURE 4.45: Historique dans PostgreSQL/PostGIS, construction de l'historique

Malgré cette modification, les insertions étaient quand même très lentes. Sur la figure 4.45,

nous rapportons les temps de construction. La courbe de l'arbre à historique est pratiquement superposée à l'abcisse. La base de données semble elle aussi être victime d'un rebalancement constant, dû au fait que les intervalles arrivent triés par temps de fin.

Il peut être intéressant de faire la comparaison en termes d'intervalles insérés, et non de taille de trace d'origine. Les traces présentées sur cette figure et les suivantes (27, 79 et 121 MB) généraient ici 3,15, 9,15 et 13,97 millions d'intervalles d'état à être insérés dans la base donnée ou l'arbre à historique.

Nous aurions pu construire des historiques pour les traces plus grandes, puisque l'information est stockée sur disque et l'utilisation reste constante. Cependant les temps de test commençaient à être très longs, particulièrement pour les tests de requêtes.

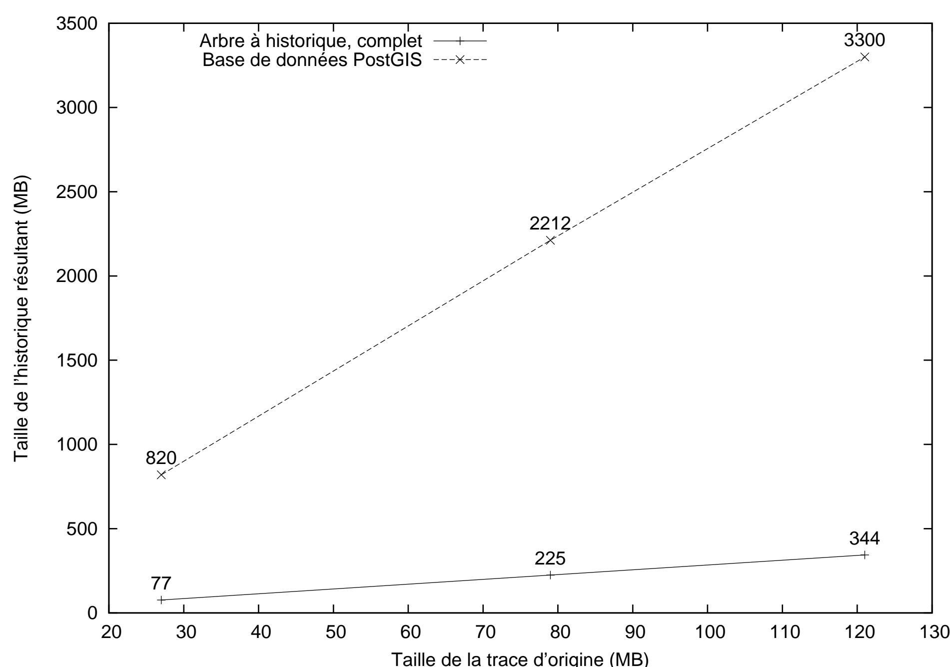

FIGURE 4.46: Historique dans PostgreSQL/PostGIS, taille de l'historique sur disque

La figure 4.46 présente l'espace disque occupé par les historiques résultants. Pour l'arbre à historique la valeur nous est rapportée par la méthode Java `File.length()`, tel que mentionné plus tôt. Pour obtenir la taille de la base de données, nous avons utilisé la commande :

```
du -s /var/lib/postgresql
```

Ce répertoire contient également la configuration initiale de la base de données (c'est pourquoi l'ordonnée à l'origine est au-dessus de 0).

On constate que l'espace occupé par la base de données est beaucoup plus grand que celui de l'arbre à historique, environ d'un facteur dix.

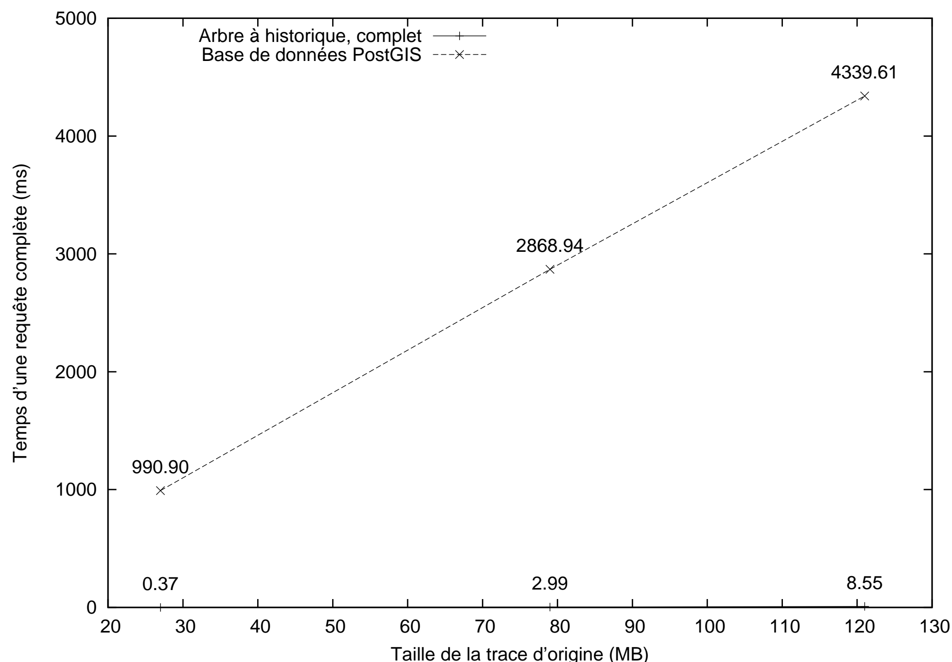

FIGURE 4.47: Historique dans PostgreSQL/PostGIS, requêtes complètes

Enfin, les figures 4.47 et 4.48 présentent les résultats des temps de requêtes. Les requêtes sont elles aussi beaucoup plus lentes pour la base de données que pour l'arbre à historique. Ce qui est intéressant à constater aussi, c'est que PostgreSQL ne semble pas utiliser la localisation spatiale des intervalles pour accélérer le temps de recherche, mais semble simplement itérer sur la totalité d'entre eux lors des requêtes. En effet, la tendance des temps de requêtes est exactement linéaire.

Les temps de requêtes étaient pratiquement les mêmes lorsque nous sautions l'étape de création de l'index. Après courte investigation, il semble qu'il n'est pas possible de forcer PostgreSQL à utiliser un index en particulier : le moteur de la base de données décide lui-même si l'utilisation d'un index en vaut la peine ou non. Nous pouvons supposer que l'index de recherche qui est créé n'est pas utile pour de l'information à seulement une dimension.

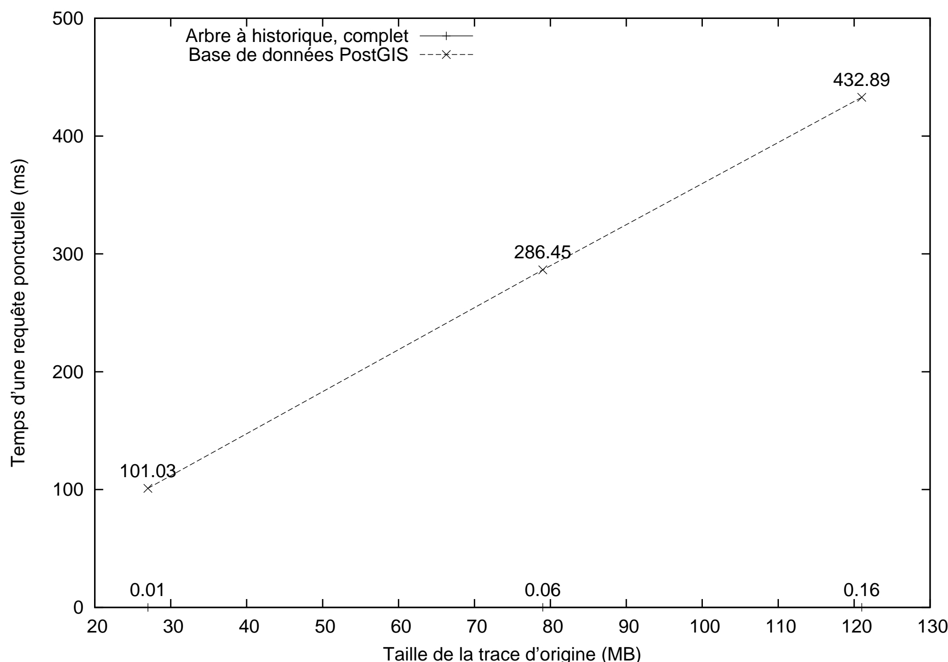

FIGURE 4.48: Historique dans PostgreSQL/PostGIS, requêtes ponctuelles

En résumé, la méthode de base de données ne sera pas recommendée, puisque en plus de dépendre d'un serveur de base de données externe, les constructions sont plus longues et les requêtes ne semblent pas pouvoir atteindre le comportement $\mathcal{O}(\log n)$ de l'arbre à historique.

4.7 Tests à grande échelle avec le format CTF

Le test final à être effectué était de tester les limites de l'arbre à historique, avec des traces aussi grandes qu'il nous était possible de manipuler. Après tout, nous voulons proposer une solution qui ne met aucune limite théorique sur la taille des traces et historiques d'états, autre que l'espace disque disponible.

En parallèle, un lecteur de traces Java pour le nouveau format CTF avait été complété. Voulant également supporter ce format dans le système d'état, un nouveau gestionnaire d'événement a été écrit, ce qui permettait de générer des historiques à partir de traces CTF.

De nouvelles traces CTF ont donc été générées, avec des tailles beaucoup plus importantes cette fois-ci. La même méthode de génération a été utilisée (script vérifiant régulièrement si la taille voulue était atteinte, et processus `ping` roulant en arrière-plan pour générer des événements). À titre de référence, la trace de 550 GB a pris environ 4 jours à être générée. Le tableau 4.3 présente les traces CTF obtenues.

TABLEAU 4.3: Traces CTF

Taille cible (GB)	Taille réelle (MB)
0	54
1	1006
5	5051
10	11350
25	28505
80	79799
550	548540

Une autre différence avec les traces LTTng qui avaient été utilisées jusqu'à présent est qu'avec les traces CTF nous n'avions pas de garantie qu'il n'y avait pas d'événements perdus (le support pour détecter les événements perdus dans la librairie de lecture n'est arrivé que plus tard). Cela n'est pas optimal, et nous verrons plus loin des effets secondaires de cette situation.

Nous avons ici seulement utilisé des historiques complets. Une des raisons était que le support pour les historiques partiels n'existait que pour les traces VSTF (il devra être réécrit ou généralisé pour permettre d'utiliser les traces CTF). Additionnellement, nous voulions obtenir les arbres à historiques les plus volumineux possibles, il est logique d'utiliser des historiques complets.

Nous avons d'abord effectué les tests avec une taille de bloc laissée par défaut, ce qui correspond à des blocs de 32 KB. À la lumière des résultats obtenus nous avons ensuite répété les tests avec des tailles de blocs plus élevées, de 1 MB et 4 MB. Les résultats de ces trois essais sont tous rapportés sur les graphiques.

La figure 4.49 présente les temps de construction des différents historiques d'état. Nous sommes rassurés de voir que le comportement linéaire du temps de construction est maintenu même pour les très grands historiques. Ceci était attendu, puisque l'utilisation de mémoire est relativement constante.

Le temps est rapporté en minutes, donc il a pris environ 7 heures pour construire l'historique de la trace de 550 GB. Cela peut sembler long, mais rappelons-nous que cette trace à mis près de 4 jours à être générée. La construction de l'historique est au moins bien plus rapide qu'en temps réel !

Nous remarquons également que les tailles de noeuds plus élevées entraînent une légère augmentation du temps de construction, particulièrement pour les traces et historiques très

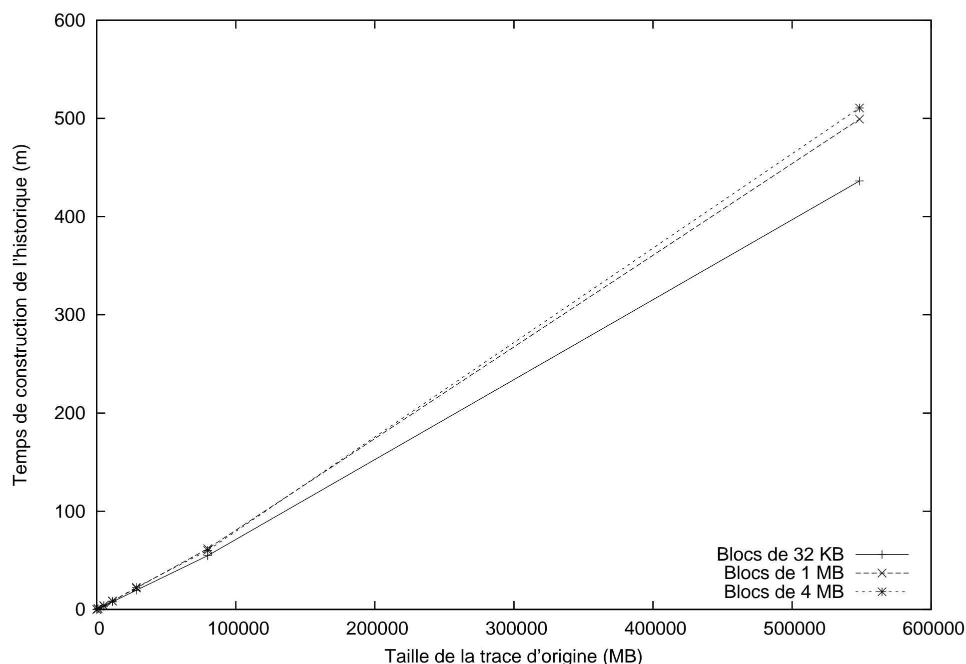

FIGURE 4.49: Traces CTF, temps de construction

grands. Ce phénomène est sans doute causé par une faiblesse de l'architecture : c'est le même fil d'exécution qui place les intervalles dans les noeuds et qui écrit ces noeuds sur disque. Des noeuds plus gros impliquent un temps d'écriture plus long, ce qui arrête le traitement pendant des périodes de temps plus longues. Ceci augmente donc les chances que les files d'attente se remplissent, ce qui stoppe toutes les autres parties en amont du système.

De plus, le fait de demander des écritures plus volumineuses à chaque fois augmente le risque de remplir la cache d'écriture du disque. Il serait donc souhaitable que l'opération d'écriture ne bloque pas le reste du système d'état.

Il serait possible de remédier à ce problème en séparant le placement des intervalles de l'écriture comme telle, soit en déplaçant le travail d'un fil à un autre, soit en en ajoutant un nouveau. Les noeuds pleins seraient donc envoyés dans une nouvelle file d'attente, et le fil d'exécution au bout n'aurait comme seule tâche d'écrire ceux-ci sur le disque. Ceci donnerait également un taux d'écriture plus constant.

Ce problème n'est apparent qu'avec des tailles d'historique très élevées, donc cette modification n'est peut-être pas la plus prioritaire.

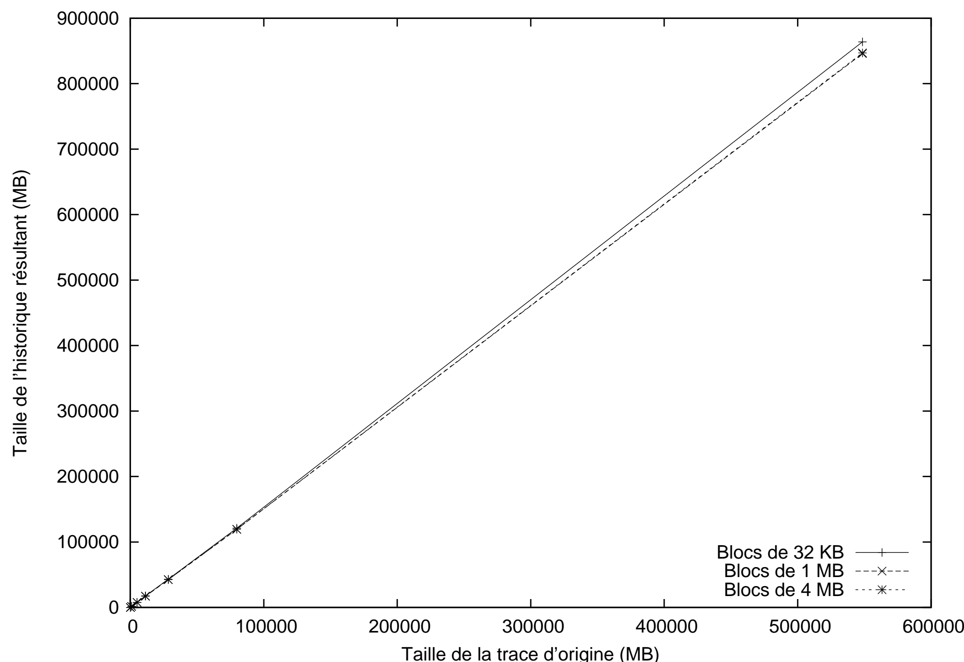

FIGURE 4.50: Traces CTF, taille du fichier d'historique

La figure 4.50 montre les tailles sur disque des historiques obtenus pour les trois tailles de blocs. Encore une fois, la taille des blocs influence peu la taille de l'historique final. On remarque cependant une légère augmentation pour les tailles de blocs plus *petites*, particulièrement pour la trace de 500 GB. Ceci est logique, puisque des blocs plus petits seront plus nombreux, ce qui augmente l'espace total utilisé par les en-têtes des blocs.

Le gesionnaire d'événements utilisé produisait ici des historiques presque deux fois plus gros que la trace d'origine. Pour un cas comme celui-ci en pratique, l'utilisation d'un historique partiel serait recommendée.

Les figures 4.51 et 4.52 montrent les temps moyens pour les deux types de requêtes. Ils sont tous deux présentés avec des axes X à échelle logarithmique. Une droite sur ces graphiques correspond donc à un comportement logarithmique, ce que nous attendons pour les temps de requêtes en général.

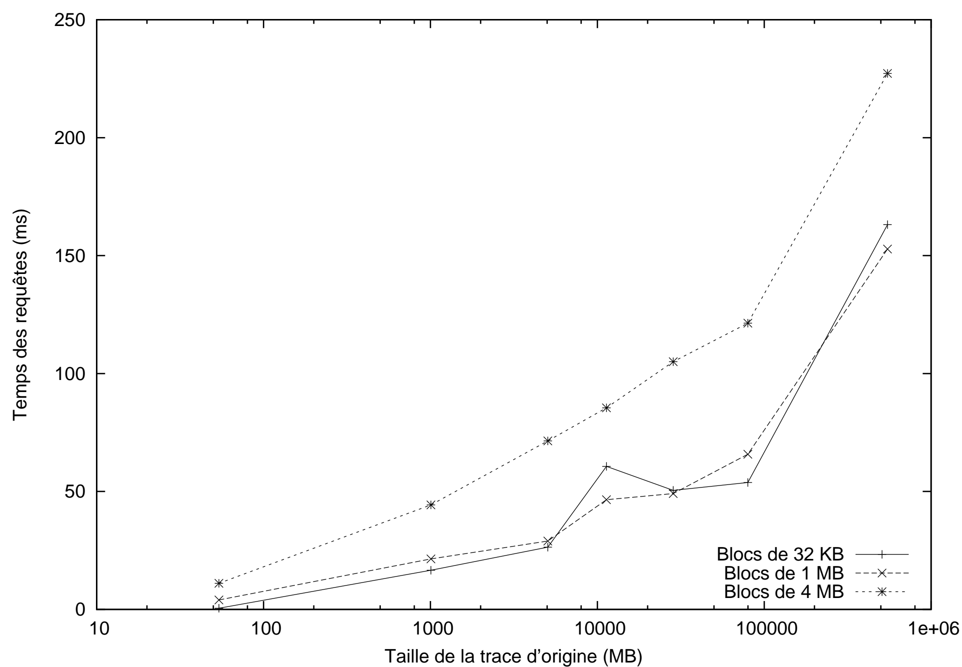

Figure 4.51: Traces CTF, requêtes complètes

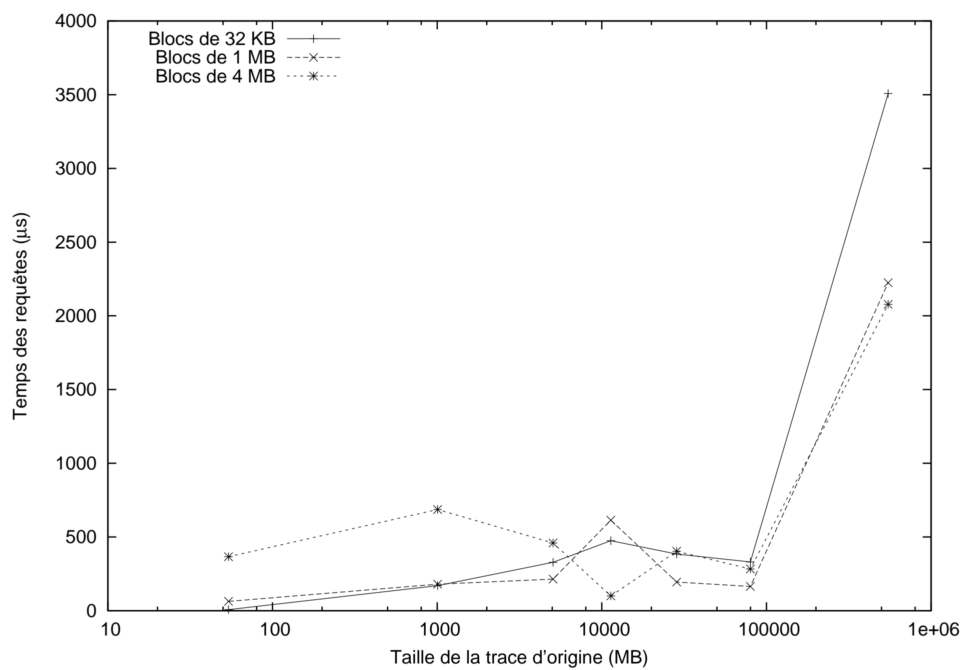

Figure 4.52: Traces CTF, requêtes ponctuelles

Il semble que pour les traces plus petites, les historiques composés de blocs de 32 KB offrent les meilleures performances. Lorsque la trace dépasse 10 GB cependant, les blocs de 1 MB prennent légèrement le dessus. La taille de blocs de 4 MB ne semble pas ici être une bonne alternative, surtout pour les requêtes complètes.

On remarque également que le comportement logarithmique des temps de requêtes est maintenu jusqu'à la trace de 80 GB. La trace de 550 GB montre par contre une augmentation des temps de requêtes au-delà de l'évolution escomptée.

Puisque les tests étaient réalisé en prenant 200 points au hasard dans les traces (entre chaque vidange des caches), nous pouvons supposer que plus un historique est grand, moins il y a de chances que des requêtes consécutives soient la même section de l'arbre. Des blocs différents doivent être chargés à chaque fois, ce qui limite l'utilité de la mémoire cache.

Néanmoins, les temps de requêtes demeurent très raisonnables sur une échelle absolue : environ 2 ms pour une requête ponctuelle et 150 ms pour une requête complète.

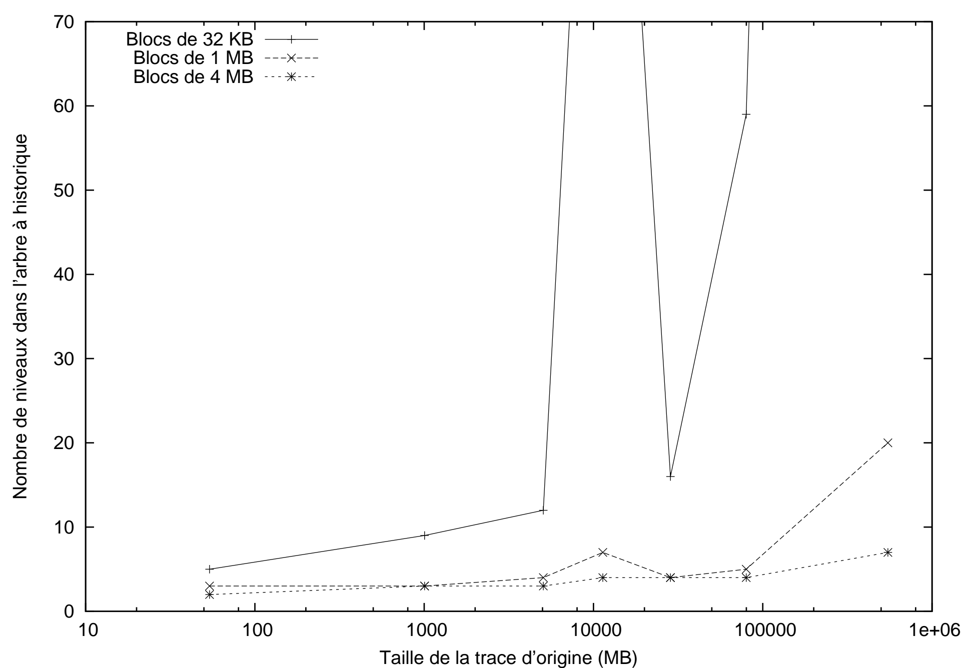

FIGURE 4.53: Traces CTF, profondeur de l'arbre à historique

Finalement, la figure 4.53 présente le nombre de niveaux dans chacun des arbres à historiques résultants. Les deux points non montrés sur la courbe de la taille de blocs de 32 KB sont à 144 et 614, pour les traces de 10 GB et 550 GB respectivement.

On remarque tout de suite cette étrange augmentation pour la trace de 10 GB (on pouvait aussi voir ses effets sur les graphiques des requêtes). Après analyse, il semble que la trace

de 10 GB comportait relativement plus d'événements perdus par rapport aux autres. Étant donné la quantité énorme d'information, il est difficile de connaître exactement l'agencement des intervalles d'état.

Il est très probable par contre que le plus grand nombre d'événements perdus génère plus d'états qui restent "ouverts" (par exemple, une entrée en appel système dont on ne voit jamais la sortie). Ces états ouverts seront éventuellement tous fermés à la fin de l'étape de construction. Mais comme ils auront habituellement été ouvert depuis longtemps, le seul noeud qui pourra les contenir sera souvent le noeud racine. Or, chaque remplissage du noeud racine amène automatiquement la création d'un nouveau niveau dans l'arbre.

À part ce comportement, tout semble normal. En utilisant une taille de noeuds plus élevée, le phénomène disparait complètement. Ce n'est pourtant pas une situation optimale. Bien entendu, la qualité de l'historique est dépendante de la qualité des traces qui sont utilisées comme entrée. Il serait toutefois souhaitable de pouvoir gérer ces problèmes d'une manière un peu plus robuste qu'une simple dégradation en une liste chaînée (menant à une augmentation du temps requis pour toutes les requêtes, ainsi qu'à un gaspillage d'espace pour des en-têtes de noeuds qui ne contiendront rien).

Nous parlerons à la conclusion d'un léger changment à l'organisation des noeuds qui devrait permettre non seulement d'améliorer cette situation, mais aussi de la détecter plus efficacement.

Chapitre 5

CONCLUSION

Nous ferons maintenant la conclusion des travaux présentés dans le cadre de ce mémoire. Nous ferons d'abord un bref résumé de la solution, puis nous discuterons des ses limitations. Enfin, nous explorerons quelques améliorations qui pourraient potentiellement y être ajoutées.

5.1 Synthèse des travaux

Nous avons tout d'abord identifié des lacunes dans la méthode actuelle de stockage de l'état de visualiseurs de traces. Pour pallier à ces lacunes, nous avons proposé une solution en deux volets.

D'abord, nous avons présenté l'arbre à historique, qui est une structure de données optimisée pour le stockage d'intervalles sur disque. Celui-ci sert de conteneur principal pour l'information d'état. Cet arbre regroupe les intervalles en noeuds disjoints, de telle sorte qu'une seule branche de l'arbre doit être explorée lors d'une requête visant à regénérer l'état complet d'un système à n'importe quel point de l'historique. Ceci permet le stockage d'historiques très volumineux, tout en maintenant une efficacité $\mathcal{O}(\log n)$ pour les requêtes.

Ensuite, nous avons proposé une modification au fonctionnement du sytème d'état en mémoire pour rendre la définition des modèles d'état plus générique. Pour ce faire, nous avançons l'utilisation d'un arbre d'attributs. Cette organisation des attributs, similaire à un système de fichiers, permet de passer efficacement entre leur représentation litérale et la valeur entière (ou quark) qui leur est associé.

Nous avons ensuite défini le fonctionnement de l'état transitoire, qui permet non seulement de conserver une image de l'état courant à mesure que nous construisons l'historique, mais qui permet aussi de convertir des changements d'états ponctuels en intervalles représentant les différents états à être insérés dans l'historique.

Enfin, nous avons suggéré une méthode de base pour construire des *gestionnaires d'événements*, qui permettent de construire un modèle de l'état en convertissant des événements

venant de traces en des changements d'états. C'est ce gestionnaire qui fait le lien entre les librairies de lecture de traces déjà existantes et le nouveau système d'état proposé.

Une variante au système de base, appelée *historique partiel*, a également été décrite. Cette variante fait quelques sacrifices au niveau de la flexibilité de la structure (il faut conserver la trace d'origine, il n'est plus possible de faire des requêtes ponctuelles ou d'explorer l'historique "latéralement") mais permet des économies d'espace disque considérables.

Nous avons également fait une implémentation de référence des différentes composantes, ainsi que d'un gestionnaire d'événements permettant de traiter des traces noyau. Ceci a permis d'avoir un système d'état fonctionnel, qui a pu ensuite être testé et dont la performance a pu être mesurée.

Ces tests ont permis en premier lieu de faire des choix quant aux options qui s'offraient dans la conception du système. Nous avons ensuite pu comparer les performances du nouveau système par rapport au système existant. Nous avons également comparé la méthode de stockage proposée avec des méthodes génériques déjà existantes, de manière à montrer que l'utilisation d'une structure sépcialement conçue pour notre cas d'utilisation était justifiée.

5.2 Limitations de la solution proposée

Bien que l'arbre à historique ait été présenté comme une structure pour stocker des intervalles sur disque de manière efficace, celui-ci requiert certaines conditions d'utilisation. Tout d'abord, les intervalles peuvent seulement être ajoutés. L'arbre ne supporte pas les effacements et les rebalancements. De plus, les intervalles doivent être insérés autant que possible par ordre croissant de temps de fin, pour garantir une organisation optimale des noeuds de l'arbre.

Une autre limitation est que les intervalles enregistrés doivent être relativement courts par rapport à la durée totale de l'historique (ou dans notre cas, de la trace). Pour des intervalles d'états provenant du système que nous avons proposé, cette condition est habituellement respectée.

Il serait possible d'utiliser l'arbre à historique comme structure d'intervalles dans un cadre autre que le stockage d'un historique d'état, tant que ces conditions demeurent respectées.

Pour ce qui est du système d'état général, celui-ci requiert l'implémentation d'un gestionnaire d'événements pour chaque type de trace que nous souhaitons analyser. Bien que des méthodes standard aient été proposées, la création d'un gestionnaire d'état n'est pas triviale.

Il faut bien comprendre le fonctionnement du système pour construire un modèle (donc, un arbre d'attribut) dont l'organisation sera efficace.

De plus, plusieurs optimisations peuvent être faites au niveau du gestionnaire lui-même, et celles-ci peuvent affecter grandement les performances du système global. Des exemples ont été vus au chapitre 4. Ces optimisations ne sont pas toujours évidentes et peuvent complexifier assez vite l'implémentation d'un gestionnaire d'événements.

Bref, le gestionnaire d'état permet à des développeurs de facilement utiliser le nouveau système pour créer et enregistrer des historiques d'état. Cependant, une fois qu'on considère les points énoncés plus haut, son utilisation n'est pas aussi simple que nous l'aurions souhaité.

5.3 Améliorations futures

Bien entendu, le système proposé n'est pas parfait, et bien des améliorations pourront y être faites dans le futur. Nous avons identifié deux d'entre elles.

5.3.1 Noeuds étendus

Une des limitations connues de l'arbre à historique est sa facilité à dégénérer en une liste lorsque les intervalles à insérer sont problématiques. Il peut s'agir d'intervalles en moyenne très longs, d'intervalles dont les temps de fin ne sont pas triés, ou encore lorsque beaucoup d'intervalles se terminent pratiquement tous en même temps.

Ce dernier problème est sans doute le plus courant, puisque beaucoup d'intervalles avec le même temps de fin seront générés lorsqu'on termine un historique d'état. Nous avons vu au chapitre 4 des cas où cette situation causait problème.

La cause principale de ce problème est que lorsqu'un noeud est plein et fermé, nous fermons également *tous les enfants récursifs* de ce noeud. Dans le pire cas, c'est le noeud racine de l'arbre qui est fermé ainsi. Si beaucoup d'insertions sont tentées mais que seul le noeud racine peut recevoir les intervalles, celui-ci risque de se remplir plus rapidement que les noeuds plus bas dans l'arbre. Chaque remplissage du noeud racine entraîne la création d'une nouvelle branche d'arbre qui risque à son tour de demeurer presque vide.

Une solution possible à ce problème serait de ne pas automatiquement fermer les enfants lorsqu'un noeud non-feuille devient plein. Plutôt, nous pourrions *étendre* ce noeud, en positionnant le prochain noeud de l'arbre *derrière* celui-ci, si nous voulons représenter le concept en trois dimensions.

Nous parlerons alors d'un *noeud étendu*. Le nouveau noeud qui sera ajouté portera le

prochain numéro de séquence disponible (et sera positionné à l'endroit correspondant dans le fichier), mais possèdera les mêmes bornes que le noeud derrière lequel il est placé.

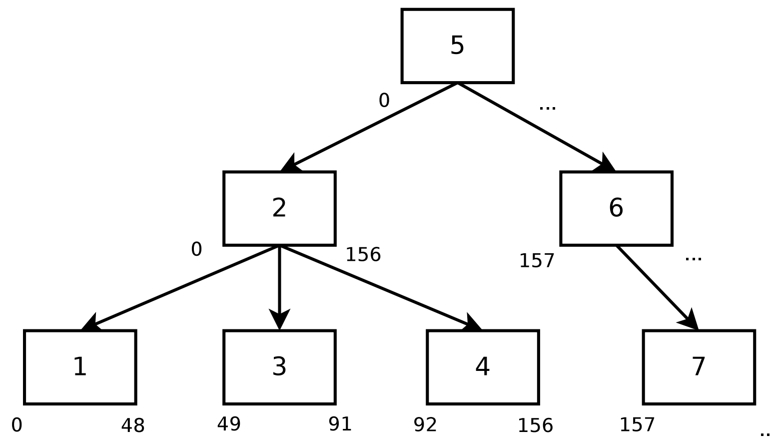

Figure 5.1: Comportement normal

Par exemple, la figure 5.1 montre un petit arbre à historique, incluant les bornes de chacun des noeuds. Les nombres inscrits dans les noeuds représentent leur numéro de séquence (donc, l'ordre dans lequel ils ont été créés).

Si le noeud #2 a été rempli avant le noeud #4, ces deux noeuds auront été fermés et la nouvelle branche 5-6-7 aura été créée, même si le noeud 4 n'était pas encore plein.

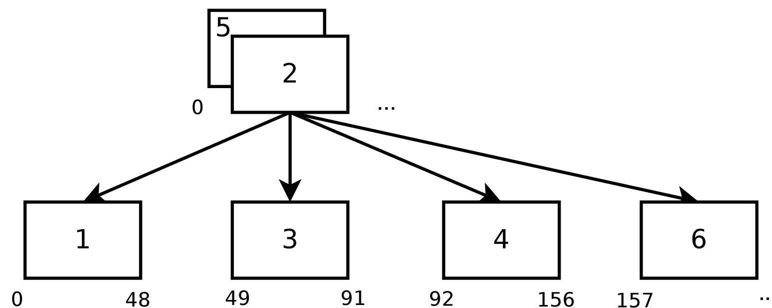

Figure 5.2: Utilisation de noeuds étendus

La figure 5.2 montre ce qui arrive lors de la même situation mais avec l'utilisation de noeuds étendus. Lorsque le noeud #2 sera rempli, au lieu de fermer #4, nous ne ferons qu'étendre le noeud #2. Le prochain noeud créé, le #5, sera placé derrière le noeud 2, et pourra continuer à recevoir des intervalles positionnés entre les mêmes bornes.

Lorsque le noeud 4 sera rempli à son tour, le prochain noeud feuille à être créé sera son frère direct, et non pas son cousin éloigné comme dans 5.1.

La seule condition pour créer une nouvelle racine et de nouvelles branches sera que le nombre maximal d'enfants soit atteint. Lorsque des noeuds non-feuille seront remplis d'intervalles, nous ne ferons que les étendre. Bien entendu, les noeuds feuilles n'auront jamais à être étendus. Ceci mènerait à des arbres beaucoup plus compacts.

Il n'y aurait techniquement pas de limite au nombre d'extensions qu'un noeud peut avoir. Par contre, il serait probablement souhaitable de limiter ce nombre dans l'implémentation (par exemple à 5 ou 10), ou au moins d'afficher un avertissement lorsque cette limite est atteinte quelque part dans l'arbre. Un tel avertissement permettrait de cibler exactement quel noeud, et donc quelle région de l'arbre, est problématique. Cela est parfois difficile à faire avec la méthode actuelle.

5.3.2 Stockage hybride

Une autre amélioration potentielle a été identifiée à plus haut niveau dans le système d'état. Nous avons parlé de l'historique partiel, et de comment celui-ci permet de réaliser des économies d'espace disque considérables par rapport à un historique complet. Cette réduction d'espace se fait au prix de temps de requêtes légèrement plus longs, quoique cette différence n'est pas très importante en pratique. Par contre, une autre limitation importante de l'historique partiel est que nous perdons la possibilité de faire des requêtes ponctuelles et des recherches latérales de manière efficace (nous pouvons toujours réaliser ces opérations, mais il faut passer par des requêtes complètes, qui sont beaucoup plus lentes).

Une possibilité intéressante serait de pouvoir utiliser différentes méthodes de stockage en parallèle. Par exemple, nous pourrions vouloir enregistrer les statistiques dans un historique partiel (où celles-ci se "compressent" très bien) mais tout en gardant un historique complet pour stocker les changements d'états des processus et des CPU, qui sont plus propices à être utilisés par des outils d'analyse.

Ceci permettrait de réaliser de bonnes économies d'espaces, tout en permettant de faire des études de l'état plus poussées, comme de l'analyse de dépendances, à l'aide des attributs jugés plus importants.

Pour que cela soit possible il faudrait pouvoir définir, au niveau du gestionnaire d'événements, quels attributs devront être stockés dans quel historique. Cette déclaration pourrait être faite au moment de la création de chaque nouvel attribut, ou encore à chaque changement d'état.

Le concept d'interface de stockage se prête déjà bien à la modularité. Nous n'aurions qu'à rajouter une petite routine pour vérifier l'information indiquée par le gestionnaire et envoyer

les intervalles d'état dans la méthode de stockage qui leur est appropriée. Une redirection similaire serait également faite lors des requêtes.

Références

ANG, C. et TAN, K. (1995). The interval B-tree. *Information Processing Letters*, 53, 85–89.

ARGE, L. et VITTER, J. S. (2003). Optimal external memory interval management. *SIAM Journal on Computing*, 32, 1488–1508.

AURORA, V. (2009). A short history of btrfs. https://lwn.net/Articles/342892/.

AUSTIN, T., LARSON, E. et ERNST, D. (2002). SimpleScalar : An infrastructure for computer system modeling. *Computer*, 35, 59–67.

BECKMANN, N., KRIEGEL, H., SCHNEIDER, R. et SEEGER, B. (1990). *The R*-tree : an efficient and robust access method for points and rectangles*, vol. 19. ACM.

CANTRILL, B., SHAPIRO, M. et LEVENTHAL, A. (2004). Dynamic instrumentation of production systems. *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2–2.

CHAN, A., GROPP, W. et LUSK, E. (2008). An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16, 155–165.

COMER, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11, 121–137.

CORBET, J. (2008). Tracing : no shortage of options. https://lwn.net/Articles/291091/.

DE BERG, M., CHEONG, O. et VAN KREVELD, M. (2008). *Computational geometry : algorithms and applications*. Springer-Verlag New York Inc.

DESNOYERS, M. et DAGENAIS, M. (2006). The lttng tracer : A low impact performance and behavior monitor for gnu/linux. *Proceedings of the Ottawa Linux Symposium*. Citeseer, vol. 2006.

DESNOYERS, M. et DAGENAIS, M. (2008). LTTng : Tracing across execution layers, from the hypervisor to user-space. *Linux Symposium*.

DESNOYERS, M. et DAGENAIS, M. (2009). LTTng, filling the gap between kernel instrumentation and a widely usable kernel tracer. *Linux Foundation Collaboration Summit*.

EDGE, J. (2009a). A look at ftrace. https://lwn.net/Articles/322666/.

EDGE, J. (2009b). Perfcounters added to the mainline. https://lwn.net/Articles/339361/.

FOURNIER, P., DESNOYERS, M. et DAGENAIS, M. (2009). Combined tracing of the kernel and applications with LTTng. *Proceedings of the 2009 Linux Symposium*.

FOURNIER, P.-M. (2009). *Analyse automatisée des causes de blocage de processus à partir d'une trace d'exécution*. Mémoire de maîtrise, École Polytechnique de Montréal.

GIRALDEAU, F., DESFOSSEZ, J., GOULET, D., DAGENAIS, M. et DESNOYERS, M. (2011). Recovering system metrics from kernel trace. *Linux Symposium*. 109.

GUTTMAN, A. (1984). *R-trees : a dynamic index structure for spatial searching*, vol. 14. ACM.

KAMEL, I. et FALOUTSOS, C. (1994). *Hilbert R-tree : An improved R-tree using fractals*. Citeseer.

KRIEGEL, H., PÖTKE, M. et SEIDL, T. (2000). Managing intervals efficiently in object-relational databases. *Proc. 26th Int. Conf. on Very Large Databases (VLDB)*. Citeseer, 407–418.

NASCIMENTO, M. et SILVA, J. (1998). Towards historical R-trees. *Proceedings of the 1998 ACM symposium on Applied Computing*. ACM, 235–240.

REMNANT, S. J. (2010). How we made Ubuntu boot faster. LinuxCon 2010, diponible : `https://events.linuxfoundation.org/slides/2010/linuxcon2010_remnant.pdf`.

RODEH, O. (2008). B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3, 2.

ROSTEDT, S. (2010). Using the TRACE EVENT() macro. `https://lwn.net/Articles/379903/`, `https://lwn.net/Articles/381064/`, `https://lwn.net/Articles/383362/`.

ROSTEDT, S. (2011). Using kernelshark to analyze the real-time scheduler. `https://lwn.net/Articles/425583/`.

SELLIS, T., ROUSSOPOULOS, N. et FALOUTSOS, C. (1987). The R+-tree : A dynamic index for multi-dimensional objects. *Proceedings of the 13th International Conference on Very Large Data Bases*. Citeseer, 507–518.

TAO, Y. et PAPADIAS, D. (2001). The MV3R-Tree : A spatio-temporal access method for timestamp and interval queries. *Proceedings of the 27th VLDB Conference*.

Annexe A

Format de fichier sur disque de l'arbre à historique

Cette section décrit le format du fichier de l'arbre à historique. Un fichier représente un arbre en entier. Il serait possible de stocker un historique en plusieurs arbres-fichiers, si cela s'avérait nécessaire.

Le fichier débute avec un en-tête de taille fixe. Nous fixons cette taile à 4 KB pour que le premier bloc (et tous les autres qui suivent) restent alignés. L'en-tête est suivi par la série de blocs, qui sont tous de la même taille (cette taille est configurable pour chaque historique). Chaque noeud de l'arbre à historique correspondra à un bloc dans le fichier.

Enfin, lorsque l'historique est terminé, nous ajoutons une troisième section pour l'arbre d'attributs. Cette section permet de recharger l'arbre d'attributs en mémoire lorsque nous ouvrirons cet historique à nouveau.

L'ordre des bits recommandé est *little-endian*.

A.1 En-tête du fichier d'historique

Le contenu de l'en-tête du fichier est le suivant :

Pos.	Type	Contenu	Commentaire
0	int	"Magic number"	Indique le type de fichier
4	int	Numéro de version	Version du format de fichier
8	bool	Mode déboggage	Indique si l'arbre est en mode déboggage
12	int	BLOCKSIZE	Taille des blocs du fichier
16	int	MAX_CHILDREN	Nombre d'enfants maximal permis par noeud
20	int	Nombre de noeuds	Nombre de noeuds existant dans l'arbre
24	int	Noeud racine	No de séquence du noeud racine de l'arbre
28	Bourrage		
4096	Fin de l'en-tête/début des blocs		

Les constantes BLOCKSIZE et MAX_CHILDREN sont les deux paramètres de configuration qui doivent être spécifiés lors de la création de l'arbre. Nous pouvons connaître la position de début de la section "Arbre d'attributs" du fichier à l'aide de *Nombre de noeuds* $*$ *BLOCKSIZE*.

A.2 Blocs

Chaque bloc du fichier a une taille égale à BLOCKSIZE, en octets, qui est spécifiée dans l'en-tête du fichier. Chaque bloc contient également son propre en-tête, une section *Data* et une section *Strings*.

L'en-tête est de taille fixe et contient l'information globale du noeud.

Chaque intervalle d'état appartenant au noeud aura une entrée dans la section *Data*. Chacune de ces entrées sera de taille fixe, par contre le nombre d'entrées maximal variera en fonction des valeurs d'états enregistrées.

Enfin, la section *Strings* est placée à la fin complètement du noeud, similairement à la section *Strings* du format de fichier ELF. Cette section sert à stocker les tableaux d'octets de longueur variable (comme les chaînes de caractères). Les entrées de la section *Data* peuvent optionnellement référer à une entrée de la section *Strings*.

Lorsqu'il n'y a plus d'espace libre entre la fin de la section *Data* et le début de la section *Strings*, le noeud est considéré plein.

A.2.1 En-tête d'un bloc

Le tableau suivant décrit le contenu de l'en-tête d'un bloc du fichier d'historique. Les positions représentent le nombre d'octets depuis le début du bloc dans le fichier.

Pos.	Type	Contenu	Commentaire
0	long	Borne initial	Temps de début du noeud
8	long	Borne finale	Temps de fin du noeud
16	int	No. de séquence	Numéro de séquence de ce noeud
20	int	Noeud parent	No. de séquence du noeud parent
24	int	Nb d'enfants	Nombre d'enfants que ce noeud possède
28	int	Nb d'intervalles	Nombre d'intervalles contenus
32	bool	terminé	Indique si ce noeud est "terminé"
33	bool	plein	Indique si ce noeud est "plein"
36	int	Pos. section *Strings*	Position où débute la section *Strings*
40	int[]	Noeuds enfants	Numéros de séquence des enfants
a	long[]	Bornes des enfants	Temps de début des enfants
b	Fin de l'en-tête / Début de la section *Data*		

$a = 40 + \text{MAX_CHILDREN} * 4$
$b = 40 + \text{MAX_CHILDREN} * (4 + 8)$

Comme nous voulons un en-tête de taille fixe, les deux tableaux à la fin sont définis de manière statique. Ils contiendront toujours de la place pour MAX_CHILDREN entrées. Cette valeur est constante pour tous les noeuds d'un arbre (mais pourrait varier d'un arbre à l'autre).

Un noeud est dit "terminé" s'il a été fermé et enregistré sur disque et qu'il existe des noeuds à sa droite dans l'historique. Un noeud est dit "plein" s'il n'a plus de place pour recevoir de nouveaux intervalles.

Cette légère nuance permet de spécifier par exemple un noeud dont le parent a été fermé mais qui pourrait encore recevoir des intervalles si ceux-ci étaient situés entre ses bornes. Dans l'implémentation de référence, cette différence n'est pas utilisée, puisqu'on n'insère jamais d'intervalles dans le passé.

A.2.2 Section "Data"

Chaque entrée de la section *Data* correspond à un intervalle contenu dans ce noeud. Les positions sont calculées ici par rapport au début de la section. La première entrée vient donc immédiatement après l'en-tête.

Pos.	Type	Contenu	Commentaire
0	long	Temps de début	Temps de début de l'intervalle
8	long	Temps de fin	Temps de fin de l'intervalle
16	int	quark	Entier identifiant l'attribut
20	byte	type	Indique comment interpréter le champ "valeur"
21	int	valeur	Voir ci-dessous
25	Début de l'entrée suivante		

Le quark est la représentation numérique de l'attribut dont l'état est représenté par cet intervalle. À l'aide de l'arbre d'attributs, on peut le reconvertir en son nom complet.

Le champ "type" représente le type de valeur d'état qui est stockée dans cet intervalle. Celui-ci indique la manière d'interpréter le champ "valeur". Pour l'instant les deux types de valeurs possibles sont :

type = 0 ⇒ La valeur d'état est un entier. Lire le champ "valeur" directement.

type = 1 ⇒ La valeur d'état est une chaîne de caractère. Le champ "valeur" indique la position d'une entrée dans la section *Strings*.

Il serait facile d'étendre le système pour supporter d'autres types de valeurs d'état (en définissant d'autres "types"). Des valeurs de 4 octets ou moins (comme des *char*, des *short int*, etc.) pourraient être stockées directement dans le champ "valeur". Des valeurs plus grandes, ou de taille variable (comme des *struct*) pourraient être enregistrées plutôt dans la section *Strings*.

A.2.3 Section "Strings"

Contrairement aux deux autres sections des blocs, la section *Strings* est alignée sur la fin du bloc et "grandit" vers l'arrière. Chaque entrée est de taille variable. Les entrées de la section *Data* contenant une valeur d'état comme une chaîne de caractères pointeront sur une entrée de la section *Strings*.

Chaque entrée débute avec un entier signé de 8 bits (ce qui correspond à la primitive *byte* en Java) indiquant la longueur de cette entrée. Vient ensuite une série d'octets, de longueur égale à ce nombre. C'est le champ "type" correspondant qui indique comment interpréter ce tableau d'octets.

Chaque entrée est terminée par un *byte* de valeur 0 pour en indiquer la fin. À l'aide du *byte* initial indiquant la longueur, celui-ci permet de s'assurer que nous venons de lire une entrée qui n'est pas corrompue.

Lorsqu'une nouvelle entrée est ajoutée à la section *Strings*, elle est écrite juste avant l'entrée précédente. On met ensuite à jour le champ "Position de la section *Strings*" du bloc.

À mesure que des entrées sont ajoutées dans les sections *Data* et *Strings*, l'espace disponible entre les deux sections va diminuer. Avant chaque insertion d'intervalle, on vérifie qu'il reste suffisamment d'espace. Si ce n'est pas le cas, on marque le noeud comme "plein", celui-ci est fermé et l'intervalle sera inséré ailleurs dans l'arbre.

A.3 Arbre d'attributs

Lorsqu'un historique d'état est complété, l'arbre d'attributs est enregistré à même le fichier de l'arbre à historique, dans sa section située à la fin complètement. L'information conservée permettra de reconstruire l'arbre d'attributs à nouveau lorsque cet historique sera rechargé plus tard.

Dans sa représentation en mémoire, les chemins d'accès sont enregistrés sous forme de références à des chaînes de caractères. De cette façon chaque nom de "répertoire" unique n'est stocké qu'une seule fois. Par contre, lorsque l'arbre d'attributs est écrit sur disque, nous écrirons le chemin d'accès complet pour chaque attribut. Ceci causera de l'information redondante à être écrite, mais permet de reconstruire l'arbre beaucoup plus facilement. De plus, la quantité d'information répétée est minime par rapport à l'espace occupé par les noeuds et les intervalles.

La position du début de cette section dans le fichier peut être déterminée à l'aide des valeurs des champs BLOCKSIZE et "nombre de noeuds", qui sont présents dans l'en-tête.

L'organisation de la section de l'arbre d'attributs est comme suit :

Type	Contenu	Commentaire
int	"Magic number"	Pour identifier la section
int	Taille totale	Taille totale de cette section (en octets)
int	Nombre d'entrées	Combien d'entrées sont présentes
byte	Taille de l'entrée	Longueur de cette entrée (en octets)
byte[]	Entrée	Le tableau de caractères représentant un attribut
byte	0	Caractère de fin de chaîne
byte	Taille de l'entrée	Longueur de l'entrée suivante
etc.		

Les chaînes de caractères comprennent les chemins d'accès complets, dont les répertoires sont séparés par des barres obliques (/). Les entrées sont inscrites par ordre de quarks. Par

exemple, la première entrée de la section correspondra au quark 0, la seconde au quark 1, et ainsi de suite.

Nous indiquons également un “magic number” spécifique à la section. Celui-ci permet de stocker l’arbre d’attributs dans un fichier à part (si nous n’utilisons pas l’arbre à historique comme méthode de stockage de l’état, par exemple).

Annexe B

Format de trace VSTF

Cette section décrit le format de trace VSTF. C'est un format de trace qui se veut très simple, d'où son nom *Very Simple Trace Format*.

Une particularité du format est qu'il ne contient aucune méta-information. Chaque événement spécifie un "type", qui n'est qu'un nombre entier unique. Le mappage entre chaque type et sa signification doit être fait à l'extérieur du format de trace.

Une trace VSTF débute par un court en-tête, qui est suivi d'une succession d'événements. Chaque événement est composé de zéro ou plusieurs champs. Il existe plusieurs types de valeurs possibles pour ces champs.

B.1 En-tête du fichier de trace

Le fichier débute par un *int* indiquant le "magic number" de ce type de fichier. Il est suivi par un autre *int* représentant le numéro de version (à des fins de compatibilité).

Vient ensuite un *byte* (entier signé de 8 bits) valant zéro. Comme nous le verrons plus loin, chaque événement se termine par un *byte* à zéro. Lorsque nous nous positionnons sur un événement de la trace, il est possible de vérifier que le *byte* juste avant la position est bien égal à zéro. Cela n'est bien sûr pas parfait, mais permet de donner une certaine garantie d'intégrité.

B.2 Événements

Le tableau suivant montre le contenu d'un événement VSTF :

Pos.	Type	Contenu	Commentaire
0	long	Estampille	Estampille temporelle de l'événement
8	int	Source	Source (#CPU) de l'événement
12	int	Type	Type d'événement
16	byte	Nombre de champs	
17	Espace pour les champs, terminé par un *byte* à zéro		

Chaque champ débute par un *byte* indiquant le type du champ. Par convention, une valeur négative indique un champ de taille fixe et une valeur positive, un champ de taille variable. Par exemple, les valeurs -2 et -3 pour les types *int* et *long*, respectivement, sont suggérées. Cette valeur viendra immédiatement après le *byte* initial.

Pour les types de taille variable, un autre *byte* suit pour indiquer la taille, en octets, occupée par la valeur. Cette valeur sera lue comme un simple tableau d'octets et sera envoyée à l'application. C'est à l'application d'interpréter ce tableau comme elle le veut, en fonction du type de champ. Le type "1" a été utilisé pour représenter des chaînes de caractères.

Enfin, une fois tous les champs terminés, un *byte* mis à zéro sert à confirmer la fin de l'événement.

Le format est fait de telle sorte qu'il faille lire toute l'information d'un événement avant de pouvoir passer au prochain. On ne peut pas sauter par-dessus un événement à cause des tailles variables impliquées, nous ne savons pas où commence le prochain. Il est cependant possible d'enregistrer la position actuelle du lecteur de traces de manière à pouvoir revenir exactement à cette position dans le futur.