Analyzing the memory behavior of parallel scientific applications

David Beniamine

▶ **To cite this version:**

David Beniamine. Analyzing the memory behavior of parallel scientific applications. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Grenoble Alpes, 2016. English. NNT : 2016GREAM088 . tel-01681008v2

HAL Id: tel-01681008

https://theses.hal.science/tel-01681008v2

Submitted on 11 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

David Beniamine

Thèse dirigée par **Bruno Raffin**
et codirigée par **Guillaume Huard**

préparée au sein **LIG**
et de **EDMSTII**

Analyzing the memory behavior of parallel scientific applications

Thèse soutenue publiquement le 5 décembre 2016
devant le jury composé de :

Pr, Martin Quinson
Professor at ENS Rennes, Président
Pr, Jesús Labarta Mancho
Professor at Universitat Politècnica de Catalunya, Rapporteur
Pr, Raymond Namyst
Professor at University of Bordeaux, Rapporteur
Dr, Lucas M. Schnorr
Associate professor at Institute of Informatics of the Federal University of Rio Grande do Sul, Examinateur
Pr, Bruno Raffin
Professeur at INRIA Grenoble, Directeur de thèse
Dr, Guillaume Huard
Associate professor at Grenoble Alpes University, Co-Directeur de thèse

Acknowledgments

I first would like to thank the members of my jury for the time they spent on my work and for their helpful remarks.

I would like to thank all the members of the Informatica team of Porto Alegre for welcoming me there. Especially Mathias and Marcos, I enjoyed working with you guys, it has been a great experience.

Eu gostaria de agradecer todos os meus colegas em Porto Alegre que me receberam de braços abertos e fizeram da minha estadia um período inesquecível. Dentre outros, gostaria de agradecer todos os moradores do condomínio Riachuelo e os visitantes regulares pelos momentos divertidos e discussões interessantes. And thank you Fernando for the translation.

Je tiens à remercier tous les membres présents et passés des équipes Polaris/Moais/Datamove/Mescal, travailler avec vous à été très enrichissant, scientifiquement comme humainement. Et surtout merci à Annie Simon et Christine Guimet qui plus d'une fois ont rattrapé mes bêtises administratives. Je voudrais aussi remercier les enseignants de l'UFR IM2AG qui m'ont motivé à pousser les études jusqu'à la thèse et ont continué à suivre mon évolution, en particulier Anne Rasse. Dans un autre registre, tous les coincheur.euse.s grâce à qui les pauses midi finissent souvent en fou rire et concours de mauvaise foi. Tous mes co-bureaux même temporaires qui ont rendu les heures de déprime de rédaction ou de debug supportables et souvent même agréables. Plus particulièrement Raphaël (avec ou sans son genou) et Alexis pour tous les interminables débats mi-Trolls mi-sérieux et les croissants du lundi. Swann qui rentre à la fois dans toutes ces catégories et dans aucune, et sans qui je n'aurais peut-être pas atterri dans ce laboratoire. Enfin je voudrais remercier mes encadrants et surtout Guillaume pour ces 5 années durant lesquelles il m'a accompagné et encouragé dans mon travail, qu'il s'agisse de la thèse ou d'autres projets, toujours avec beaucoup d'humour, parfois même sur un vélo ou derrière un verre (heureusement jamais les deux à la fois). Bref j'ai eu de la chance d'avoir un aussi bon

encadrant[1], j'espère vraiment que nous continuerons à travailler ensemble.

Je voudrais finalement remercier tou.te.s mes ami.e.s qui depuis des années me supportent avec tous mes défauts et sont toujours présent malgré tout. Tous les sportifs occasionnels ou réguliers, sans nos sorties vélo, via, rando, etc. je serais probablement devenu fou. Parmi eux je tiens à remercier spécifiquement Seb pour sa capacité à transformer n'importe quelle "petite sortie d'une heure" en aventure, parfois en y laissant une roue, des sacoches ou carrément un genou (encore un). Je tiens aussi à remercier mes parents et Sacha pour avoir subi mon incapacité à organiser ma vie hors de la thèse sans jamais se plaindre ni m'en tenir rigueur mais aussi pour leurs encouragements. Enfin, Flo, je ne peux pas imaginer ce qu'auraient été ces dernières années ni comment aurait fini cette thèse sans toi, je ne peux pas non plus te dire tout ce pour quoi je veux te remercier sans doubler (au minimum) la taille de ce manuscrit, alors je dirais juste: "thanks for all the fish" (mais pas "so long", bien au contraire).

[1] "le mauvais encadrant, il voit un thésard, il l'encadre, alors que le bon encadrant, il voit un thésard, il l'encadre, mais c'est un bon encadrant. . . "

Abstract

Since a few decades, to reduce energy consumption, processor vendors builds more and more parallel computers. At the same time, the gap between processors and memory frequency increased significantly. To mitigate this gap, processors embed a complex hierarchical caches architecture. Writing efficient code for such computers is a complex task. Therefore, performance analysis has became an important step of the development of applications performing heavy computations.

Most existing performance analysis tools focuses on the point of view of the processor. Theses tools see the main memory as a monolithic entity and thus are not able to understand how it is accessed. However, memory is a common bottleneck in High Performance Computing (HPC), and the pattern of memory accesses can impact significantly the performance. There are a few tools to analyze memory performance, however theses tools are based on a coarse grain sampling. Consequently, they focus on a small part of the execution missing the global memory behavior. Furthermore, these coarse grain sampling are not able to collect memory accesses patterns.

In this thesis we propose two different tools to analyze the memory behavior of an application. The first tool is designed specifically for Non-Uniform Memory Access (NUMA) machines and provides some visualizations of the global sharing pattern inside each data structure between the threads. The second one collects fine grain memory traces with temporal information. We can visualize theses traces either with a generic trace management framework or with a programmatic exploration using R. Furthermore we evaluate both of these tools, comparing them with state of the art memory analysis tools in terms of performance, precision and completeness.

Résumé

Depuis plusieurs décennies, afin de réduire la consommation énergétique des processeurs, les constructeurs fabriquent des ordinateurs de plus en plus parallèles. Dans le même temps, l'écart de fréquence entre les processeurs et la mémoire a significativement augmenté. Pour compenser cet écart, les processeurs modernes embarquent une hiérarchie de caches complexe. Développer un programme efficace sur de telles machines est une tâche difficile. Par conséquent, l'analyse de performance est devenue une étape majeure lors du développement d'applications exécutant des calculs lourds.

La plupart des outils d'analyse de performances se concentrent sur le point de vue du processeur. Ces outils voient la mémoire comme une entité monolithique et sont donc incapables de comprendre comment elle est accédée. Cependant, la mémoire est une ressource critique et les schémas d'accès à cette dernière peuvent impacter les performances de manière significative. Quelques outils permettant l'analyse de performances mémoire existent, cependant ils sont basés sur un échantillonnage à large grain. Par conséquent, ces outils se concentrent sur une petite partie de l'exécution et manquent le comportement global de l'application. De plus, l'échantillonnage à large granularité ne permet pas de collecter des schémas d'accès.

Dans cette thèse, nous proposons deux outils différents pour analyser le comportement mémoire d'une application. Le premier outil est conçu spécifiquement pour pour les machines NUMA (Not Uniform Memory Accesses) et fournit plusieurs visualisations du schéma global de partage de chaque structure de données entre les flux d'exécution Le deuxième outil collecte des traces mémoires à grain fin avec information temporelles. Nous proposons de visualiser ces traces soit à l'aide d'un outil générique de gestion de traces soit en utilisant une approche programmatique basé sur le langage R. De plus nous évaluons ces deux outils en les comparant à des outils existant de trace mémoire en terme de performances, précision et de complétude.

Résumé étendu

Les scientifiques de toute les disciplines utilisent des ordinateurs pour faciliter leurs calculs et exécuter des simulations afin de tester leurs hypothèses. Plus la science avance, plus ces simulations deviennent complexes, les scientifiques ont donc toujours besoin de plus de capacité de calcul. Pour augmenter la capacité de calcul de leurs processeurs, les constructeurs ont dans un premier temps augmenté la fréquence de ces derniers. Cependant cette approche a vite été stoppée par plusieurs limites physiques. Afin de contourner ces limites, les constructeurs se sont mis à concevoir des processeurs parallèles.

La première limite provient de l'énergie nécessaire pour augmenter la fréquence d'un processeur. En effet, d'après Intel, augmenter la fréquence d'un processeur de 20 % n'augmente les performances que d'un facteur 1.13 mais requière 1.73 fois plus d'énergie. A l'opposé, utiliser un processeur identique mais avec deux cœurs de calculs au lieu d'un en diminuant la fréquence de ce dernier de 20 %, permet d'obtenir 1.73 fois plus de performances pour uniquement 1.02 fois plus d'énergie. La deuxième limite est la vitesse de la lumière : en effet les données doivent transiter de la mémoire jusqu'au processeur et ne peuvent pas se déplacer plus vite que la lumière. Si cette limite peut paraître élevée, elle a déjà été atteinte. En effet si nous voulons construire une machine séquentielle capable de traiter 1 To de données par seconde, du fait de cette limite, il faudrait faire tenir 1 To dans une aire de $0.3\,\text{mm}^2$, ce qui signifie que 1 bit occupe uniquement 0.1 nm, la taille d'un petit atome.

Si ces processeurs parallèles sont en théorie plus puissants que les séquentiels, les utiliser efficacement est bien plus complexe et relève de la responsabilité des développeur.euse.s. De plus, depuis plusieurs années nous sommes face à une troisième limite physique. Nous somme capable de réduire la taille des transistors, donc d'augmenter le nombre de transistors sur une puce. Cependant, plus de transistors signifie plus de chaleur et la quantité de chaleur qu'une puce peut produire avant que des effets indésirables tel que des courants de fuites

se produisent. De ce fait, les constructeurs font désormais des machines avec plusieurs puces (processeurs), chacune étant composée de plusieurs cœurs.

Dans le même temps, les processeurs sont devenus significativement plus rapides que la mémoire; pour palier à cet écart, ils embarquent des petites mémoires cache. Ces caches sont conçus pour tirer profit de deux schémas d'accès communs à la plupart des programmes : la localité spatiale et temporelle qui correspond respectivement au fait d'accéder à des données proches et d'accéder plusieurs fois aux mêmes données dans un temps restreint. Une des choses qui rend ces caches plus rapides que la mémoire est leur taille, plus ils sont petits, plus il est rapide d'y trouver une donnée. C'est pourquoi les processeurs embarquent plusieurs niveaux de cache (en général trois). Le premier est très petit et rapide, quelques kilo octets, et conçu pour des accès très proches (boucles sur un tableau), le dernier plus grand et lent, environ 10 Mo, et conçu pour des accès plus espacés. Comme les processeurs sont multi-cœurs, ces caches sont organisés hiérarchiquement, le dernier niveau est partagé par tous les cœurs, et chaque cœur a accès à un cache de niveau 1 privé. Cette hiérarchie permet d'isoler les données privées et de tirer profit des partages correctement structurés; de plus cela réduit la bande passante nécessaire entre les caches. Pour des raisons similaires, les ordinateurs comportant plusieurs processeurs ont une organisation mémoire non uniforme (NUMA) ce qui signifie que chaque puce a un accès privilégié à une sous-partie de la mémoire. Par conséquent, les schémas d'accès à la mémoire d'une application peuvent avoir un impact significatif sur ses performances.

En fin de compte, écrire un programme efficace nécessite de prendre en compte l'architecture de la machine qui va l'exécuter, les schémas d'accès et leur adéquation, même si le programme est séquentiel. Bien qu'il y ait des règles générales : privilégier les accès séquentiels, travailler sur des petits ensembles de données, cette tâche est extrêmement complexe même pour des spécialistes de calcul haute performance (HPC). Les outils d'analyse de performances sont donc extrêmement utiles pour comprendre et optimiser les performances d'une application.

La première étape lors de l'optimisation des performances d'une application consiste à identifier les points chauds, c'est à dire les parties du code qui sont inefficaces et comprendre la nature des erreurs qui entrainent cette inefficacité. C'est uniquement après cette étape qu'il est possible de décider quelle partie du code peut être améliorée et comment. Il existe de nombreux outils conçus pour analyser les performances d'une application [Pillet et al., 1995, Browne et al., 2000, Shende and Malony, 2006, Treibig et al., 2010, Adhianto et al., 2010], la plupart d'entre eux utilisent les compteurs de performances pour collecter la trace d'une application. Ces compteurs sont des registres processeurs dédiées à l'analyse de performances qui permettent la collecte efficace de données concernant les performances.

Dans cette thèse nous avons mené une étude de cas sur l'analyse de performance d'un outil de simulation physique : Simulation Open Framework Architecture (Sofa). Afin d'analyser les performances de cette application, nous avons utilisé "Like I Knew What I am Doing" (Likwid) [Treibig et al., 2010], un outil classique d'analyse de performances, et nous avons tracé plusieurs métriques concernant l'utilisation de la mémoire. Avec cet outil nous avons été capable de détecter des problèmes de performances liés à la mémoire, mais il était impossible de trouver leur position dans la mémoire et les schémas responsable

des mauvaises performances. En effet, si ces compteurs peuvent s'avérer très utiles, ils voient la mémoire comme une entité monolithique ce qui n'est pas le cas pour les architecture récentes. Par conséquent, des outils spécifiques doivent être utilisés pour analyser les performances du point de vue de la mémoire.

Analyser les performances d'une application au regard de la mémoire soulève deux défis techniques : le premier est la collecte de la trace elle-même. C'est une tâche compliquée car il n'existe pas de matériel comparable aux compteurs de performances pour tracer les accès mémoire. De plus, chaque instruction d'un programme déclenche au moins un accès mémoire : collecter chaque accès mémoire d'une application n'est donc pas possible. Par ailleurs, l'absence de matériel pour tracer les accès mémoire implique qu'un outil de collection peut facilement devenir envahissant et modifier significativement le comportement de l'application analysée. Le deuxième défi technique consiste à présenter la trace de manière simple et compréhensible. En effet, les traces mémoire sont extrêmement complexes puisqu'elles sont étalées sur cinq dimensions : le temps, l'espace d'adressage, la localité processeur, les flux d'exécution et le type d'accès. De plus, certaines de ces dimensions ne sont pas triviales à représenter, par exemple l'espace d'adressage peut être physique ou virtuel, et la localité processeur est organisée de manière hiérarchique. Finalement, les outils d'analyse mémoire doivent extraire les données pertinentes et les présenter de manière compréhensible.

Un outil idéal d'analyse mémoire devrait être capable de présenter les schémas d'accès mémoire d'un programme à ses développeur.euse.s, en incluant des informations concernant le partage de données entre flux d'exécution et la localisation des accès sur l'architecture de la machine. De plus, un tel outil devrait mettre en avant les schémas inefficaces.

Plusieurs outils ont été conçus dans le but d'analyser les performances mémoire [Lachaize et al., 2012, Liu and Mellor-Crummey, 2014, Giménez et al., 2014], cependant la plupart d'entre eux collectent la trace à l'aide de la technique d'échantillonnage d'instructions. L'échantillonnage d'instructions est une technique assistée par le matériel, qui permet de tracer certaines instructions à une fréquence[Drongowski, 2007, Levinthal, 2009] définie. Si cette méthode permet de collecter rapidement une trace, elle manque la plupart de l'espace d'adressage. De ce fait, il est impossible de visualiser les schémas d'accès mémoire à partir de la trace collectée.

Contributions

Dans cette thèse, nous proposons deux outils pour analyser le comportement mémoire d'une application. Notre premier outil, nommé Tool for Analyzing the Behavior of Applications Running on NUMA ArChitecture (Tabarnac), collecte des traces globales de l'utilisation de la mémoire sans informations temporelles et présente une vue d'ensemble des schémas de partage à l'intérieur des structures de données entre les flux d'exécution. Le deuxième, nommé Memory Organisation Cartography & Analysis (Moca), collecte des traces mémoire génériques à grain fin, avec informations temporelles. Nous proposons deux approches différentes pour visualiser les traces collectées par Moca, la première est basée sur FrameSoc, un outil existant d'analyse de gestion de traces, la deuxième approche est basée sur une exploration programmatique utilisant le langage R.

Conduire une campagne d'expérience en informatique peut être extrêmement simple, mais le faire de manière reproductible requiert davantage de planification et de méthodologie. L'analyse de performances, qu'elle soit conduite dans le but d'optimiser une application ou pour évaluer un outil requiert une campagne d'expériences complète. Dans cette thèse, nous avons porté une attention particulière à rendre nos expériences aussi reproductibles que possible. Dans ce but, nous décrivons clairement notre méthodologie expérimentale et distribuons tous les fichiers requis afin de reproduire chaque étape des expériences présentées.

Vue d'ensemble des schémas de partage

Nous avons conçu Tabarnac afin d'analyser les schémas de partage d'applications s'exécutant sur des machines NUMA. Cet outil est basé sur une instrumentation binaire légère pré-existante, elle-même basée sur la bibliothèque Pin d'Intel. Cette instrumentation compte le nombre d'accès de chaque flux d'exécution à chaque page mémoire d'une application. Nous avons ajouté à cette bibliothèque la capacité de retrouver des informations contextuelle afin d'associer les adresses mémoire à des structures de données (statiques ou allouées). De plus, nous avons conçu plusieurs visualisations simples et compréhensibles pour les traces collectées. En utilisant ces visualisation, nous avons pu identifier des problèmes d'utilisation mémoire et augmenter de 20 % les performances d'une application de test largement étudiée.

Ces résultats ont été publiés dans un article à Visual Performance Analysis (VPA) 2015, un séminaire de Super Computing [Beniamine et al., 2015b]. De plus, Tabarnac est distribué en tant que logiciel libre sous la licence General Public License (GPL) : https://github.com/dbeniamine/Tabarnac. Ce travail est le fruit d'une collaboration avec M. Diener et P.O.A Navaux de l'équipe informatica de l'Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Bresil, financé par CAMPUS France.

Collecte de traces mémoires à grain fin

Moca est notre contribution principale. Cette outil est basé sur un module noyau Linux pour collecter efficacement des traces mémoires à grain fin. Ce module noyau intercepte les défauts de pages, qui sont déclenchés par le processeur et gérés par le système d'exploitation, afin de tracer les accès mémoire. Comme ces défauts de pages n'ont pas lieu fréquemment, il injecte aussi périodiquement de faux défauts de pages. De plus nous avons porté notre bibliothèque qui trace les structures de données dans Moca en s'affranchissant de sa dépendance à Pin. Nous avons aussi exécuté une campagne expérimentale approfondie, comparant Moca à Tabarnac ainsi que deux outils existants de collecte de traces mémoire, en termes de surcout, précision et complétude.

Ce travail est le sujet de deux rapport de rechercher Inria [Beniamine et al., 2015a, Beniamine and Huard, 2016] et a été soumis à Cluster, Cloud and Grid Computing (CCGRID) 2017. Comme l'outil précédent, Moca est distribué sous licence GPL : https://github.com/dbeniamine/Moca.

Analyse de traces mémoires à grain fin

Nous proposons deux approches différentes pour visualiser les traces de Moca. La première est basée sur un outil générique de gestion et d'analyse de traces : FrameSoc [Pagano and Marangozova-Martin, 2014]. Plus particulièrement, elle repose sur Ocelotl [Dosimont et al., 2014], un outil FrameSoc qui agrège les parties similaires de la trace et présente une vue globale simplifiée qui met en avant les anomalies. Avec cet outil nous avons pu identifier des schémas inefficaces en mémoire connus sur une application de test. Cependant nous avons rencontré plusieurs problèmes de passage à l'échelle dus à la représentation générique de la trace dans FrameSoc.

Afin de dépasser ces limites, nous avons aussi exploré plusieurs traces Moca avec une approche programmatique utilisant le langage R. Cette approche permet d'utiliser des filtres et zooms avancés, et de concevoir des visualisations spécifiques pour chaque trace. Avec cette méthode, nous avons pu analyser des traces plus complexes et détecter des comportements mémoires inconnus et intéressants. Afin d'être reproductibles, ces analyses sont sauvées et versionnées dans un cahier de laboratoire publiquement accessible sur github :
https://github.com/dbeniamine/Moca_visualization.

Organisation de cette thèse

Cette thèse est organisée de la manière suivante : dans le chapitre 2, nous présentons une étude de cas sur l'analyse de performances de Sofa, un outil de simulation physique. Ce chapitre commence par présenter Sofa, ses spécificités ainsi que les tentatives précédentes de l'optimiser et souligne le besoin d'analyser les performances de Sofa. Ensuite, nous discutons des outils génériques d'analyse de performances existants et notre méthodologie expérimentale. Cette étude de cas met en avant le besoin d'outils spécifiques pour l'analyse de performance du point de vue de la mémoire. Dans le chapitre 3, nous présentons certaines spécificités des architectures mémoires récentes, des problèmes de performances classique liés à l'utilisation de la mémoire et des façons de les contourner. Puis nous traitons des outils d'analyse de performances mémoire existants, de leurs limites et ce que l'on attendrais d'un outil idéal. Après cela nous présentons Tabarnac, notre première contribution, dans le chapitre 4. Nous présentons sa conception et son utilisation, évaluons son surcout et finalement présentons des optimisations de performances réalisées grâce à la connaissance acquise en analysant des traces produite par Tabarnac. Dans le chapitre 5, nous décrivons notre contribution principale : Moca. Nous expliquons d'abord en détail les mécanismes utilisés par Moca, sa conception interne et comment il répond aux défis soulevés par la collecte de traces mémoire à grain fin. Puis, nous proposons une analyse expérimentale approfondie comparant Moca à deux outils existants de collecte de traces mémoires ainsi qu'à Tabarnac en termes de performances, précision et complétude. Le chapitre 6 traite de la visualisation des traces de Moca. Nous présentons d'abord FrameSoc et Ocelotl puis les résultats obtenus avec ces outils. Après cela nous proposons une approche programatique et présentons les visualisations obtenues avec cette approche. Enfin nous tirons nos conclusions et proposons des perspectives de travail futures dans le chapitre 7.

Outline

Acknowledgments **iii**

Abstract **vii**

Résumé **ix**

Résumé étendu **xi**

I Introduction 3

1.1 Contributions 6
1.2 Thesis organization 7

II Case Study 11

2.1 Motivations 12
2.2 Profiling tools 15
2.3 Experimental methodology 16
2.4 SOFA Analysis 25

III Memory Performance Analysis 31

3.1 Architectural considerations 32
3.2 Existing tools 39
3.3 Conclusions 43

IV Collecting and Analyzing Global Memory Traces 45

4.1 Design 46

4.2 Experimental validation 50
4.3 Results and discussion 57

V Collecting Fine Grain Memory Traces 61

5.1 Moca components 62
5.2 Background knowledge 63
5.3 Design 63
5.4 Experimental validation 70
5.5 Conclusions 78

VI Analyzing Fine Grained Memory Traces 81

6.1 Interactive visualization of aggregated trace 82
6.2 Programmatic exploration 90
6.3 Conclusions 96

VII Conclusions and perspectives 99

7.1 Contributions 100
7.2 Perspectives 101

Contents 105

List of Figures 109

List of Tables 113

Acronyms 115

Glossary 119

Bibliography 125

Chapter I

Introduction

Contents

1.1 Contributions **6**
1.1.1 Global overview of the memory sharing patterns **6**
1.1.2 Fine grain memory traces collection **6**
1.1.3 Fine grain memory traces analysis **7**
1.2 Thesis organization **7**

Scientists from all fields use computers to ease their calculations and run simulations to test their hypothesis. These simulations are more and more complex as science advances and therefore requires always more computing power. In a first time, computer vendors increased this power by increasing the frequency of their Central Processing Units (CPUs), but this approach reached quickly several hard physical limits. To overpass them, they started to build parallel processors.

The first limit comes from the energy required to increase the frequency of a CPU. Indeed, according to Intel [Ganesan, 2016], over-clocking a processors by 20 % only increase the performance by a factor 1.13 but requires 1.73 times more energy. At the opposite, using an identical processor with two cores instead of one with a frequency 20 % lower provides 1.73 times more performance for only 1.02 times more energy. The second limit is the speed of light: data have to travel from memory to the CPU and cannot go faster than the speed of light. While this limit may seem high, we have already reached it. Indeed, if we want to build a sequential machine able to process 1 TByte of data per second, due to this limit it would require to stick 1 TByte of data on an area of 0.3 mm^2 which mean that 1 bit occupies only 0.1 nm, the size of a small atom.

While these parallel processors are theoretically more powerful than sequential ones, it is way more complex to use them efficiently and it is the responsibility of the developer to do so. Moreover since a few years we are reaching yet another physical limitation. We are capable of reducing the size of transistors, hence increasing the number of transistors in a chip. Still, more transistors means more heat, and there is a maximum of heat that an area can produce before unexpected effects such as leakage occurs. As a result, vendors are now building machines with several sockets, each one embedding several cores.

At the same time, processors became significantly faster than memory, thus, CPUs embed small caches memory to limit the impact of this gap on performance. Theses caches are designed to benefit from two patterns that occurs in most programs: spacial and temporal locality, which respectively means using data close in memory and using several times the same data in a short time lapse. One of the things that makes the caches faster than the main memory is their size, the smaller they are, the faster it is to access them. Therefore CPUs embed several level of caches (usually three), the first level is very fast and small few kilo bytes and designed for very close accesses (loop on an array), while the last level is bigger and slower, about 10 MBytes and designed for more distant accesses. As the CPUs embed several cores, theses caches are organized hierarchically, the last level is shared by all cores while each core has a private access to a level one cache. This hierarchy helps isolating private data and benefit from well structured sharing, moreover it reduces the required bandwidth in the caches. For similar reasons, computers with several sockets have a Non-Uniform Memory Access (NUMA) which means that each socket has a privileged access to a subpart of the memory. Consequently, the memory access patterns of an application can significantly impact its performance [Drepper, 2007]. Indeed, four threads working on small and separate piece of data will benefit from their private caches while patterns such as all to all sharing will result in a lot of conflicts in the caches. Moreover, if some sharing occurs between threads that are close in this hierarchy, the shared caches will contain shared data and one thread will benefit from the accesses of the other. At the opposite, if theses threads are far in the hierarchy, the sharing will generate some noise and maybe

some contention on the memory bus.

In the end, writing an efficient program requires to consider the architecture of the computer that will run it, and the patterns, and their matching, even if the program is sequential. Although there are some general rules: privileging sequential accesses, working on small set of data, this task is extremely complex, even for High Performance Computing (HPC) specialists, as every accesses matters. Thus, performance analysis tools are extremely helpful to understand and optimize the performance of any application.

The first step to optimize the performance of an application is to find the hotspots, which means the parts of code that are inefficient and understand their nature. Only at this point it is possible to decide what part of code should be improved and how. There are many tools designed to analyze the performance of an application [Pillet et al., 1995, Browne et al., 2000, Shende and Malony, 2006, Treibig et al., 2010, Adhianto et al., 2010] most of them rely on performance counters to collect a trace of the application. These are CPU register dedicated to performance analysis which enable efficient collection of performance data.

In this thesis we ran a case study on the performance analysis of a physical simulation framework: Simulation Open Framework Architecture (Sofa). To analyze the performance of this application, we used "Like I Knew What I am Doing" (Likwid) [Treibig et al., 2010], a classical performance analysis tool and traced several metrics concerning the memory usage. With this tool we are able to detect memory related performance issues and guess the nature of some of them, but it was impossible to spot their location on the memory and the patterns responsible for the bad performance. Indeed, if these counters can be very useful, they consider the memory as a monolithic entity which is not the case on recent architecture. Thus, specific tools should be used for analyzing performance in view of memory.

Analyzing an application performance in view of the memory raises two challenges: the first one is the collection of the trace itself. This is a complex task as there is no hardware comparable to the performance counters for tracing memory accesses. Furthermore, every instructions of a program triggers at least one memory access, thus, collecting every single memory accesses of an application is not possible. Additionally, due to the lack of hardware tracing, a memory collection tool might easily become invasive and significantly change the behavior of the analyzed application. The second challenge is the presentation of the trace. Indeed, memory traces are extremely complex as they are spread over five dimensions: time, address space, CPU location, threads and access type. Furthermore, some of these dimensions are not trivial to represent, for instance the address space can be virtual or physical and the CPU location is organized hierarchically. In the end, memory analysis tools have to extract pertinent data and present them in an understandable way.

An ideal memory analysis tool should be able to present the memory access patterns of a program to its developer, including information about data sharing between threads and the location of the access on the machine architecture. Furthermore, such tool should highlight inefficient patterns.

Several tools were designed for memory performance analysis [Lachaize et al., 2012, Liu and Mellor-Crummey, 2014, Giménez et al., 2014], however most of them addresses the trace collection challenge by doing an instruction sampling. Instruction sampling is a hardware based technique that enable tracing some

instructions at a defined frequency [Drongowski, 2007, Levinthal, 2009]. While this method enables efficient tracing, it does not trace the whole memory space addressed. As a result, it is impossible to visualize memory patterns from the collected trace.

1.1 Contributions

In this thesis we propose two tools to analyze the memory behavior of an application. Our first tool, called Tool for Analyzing the Behavior of Applications Running on NUMA ArChitecture (Tabarnac), collects global memory traces without temporal information and presents an overview of the sharing patterns inside the data structures, between the threads of the execution. The second one, called Memory Organisation Cartography & Analysis (Moca), collects generic, fine grained memory traces with temporal information. We propose two approaches to visualize Moca traces, the first one is based on FrameSoc an existing generic trace analysis framework, while the second one relies on a programmatic exploration using R.

Conducting experiments in computer science can be extremely simple, but doing it in a reproducible way requires more planning and methodology. Performance analysis, whether it is for optimizing an application or to evaluate a tool requires to do complete experimental campaigns. In this thesis, we take a particular attention at making our experiments as reproducible as possible. To do so, we clearly describe our experimental methodology and distribute the files required to reproduce each step of the presented experiments.

1.1.1 Global overview of the memory sharing patterns

We designed Tabarnac to analyze the memory sharing patterns of applications running on NUMA machines. This tool relies on an existing, lightweight binary instrumentation, based on the Intel Pin library, which counts how much each thread of an application accesses each page. We added to this library the capacity to retrieve contextual information to associate memory addresses to data structures (static and allocated). Moreover we designed several comprehensive visualizations of the collected traces. Using these visualizations we were able to identify some performance issues and improve the performance of the NAS Parallel Benchmarks (NPB), `IS` by 20 %.

These results were published in an article at Visual Performance Analysis (VPA) 2015 a Super Computing workshop [Beniamine et al., 2015b]. Furthermore Tabarnac is distributed as a free software under the General Public License (GPL): https://github.com/dbeniamine/Tabarnac. This work is the result of a collaboration with M. Diener and P.O.A Navaux from the Parallel and Distributed Processing Group (GPPD) of the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, financed by CAMPUS France.

1.1.2 Fine grain memory traces collection

Moca is our main contribution. This tool relies on a Linux kernel module to collect efficiently fine grain memory traces. This kernel module intercepts

page faults, which are triggered by the hardware and handled by the Operating System (OS), to trace memory accesses. As these page faults does not occur frequently, it also injects periodically false page faults. It handles memory traces in the kernel space and flush them to userspace periodically. Moreover we incorporated our data structures tracking library in Moca without the dependency to Pin. Additionally we ran an extensive experimental comparison of Moca comparing it to Tabarnac and two state of the art memory analysis tools in terms of overhead, trace precision and completeness.

This work is the subject of two Inria research reports [Beniamine et al., 2015a, Beniamine and Huard, 2016] and has been submitted at Cluster, Cloud and Grid Computing (CCGRID) 2017. As the previous tool, Moca is distributed under the GPL license: https://github.com/dbeniamine/Moca.

1.1.3 Fine grain memory traces analysis

We proposed two different approaches to visualize Moca traces. The first one is based on an existing general trace management and analysis framework called FrameSoc [Pagano and Marangozova-Martin, 2014]. More precisely, it relies on Ocelotl [Dosimont et al., 2014] a FrameSoc tool that aggregates similar parts of the trace and present a simplified overview highlighting anomalies. With this tool we were able to identify classical inefficient memory patterns on a test application. Nevertheless we encountered several scalability issues due to the generic representation of the trace inside the tool.

To overpass theses scalability issues, we also explored several Moca traces with a programmatic approach using R. This approach enables using advanced filtering and zooms, and to design specific visualization for each traces. With this method we were able to explore more complex traces and detect some interesting and unknown memory patterns. For reproducibility, these analysis are saved and versioned in a labbook publicly available at github:
https://github.com/dbeniamine/Moca_visualization.

1.2 Thesis organization

The remaining of this thesis is organized as follow: in Chapter 2 we present a case study on the performance analysis of Sofa, a physical simulation tool. This chapter first introduces Sofa, its specificities and previous attempts to optimize it, and highlight the need for performance analysis on Sofa. We then discuss the existing generic performance analysis tools and our experimental methodology. This case study emphasize the need for specific memory performance analysis tools. In Chapter 3, we introduce some specificities of recent memory architectures, usual memory performance issues and workarounds. Then we discuss the existing memory performance analysis tools, their limitations and what we would expect from an ideal tool. After that we present Tabarnac, our first contribution, in Chapter 4. We discuss its design and usage, evaluate its overhead and finally present some performance optimization done with the knowledge obtained thanks to Tabarnac. In Chapter 5, we describe our main contribution, Moca. We first explain in details the mechanisms used by Moca, its internal design and how it handles the challenges raised by fine grain memory trace collection. Then we provide an extensive experimental evaluation comparing Moca

to two state of the art memory analysis tools and Tabarnac. Chapter 6 discusses the visualization of Moca traces. We first introduce FrameSoc and Ocelotl and then discuss the results obtained with these tools. Then we propose a programmatic approach and present the visualizations and results obtained with it. Finally we draw our conclusions and present some perspectives of future work in Chapter 7

Chapter II

Case Study

Contents

2.1 Motivations **12**
2.1.1 SOFA: a physical simulation framework **12**
2.1.2 Previous efforts toward SOFA parallelization **14**
2.2 Profiling tools **15**
2.3 Experimental methodology **16**
2.3.1 Reproducible research **16**
2.3.2 Experimental workflow **18**
2.3.3 Methodology **20**
2.4 SOFA Analysis **25**
2.4.1 Experimental plan **25**
2.4.2 Results and discussion **26**

Optimizing a computational kernel is a complex task, which requires a deep understanding of both the algorithm mechanics and the machine that will execute it. Simple computational kernels such as matrix multiplication or Cholesky factorization have been subject of years of optimizations, yet some scientists still manage to improve them. The task is even more complex when it comes to optimizing a whole actual application. We first need to identify hotspots which means understand where and why the performance are suboptimal. Then we have to understand their nature and localization both in terms of code and data structure (if they are memory related). Finally, when optimizing a real life application, it is important to make the code modifications as clear as possible. Indeed not all developers are specialized in High Performance Computing (HPC) and a code is only maintainable if understandable.

Simulation Open Framework Architecture (Sofa) [Allard et al., 2007] is a simulation framework designed for exact and interactive physical simulation, it aims at assisting surgeons with real time medical simulation. Hence it cannot afford to improve its efficiency by relying on approximations. Therefore optimizing Sofa performance is crucial. Yet, most of the developers are not from the HPC community. To guide them into this optimization process, we need to analyze the performance of Sofa and identify precisely the hotspots.

In this chapter, we present a case study about the performance optimization of Sofa. This case study aims at demonstrating the usage of classical analysis tools and emphasize their limits concerning memory related performance issues. It is organized as follow: first we present Sofa, its specificities and previous attempt to parallelize or optimize it in Section 2.1. Then, we discuss the existing profiling tools that can be used to analyze the performance of application in Section 2.2. After that we detail our experimental methodology and discuss reproducibility matters in Section 2.3. Finally we present our analysis and first conclusions in Section 2.4.

2.1 Motivations

Several efforts were made to parallelize the different part of Sofa, using its specificities. Before discussing these efforts, we need to present more precisely the Sofa framework.

2.1.1 SOFA: a physical simulation framework

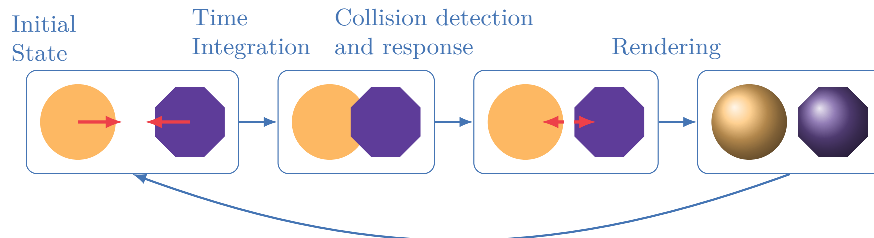


Figure 2.1 – The simulation loop.

In Sofa, simulation can be seen as a loop depicted in Figure 2.1: we start from an initial configuration were a set of objects are subject to a force field. The second step, called time integration, solves a system of equation to compute the next position of each object. At that point, some objects might be overlapping. Thus the third step consist in detecting these overlaps and applying repulsing forces to simulate the collision. Finally the result of these step is displayed (rendered) and we are back the beginning. The time integration and the collisions detection are the most costly steps. Hence many algorithms were developed to compute them efficiently, each algorithm being more appropriate for simulating one type of object depending on its form and stiffness.

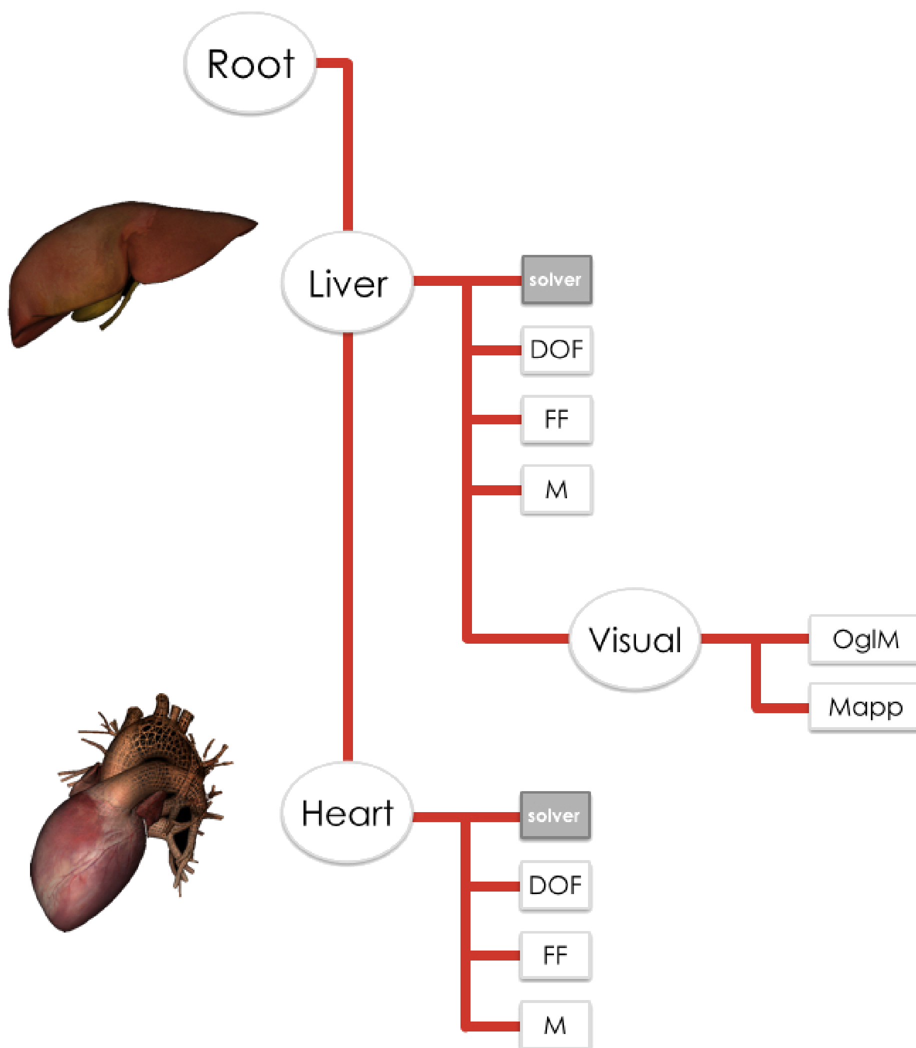


Figure 2.2 – SOFA representation of a scene with two objects: a liver and a heart. Each node of the scene can embed its own set of solvers and visual representations.
Image from SOFA documentation [SOFA, 2016].

One of Sofa main specificity is that it has a multi-model representation of each component. A simulation scene is represented as a tree, as shown in Figure 2.2 where each physical object is a node. Each level of the graph can embed solvers, collision detector and visual representation, overriding the defaults. This hierarchical representation enable dependencies management between objects and the representation of complex embedded objects [Nesme et al., 2009, Faure et al., 2011].

2.1.2 Previous efforts toward SOFA parallelization

It is important to note that the main developers of Sofa are mostly computer scientists with a physical or medical background but not specialized in HPC. Several efforts were made by external developers to parallelize Sofa, most of these efforts consist in optimizing some algorithms which, according to Sofa developers, are time consuming. For instance, Everton Hermann proposed an efficient (sequential) collision detection algorithm based on ray tracing and a parallelization of this algorithm [Hermann et al., 2008]. More recently, Julio Toss has been developing several algorithms on Graphical Processing Units (GPUs) to improve the computation time of Voronoi diagrams [Toss and Comba, 2013, Toss et al., 2014]. Although their computation generates a considerable overhead before the simulation, these diagrams enable efficient simulation of forces propagation in heterogeneous materials [Faure et al., 2011]. Hence optimizing their computation is critical for Sofa.

E. Hermann also proposed a more global approach, exploiting the hierarchy of the scene tree, to parallelize the time integration step [Hermann et al., 2009]. This parallelization relies on the KAAPI runtime [Gautier et al., 2007] which consider an application as a set of tasks and dependencies between these tasks. Each task can provide one or several implementations (Central Processing Unit (CPU), GPU ...), the runtime choose online which implementation to use depending on the current performance, which results in portable performance. With this method, the amount of parallelism depends on the number of objects simulated. As most Sofa scenes only include a few objects, this parallelization is not suitable for them.

While this approach is more generic it was never actually used by Sofa developers for several reasons. First of all, they are not specialized in parallelism, hence not used to write programs in parallel languages. Second, KAAPI is a research runtime that evolve quickly, maintaining code based on it seemed to costly for them. Last but not least, while this approach helps to parallelize the code, it does not help finding hotspots and optimizing existing code. In the end, Sofa is currently parallelized using simple Open Multi-Processing (OpenMP) `#pragma`. These `#pragma` are compiler directives to tells the runtime that a region of code should be run in parallel. While adding such annotation to an existing code is trivial, it requires to spend some time at improving data structures and algorithms to obtain an efficient parallelization. This method impacts small chunks of code and miss the global aspect of the previous one. Thus it is considerably less efficient than the KAAPI version.

To conclude, it appears that optimizing algorithms or chunk of codes pointed out by Sofa developers is not sufficient. Indeed this approach can miss unknown hotspots and therefore opportunities for optimizations. Moreover, while the global, runtime based approach seems potentially more efficient than local opti-

mizations, it does not overpass this limitation. At the end of the day, it appears that identifying precisely unknown hotspots and digging into Sofa performance would be more profitable than writing pieces of highly optimized code.

2.2 Profiling tools

Performance analysis consists of two steps: data collection and presentation. The first step aims at extracting as much pertinent information as possible from an execution. Nevertheless, observing an execution is not free: it takes time to count and record events. As a result, it can impact the application performance or, worst, modify its behavior. Consequently, any analysis provides a trade-off between the amount of data collected and the impact on the monitored application. The second step is also challenging as the analysis tool has to to find out which data are pertinent and to present them in a meaningful way to the user. Many tools were designed to address one or both of these challenges.

Performance counters are dedicated CPU registers that were originally designed by vendors to debug their processor prototypes. They count events such as cache misses, or branch miss-predicts at a very low cost compared to software based solutions. They can directly be accessed using the Perf driver which is part of the Linux kernel since version 2.6.31. Yet, as a result of their initial aim, the available counters depends of the CPU model and vendor. Furthermore, it requires a deep knowledge of CPUs mechanisms to understand the meaning of some counters. Hence, higher level libraries such as Performance API (PAPI) [Browne et al., 2000, Malony et al., 2011, Weaver et al., 2013] and "Like I Knew What I am Doing" (Likwid) [Treibig et al., 2010] were designed to make performance counter access and interpretation more convenient. These libraries provide performance groups and automatically compute comprehensive metrics. In addition they provide markers that can be used in the code in order to collect counters only during some parts of the execution. This is useful once hotspots are identified, but can lead to miss a part of the execution if used too early.

Another approach to make performance counters more understandable consists in combining them with contextual informations. Such informations can be obtained by intercepting libraries calls (system calls, C standard library, Message Passing Interface (MPI), OpenMP ...). The easiest way to intercept a library call is by overriding it at runtime with the `LD_PRELOAD` environment variable[1]. A second method is to rely on binary instrumentation libraries such as Intel Pin [Luk et al., 2005] or Dyninst from the Paradyn Project [Miller et al., 1995]. This method is more flexible and usually enable higher level data collection, but it is more intrusive, thus it can impact the behavior of the studied application. Simulators such as SimGrid [Casanova et al., 2014] can be used to overpass these limitations. However HPC simulators often focus on explicit communications (via MPI or OpenMP) discarding the actual computations. Hence they might miss memory related issues. Several tools such as HPCToolkit [Adhianto et al., 2010], PARAllel Visualization end Events Representation (PARAVER) [Pillet et al., 1995], Tunning and Analysis Utilities (TAU) [Shende and Malony, 2006], Modular Assembler Quality Analyzer

[1] One can use the `LD_PRELOAD` environment variable to tell the linker to load a library before running a program. As a result, each call to a function from an external library overwritten in the preloaded library will be intercepted.

and Optimizer (MAQAO) [Djoudi et al., 2005], AMD CodeXL [AMD, 2016] (the successor of AMD CodeAnalyst [Drongowski, 2008]) and Intel VTune [Reinders, 2005] combine several of these methods to collect traces.

When it comes to presenting performance traces in a readable way, we can split these tools in three groups. The first ones only provides textual traces and let the user extract pertinent information from them. It includes the Perf driver, Likwid and PAPI library as well as several Pintools. Such tools are very useful for small applications as they do not require complex tools to be read. Moreover they are usually easy to parse and one can build more complex visualization on top of it using R, for instance. Tools from the second group, that includes VTune, CodeXL tries to present data in a more readable way. Usually their visualization consist in a set of tables and plots, where they highlight values that seem to be pertinent (for instance cache miss above a fixed threshold) pointing important parts to the user. Finally, while tools like MAQAO, HPCToolkit and PARAVER propose similar visualizations, but they also provide Application Programming Interfaces (APIs) to design new visualizations or import external traces. FrameSoc [Pagano et al., 2013, Pagano and Marangozova-Martin, 2014] is very similar to the previous tools, the main difference is that it is designed for trace management and analysis. Consequently it does not provide any way to collect traces but is able to import traces collected by the user. It describes them with a generic representation and enable easy navigation through different visualizations of the same trace.

To conclude, many tools were developed to monitor the performance of an application. The best tool depends on the kind of issues we are looking for and the application that is monitored. Most of the tools discussed here are based on performance counters and thus present data from the point of view of the CPU. In our specific case, we are studying a complex application, yet we are in contact with the developers of Sofa. Consequently they can give us hints about the important parts and the kind of issues we should look for. Therefore low level tools such as Likwid are well suited as they can both compute pertinent metrics and focus on specific parts of the application.

2.3 Experimental methodology

In computer science we can easily monitor our experiments and restart them quickly if something goes wrong. At the opposite, in other domains, such as biology, this reactivity is not possible, scientist are forced to write very precise protocols to avoid loosing large amounts of time and money. By inspiring ourselves from their protocols we could make our experiments reproducible and our research more trustable.

In this section we first introduce reproducibility and how people have tried to reach it in HPC, then, we present the methodology we have developed during this thesis to make our experiments as reproducible as possible.

2.3.1 Reproducible research

Measurement bias, which means attributing a consequence to the wrong cause due to an issue in our measurement and analyze method, is a widely known phenomena in scientific communities and is analyzed in most fields. Mytkowicz

et al. [Mytkowicz et al., 2009] highlighted several ways to introduce significant measurement bias in computer science experiments without noticing it. Its experiments showed that measurement bias is both commonplace and unpredictable in our field. Therefore, the easiest way to deal with this bias is to reproduce studies published by other teams in order to confirm or invalidate their results. Still, reproducing experiments in computer science, and more specifically in HPC, is not trivial.

A previous study [Collberg et al., 2015] tried to evaluate how reproducible the experiment presented in computer science article are. To do so, they only focused on the capacity to compile the experimental code and evaluated 601 articles published in "top ACM conferences and journal". From these 601 articles they were only able to build the environment of 217 articles. Moreover it took more than half an hour to build the experimental code of 64 of these papers and 23 others required the intervention of the authors.

At this point we need to define precisely reproducibility, for the remaining of this thesis, we will use the definition proposed by Dror G. Feitelson [Feitelson, 2015]:

> **Repeatability** concerns the exact repetition of an experiment, using the same experimental apparatus, and under the same conditions.
>
> **Reproducibility** is the reproduction of the gist of an experiment: implementing the same general idea, in a similar setting, with newly created appropriate experimental apparatus.

Repeating an experiment in the sense of Feitelson in HPC is nearly impossible. Indeed repeating an experiment first requires the access to the machine that executed the original one, with the exact same software stack and all the scripts to run it. Yet, several unpredictable factors can impact the repeatability: between the two experiments, some hardware (for instance a disc) could have been replaced by one faster. While we can log the whole hardware configuration during the experiment, some other factors, such as the room temperature, can impact the performance and are nearly impossible to measure during the experiment and impossible to reproduce. Still there is a gap between the definitions of repeatability and reproducibility. Indeed if we have access to a machine, with the exact same software stack and comparable hardware, as well as the experimental scripts, we can repeat the experiment in **similar** conditions. This definition of *similar* repeatability is stronger than reproducibility as we do not re-implement the experiment but re-run it on a similar settings. Nevertheless we cannot expect the exact same results. If the experiments compares raw execution times, changing the machine may significantly changes the results, although if we compare relative time (speedup or slowdown) we are more likely to lessen the discrepancy. Finally, adaptive algorithms may be a limit to similar repetition. Indeed, small hardware differences might be enough to trigger a change on the executed code of an adaptive algorithm.

Several tools can help us making experiments more repeatable. For instance, by running our experiment on a shared platform such as grid5000 [Cappello et al., 2005], we can argue that other people have access to the same set of machines. Moreover on these machines, it extremely easy to make a deployable image of our environment. Using an image provides controls on the installed

library, but it is impossible to know or change the version a library without deploying it. Kameleon [Ruiz et al., 2015] overpass this limit by describing an environment as a recipe. It also make the distribution of the environment easier as we only need to distribute the recipe which is a lightweight piece of code instead of an archive containing a whole Operating System (OS).

To reproduce an experiment it is important to understand how it has been designed and how it has evolved from the first version to the results presented in the paper. Stanisic et al. [Stanisic, 2015, Chapter 4, p31-44] described an experimental workflow based on Git and Org-mode to keep track of these evolutions and make easy for anyone to understand it. One of the main drawback of this workflow is that it is not suitable for experiment generating huge ($\geq$500 Mib) trace files as Git is not designed to handle such files.

Many tools were designed to conduct experiments in computer science however they are not designed for HPC and using them in our context would require some adjustments. A comprehensive survey can be found in [Stanisic, 2015, Chapter 3, p17-19].

2.3.2 Experimental workflow

We design experiments to analyze the behavior of an application, compare it to other applications or to test it under specific circumstances. The goal of the experiment is to answer a question or confirm an hypothesis. Answering scientifically a question requires to define:

- The environment: the circumstances under which we do the measure. In computer science that includes the OS, libraries and any user configurations.
- The reference cases to which we will compare our results. These are usually state of the art existing applications comparable to the one we evaluate.
- The inputs given to the tested applications.
- The parameters used for each application.
- The metrics: a set of quantifiable and measurable indicators used to evaluate our results.
- The expected results: what behavior do we expect, what should be considered as abnormal.

Designing an experiment consists in translating high level questions into an experimental plan that answers them scientifically.

Complete experimental plan: We consider an experiment as a three steps workflow depicted in Figure 2.3 determined by a *Complete experimental plan*. We define the *Complete experimental plan* as the smallest set of scripts and documentation required to repeat an experiment. It includes the main script that actually run the experiment with all its dependencies. These dependencies consists in the tested applications (Git version, modifications) along with their inputs or the benchmarks used for testing them and the environment on which it is run. The complete experimental plan also contains the description of the

experimental machine(s) and the command(s) or script(s) used to deploy the environment on them and start the main script. Finally all the scripts used for parsing and analyzing the experimental results are part of the complete experimental plan as they are required to repeat it. Moreover, designing the analysis at the same time as the experiment help reduce some bias. Indeed, preparing data presentation before obtaining the actual data forces us to express our expectations. In the end, if the results do not match theses expectations we are more likely to notice it.

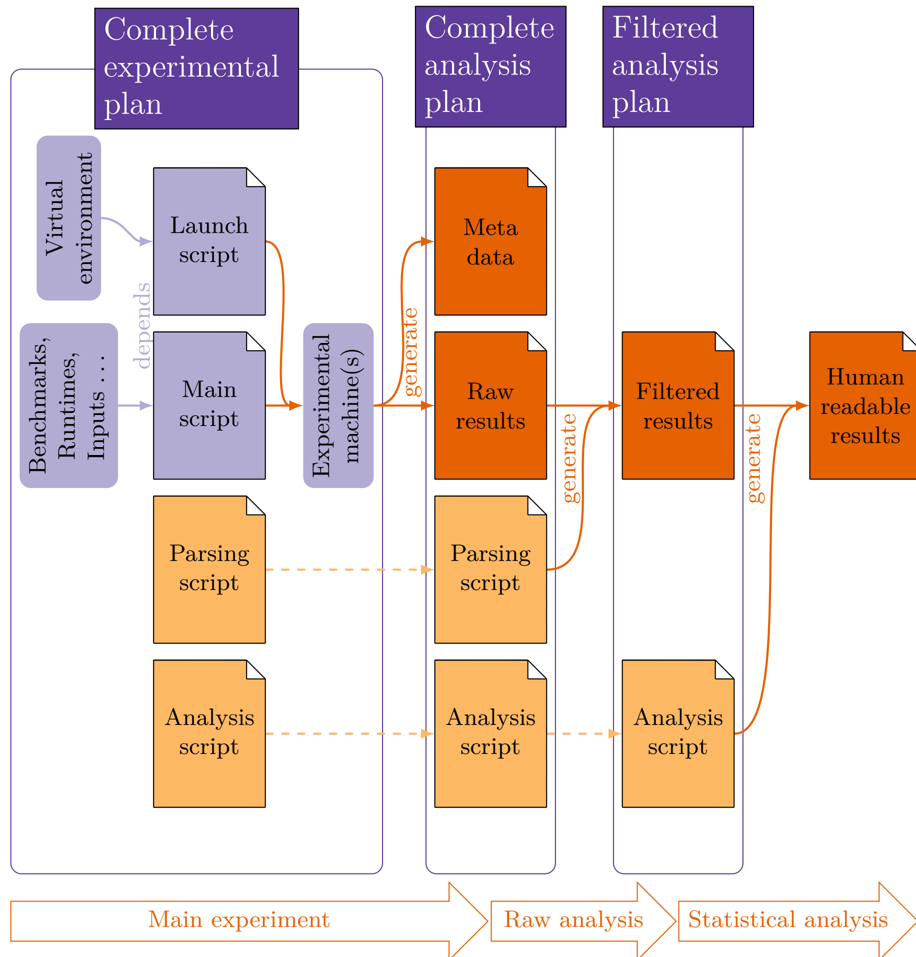


Figure 2.3 – Experimental workflow.

Main experiment: The first step of the experimental workflow consist in execution the main experiment on the experimental machines. This step produces two types of results: the raw results which are the actual output of the experiment and the meta data. These meta data include all pertinent informations on how the data were produced (information about the environment, commands executed, Git version of every applications) and how to interpret them.

Raw analysis: We call *Raw Analysis* the second step that extract the raw values needed to compute the metrics defined in the complete experimental plan from the raw results. This step only aims at reducing the amount of data to analyze (from 100 Gio to few Mib in some of our experiments). The result of the raw analysis is usually one or two Comma-Separated Values (CSV) files that can be easily read by any statistical tool.

Statistical Analysis: Finally comes the *Statistical Analysis* which first reads the filtered results and computes statistics such as means, standard error, slowdown and speedup etc. The second aim of this step is to present a comprehensive visualization of these statistics.

While the complete experimental plan is required to repeat the experiment, some people may want to reproduce only the statistical analysis to change it and inspect the results from another point of view. Furthermore they might want to extract other values from the raw results or get more information about the experimental environment from the meta data. To enable such partial reproduction we can distribute the *Complete analysis plan* and the *filtered analysis plan* that each includes all the files generate by the previous step and all the files required to redo the next step. Finally while repeating the whole experiment requires access to the experimental machines, the two last steps are not machine dependent, thus, are easily repeatable.

2.3.3 Methodology

Implementing such an experimental plan is not trivial, we describe here how we design, implement and distribute our plans.

Construction of an experimental plan

In HPC, an experiment usually consists in evaluating the performance or the correctness of an application or of a tool that uses an application as an input (scheduler, simulator, analysis tool . . .). For both we need to find a set of *benchmarks* to conduct our experiments, which means an input representative of the case we want to test. A benchmark can be either the input of a computational kernel or an actual application used by the tool we evaluate. Additionally, if the aim of the experiment is to evaluate the performance of a code that we have developed and there are some existing comparable applications, we need to test it against these applications. Yet, a set of benchmarks might not be a sufficient input as HPC applications are often highly configurable. Consequently, we must determine for each application which parameters are the most efficient for the evaluated metrics and the defined benchmarks. In the end, we evaluate each program under at least two set of parameters: the default and a tuned version.

Once the benchmarks are chosen, we need to decide on one machine or a set of machines on which we will run the experiment. This step is crucial as computers architectures are getting more and more complex and applications are (usually) designed for one type of machines. Furthermore some tools (such as Pin, Precise Event Based Sampling (PEBS) [Levinthal, 2009], Instruction Based Sampling (IBS) [Drongowski, 2007]) are either vendor specific or optimized for

some architecture. Therefore, we often have to repeat the same experiment on several different machines to conduct a fair comparison.

At this point, we need to find some relevant metrics to answer the questions that we are asking. These metrics should remain simple as they will be interpreted by humans. Still, they must also cover every aspect that we are studying. A complete set of simple measurable metrics is often easier to understand that one complex metric providing an overall score in an obscure unit.

An experimental environment should be both *minimalist* and *sufficient*. Indeed if the environment is not sufficient we will have to install packages or library before the experiment. Hence the installed version will depend on the date at which the experiment is run, making it almost impossible to repeat. At the opposite if we include more libraries than we need or worst several version of the same library when it is not required, finding the version actually used during the original experiment might require to pay an extra attention to the whole building pipeline.

Finally it is crucial to write both parsing and analysis scripts before running the experiment. To do so, we can generate a filler set of (fake) results representing our expectations. From this set we can design some data visualizations. Furthermore using this set we can complete the plots with some text describing these expected results. Such information will prevent us from trying to explain a posteriori results that infirm our hypothesis.

Automation and documentation

It is crucial that a all the steps of the experiment from the deployment of the environment to the final analysis are properly scripted in a language that can be understood by other developers. Any manual step can make the experiment impossible to repeat and an obscure code might make it difficult to modify the experiment without breaking it.

Moreover, the traces generated by an experiment must be *self explanatory* or at some point they will only be a (large) set of meaningless Bytes on a hard drive. We consider that a trace is **self explanatory** if and only if we can easily answer the following questions from the raw trace:

- How the trace was generated, what is the exact command that launched it ?
- What software were used (including their versions and possible modifications) ?
- What were the hypotheses and expectations ?
- When was it executed ?
- On which machines (description, name and physical location) ?
- How are the trace files organized (file hierarchy) ?
- What scripts are used to do the analysis ?

To do so, all our experimental scripts starts the same way: they first create a new directory that will hold the traces. Then they copy themselves with all the scripts which are inside their directory to this new directory. After that

they duplicate their output to a file in this new directory and log every sensitive meta data as show in Listing 2.1. During the experiment, each command is echoed before executing it. Regarding the data analysis we use R as it provides a large set of reliable statistic analysis libraries. Additionally, thanks to R-markdown, we can produce a standalone structured output that contains the original questions, our assumptions, the results and plots, our observations and comments. Finally, before distributing an experimental trace, we write a small `Readme` that explain the file hierarchy and the experiment design (although these informations could be recovered from the meta data and by reading the experiment code).

Listing 2.1 Logging experimental informations.

```
1  exec > >(tee $OUTPUT) 2>&1
2
3  echo "Expe started at $START_TIME"
4  echo "#### Cmd line args : ###"
5  echo "$CMDLINE"
6  echo "EXP_NAME $EXP_NAME"
7  echo "OUTPUT $OUTPUT"
8  echo "NUMBER OF RUNS $RUN"
9  echo "########################"
10 echo "#### Hostname: #########"
11 hostname
12 echo "#### Kernel:   #########"
13 uname -a
14 echo "#### Path:     #########"
15 echo "$PATH"
16 echo "########################"
17 echo "##### git log: #########"
18 git log | head
19 echo "########################"
20 echo "#### git diff: #########"
21 git diff
22 echo "########################"
23 lstopo --of txt
24 cat /proc/cpuinfo
25 echo "########################"
26
27 # Copying scripts
28 cp -v $0 $EXP_DIR/
29 cp -v ./*.sh $EXP_DIR/
30 cp -v *.pl $EXP_DIR/
31 cp -v *.rmd  $EXP_DIR/
32 cp -v Makefile  $EXP_DIR/
```

To evaluate the variability and reduce the effect of external noise, we run each configuration several times. The more variability there is, the more runs we need to execute. These runs are all independent therefore we must avoid to make them artificially dependent. The Algorithm 2.2 shows a very simple experiment where the runs are launched by the process that actually do the experiment. As we do not create a new process for every run, some cache effects

might appear after the first the run improving the performance of subsequent runs. Thus, these runs are artificially dependent. An easy way to avoid this bias is to start a new process for each run, as shown in Algorithm 2.3. However, this is not enough to protect our experiment from system noise Indeed if at some point of the experiment a system process (such as `logrotate`) interfere with our experiment, the performance of a set of runs will drop. As the runs are executed in order, it might be correlated with the size, and therefore we will not realize that it is due to external noise, and we will consider it as a consequence of the size. While, if we randomize the runs, the performance drop will affect several runs (but not all) for different sizes. As a result we will observe abnormal results for the impacted sizes and conclude that something might have interfered. To do so, our experimental scripts generate a list of runs that should be executed, shuffle it and then execute them in this randomized order.

Algorithm 2.2 Dependent runs.

```
MatAdd:
    function DO_RUN(size)
        for i in 1..size do
            for j in 1..size do
                C[i][j] = A[i][j]+B[i][j]
            end for
        end for
    end function

    function MAIN
        for size in 1..N do
            for run in 1..R do
                DO_RUN(Param)
            end for
        end for
    end function
```

Algorithm 2.3 Independent runs.

```
MatAdd:
    function MAIN(size)
        for i in 1..size do
            for j in 1..size do
                C[i][j] = A[i][j]+B[i][j]
            end for
        end for
    end function

Experiment:
    function MAIN
        for size in 1..N do
            for run in 1..R do
                EXEC(MatAdd size)
            end for
        end for
    end function
```

Each run consists of three step: a pre-command that can set specific environment values, the actual command that execute the benchmark and the post-command that can save data, pre-process it, compute metrics must restore the normal state. Pre-processing data during the experiment can reduce the raw trace, and interfere with the raw analysis, but it is sometimes unavoidable, for instance when some evaluated tools generate binary traces that can only be interpreted inside the experimental environment. The first and last steps are not mandatory, however any environment change must be restored on the last step. Each of the commands executed by these 3 steps are logged before execution as shown in Listing 2.4.

Distribution

The experimental plan is mostly composed of source code, thus it can easily be distributed using a Version Control System (VCS). As it often depends on

Listing 2.4 Execution of a run.

```
1  do_run()
2  {
3      run=$1
4      conf=$2
5      benchname=$3
6
7      benchname=$(basename $bench)
8      echo "$benchname"
9
10     LOGDIR="$EXP_DIR/$benchname/run-$run"
11         echo $LOGDIR
12     mkdir -p $LOGDIR
13
14     set -x
15     # Prepare run
16     if [ ! -z "${PRE_ACTIONS[$conf]}" ]
17     then
18         bash -c "${PRE_ACTIONS[$conf]}"
19     fi
20
21     # Do experiment
22     cmd="${TARGETS[$conf]} $bench"
23     $cmd > $LOGDIR/$conf.log 2> $LOGDIR/$conf.err
24
25     # Restore everything
26     if [ ! -z "${POST_ACTIONS[$conf]}" ]
27     then
28         bash -c "${POST_ACTIONS[$conf]}"
29     fi
30     set +x
31 }
```

external programs, it is preferable to use VCS which can manage dependencies and patches such as Git or mercurial. We distribute our experimental plans as Git repositories, were each dependency is tracked as a submodule. This repository includes an initialization script that retrieves all dependencies and apply the required patch. If the virtual environment is written as a Kameleon recipe, it is trivial to distribute it inside the Git repository. Yet in this case, we have to build the image from the recipe during the initialization. Often the environment is just an archive, in that case, we make it available on a http server, and the initialization script take charge of downloading it. Finally, we add a Readme describing how to run the experiment, on which machines and what variables should be changed (if any). Ideally this Readme should be generated automatically.

Distributing the raw analysis is more difficult as some trace files can be quite heavy and code repositories are usually not designed to handle large files. Large file hosting services such as Renater FileSender [Renater, 2011] or Zenodo [Cern and OpenAire, 2013] are well suited to upload these files. Moreover they gener-

ate Digital Object Identifier (DOI) that can be used both in the article and in the experimental Git to link to the trace.

The statistical analysis could be distributed via code repository or with large file hosting service as it does not include large files. Still, large file hosting seems more suitable to us as the filtered traces are not code but experimental results.

2.4 SOFA Analysis

In this section, we present first the experiment designed to produce a performance analysis of Sofa, then we discuss about the obtained result and the possibility of improvements.

2.4.1 Experimental plan

The current version of Sofa is mostly sequential with a simple OpenMP parallelization in some places. Yet, this parallelization is known to be often inefficient. The Sofa developers gave us four simulation scenes specifically designed to test the most popular parts of Sofa, and hints about functions that are known as hotspots.

We decided to use Likwid to analyze Sofa performance as it is quite flexible allowing both global (wrapper mode) and local (marker mode) analysis and provides several useful metrics and performance groups out of the box.

More specifically, we focused on the following metrics:

- *Idle time ratio*: The CPU is idle when it is waiting for a transfer to memory, disc or network Input / Outputs (I/Os). Idle time can often be reduced either by overlapping it with computation or by improving memory (or I/Os) access patterns. This metric gives the ratio between the time spent idle and the total execution time. It is computed from two likwid metrics: `Runtime (RDTSC)` (total time) and `Runtime unhalted` (total time not idle): $idleTime = 100 * (1 - \frac{RuntimeUnhalted}{RuntimeRDTSC})$.
- *Simulation time*: The time spent inside Sofa simulation loop, this is only useful to compare parallel executions to sequential ones.
- *Runtime ratio*: For each function, the ratio of the time spent in this function and the total time spent in the simulation loop. This metric highlights functions that are actually hotspots.
- `L2`, `L3` and `MEM` predefined groups from Likwid. These groups mainly compute the data volume exchanged and the bandwidth at their level (respectively second level of cache, third level of cache and main memory). These two metrics and more precisely their evolution between sequential and parallel executions tell us how data is shared between the threads and if it is shared efficiently or not. These metrics are computed by counting the misses at one level lower (hence the absence of `L1` group).

As Sofa is mainly used on personal machines and not HPC ones, we ran our performance evaluation on a recent desktop machine called *Naskapi*. The hardware and software specifications of this machine are described in Table 2.1.

CPU	Vendor	Model	Core	Threads	Frequency
	Intel	Xeon E5-1607	4	4	3.00 Ghz
Cache and Memory	L1	L2		L3	Memory
	Private			Shared	
	32 Kib	256 Kib		10 Mib	16 Gib
Software	kernel	Distribution		Bios configurations	
	Linux 3.2.0-4	Debian Wheezy		No hyperthreading	

Table 2.1 – Hardware and software configuration of Naskapi.

The experimental methodology described in Section 2.3 evolved during this Ph.D thesis, hence the experiments presented here do not fit it perfectly. Mostly the runs were independent but not randomized, and the experimental logs were not as complete as they should. Furthermore, as *Naskapi* is a desktop machine and not a node from a cluster, we did not deploy an image of the system. Hence, repeating these experiment in the exact same condition do not seem possible. Nevertheless, we distribute the experimental plan and the trace, which should make the experiment easy to reproduce on github:
https://github.com/dbeniamine/Sofa_expe.

2.4.2 Results and discussion

To present the results of our evaluation, we will focus on two Sofa scenes that illustrate the kind of results we obtained. The first scene is called *linearQuadraticFrame*, for this scene the OpenMP parallelization provides a speedup of 1.37 with four threads. This speedup seems rather low, yet, as the parallelization is only partial we need to determine if we can actually reach a higher speedup. However the second scene *linearRigidAndAffineFrame* has a speedup of 0.87 which means that the parallel version is slower than the sequential one. These difference of performance probably means that theses scenes are not executing the same core. Our first goal is to understand why some part of Sofa are more efficient with the OpenMP parallelization while some other are slowed down by it.

Before digging into the differences between these scenes, there are two noticeable similarities between them to discuss. First the idle time ratio reported by Likwid showed that, for the sequential version of the code, during almost half of the simulation the processor is idle, which means that it spend a lot of time waiting for I/Os or memory accesses. For Sofa, all the I/Os occurs at the beginning of the execution, before the simulation loop, therefore, all the idle time captured in the simulation loop is due to memory accesses. In parallel executions, this idle time increases. This behavior is expected as the OpenMP based parallelization only affect some small parts of the code, hence as soon as the code is not in a parallel loop, 3 threads out of 4 are idle. Still, it means that there is room for improvement even for the sequential version. The second noticeable similarity is that the functions pointed by the developers only represent around 30 % of the simulation time, which means that there are some unknown

hotspots left.

From here, we know that there might be a memory issue (as the processor spend a lot of time idle). We compared the bandwidth and data volume going into each cache level and the main memory for the two scenes and for the functions indicated by the developers. Those functions are generic and thus their actual code varies depending on the parameters of the simulation. Therefore, while the compared functions are similar in terms of role in the simulation, the manipulated data as well as the executed code is different for the two scenes.

Figure 2.4 shows the bandwidth (left side) and the data volume (right side) for the two functions (*linearQuadraticFrame* above, *linearRigidAndAffineFrame* below) between `L2` cache to the main memory. Each point represents the mean of 60 runs and the error bars represents the standard error. To make the plot more readable, we have normalized each value by dividing it by the value for the corresponding sequential run. Thus these plots show the evolution of the bandwidth and data volume when we use the OpenMP version of Sofa.

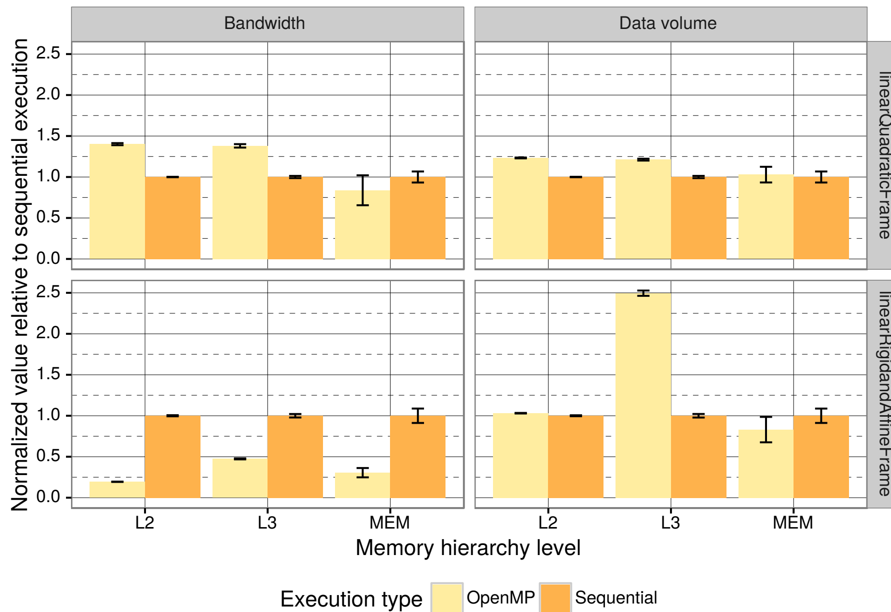


Figure 2.4 – Differences of bandwidth and data volume, from L2 cache to main memory, between sequential and parallel execution of Sofa on two different simulation scenes.

We can see that for the first scene, both the bandwidth and data volume increases by a factor of 1.5 in the cache and stays comparable in the main memory when we use the parallel version. Yet, in the parallel version, the variability of the memory bandwidth increase significantly and we are not able to explain this behavior with our traces. We can see the same behavior for the second scene and the memory data volume. Still, the overall behavior shows that there are more useful data in the cache in parallel. If this was due to each thread working on a smaller set of independent data, the data volume in

memory would decrease in parallel. As it is not the case, these useful data in the cache are due to an efficient data sharing between the threads. At the opposite, for the second scene, the bandwidth drops down at each level while the data volume stays still except at the `L3` level where it increases by a factor 2.5. It is important to note that the `L3` cache is the only shared cache of this machine. It means that several threads are writing the same data, invalidating each other private cache and requiring the coherency protocol to interfere. This behavior is called false sharing and is a well known performance issue that have been introduced by Torellas et al. [Torrellas et al., 1994].

At this point we can say that false sharing occurs in this precise function, yet we do not know on which data structure. We could dig onto the code and try to understand the memory access pattern of each data structure to see where the false sharing does occurs. By looking at Sofa's code, we can see that the main difference between the two version of the function is that for `linearRigidAndAffineFrame` there is one more loop in the computations. However, this code manipulate several data through many indirections. Consequently, determining on which data the false sharing is happening could be extremely cumbersome. Furthermore, this approach is not generic at all. Once we have done it for one particular function, we would have to redo the same analysis and optimization for each potential hotspot.

As we have seen with the runtime ratio, this particular false sharing issue only represent a small part of the execution. Moreover, our analysis highlighted some variability in the memory bandwidth and data volume in parallel that we are not able to explain. Therefore, a more generic approach would be welcome.

From that point it seems clear that Sofa is having memory performance issues. Yet, we need a generic tools to analyze its performance from the memory point of view. Such tool should be able to establish a complete diagnosis of the memory usage. To do so, this tool need to identify memory access patterns and differentiate the efficient ones from the others. Additionally, to correct an inefficient pattern we need to know precisely why it is inefficient. Thus, this tool should classify them in terms of performance characteristics. Finally, it needs to localize these patterns which means determine where it occurs in term of data structure, which thread(s) are responsible for the accesses, to what lines of code and at what time it correspond. To sum up, this ideal tool will trace every memory access of an application and give to the user a list of inefficient patterns. For each pattern it will specify very precisely what kind of issue is occurring, where it happens and how we could fix it.

Chapter III

Memory Performance Analysis

Contents

3.1 Architectural considerations **32**
3.1.1 Caches **32**
3.1.2 Memory hierarchy **37**
3.2 Existing tools **39**
3.2.1 Memory traces collection **40**
3.2.2 Memory traces analysis **41**
3.3 Conclusions **43**

The results of our case study (Chapter 2) showed that traditional performance analysis tools can help identify memory related performance issues. Yet they are not able to tell precisely where, in terms of data structures, the issue occurs. Thus it is still required to analyze the code manually. As memory is often a performance bottleneck, several tools where developed to analyze performance in regards of the memory. However, none of these tools is currently able to present the memory access patterns of an application.

This chapter discuss memory analysis tools, first we present the specificities of recent memory subsystems and the usual mistakes that can generate performance drop in Section 3.1. Then we present existing memory performance analysis tools, how they relate to usual memory issues and discuss their limitation in Section 3.2. Finally we describe what would be in idealistic memory performance analysis tool Section 3.3.

3.1 Architectural considerations

During a few decades, processor frequency increased significantly more than memory frequencies, resulting in a considerable gap between these two resources. Nevertheless most applications use spatially close data (spatial locality) and temporally close data (temporal locality). To benefit from these localities, Central Processing Units (CPUs) embed some caches reducing the number of accesses actually generating a fetch from the main memory.

3.1.1 Caches

There are two phenomena called *spatial locality* and *temporal locality* that can be observed in most programs and that are the main principles used for cache optimization. Spatial locality is the fact that most programs do not usually access the memory in a completely random way, they usually access data that are near to each other in a small time lapse. Similarly, temporal locality is the fact that a memory address is often used several times during a small interval of time. Consequently the main role of caches is to keep data that have already been accessed close to the CPU when they are reused to reduce the number of actual memory accesses.

These caches are organized hierarchically, thus, the time required to access a piece of data depends on the cache level in which it is present. One access to the fastest cache usually cost about 4 CPU cycles while retrieving one piece of data from the main memory cost around 180 CPU cycles [Levinthal, 2009]. As these caches are small, each time a data is added in a cache it replaces (evict) an existing one. Several mechanisms where designed to guess which data should be in the cache and which one should be evicted. As a result, a developer seeking for performance must consider the architecture of these caches and the way the work to benefit from them.

Cache lines and alignment

To benefit from spatial locality, the memory is divided on lines (usually 64 Bytes), and every time an address must be fetched from the main memory, the whole line is copied to the cache. Therefore accessing 8 successive doubles (usually

one double equals 8 Bytes) requires only one memory transaction if they all are in the same line of cache. As a result, aligning data structure to cache line can improve the memory access time.

For instance if we consider accessing an array of two lines of caches as illustrated in Figure 3.1. If the data structure is aligned to cache lines, two accesses will be required to retrieve the whole data structure, and no unused data will be introduced in the cache. At the opposite, if it is not correctly aligned, not only one more memory fetch will be required but one useless line will be introduced inside the cache. As caches are designed to keep only a limited volume of data, inserting one line inside a cache means evicting another one, thus, introducing unused data must be avoided as much as possible.

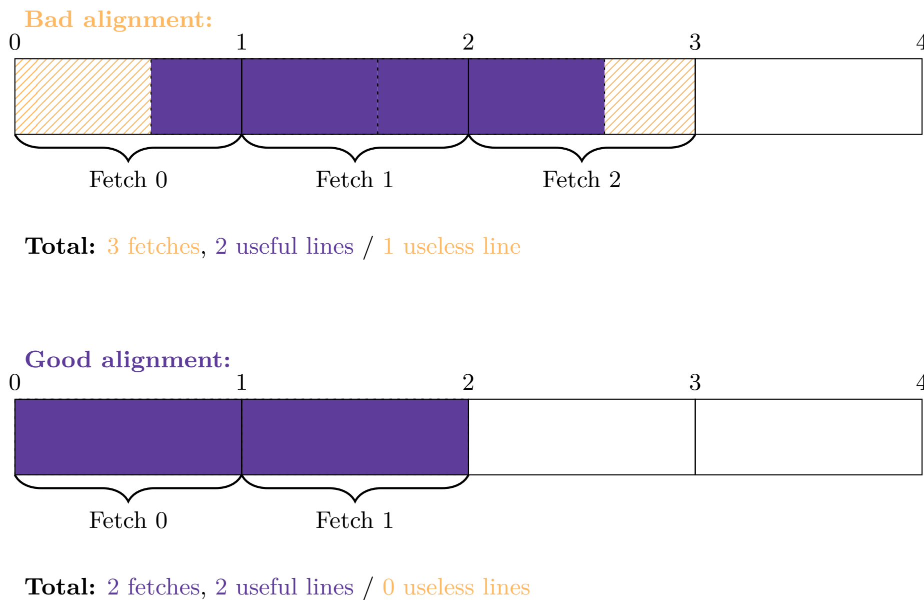


Figure 3.1 – Retrieving two lines of cache with one or two fetches depending on the alignment of the lines.

Cache management policies

When an address is accessed, the CPU must search for the corresponding line in the cache. If it is not present, the CPU must fetch it from the memory and decide where to copy it into the cache, which means choosing one line to evict. To make the cache efficient, these decisions must be done quickly.

There are three distinct ways to decide where a line should be placed in the cache. The simplest way to do so is called *direct mapping* and consists in associating each memory line to one specific line of cache in a round robin way: the line number l can only be in the cache line $l \mod L$ where L is the total number of available lines in the cache. With this policy it is only required to look at one line of cache to check whether a line of memory is present or not in the cache. Furthermore, it is not required to decide which line should be

evicted as their is only one possible line. Still, this policy is inefficient with some memory patterns, for instance when a program accesses data regularly spaced but misaligned in the memory. Indeed, for such pattern, only a small subset of the cache will be used. At the opposite, *fully associative caches* allows any memory line to be mapped anywhere in the cache. As a result we always exploit the maximum size of the cache, but it requires to look at the whole cache to find if a line is present, and deciding which line should be evicted. Usually caches are *N-way associative* which is a compromise between those two policies. A N-way associative cache is divided in N sets and the memory line number l could be in any line of the set number $l \mod N$.

This associativity can be used to create (virtual) partitions in the cache and then allocate data structures in these partition giving more cache to the ones that are more reused and thus will benefit from it [Perarnau et al., 2011]. However, this technique is not possible anymore with recent Intel processors as they change dynamically the associative sets.

Choosing the best line to evict is impossible as it would require to predict the future steps of the execution. A realisable and often efficient policy consists in evicting the Least Recently used (LRU) line that could be replaced by the one fetched. Nevertheless, implementing an actual LRU policy is costly as it requires to timestamp every line inside the cache. Consequently most caches use pseudo-LRU heuristics.

A naive example

The naive matrix multiplication is a good illustration of how a simple program can benefit from the caches or not. Figure 3.2 illustrate a naive, sequential matrix multiplication algorithm that computes $C = A * B$. For the matrices A and C, this algorithm loops over a whole row before going to the next one, while it goes through B by columns first. To understand why this pattern matters, we have to consider the memory representation of these matrices. As the memory address space has only one dimension, a matrix is a contiguous block of memory. Usually it is stored row major, which means that $A[i][j]$ is actually $A[i*N+j]$ where N is the size of a row. This means that, when we loop through a matrix by rows first, we scan linearly the address space, while if we do it by columns first, we jump N elements between two accesses. In this example we use $N = 8192$, the first access to a row of B will trigger a cache miss, and the whole cache line will be fetched. Before we access the second element of this row, we will have to fetch one line of cache for each row of the matrix which means $8192 * 64 = 512$Kb which is a little more than the size that the L2 cache can accommodate. Therefore each access to B will trigger at least a L2 cache miss resulting in a lots of traffic in the L3 cache. With huge matrices, it may not even fit in the L3 cache resulting in contention on the memory bus. The simplest way to fix this issue (although it is not the optimal algorithm for the matrix multiplication) consists in swapping the two inner loop of the algorithm, as shown by the dotted arrows. Indeed the order of the operations does not change the result of the multiplication (all computations are independent in this example) this swap only changes the order of the accesses concerning the matrix B.

Recent caches also embed a prefecther that tries to detect memory accesses patterns to retrieve several lines of cache at the same time from the main mem-

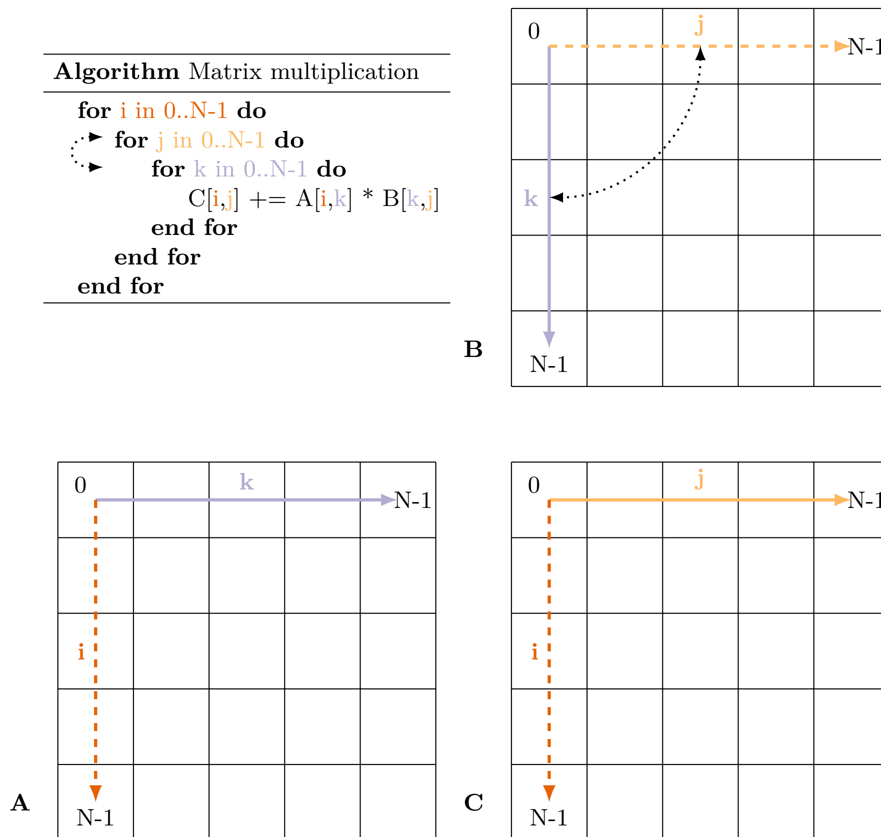


Figure 3.2 – Example of non linear memory accesses: the naive matrix multiplication.

ory. This mechanism is particularly efficient with linear or regularly spaced accesses. Yet, for sparse and random accesses it might prefetch unused lines evicting potentially useful ones. Consequently, this mechanism amplify the impact of the accesses regularity.

Memory caches and parallelism

Several reasons have led vendors to make their CPUs more and more parallel. For instance, increasing by 20 % the frequency of a CPU requires much more energy than using two cores at the original frequency and results in less computational power. Furthermore, increasing the frequency of a processor increases also the amount of heat produced, and there is a physical limitation on the maximum heat that an area can produce before current leakage happens. As a result, modern CPUs embed several cores.

Using one huge private cache per core would be costly and space consuming, at the opposite if we use only one cache shared by each thread, they will interfere with each other all the time. Therefore, the caches are organized hierarchically. The highest level of cache is shared by each core while the lowest is private, the intermediate levels are usually shared by a subset of the cores. Figure 3.3

depicts a quad core machine with hyperthreading enabled (two threads per physical core). This machine embed three levels of cache, the first one is private for each core, while the second one is shared by half the cores and the last one is shared by every cores.

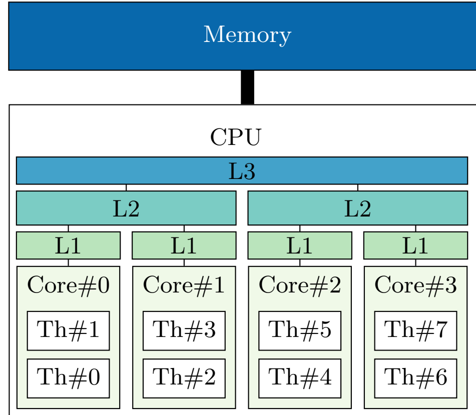


Figure 3.3 – Topology of a quad core parallel machine.

A consequence of that hierarchy is that the physical location of thread sharing data can have a considerable impact on the performance. Indeed if two threads on the same core read the same line, the first one will copy it to the L1 cache and the second one will access it extremely fast. If they are executed inside different cores, the sharing will be efficient at the lowest common level (L2 or L3). At the opposite, if two threads write data in the same cache line, a conflict occurs and must be solved by the cache coherency protocol. The coherency is done at the lowest shared level of cache and requires to lock the memory subsystem. Therefore conflicts are extremely costly, and must be avoided when possible, or at least must occurs between threads as close as possible to avoid locking the L3 cache.

For instance, we consider a simple example where two threads are working on a small array of 8 doubles. As a double is usually coded on 8 Bytes, this array is exactly the size of one cache line. Each thread is doing independent computations on a half of the array as illustrated in Figure 3.4. The first access will copy the whole array from the memory to all the caches levels of the thread that triggers it. When the second thread reads the array, it will copy it from the lowest shared cache to its private cache. If the threads were only reading it, no more memory access would be required. Yet, in our example, each thread updates the value of each entry of its array after its computation. Each time a thread writes a value of the array, it invalidates the whole line. Therefore the coherency protocol must interfere at the lowest shared level, while the two threads are not actually sharing any data. Hence the name false sharing. Not only the accesses to this array are inefficient but they generates lots of useless data traffic in the memory bus which can create some contention slowing down the whole application. The easiest way to fix such issues, consists in padding the data structure, which mean introducing zeroes between each element of the data structure so that each thread works on a different cache line.

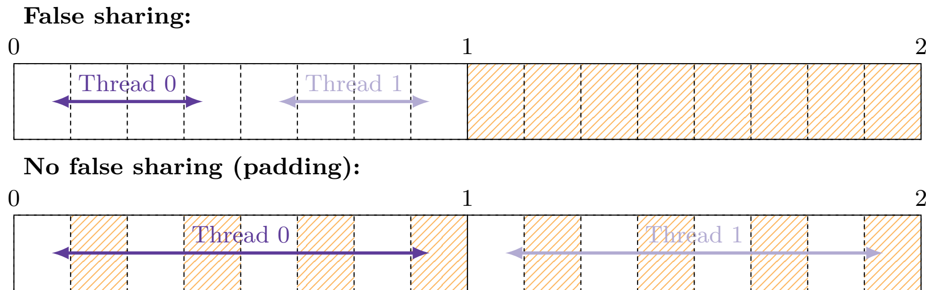


Figure 3.4 – Two threads writing 4 consecutive doubles on the same line of cache, without any actual sharing, resulting in false sharing and an easy fix by padding the data structure.

3.1.2 Memory hierarchy

Increasing the parallelism inside a chip means fitting more transistors in a limited space. Therefore, it requires to reduce the size of the transistors and find a way to dissipate the heat produced by them. Consequently it is not possible to increase indefinitely this parallelism. As a results, modern computers often embed several CPUs sockets to overpass this limitation. A machine with several sockets can either give them a uniform access to the memory by sharing the memory bus or split the memory into banks and giving non uniform access to the sockets, such machine is called Non-Uniform Memory Access (NUMA). While the first option seems simpler to use, it means that the bandwidth is shared by all the threads, therefore contention can easily appear. At the opposite, the second option provides a maximal bandwidth for each socket. Figure 3.5 depicts a NUMA machine, we can see that each socket has a privileged access to one memory bank. Furthermore, the sockets are linked via a interconnect with a ring topology. As a result, each socket has a direct access to its memory bank, a slower one the banks of its neighbors and an even slower access to the last bank. Writing code that uses efficiently this specific architecture remains the burden of the developer who therefore needs to explicitly consider the physical location of its data. Table 3.1 provides approximate accesses latencies depending on the memory hierarchy level for the Intel I7 Xeon 5500 Series.

To use NUMA machines efficiently, we need to understand how the Operating System (OS) handles the memory. From the OS point of view, the memory is split into contiguous chunks called pages, usually one page corresponds to 4 Kb (some High Performance Computing (HPC) applications uses huge pages of 4 Gb). Each userspace program works on virtual pages, which means that, when it accesses an address, the OS must first translate it to find the actual physical address of the page. This pagination is used to provide the abstraction of virtual memory and the memory isolation. Linux is a lazy OS thus it will never map a page until a program has written something to it. Indeed if a program reads the contents of a new page, Linux can simply return a zero. To do so, one page full of zeroes is always present in the memory and any virtual page points to this specific page until a program write or explicitly touch it. This means that the physical location of a piece of data is determined the first time that a program write something on the virtual page containing this

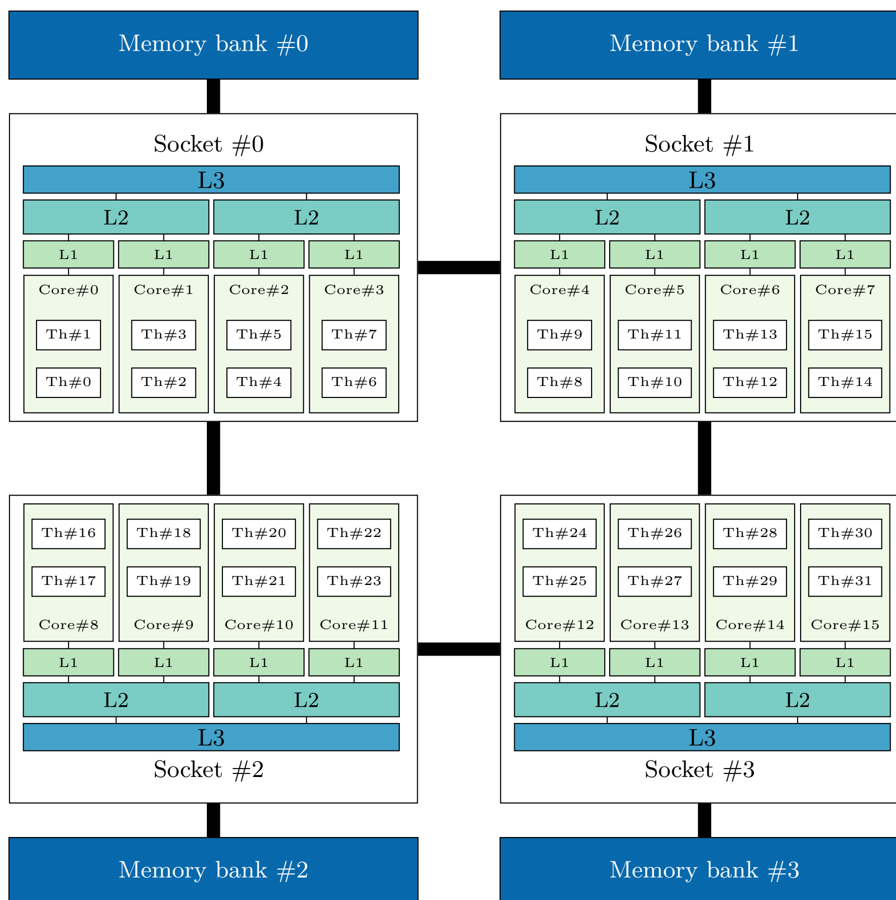


Figure 3.5 – Topology of a 32 cores NUMA machine.

data. At this point, the OS needs to decide where it is going to map the page. One of the most classic and simplest policy, used by most OSes is the first touch policy [Marchetti et al., 1995]. This policy maps a page to the memory bank closest to the socket where the thread responsible for the first access is executing. As a result, the thread(s) initializing a data structure will determine its mapping. A classic performance issue with NUMA machines comes from the initialization of all the data structures using only one thread. When doing so, all pages are mapped on the same memory bank, and each further access from another socket will be remote and, thus, slow.

Kleen et al. developed an interface to map the pages on NUMA machines using more advanced policies than first touch [Kleen et al., 2005]. This Application Programming Interface (API) can be accessed either via the `numactl` command or via a library called `libnuma`. The `numactl` command is useful to apply a global policy on all the page of the application. It is often used to apply the *interleave* policy that distribute the pages over the NUMA nodes in a round robin way. While it does not reduce the overall number of remote accesses, it distribute them among the nodes and therefore reduces the contention when there are more than two sockets. At the opposite the `libnuma` provides fine

Data source	**Latency (cycles)**	**Latency (ns)**
L1 Cache hit	~**4**	~1.6
L2 Cache hit	~**10**	~4
L3 Cache hit, unshared	~**40**	~16
L3 Cache hit, shared in other core	~**65**	~26
L3 Cache hit, modified in other core	~**75**	~30
Remote L3 Cache	~**100-300**	~40-120
Local Memory	~180	~**60**
Remote Memory	~250	~**100**

Table 3.1 – Order of magnitude of access latency depending on the memory hierarchy level. Values in bold extracted from Levinthal report on performance analysis for Intel I7 Xeon 5500 Series [Levinthal, 2009]. Conversion cycles to latency computed for a CPU frequency of 2.5GHz.

grain pages and threads mapping. The user can use it to explicitly allocate data structures and bind threads to nodes.

Still, finding the optimal mapping for one machine is not trivial and, mapping threads and data structures in an adaptive way is even harder. Therefore, several tools were developed to automatically map threads and pages online [Diener et al., 2014, Corbet, 2012]. Such tools count remote accesses for each page, and move them when they are more accessed remotely than locally. While adaptive tools often improve the performance, they cannot reach the same performance as an application optimized while considering memory issues. Indeed, these tools, by conception, require a time to detect inefficient patterns and adapt and consequently lose opportunities of optimizations.

An easy way to overpass this issue for small computational kernels consists in running a loop of computation on the data before initializing them. Indeed, by doing so, each page will be mapped as close as possible to the first thread that will actually use it. Nevertheless, this technique is only efficient for kernel that repeat several iterations of the same computations and is not suitable for more complex applications

To conclude, using efficiently the memory is challenging. Indeed, due to the organization of the memory in pages and cache lines, we must consider where and how our data structures are allocated. Furthermore the hierarchical organization of the memory and the cache must be correlated with the thread placement and data sharing. In summary, the developer must consider the machine architecture and the memory access patterns over the address space, time and threads. Therefore visualizing the memory access patterns of an application is a great help to optimize it.

3.2 Existing tools

Presenting memory access patterns to the user raises two challenges. The first one is to collect efficiently a detailed and precise trace without interfering with the normal execution. Collecting such traces is challenging as each instruction of a program triggers at least one memory access. Once this trace is collected,

presenting it in a meaningful way to the user is also a challenge. Indeed such traces are spread over at least five dimensions: memory address space (physical and virtual), time, threads, cores and type of accesses. Furthermore they are potentially huge, and identifying relevant information is complex.

3.2.1 Memory traces collection

To help the developer solve memory related issues, an ideal tool should provide enough data to build a map of the memory accesses locations over the time, and identify memory accesses patterns such as false sharing. Therefore the trace it collects must have the following properties:

- **Detailed:** a memory trace is *detailed* if it includes information about time, space (at which address the event occurs), location (on which CPU it occurs) and nature of access (is this a read, a write, by which thread).
- **Precise:** to be *precise* enough, a trace should include a sufficiently large number of events in order to enable identification of memory accesses patterns.
- **Complete:** We say that a trace is *complete*, at a given granularity, if and only if the events it contains covers the whole address space at this granularity.

Several studies propose to analyze memory by looking solely at the information collected through PAPI and Likwid libraries [Majo and Gross, 2013, Jiang et al., 2014, Bosch et al., 2000, Weyers et al., 2014, Tao et al., 2001, DeRose, 2001]. Generic tools have been designed on top of hardware performance counters to analyze and improve parallel applications performance, such as Intel's VTune [Reinders, 2005], Performance Counter Monitor (PCM) [Wilhalm et al., 2012], the HPCToolkit [Adhianto et al., 2010], and AMD's CodeAnalyst [Drongowski, 2008]. Although performance counters provide information about the memory use (bandwith, volume of data transferred . . .), they consider the memory as one huge entity and do not differentiate distinct addresses or at least distinct pages. Thus, these methods lacks of precision as they are not able to locate issues in the memory and determine in which data structure they happen.

Tracing all the memory accesses without information loss is nearly impossible as almost each instruction can trigger a memory access in addition to its fetch. Nevertheless, several methods can record a *detailed* memory trace with a good *precision*. Budanur et al. [Budanur et al., 2011] use an instrumentation based tool to collect all the memory accesses. They loose *precision* by doing online compression and merging accesses into a higher level model, but this is necessary to reduce both the trace size and its overhead. Still, on a small matrix multiplication (size $48 * 48$ with four OpenMP threads) they already slow the execution down by a factor 50. Another method consists in using hardware sampling tools such as AMD's IBS [Drongowski, 2007] or Intel PEBS [Levinthal, 2009] to trace a subset of the memory accesses. This method is used by many several tools, including the memory trace module of HPCToolkit [Liu and Mellor-Crummey, 2014], Memphis [McCurdy and Vetter, 2010], MemProf [Lachaize et al., 2012], and Mitos [Giménez et al., 2014]. This method provides *incomplete* sampling: some parts of the memory can be accessed without being noticed by the tool if

none of the associated instructions are part of the sampled instructions. Thus, it is possible that they ignore memory areas less frequently accessed, but in which optimization could take place. Applications sensitive to spurious performance degradation, such as interactive applications, could be hindered by these unnoticed accesses, despite their low frequency. Furthermore, to be able to detect patterns such as false sharing, theses sampling mechanisms should be able to collect several samples every 10 cycles which means around 100 ns.

These sampling mechanisms monitor events set given by an instruction type. They can monitor several events sets at the same time but the number of monitored sets is limited by the hardware capabilities (number of available registers). Unfortunately, the number of existing events sets that relate to the memory hierarchy is large, because of its complexity. This makes difficult the task of tracing all the relevant memory accesses with just a single analysis. One way to lessen the impact of this limitation is to run several times the instrumentation and use advanced methods such as folding [Servat Gelabert, 2015] to generate a more accurate summary trace. Nevertheless, this makes the instrumentation cost grow accordingly. Moreover, writing (and sometimes) using tools that relies on hardware mechanisms requires a deep knowledge of the processor. As processors evolve, such tools are hard to maintain and can quickly become outdated. We regard all these limitations as too constraining for a general purpose memory analysis tool.

Finally other studies rely on hardware modifications either actually implemented or simulated [Bao et al., 2008, Martonosi et al., 1992]. Although they are eventually able to collect more *precise* traces efficiently, these techniques are limited to hardware developers. Indeed, to use these hardware extensions one has either to obtain (or build) a prototype or to use a suitable simulator. Such configuration is not realistic for general purpose memory analysis.

To conclude, existing memory trace collection tools are not able to collect traces *precise* and *detailed* enough to present memory patterns to the end user. Nevertheless, two tools: MemProf [Lachaize et al., 2012] and Mitos [Giménez et al., 2014] collects *incomplete* and *detailed* traces. While this is not enough for the kind of analysis we want to run, these tools provides an interesting comparison point. Both of them collects trace using event sampling (PEBS or IBS, respectively). Therefore their trace are *precise* on the parts of the memory that are accessed the most.

3.2.2 Memory traces analysis

Some memory oriented analysis tools such as Memphis [McCurdy and Vetter, 2010] and Memspy [Martonosi et al., 1992] only provide a textual output. MemProf [Lachaize et al., 2012] also provide a command line interface to inspect the trace. Although these tools highlight relevant informations, it is hard to get an overview of the memory behavior from such output. The developer might be presented with a huge amount of information and, thus, unable to differentiate normal behaviors from problematic ones nor identify memory patterns.

Several studies [DeRose, 2001, DeRose et al., 2002, Bosch et al., 2000] rely on generic performance traces and present them in a *data centric* way, in the sense that they correlate the metric values with source code and data structures. These studies are based on performance counters and present derived metrics such as cache misses or memory bandwidth. Weyers et al. [Weyers et al., 2014]

have a slightly different approach: they present the same kind of data but correlate them with the NUMA architecture instead of the data structures. All these studies can help localizing the issue in the code and find the data structures involved in it. Furthermore, they provide comprehensive visualization that are easier to understand than plain text traces. However, they are not able to present the memory patterns, and the developer still has to figure out by itself the nature of the issue.

A part of this limit is due to the fact that the previously cited tools work on generic traces instead of memory traces, hence they do not have the information required to identify access patterns. Liu et al [Liu and Mellor-Crummey, 2013, Liu and Mellor-Crummey, 2014] proposed an HPCToolkit extension that let the user visualize the number of accesses done by each thread to a data structure. This visualization gives already more information about data sharing, but the granularity is quite high. Consequently, we cannot identify patterns inside the data structure nor pattern change over time. Tao et al [Tao et al., 2001] proposed a more fine grain visualization, showing for each page the number of remote and local accesses. Yet, compared to the previous study they loose the notion of sharing.

Finally, MemAxes [Giménez et al., 2014], which is the visualization tool for Mitos, provide a unified view of the trace. This view, presented in Figure 3.6, correlates the information collected in the samples (bottom pane) with the architecture (middle pane) and the source code and data structures (left pane). It enables to do some selection on any part of the visualization to focus on some code or a NUMA node etc. Furthermore, Husain et al. [Husain et al., 2015] have recently added a layer to MemAxes that enable to trace simulations and link the simulation visualization to MemAxes. Still, this visualization does not show sharing pattern or access patterns, it only helps identifying the lines of code and data structures responsible for the bad performance (hotspots) As a result, the user still has to correlate the samples information to understand the nature of the issue and how to fix it.

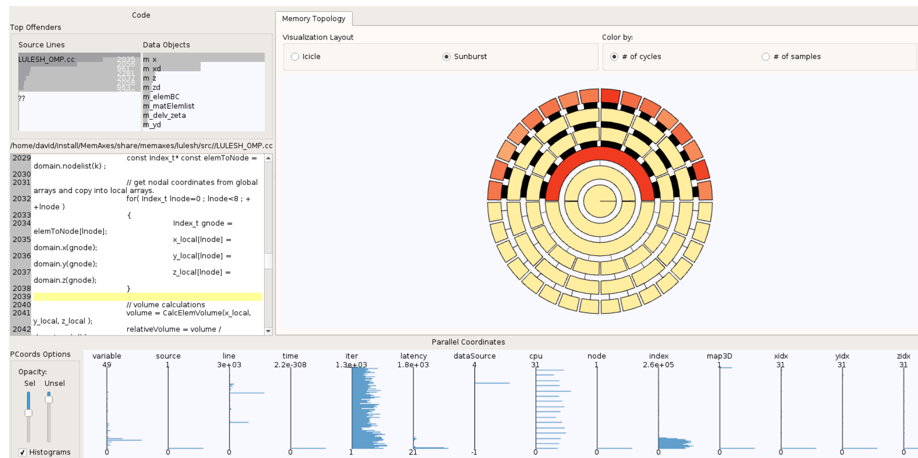


Figure 3.6 – Screenshot from MemAxes on the example data trace provided with the tool.

3.3 Conclusions

The memory subsystems have became more and more complex over the last few decades. As a result, the way a program allocates and access its memory has a significant impact on the performance. Eventually, a developer looking for performance must consider the memory access patterns of its application. Therefore a tool able to collect a memory trace and to display accesses and sharing patterns would be useful for performance optimization.

Most existing memory trace collection tools consider the memory as a monolithic entity and only provide global information such as the bandwidth. Some tools provide more detailed memory traces. Nevertheless, they rely either on hardware based sampling in which case the resulting trace only shows a small subset of the memory, or on hardware modifications and are thus not usable by real life developers.

When it comes to visualizing these traces, many techniques were developed to identify precisely in the code and data structures where performance are suboptimal. Yet, most of the existing tools are not able to show memory patterns of any kind. A few advanced tools picture the number of accesses per data structure and per thread or the number of remote accesses per page. Still this is not sufficient to understand precisely sharing patterns or memory access patterns.

To conclude we need both to collect precise memory traces and to present them to the user in a comprehensive way that enable identification of sharing and access patterns.

Chapter IV

Collecting and Analyzing Global Memory Traces

Contents

4.1 Design **46**
4.1.1 Trace collection **46**
4.1.2 Ease of use and portability **47**
4.1.3 Visualization **48**
4.2 Experimental validation **50**
4.2.1 Methodology **50**
4.2.2 Ondes3D **51**
4.2.3 The `IS` benchmark **52**
4.2.4 Tracing overhead **56**
4.3 Results and discussion **57**

This chapter presents Tool for Analyzing the Behavior of Applications Running on NUMA ArChitecture (Tabarnac), our first attempt to collect memory traces and use them for performance optimizations. Previous work [Beniamine, 2013] showed how difficult it is to capture *complete* memory traces with temporal information. Therefore Tabarnac focuses on the global memory behavior. Furthermore it aims specifically at improving Non-Uniform Memory Access (NUMA) related performance issues. Tabarnac, relies on a low overhead and lock free instrumentation library to provide a global overview of the memory usage. Moreover, it also provides simple yet meaningful visualizations.

The contribution presented in this chapter were published at Visual Performance Analysis (VPA) 2015 a Super Computing workshop [Beniamine et al., 2015b]. Furthermore Tabarnac is distributed as a free software under the General Public License (GPL) license: https://github.com/dbeniamine/Tabarnac. This work is the result of a collaboration with M. Diener and P.O.A Navaux from the Parallel and Distributed Processing Group (GPPD) of the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil.

This chapter first presents the design and usage of Tabarnac in Section 4.1. Then Section 4.2 present an experimental validation of Tabarnac including performance optimization of well known benchmarks and overhead analysis. Finally we discuss the results obtained with Tabarnac in Section 4.3.

4.1 Design

On NUMA machines, optimizing memory performance often means reducing the number of remote accesses. Indeed, to optimize the performance a memory page must be mapped to a physical memory bank as close as possible to the threads that actually uses it. Therefore to optimize memory performance on NUMA system, it is required to know how much each page is accessed by each thread. Tabarnac is designed to collect specifically this information. It gives also hints about imbalance in the accesses to data structures.

4.1.1 Trace collection

Tabarnac is based on Numalyze [Diener et al., 2015] instrumentation which rely on the Pin [Luk et al., 2005] library. This instrumentation is lock free by design: it traps on each memory accesses but only maintain one counter per pages and per threads. Adaptive NUMA mapping tools collect similar traces but uses them online to decide whether they should migrate a memory page from a node to another. Numalyze was originally designed to estimate the efficiency of such tools. Indeed, adaptive tools work with partial traces as they need to decide on page migration before the end of the execution. Thus, comparing the mapping obtained with a partial trace to the one obtained with the complete trace helps finding out the minimum size of partial trace required to compute an efficient page mapping.

We proposed to expand these traces and use them for offline analysis. While Numalyze only collects one counter per page and per thread, our instrumentation also differentiate reads and writes, as shown in Algorithm 4.1. This distinction is important for memory analysis as reads and writes do not have

the same impact on performance. Furthermore, an easy way to reduce the number of remote accesses produced by a data structure is to duplicate it and map a copy to each node, but this optimization is only useful to data structures which are mostly read. Moreover, as page numbers are not really meaningful for humans, we added contextual information to Numalyze traces. Indeed, Tabarnac collects data structure information by three different means. First: each time a thread is created, it computes its stack bounds and create a virtual structure named `Stack#N` where N is the thread Id. Then, every time a binary file is loaded (main file or shared library), it inspect the binary, looking for static data structures. Finally it intercepts all calls the `malloc` functions family, keeping track of allocated data structures. Only structures that are bigger than one page (usually 4Kib in current x86_64 architectures) are recorded as our analysis granularity is the memory page. The data structure informations (name, size and address) are only used to generate the visualization, after the end of the instrumentation.

Algorithm 4.1 Handling of memory accesses by Tabarnac.

```
function MEM_ACCESS(unsigned long address, int threadId, char type)
    uint64_t page = address » page_bits;
    accesses[threadId][page][type]++;
end function
```

The name detection of allocated data structure is heuristic and based on source code analysis. When a data structure is allocated, and if the binary was compiled with the debug (`-g`) flag, Tabarnac inspects the source code at the address responsible for the allocation. It looks for the first identifier before the "`=`" sign and consider it the name of the data structures. For instance for the code presented in Listing 4.2, the name of the allocation will be `MyDataStructure42`. If the source code is not accessible the data structure will be name `malloc#N` where N is the number of call to the `malloc` function. The beginning and ending addresses of the data structure correspond respectively to the address returned by `malloc` and the size requested in the call.

Listing 4.2 A simple allocation.

```
1       double * MyDataStructure42 =
2           // This is an allocation
3           (double *) malloc(N*sizeof(double));
```

4.1.2 Ease of use and portability

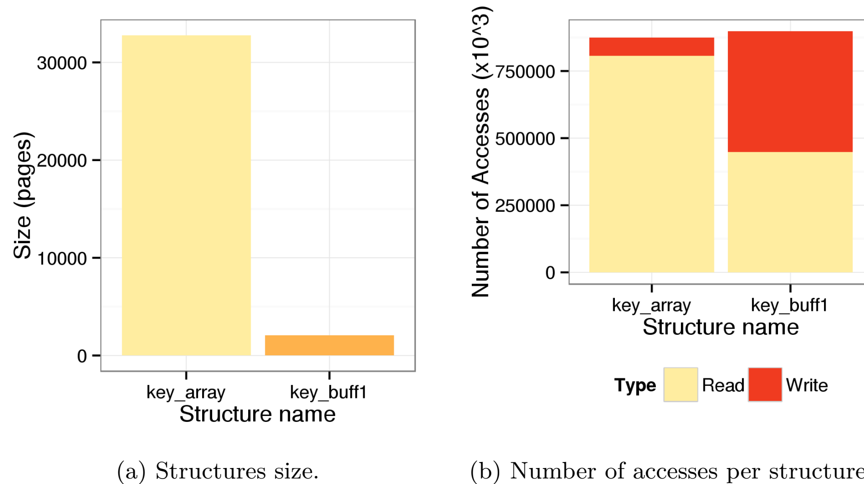
At the end of the execution, the tool generate three CSV files. The first contains the list of pages and the number of reads and writes per thread. The second contains the list of structures with their names, sizes and start addresses, the last file contains the stacks size and addresses. Then, a R-markdown script which reads the trace, retrieves the page to data structure mapping and generates the final visualization (as an HTML page).

Tabarnac only depends on Pin for the trace collection and R for the visualization, and can, thus, be installed easily. If all the R libraries required to generate the visualization are not present, it is able to install them automatically. By default Tabarnac generates both the memory trace and the visualization, but the user can also collect the trace and generate the visualization separately on different machines.

4.1.3 Visualization

Tabarnac visualization provides a summary of the trace through several plots. It aims at showing how pages are shared inside each data structure. Therefore, it provides two types of plots, the first highlights the importance of each data structure, while the second describes the sharing patterns inside these data structures. Each plot is introduced by an explanation of the characteristic it emphasizes, what common issues it can help to understand and hints about how to fix these issues. We present here each plot included in Tabarnac visualization. Additionally, A full visualization, including comments and hints, is available online: https://dbeniamine.github.io/Tabarnac/examples/.

The visualization starts with a small introduction, summarizing the main principles while developing for NUMA machines, and shows the hardware topology of the analyzed machine extracted with Hwloc [Broquedis et al., 2010]. After the introduction, the visualization focuses on the usage of data structures. Some structures are not displayed if less than 0.01 % of the total accesses happen within them. This makes the output more readable by focusing on the most important structures. However, it is possible to ask Tabarnac to include them for a more detailed view.



(a) Structures size.

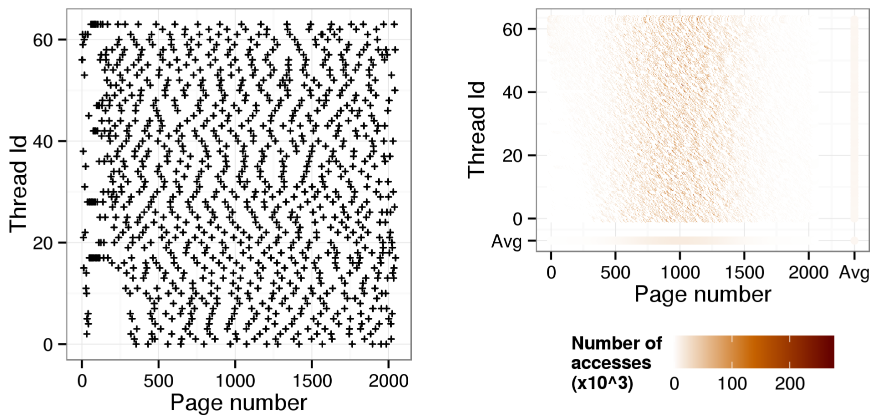
(b) Number of accesses per structures.

Figure 4.1 – Global views of the memory usage.

The first series of plots presents information about the relative importance of the data structures. It consists of two plots, showing the size of each data structure, as in Figure 4.1a, and the number of reads and writes in each structure as in Figure 4.1b. These plots give a general idea of the importance of the

structures used by the parallel application. Moreover, knowing the global read-/write distribution is very useful as it determines the possible optimizations. For instance, small data structures written only during initialization or very rarely can be relatively easily duplicated, such that each NUMA node works on a local copy.

The second series of plots is the most important one. It shows, for each page of each structure, which thread is responsible for the first touch (Figure 4.2a). This information is important as the default policy for Linux and most other operating systems is to map a page as close as possible to the first thread accessing it. If the first touch distribution does not fit the overall access distribution, the default mapping performed by Linux might not be efficient. To address this issue, the developer can either correct the first touch or do some manual data mapping to ensure better memory access locality and balance during the execution.



(a) First touch distribution.

(b) Per thread access distribution.

Figure 4.2 – Per structure view of the memory usage.

This second series also shows the density of accesses performed by each thread and their global distribution. In the example shown in Figure 4.2b, each horizontal line represents the number of accesses performed by each thread to the memory space. There is also one line for the average number of accesses. Additionally, for each thread the average number of accesses to the structure is indicated by its tone. Darker points indicate more memory accesses to the page. This visualization gives an easy way to understand the data sharing between threads, as well as the balance between pages and threads. These plots can be used to identify inefficient memory sharing and to determine the best NUMA mapping policy. For instance in Figure 4.2b, we can see that the addresses in the middle of the data structure (from 500 to 1500) are accessed by each threads and significantly more than the other pages. The consequences on the performance of this pattern and how we can improve it are discussed in Section 4.2.3.

4.2 Experimental validation

To evaluate Tabarnac, we present two examples of benchmark optimization achieved with the help of Tabarnac. For each benchmark, we compare the speedup obtained with the modifications done using the knowledge provided by Tabarnac to the one obtained by using several classic NUMA mapping tools and policies. We also evaluate the overhead of Tabarnac on each of the NAS Parallel Benchmarks (NPB).

4.2.1 Methodology

We used two NUMA machines for our experiments, `Turing` and `Idfreeze`. The second machine has only been used to compare the instrumentation overhead between Intel and AMD machines. All the other experiments ran on `Turing`. The hardware and software configurations are summarized in Table 4.1.

	Hardware totals				
	Nodes	Threads	Vendor	Model	Memory
`Turing`	4	64	Intel	Xeon X7550	128 Gib
`Idfreeze`	8	48	AMD	Opteron 6174	256 Gib
	Hardware per node				
	Cores	Threads	Frequency	L3 Cache	Memory
`Turing`	8	16	2.00 Ghz	18 Mib	32 Gib
`Idfreeze`	6	6	2.20 Ghz	12 Mib	32 Gib
	Software				
	Kernel	Distribution		Bios configurations	
`Turing`	Linux 3.13	Ubuntu 12.04		Hyper threading	
`Idfreeze`	Linux 3.2	Debian Jessie		No hyperthreading	

Table 4.1 – Hardware and software configuration of the evaluation systems for Tabarnac.

We evaluated the overhead of Tabarnac over all the NPB [Jin et al., 1999]. Moreover we present the performance optimization of the following applications with Tabarnac: the `IS` benchmark from the NPB and `Ondes3D`. They were chosen to demonstrate different memory access behaviors with different strategies to improve them.

All applications use OpenMP for parallelization, they were compiled with `gcc`, version 4.6.3, with the `-O2` optimization flag. Both analysis and performance evaluation are performed with the maximum number of threads that the machine can manage with its hardware, which means 64 threads for `Turing` and 48 for `Idfreeze`.

For the application we modified thanks to the knowledge obtained with Tabarnac, we also compare the performance obtained with following two traditional mapping policies. The *interleave* policy is performed with the help of the `numactl` tool [Kleen et al., 2005]. The recently introduced *NUMA Balancing*

technique [Corbet, 2012], which is executed with its default configuration. Our baseline for the experiment is an unmodified Linux kernel, version 3.13, with the first-touch policy. The NUMA Balancing mechanism is disabled in this baseline. To lessen the overhead of Tabarnac, we analyzed the applications with smaller input than the one used for their performance analysis.

For the plots presenting speedups, each configuration was executed at least 10 times. Each point shows the arithmetic mean of all runs. The error bars in those plots represent the standard error.

All the files required to reproduce the experiments or the analysis described here online: https://github.com/dbeniamine/tabarnac_expe.

4.2.2 Ondes3D

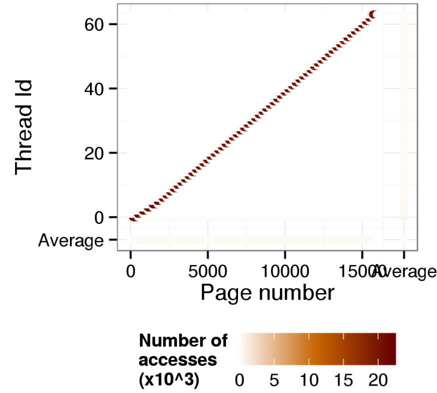
`Ondes3D` is the main numerical kernel of the Ondes3D application [Dupros et al., 2008]. It simulates the propagation of seismic waves using a finite differences numerical method. We analyzed `Ondes3D` with Tabarnac on a small input resulting in 0.7 Gib of memory usage. For the performance analysis, we increased the size of the input reaching 11.3 Gib of memory used.

The analysis of the accesses distribution in `Ondes3D` shows that each structure seems to be well distributed between the threads, as we can see for structure `vz0` in Figure 4.3a. However, for all structures, thread 0 is responsible for all first accesses, as we can see in Figure 4.3b. Due to this pattern, if we run `Ondes3D` without any improved mapping policy, every page will be mapped to the NUMA node that executes the thread 0, resulting in remote accesses for the other threads. An easy fix is to perform the initialization in parallel and to pin each thread on a different core. Such a modification results in the first touch distribution shown in Figure 4.3c, for which pages are distributed among all the threads.

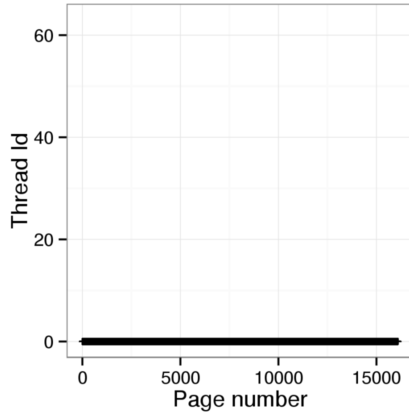
We compare the performance of our modified version called *First Touch* to three variants:

- The original (*Base*) version running on the unmodified Operating System (OS).
- A version using *NUMA Balancing* to move pages online.
- A version with *Interleave* policy that aims at reducing the impact of remote accesses.

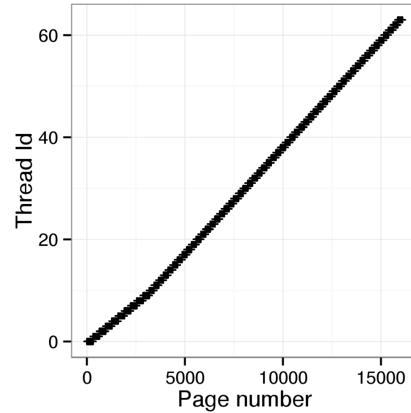
Figure 4.4 present the results of this evaluation. We can see that all methods improve the execution time compared to the base. Still, but *NUMA Balancing* provides less than 30 % speedup, while the static mappings (Interleave and the modified code) increase performance by 60 %. Indeed, with *NUMA Balancing*, all pages are initially mapped by the OS to the NUMA node of thread 0, and are only moved later on, after several remote accesses have already occurred, losing some optimization opportunities. This is a case where static mapping can be substantially better than automated tools. The *Interleave* policy provides a similar speedup as *First Touch* since it distributes the pages over the NUMA nodes at the beginning of the execution. In the end, for this example, using a classic static policy would have been enough to fix the performance issue, yet Tabarnac helped understanding and fixing it in a definitive way.



(a) Access distribution



(b) Original first-touch.



(c) Improved first-touch.

Figure 4.3 – Access distribution and first-touch for structure `vz0` from `Ondes3D`.

4.2.3 The `IS` benchmark

According to the NPB website[1], `IS` has a random memory access pattern, although we observed a very specific pattern. In this section we explain this pattern and how we used the knowledge about this pattern to improve the performance of `IS`.

`IS` was executed with input class D for the performance evaluation, resulting in a memory usage of 33.5 Gib, and class B for the analysis, with a memory usage of 0.25 Gib.

Figure 4.5 shows the original access distributions for the three main structures of `IS`. We can see that each structure has a different access pattern: for `key_array` (Figure 4.5a) each thread works on a different part of the structure, which let automated tools perform an efficient data/thread mapping on it. On the other hand, `key_buff2` (Figure 4.5b) is completely shared by all threads,

[1] http://www.nas.nasa.gov/publications/npb.html

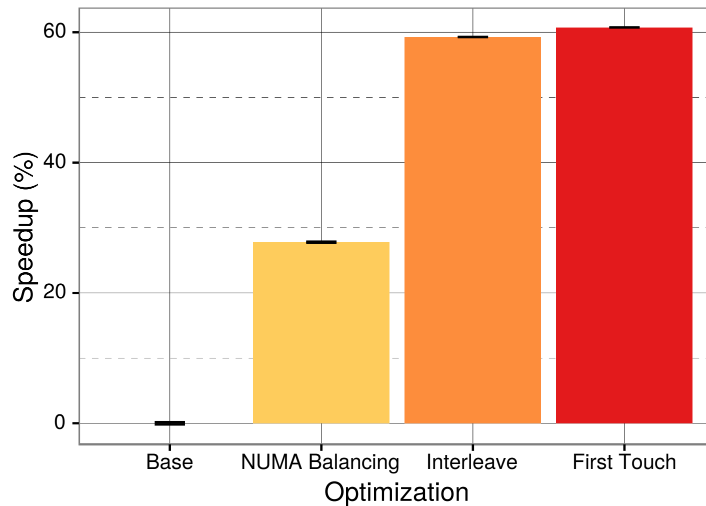


Figure 4.4 – Speedup for `Ondes3D` compared to the baseline.

we can barely see a diagonal pattern, indicating a small affinity between some threads and some pages. `key_buff1`'s access distribution (Figure 4.5c) is the most interesting one. We can see that almost all accesses occur in pages in the middle of the structure (from page 500 to 1500), and those pages are shared by all threads. This means that the number of access per page for each thread follows a Gaussian distribution centered in the middle of the structure.

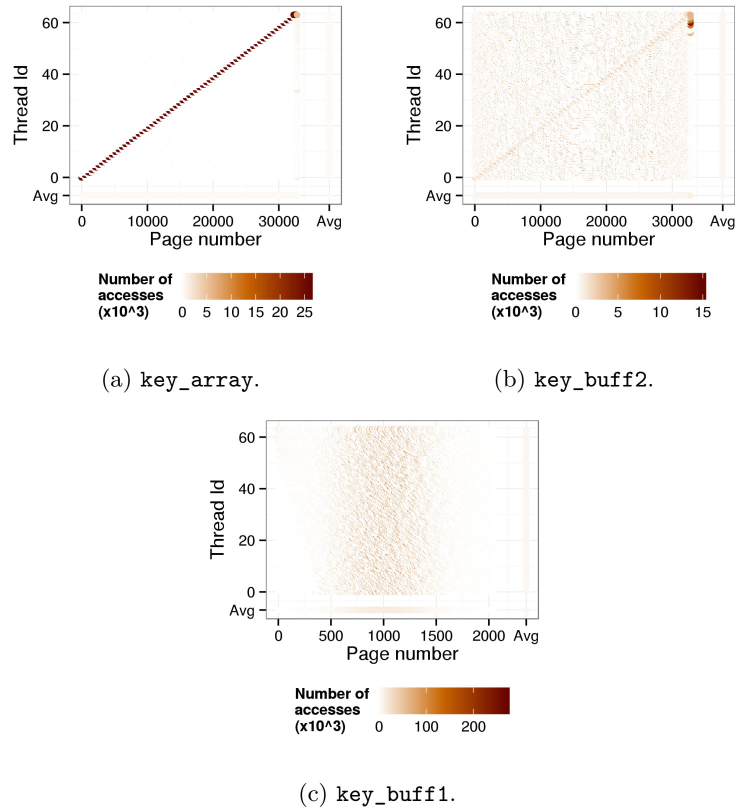
We can identify the source of this pattern in the `IS` source code. Indeed, all the accesses to `key_buff1` are linear, except the ones shown in Listing 4.3, Line 4, which depend on the values of `key_buff2`[2]. Those accesses happens in an OpenMP parallel loop that has the particularity to be scheduled dynamically. The comments on the `IS` code explains that the values of `key_buff2` follows a Gaussian distribution, therefore using a dynamic scheduling provides a good load balancing between the threads, while a cyclic distribution would results in some threads generating significantly more memory accesses than the others. Still it is possible to use a cyclic scheduling instead of the default one (dynamic) by defining a variable at the compilation time.

Listing 4.3 `IS` code responsible for the distribution of memory accesses.

```
1 #pragma omp for schedule(dynamic)
2 for( i=0; i< NUM_BUCKETS; i++ )
3   for ( k = m; k < bucket_ptrs[i]; k++ )
4     key_buff1[key_buff2[k]]++;
```

As simple cyclic distribution of the loop would result in unbalanced work, we can design a slightly more complex distribution that provides a fair load

[2] The code has been slightly modified to make it more readable. In the original version, the arrays generic pointers. Furthermore we removed several lines of code inside the loop that are not required to understand the memory pattern.



(a) `key_array`.

(b) `key_buff2`.

(c) `key_buff1`.

Figure 4.5 – Original memory access distribution for the main structures of `IS`.

balancing while enforcing locality of data. To do so, we split the loop into two equal parts and distribute each part among the threads in a cyclic way as shown in Figure 4.6 where each color represent a thread. This can be done by modifying the OpenMP pragma (Line 1 in the original code), as shown in Listing 4.4. Obviously this distribution does not provide an optimal load balancing, but only a relatively fair one.

Listing 4.4 Modified IS code.

```
1 #pragma omp for schedule(static,NUM_BUCKETS/
2     (2*omp_get_num_threads()))
```

Figure 4.7 show the accesses distribution obtained with this code modification. We can see that now each thread accesses a different part of `key_buff1`. Furthermore, if most of the accesses still occur in the middle of the structure, the average number of accesses across the whole structure is the comparable same for all threads, which means that our distribution preserves the good load balancing. Our modification has also changed `key_buff2`'s accesses distribution. Indeed, we can now clearly see a sharing pattern very similar to the one obtained for `key_buff1`.

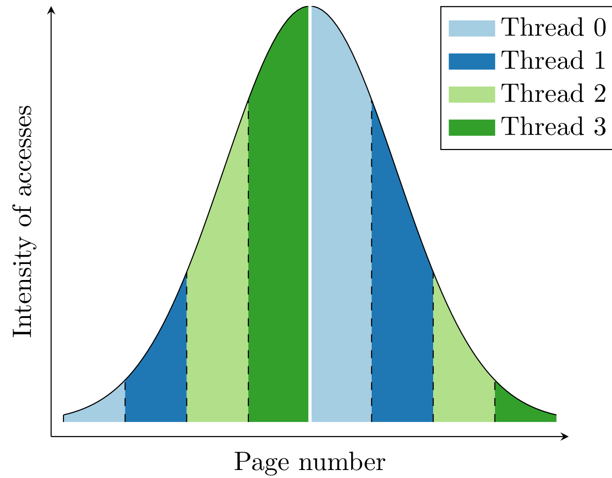
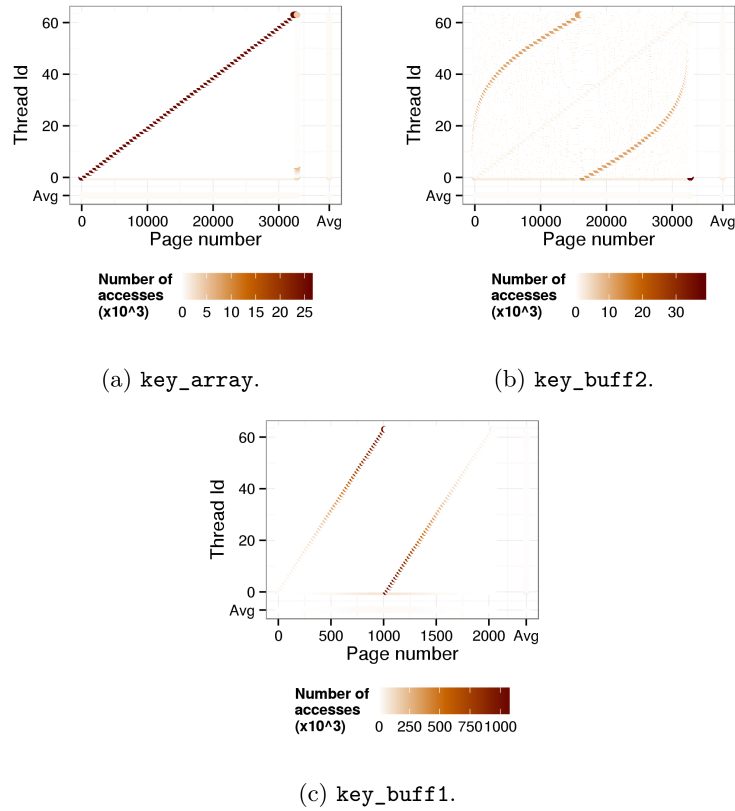


Figure 4.6 – Fair distribution of the pages of `key_buff2` among the threads.

The main point of our code modification is to improve the affinity between thread and memory, therefore we need to pin each thread on a core to keep them close to the data they access. To perform the thread mapping, we use the `GOMP_CPU_AFFINITY` environment variable. Tabarnac also showed us that the first touch is always done by the thread actually using the data for IS, therefore we do not need to explicitly map the data to the NUMA nodes.

We compare the execution time of `IS` (class *D*) for the three scheduling methods, *Dynamic*, *Cyclic* and our custom *Fair* distribution. We evaluated each scheduling method in each of the following setup: with our *Base* unmodified OS, with the *Interleave* policy and with *Numa Balancing* activated. As our modification rely on the firs-touch it does not make sense with Interleave and NUMA Balancing, therefore these policies are not evaluated for our *Fair* distribution.

Figure 4.8 shows the speedup of `IS` compared to the default version (*Dynamic*) for each scheduling method and for each mapping policy. The first thing to notice is that with the default *Dynamic* scheduling, both Interleave and NUMA Balancing slow the application down, by up to 10 %. This slowdown is due to the fact that two mechanisms (the mapping policy and the OpenMP runtime) or modifying the application behavior with conflicting interests and without communicating. Then we can note that *Cyclic* scheduling, proposed in the original code, already provides up to 13 % of speedup. Yet, both interleave and NUMA Balancing are still reducing the performance gains. Finally, the *Cyclic-Split* version provides more than 20 % of speedup with a very small code modification. This example shows that while mapping policies (static or dynamic) can conflict with the parallel runtime and slow the execution time, analyzing an application's memory behavior can help fixing inefficient behavior resulting in significant performance gain.



(a) `key_array`.

(b) `key_buff2`.

(c) `key_buff1`.

Figure 4.7 – Modified memory access distribution for the main structures of `IS`.

4.2.4 Tracing overhead

Finally, we evaluate the instrumentation cost of Tabarnac. To do so, we executed all of the NPB in class B[3] with 64 threads on both evaluation systems and compared the original execution time to the execution time with instrumentation enabled.

As we can see in Figure 4.9, on the Intel machine, the instrumentation slows the execution down by a factor ranging from 10 to 30. Nevertheless, on the AMD machine, the instrumentation is 10 to 50 % slower for most benchmarks, and two to three times slower than on the Intel machine for pathological cases. This behavior was expected as Pin is an instrumentation library developed by Intel. Although this overhead is not negligible, we have to consider the fact that often we can instrument smaller versions of the applications, as we focus on the general behavior. Moreover, our method is more precise than sampling and thus one run is often enough. Finally, as our analysis is designed to be used during the development phase and not at runtime in an automated tool, we consider that this overhead is acceptable.

[3] `DC` was run in class A as it was too slow in class B to run the full experiment.

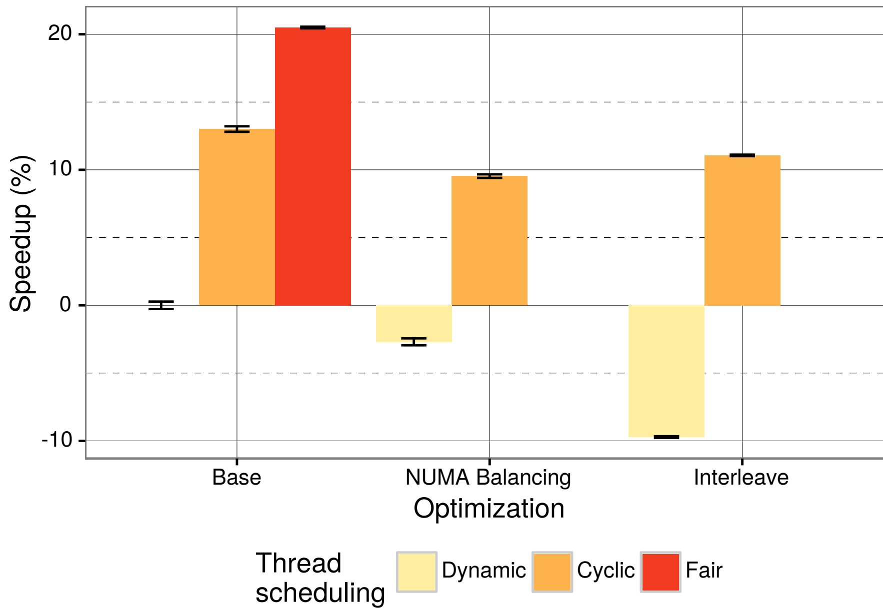


Figure 4.8 – Speedup for IS (class D) compared to the baseline.

4.3 Results and discussion

Our experiments have highlighted the fact that using blindly static mapping policies such as Interleave or dynamic tools such as NUMA Balancing can result in significant performance loss. Furthermore in the experiments where both dynamic and static policies increased the performance, the difference between the gain provided by the two policies was significant. The only way to predict which tool is the most suited to an application is to understand the sharing patterns of the memory of this application by the threads.

Tabarnac enables developers and users to achieve performance improvements in two ways. First, by providing a deep understanding of the memory sharing pattern, it enables the user to find the best existing mapping policy. Second, this knowledge can be used to identify and fix inefficient memory behavior, for instance by designing a specific thread scheduling taking the sharing patterns into account. Our experiments showed that both situations result in significant performance gains.

While Tabarnac helps the developer identify and fix some inefficient sharing patterns, it only provides a global overview of the memory usage. Indeed it does not collect any temporal information. Therefore, it does not enable identification of the distribution of inefficient memory access patterns over the time, such as the ones depicted in Section 3.1 with the naïve matrix multiplication.

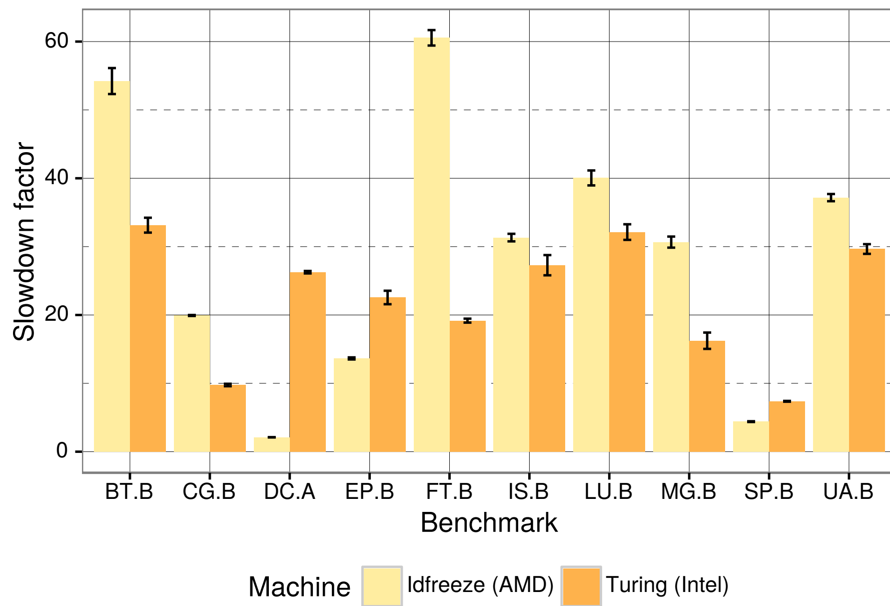


Figure 4.9 – Tabarnac's instrumentation overhead.

Chapter V

Collecting Fine Grain Memory Traces

Contents

5.1 Moca components **62**
5.2 Background knowledge **63**
5.3 Design **63**
5.3.1 Page faults interception and injection **64**
5.3.2 Internal design **65**
5.4 Experimental validation **70**
5.4.1 Methodology **70**
5.4.2 Moca validation **72**
5.4.3 Comparison to other memory trace collection tools **72**
5.4.4 Results and discussion **77**
5.5 Conclusions **78**

Tabarnac manages to keep it's overhead reasonable by two means. First it sees the memory at the granularity of the page, loosing any information about the internal use of a page. While this is enough to debug issues specific to Non-Uniform Memory Access (NUMA), it is not *detailed* enough to build a map of the memory accesses. Second, it does not collect any temporal information, reducing the size of its trace and avoiding to maintain temporal order. An instrumentation similar to Tabarnac but with temporal information with a finer grain than the page would have a considerable overhead. Therefore it is required to find a lighter mean to collect memory traces. This chapter presents Memory Organisation Cartography & Analysis (Moca), a memory trace collection system that relies on an Operating System (OS) level mechanism called page fault interception and injection. Moca addresses the challenge of collecting *detailed*, *precise* and *complete* traces, that is with temporal information.

This work is the subject of two Inria research reports [Beniamine et al., 2015a, Beniamine and Huard, 2016] and has been submitted at Cluster, Cloud and Grid Computing (CCGRID) 2017. Moca is distributed under the GPL license: https://github.com/dbeniamine/Moca.

This chapter presents first the main components of Moca in Section 5.1. Then we present some background knowledge required to understand Moca design in Section 5.2. After that we discuss the implementation details of Moca Linux kernel module in Section 5.3. Then we present an experimental validation including an extensive comparison with existing memory trace collection tools in Section 5.4. Finally we provide our conclusions about memory traces collection in Section 5.5.

5.1 Moca components

Moca consists of a Linux kernel module that can be loaded at runtime, a script in charge of both loading this module with the proper parameters and launching the monitored application on the user behalf and an optional context library able to retrieve data structures information. It neither relies on architecture specific technologies such as AMD IBS or Intel PEBS, nor on architecture dependent kernel code, kernel patch or kernel modifications. Therefore it is portable and can be run on any Linux kernel from the 3.0.

When the user wishes to retrieve data structure informations, Moca runs the application twice with virtual address space randomization disabled. The first time, the application is run with the context library preloaded but without the Linux kernel module. At the opposite, during the second run, only the Linux kernel module is loaded. This library is a port of the one used by Tabarnac without the dependencies to Pin. It reads static data structures information in each executed binary file and stores data about structures larger than one page. It also intercepts all calls to `malloc` family functions and names these calls according to their stack trace and retrieve stacks from the file `/proc/<pid>/maps`. Naming the data structures by call stack instead of by looking at the source code is an improvement, as some code uses temporary pointers for allocations. Moca and the library are run separately as it would be costly for Moca to check if each intercepted access has been triggered inside the context library. Our context library is lightweight, the added cost is the cost of a regular execution added to a small constant overhead for each binary opened and each allocation performed.

Thus the overhead of using Moca and the library is basically Moca overhead plus one time the normal execution time.

5.2 Background knowledge

As explained in Chapter 3, in recent Linux kernel, physical memory pages are lazily allocated to page frames during the execution. The first access to a page in the virtual address space triggers a page fault. To handle this page fault, Linux allocates a physical page to the requested page frame. Such a page fault can also be triggered when a thread access a shared page modified by another thread. Linux provides the possibility to intercept these page faults. Page faults interception is an efficient mean to collect information about memory usage at the run time. Such a mechanism has been used in several existing works : in parallel garbage collectors [Boehm et al., 1991], in memory checkpointing [Heo et al., 2005] or in the domain of virtualization to provide the hypervisor with information about the memory usage of the guest OS [Jones et al., 2006]. However, page faults only occur when caused by predetermined events in the system (copy-on-write, paging, ...). Thus, intercepting existing page faults at a relatively low frequency do not provide the precision required for a thorough analysis. To improve this method, it is also possible to fake invalid pages at regular intervals in order to generate false page faults [Bae et al., 2012, Diener et al., 2013] at a higher frequency than regular ones. These false page faults are just triggered during regular memory accesses, that would not have caused a page fault if the page were not faked as invalid. The advantage is that they create additional events for the monitoring tool to collect, thus more *precision*, but the tool must be able to identify faked invalid pages. Fake page faults can be identified either by setting some bits of the Page Table Entry (PTE) that are not used by the OS, yet this kind of hack is not portable and might interfere with runtimes or by maintaining the set of faked invalid pages in the monitoring tool.

Existing memory profiling tools do not use false page faults injection and only need to store the location of memory pages and the threads that access them. As a consequence, they require a relatively small data structure in memory for their own usage. Moca is a new *complete* memory trace collection system, based on page fault interception and false page faults injection, able to capture *precisely* the temporal evolution of memory accesses performed by a multithreaded application. To reach a satisfying *precision*, Moca has to maintain in memory both the trace data and the set of faked invalid pages. Overall, storing and exploiting efficiently these data within the kernel space and outputting them in real time to the user space is a challenge and is the main contribution of our work.

5.3 Design

Two tasks are addressed by the kernel module included in Moca. The first one is to keep track of the set of pages accessed by the application during an elementary monitoring interval. The second one is to manage the huge quantity of data produced by the trace collection within the kernel space in-between

regular flushes toward the user space. Of course, these two tasks should be as slightly intrusive as possible.

The remaining of this section discusses the design of Moca Linux kernel module as it is both the most important and the most complex part of Moca.

5.3.1 Page faults interception and injection

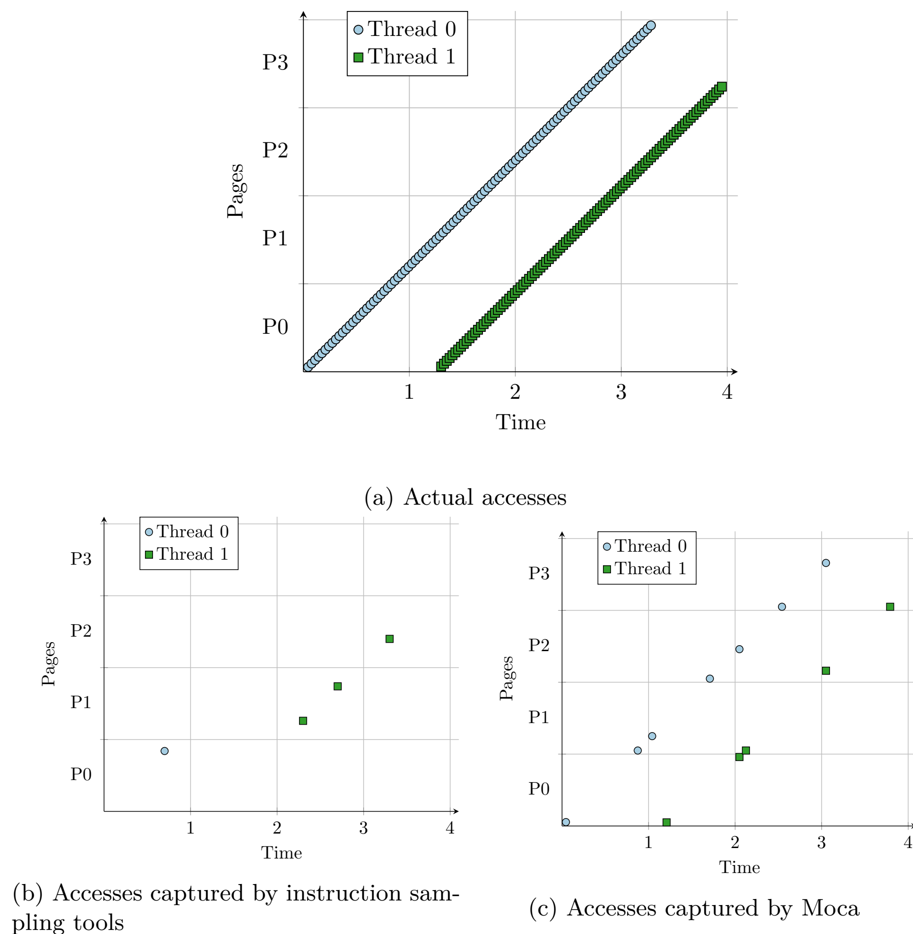
Moca collects *complete* traces in the sense that the exact set of pages accessed by the application is deduced from the collected events at all times during the execution. Thus, it is *complete* at the page granularity. Other information such as exact addresses and access times are a sample of the set of all the accesses.

Moca is built upon the possibility to register an additional callback on Linux page faults. Nevertheless, a page fault does not occur at each memory access. To monitor memory accesses during the course of the execution, we need to reenable a page fault similar to the first access, but performed on a regular basis and on behalf of Moca. In other words, we need to inject false page fault by periodically marking as *not present* the pages accessed by the application. In Linux terminology, marking means that the next access to the page will trigger a page fault which will have to be handled, in this case, by a handler contained in Moca.

This method has several advantages over hardware sampling or instrumentation. First, it provides a superset of all the memory accesses, because it guarantees that each page accessed by the monitored application will fault once and will be traced. Second, at the end of each monitoring interval, we know the exact set of accessed pages from which we deduce a superset of actual memory accesses. This comes in addition to the fact that each false page fault generated provides Moca with exact information about one memory access. This means that Moca also performs a sampling of all the memory accesses at the granularity of the Byte. Because it is designed to manage large chunks of trace data within the kernel space, it also stores all the details about these samples in the collected trace.

Moca differs from instruction sampling because it is not necessary to increase the monitoring frequency of Moca to collect a *complete* trace. On the contrary, when using instruction sampling, if the pages of the application are accessed in an unbalanced manner, it is necessary to increase the sampling frequency to get a precise picture of the memory working set of the application. Nevertheless, there can be no guarantee that a chosen sampling frequency will result in a trace that contains all the pages on which the application works. To illustrate this difference, we consider two threads accessing a set of four pages in a linear way over the time as represented in Figure **??**. With a classic instruction sampling based tool, we will intercept one access every n instruction, which will result in a trace similar to the one depicted in Figure 5.1b. From such trace, it is impossible to identify the access pattern. At the opposite, a tool such as Moca using page sampling, will intercept every first access to a page during each chunk of time, resulting in the trace depicted in Figure 5.1c.

Moca also differs from instrumentation based tools because, just as in the case of sampling, memory accesses that are not collected in the trace are not trapped at all by a false page fault. Furthermore, the remaining memory accesses, which are collected, are trapped using a hardware mechanism and Linux kernel probes. Both are lightweight mechanisms, which means that the overall



(a) Actual accesses

(b) Accesses captured by instruction sampling tools

(c) Accesses captured by Moca

Figure 5.1 – Accesses done by two threads on a set of pages, captured by instruction sampling tools and by Moca.

instrumentation overhead of Moca is likely to be low. Indeed, instrumentation based methods often work at a high granularity, collecting few information, in order to keep their naturally high overhead in control.

5.3.2 Internal design

During the execution, our Linux kernel module needs to store three kinds of information:

1. The set of *tasks* (Linux internal representation of threads and processes) which are monitored. This is necessary because page faults will also be triggered by tasks which do not belong to the monitored application.

2. The set of all page faults which have been injected by Moca, required to distinguish false page faults from regular ones, because their handling differs. False page fault could also be identified by setting some bits of the PTE that are not used by the OS. While this solution avoid to maintain

the set of false page fault, it is a hack not portable and could interfere with a runtime.

3. The set of addresses recently accessed by each task, this set correspond to the actual memory trace. It is required to keep it in kernel space as we need to reinject these false page faults at the end of each monitoring interval. Afterwards, this set is transferred to the user space by a dedicated process and appended to the resulting trace.

These sets of data are stored in hashmaps. The Linux kernel provides some helpers to manage such hashmaps, still these maps generate an allocation each time we add an element to it. Therefore we implemented generic pre-allocated hashmaps in our kernel module. As memory space is restricted in the kernel space, these hashmaps are not statically allocated, but they only call the *kmalloc* function during their initialization. Furthermore, they are initialized as soon as possible which means at the beginning of the execution for the two first sets, and each time a new thread is created for the third. As Moca does not have any a priori knowledge of the monitored application, it allocates huge hashmaps. If the monitored application has a considerable memory footprint they might be too small, in this case, Moca will drop a part of the trace and notify the user that it is incomplete. The user can then either monitor its application on a smaller input or use Moca parameters to change the size of these hashmaps.

The two first sets of data are read at each page fault but rarely written, only when a new task of the monitored application triggers its first page fault or when Moca generate new false page faults. We can protect them with Linux kernel built-in *rwlocks*. The third type of information is the actual trace, divided, for each task, in a private set of *chunks*. A chunk is the set of accesses that have been collected during the monitoring time interval and is implemented as a hashmap. Chunks provide a discretization of time, each chunk embed two timestamp to mark its temporal bounds. To reduce the volume of information stored, the accesses are not timestamped. However, the arrival order in the chunk is preserved in the final trace file.

The discretization of time, materialized as a sequence of chunks, is useful as it let the different components of Moca work concurrently on different chunks. Indeed, the traced program always works on *current* chunks, one for each core, while the logging daemon, which flushes the trace from memory to permanent storage works on *completed* chunks. A monitoring kernel thread, manages the progress of this logical time. It periodically wakes up, marks the current chunks as *ending* and invalidates all the pages they reference. Once all pages of the *ending* chunks have been invalidated, it marks these chunks as *completed*. Finally, the logging process flushes *completed* chunks to the filesystem at a lower rate, in order to reduce the overhead of I/Os requests, and recycle them as empty places for upcoming chunks. Figure 5.2 depicts the interaction between the different processes and threads of Moca, its data structures and Linux.

Eventually, Moca generates one CSV file, each line of this file describe one access giving its physical and virtual address, the number of read and writes captured, a bitmask indicating on which CPU the access occurred, the start and end timestamp of its chunk and the internal identifier of the task which triggered it. A set of access sharing the same timestamps and task identifier correspond to a chunk. The order of accesses inside a chunk is preserved.

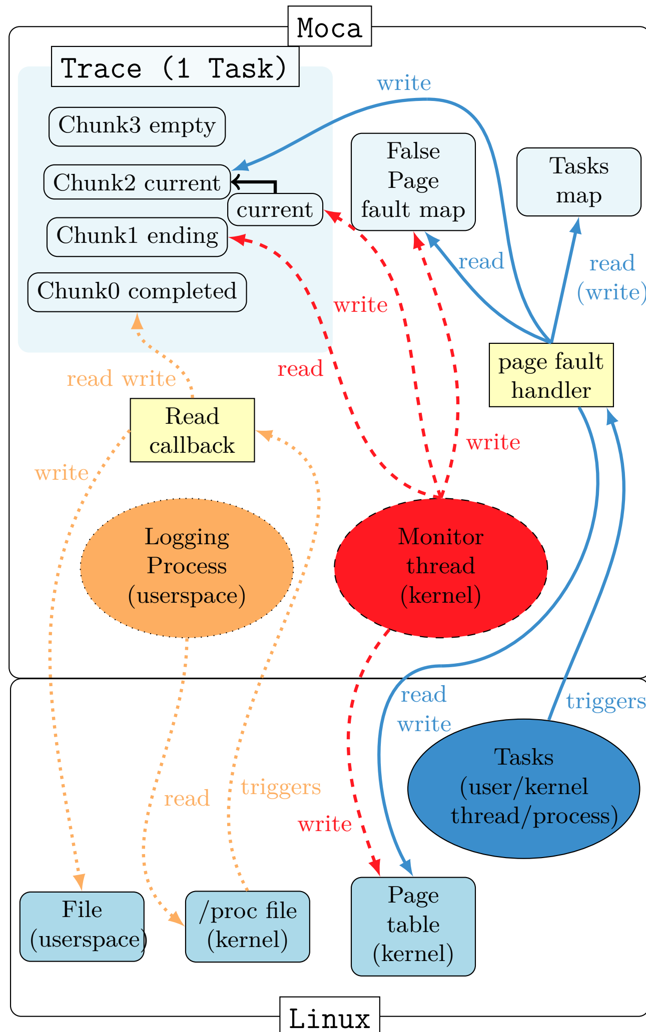


Figure 5.2 – Interactions betweens Moca and Linux.

The monitoring thread, which is a kernel thread, that uses the Algorithm 5.1, is in charge of enabling false page faults destined for Moca. It performs its task by removing the `PRESENT` flags from the PTE that corresponds to each recently accessed addresses. Of course, the shorter is the period between two wakeups, the more precise the trace is. This period is called *monitor thread wakeup interval* (or *monitoring interval*). But invalidating all the recently accessed pages takes time as it requires to take a write lock on the page faults hashmap. This write lock delays any pending false page fault in the monitored application. Thus the wakeup frequency of the monitoring thread cannot be too high, otherwise its action becomes too intrusive. The `MonitorThreadWakeUpInterval` Moca parameter lets the user change the default setting that we have empirically chosen after a few experiments that are presented in Section 5.4.2.

The logging daemon, that uses Algorithm 5.2, is a userspace process which periodically reads `/proc` pseudo files used by Moca kernel module to export its

Algorithm 5.1 Monitoring thread algorithm

```
1: while NotFinished() do
2:     for all t in MonitoredTasks() do
3:         EndCurrentChunk(t)
4:         for all Addr in PreviousChunk(t) do
5:             WriteLockPF()
6:             AddFalsePF(Addr)
7:             WriteUnlockPF()
8:         end for
9:         MarkPreviousChunkFinished(t)
10:    end for
11:    sleep(MonitorThreadWakeUpInterval)
12: end while
```

data to userspace. Those reads trigger a callback method in our tool which flushes completed chunks from memory to disc. As it works on completed chunks, it does not directly interfere with the normal application execution. Especially, no lock is required to access to these completed chunks, and, as it mostly generates disc I/O, it does not compete much for CPU. Moca has just to wake him up sufficiently often so that the kernel module does not run out of free space to store upcoming chunks.

Algorithm 5.2 Logging daemon algorithm.
Note that no locks are required to work on completed chunks.

```
1: while NotFinished() do
2:     for all t in MonitoredTasks() do
3:         for all c in FinishedChunks(t) do
4:             WriteTraceToDisk(c)
5:             ReinitChunk(c)
6:         end for
7:     end for
8:     sleep(LoggingDaemonWakeupInterval)
9: end while
```

Each time a page fault occurs, it is trapped by the handler registered by Moca. As depicted by Figure 5.3, it first finds out if the task (thread or process) involved by the page fault is monitored or not. If not, it has to check if the task is a child of a monitored task and, in this case, it starts monitoring it. The check is done with a simple read lock on the hashmap containing the monitored tasks, the write lock is only taken if the task must be added. This last case occurs only at the first page fault from a new monitored process or thread which is quite rare and usually occurs only at initialization time. For instance, in the benchmarks used for the evaluation it happened 8 times out of 5×10^6 accesses. At the end of this phase, if the task is still not monitored, we let Linux handle the page fault as usual.

When a monitored task triggers a page fault, the access is first added to its current chunk. For each access, Moca stores the exact address, its type (read or write) and the CPU on which the fault occurred. Then, it checks if the page

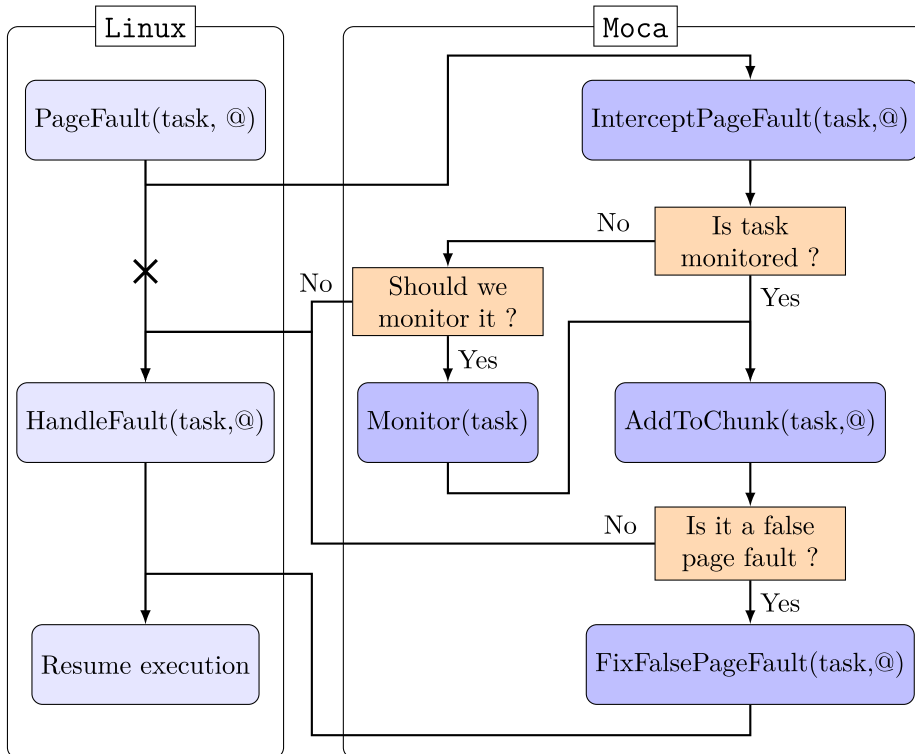


Figure 5.3 – Flow chart of Moca's page faults

fault has been injected by Moca or if this is a legitimate page fault. To do so, it needs to take a read lock on the false page fault hashmap. In the first case, Moca *fixes* it by setting the `PRESENT` flag on the PTE. The hashmap entry indicating that the fault was injected by Moca should then be removed, but this would required a write lock, so we only mark the hashmap entry as `BAD`. PTE are stored in Page Middle Directory (PMD) themselves stored in Page Global Directory (PGD), to write a flag on a PTE it is required to take a lock at the PMD level. Thus, if two threads faults at the same page in parallel, they will be serialized at this step and only one of the thread will change this flag. Therefore, it is safe to mark the page entry as `BAD` without holding the write lock. When the monitoring thread, which already holds a write lock on this hashmap, injects false page faults it might run out of space. In this case, it walks the hashmap freeing all `BAD` entries it enconters. This lazy removal reduce the overhead of Moca in three different ways. First, it avoid serializing page faults. Then, it avoid removing entries when there are still enough room, which is highly probable as we allocate huge hashmaps independently from the memory footprint of the monitored application. Finally, it enables recycling entries for page that are often accessed, speeding up the monitoring thread.

If a fix occurred in the Moca handler, Linux silently aborts the page fault when it resumes its execution. In the other case, it executes a normal page fault handling. Each page fault increases an atomic clock that is used to timestamp the beginning and end of the chunks. A race might occur if the monitoring

thread enables a false page fault between the end of our handler and the end of Linux page fault handler. To avoid that, Moca stores for each CPU the last address that faulted, and does not clear it right away but at the end of the next chunk.

5.4 Experimental validation

In this section, we compare Moca and Tabarnac to other existing memory analysis tools. In a first time, we present their main differences in terms of portability and capabilities. Then, we present two sequences of quantitative experiments, one that outlines the importance of the default parameters chosen for Moca and the other that compares the precision and performance of all the tools.

5.4.1 Methodology

Our main experiments were run on machines from Grid5000 `Edel` cluster (Intel machines). As some state of the art tools can only run on AMD machines, we also ran some of the experiments on `StRemi` machine from Grid5000 grenoble. These machines hardware and software specifications[1] are summarized in Table 5.1.

	Hardware totals				
	Nodes	Threads	Vendor	Model	Memory
`Edel`	2	8	Intel	Xeon E5520	24 Gib
`StRemi`	2	24	AMD	Opteron 6164 HE	48 Gib
	Hardware per node				
	Cores	Threads	Frequency	L3 Cache	Memory
`Edel`	4	4	2.27 Ghz	8 Mib	12 Gib
`StRemi`	12	12	1.70 Ghz	12 Mib	24 Gib
	Software				
	Distribution		Kernel	Bios configurations	
`Edel`	Debian Jessie		Linux 3.16.0-4	No hyperthreading	
`StRemi`	Debian Jessie		Linux 3.16.0-4	No hyperthreading	

Table 5.1 – Hardware and software configuration of the evaluation systems for Moca.

We disabled address space randomization to make the comparison between different traces more practical. As our two evaluation machines do not have the same hardware, we limited the number of threads used by OpenMP to 8, that is the largest number of hardware threads available on both machines.

We evaluate Moca and Tabarnac by comparing them to the following state of the art tools. The first one, Mitos, is the tracing tool from MemAxes [Giménez

[1] Grid5000 provides an online hardware description:
https://www.grid5000.fr/mediawiki/index.php/Grenoble:Hardware#Edel
https://www.grid5000.fr/mediawiki/index.php/Reims:Hardware#Stremi

et al., 2014] and relies on Intel PEBS technology. The second one, MemProf [Lachaize et al., 2012], is designed to analyze NUMA performance issues and relies on AMD IBS. The main differences between these memory profiling tools are summarized in Table 5.2.

	Moca	Tabarnac	Mitos	MemProf
Design				
Mechanisms	Page faults	Inst*	PEBS + Inst*	IBS
Architecture	**Any**	Intel (AMD)	Intel	AMD
Completeness				
Trace Granularity	**Address**	Page	**Address**	**Address**
Superset	**Page**	**Page**	None	None
Detail				
Temporal data	**Yes**	No	**Yes**	**Yes**
CPU location	**Yes**	No	**Yes**	**Yes**
Nature	**Yes**	**Yes**	Yes**	Yes**

Table 5.2 – Comparison of different memory traces tools.
*Inst: Instrumentation.
**Type (Read/Write) must be deduced from the instruction name.

Name	Footprint*	Description	Group
`IS`	132 Mib	Integer Sort	Memory Intensive
`CG`	125Mib	Conjugate Gradient	
`MG`	508Mib	Multi-grid	
`FT`	398Mib	Discrete 3D FFT	
`UA`	112Mib	Unstructured Adaptive mesh	Unstructured
`DC`	1.46Gib	Data Cube	
`BT`	120Mib	Block Tri-diagonal solver	Pseudo Applications
`SP`	122Mib	Scalar Penta-diagonal solver	
`LU`	118Mib	Lower-Upper Gauss-Seidel solver	
`EP`	78Mib	Embarrassingly parallel	CPU bound

Table 5.3 – Description of the NAS Parallel Benchmarks (NPB).
All benchmarks are in class B.
*Footprints: maximum memory used, measured with Valgrind tool: Massif.

In the following sections, all the tools are evaluated on each of the 10 NAS Parallel Benchmarks (NPB) [Jin et al., 1999], which are presented in Table 5.3, according to the information available on the NASA website[2]. In this table, we included the footprint of each benchmark, that is the maximum memory used, as reported by Valgrind's tool Massif.

In each experiment, Moca and Tabarnac were run with their default parameters, except for the experiment about the influence of Moca parameters. For

[2] http://www.nas.nasa.gov/publications/npb.html

Moca, their default values are: a wakeup interval of 0.5 s for the logging process and 50 ms for the monitoring thread. Each point in each plot is the average of at least 30 executions. Along with each point, the error bars represent the standard error.

As explained in Chapter 2, we distribute[3] all the files needed to reproduce our experiments at three different levels: The first level contains the filtered results (CSV files) from the experiments along with the `R-markdown` scripts that generated the plots presented in this chapter, making possible the reproduction of our statistic analysis. The second consists of the full raw traces generated by our experiments along with the scripts used to extract the filtered traces (CSV files from the previous level) and the scripts used at the previous level to perform the analysis. Finally, at the most comprehensive level, we provide a git repository that includes our deployment environment, dependencies to all the tools and files required and instructions that explain how to reproduce the experiment with or without access to Grid5000.

5.4.2 Moca validation

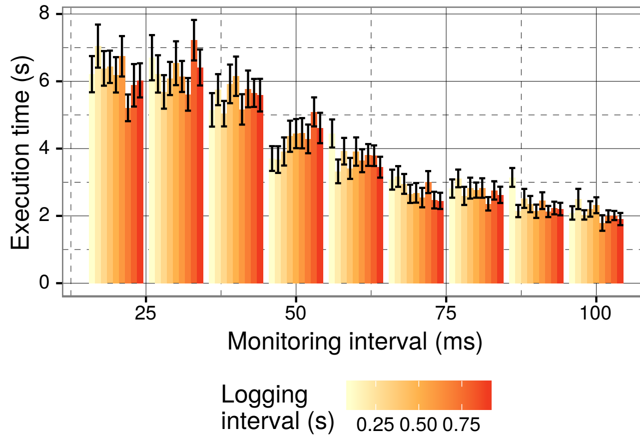
Before comparing Moca to existing tools, we need to evaluate the impact of the wakeup intervals (logging daemon and monitoring thread) on the trace *precision* and on the overhead. To do so, we ran the `IS` benchmark instrumented by Moca with a wakeup interval ranging from 0.1 s to 0.9 s for the logging daemon and from 20 ms to 100 ms for the monitoring thread. We decided to evaluate monitoring thread wakeup interval close to the Linux scheduler interval 50 ms as these events seems related to us. Concerning the logging interval we need it to be longer than the previous one to limit the impact of Input / Outputs (I/Os). For each run, we measure `IS` execution time and the number of accesses captured. We have chosen `IS` for this evaluation as it is one of the memory intensive out of the NPB and quick experiments with other ones confirmed these results. This experiment was run on a machine from the `Edel` cluster.

We can see on the Figure 5.4a that the execution time increases when we reduce the monitoring wakeup interval. At 40 ms it seems to reach its worst level, thus we should keep it larger. At 50 ms, the default value we have chosen, the Figure 5.4b shows that we obtain more than two thirds of the events captured at smaller intervals, which seems sufficient to us. Regarding the logging interval, our experiments do not exhibit a clear trend. Changing it seems to interfere with the system I/Os scheduler resulting in chaotic variations both in the execution time and the number of captured events. The fact that variations in execution time result in matching variations in the number of captured events is due to the fixed length of monitoring intervals : the longer the execution, the more monitoring intervals there are and the more events the trace contains. Overall these variations are not significant as all the confidence intervals intersect. Finally we have chosen a logging interval of 0.5 s, the median value, in order to avoid unnoticed effect caused by extremum values.

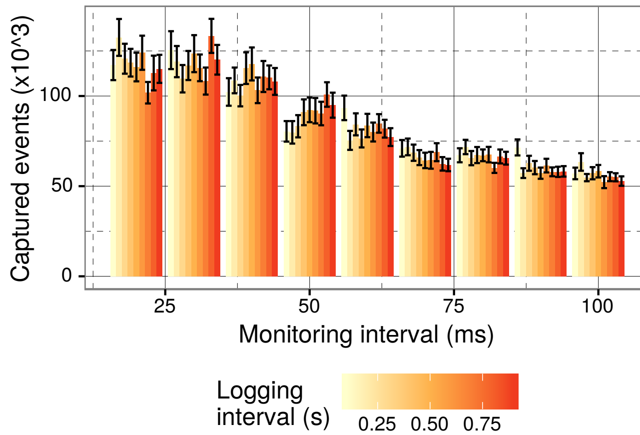
5.4.3 Comparison to other memory trace collection tools

Preliminary experiments showed us that Mitos capture by default way less distinct pages than Tabarnac and Moca. Thus, we tried to change Mitos sampling

[3] See our experiment repository: https://github.com/dbeniamine/Moca_expe



(a) Execution time.



(b) Number of captured events.

Figure 5.4 – Influence of the wakeup intervals on `IS`, class A.

period in order to make it capture as many pages as possible, we name this version MitosTun. Surprisingly, Mitos behavior regarding this sampling period is not monotonous, we had to try many different periods to find the proper one.

The default MemProf distribution did not work with our experimental setup. With the help of their support team[4], we managed to make it work by disabling the library used to retrieve data structures names. For the same reason as in the case of Mitos, our study includes two versions of MemProf: the default version and MemProfTun in which we have increased the sampling rate to its maximum.

Finally our evaluation also distinguishes Moca (kernel module only) from MocaPin, which also retrieve the data structure information using a Pin instrumentation. In recent version of Moca, the library that retrieve data structure

[4] see issue at github.com/Memprof/scripts/issues/1

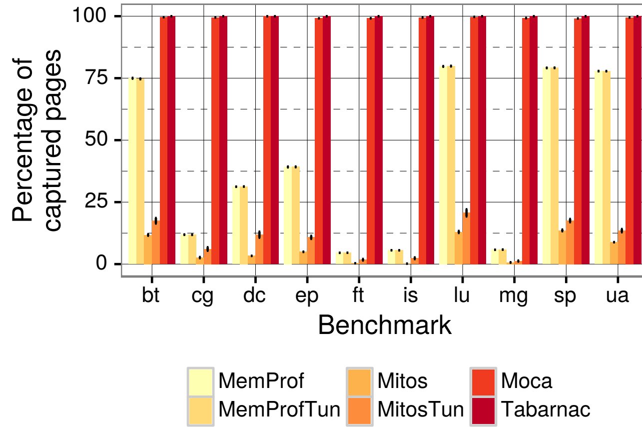
information have been ported out of Pin, as described in Section 5.3. Still it does the same work as the Pin instrumentation but without the weight of Pin therefore has a similar or lighter overhead.

We compare the different tools regarding two metrics, trace *precision* and induced slowdown. Regarding the trace *precision*, the first experiment compares the tools using two criteria, the percentage of captured pages and the number of captured events. We use Tabarnac as a reference to compute the total number of pages accessed by the application because, by design, it traps all the memory accesses to compute the number performed in each page. This metric is representative of the coverage of the memory space, that is the capacity of the tool to outline the whole memory area accessed by the application. Regarding the number of captured events, we present the percentage relative to Moca, as it is the tool that usually provides the more *precise* traces. We define one event as one timestamped access found in the trace file outputted by a tool. According to this definition, Tabarnac does not capture any event as it only keeps one counter per page and per thread without any temporal information. Thus, Tabarnac is excluded from this comparison. The number of captured events is representative of the *precision* of a monitoring tool, its capacity to keep track of all the evolutions of the access patterns during the course of the execution. The idea is that, the more the tool captures events, the less it misses changes in the access patterns. The second experiment compares the slowdown factor of the different tools. All these experiments have been run on each of the NPB on class A.

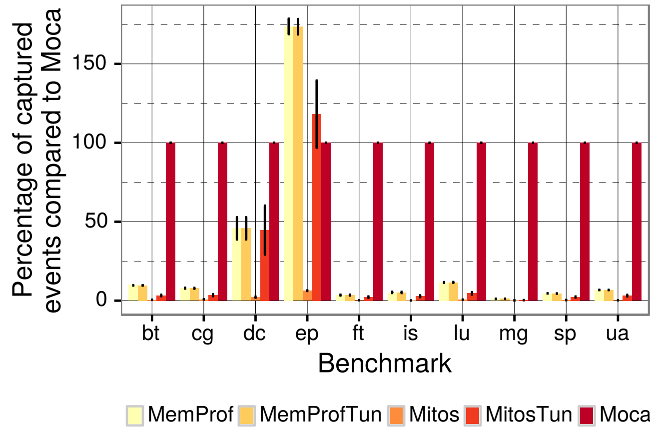
Figure 5.5 presents the results of the precision evaluation for the different tools. The values used for Mitos, MitosTun, Moca and Tabarnac result from runs on `Edel` machines, while MemProf and MemProfTun values result from runs on `StRemi`. We can see on Figure 5.5a that Moca captures almost as many pages as Tabarnac. Regarding their design they should capture as many pages. Nevertheless, there is a slight bump in the number of pages used by applications monitored by Tabarnac due to the Pin instrumentation. Indeed, its JIT instrumentation recompiles the executable on the fly and changes the memory footprint (of the stack, mainly). Thus, we can safely ignore these differences.

Mitos usually collect less than 12.5 % of the pages, adding some fine tuning can almost double this number but it still misses most of the address space. Regarding MemProf, changing the default sampling rate does not seem to have any noticeable impact on the end result. Both MemProf and MemProfTun capture significantly more pages than Mitos and MitosTun. Nevertheless, for half of the studied applications it does not see more than 50 % of the addresses space. Only for `BT`, `LU`, `SP` and `UA`, MemProf manages to capture around 75 % of the accessed pages. This is explained by the fact that all these benchmarks are using uniformly most of their address space, and that many pages are frequently accessed. This is coherent with the fact that MemProf is solely based on instructions sampling and only sees the most accessed pages.

From Figure 5.5b we can see that, as expected, for almost every benchmarks, Moca collects significantly more events than the other tools. The only benchmark for which Moca is not the more *precise* tool is `EP` which is an Embarrassingly Parallel application with very few memory accesses. This outlines the fact that Moca captures events in an uniform way, timed by the monitoring interval. On the contrary, the other tools might capture more events in a few



(a) Percentage of captured pages.



(b) Percentages of events captured (compared to Moca).

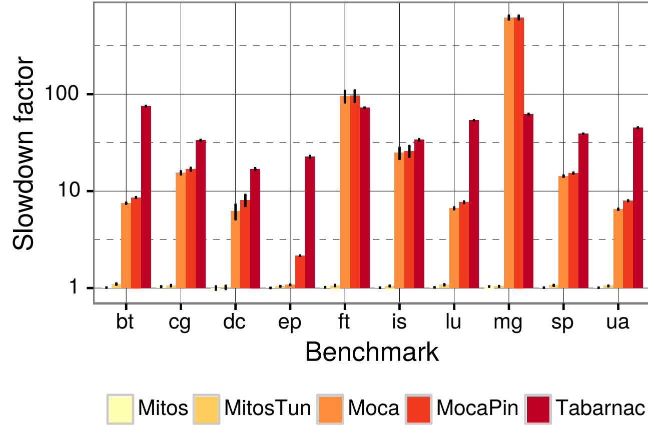
Figure 5.5 – Precision of the traces generated by each tool.

hotspots presents in the application but miss sparse accesses during the rest of the execution. For almost every other benchmarks both Mitos (with or without tunning) and MemProf hardly reach 10 % of the accesses collected by Moca, the only exception is `DC` for which MemProf captures from 25 % to 50 % of the accesses collected by Moca.

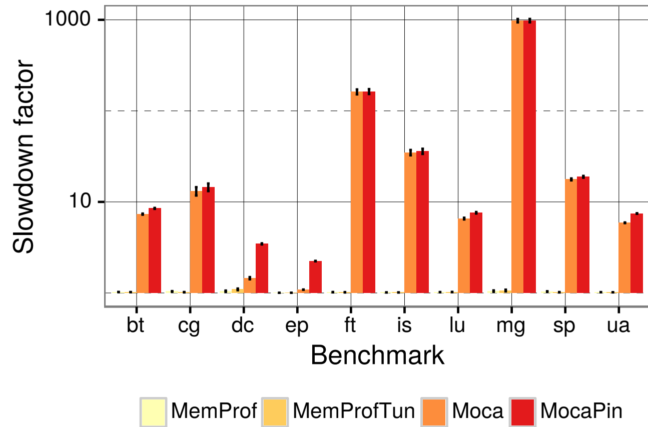
These results prove that most existing tools can miss a considerable part of the address-space while Moca guarantee that it traces covers all the accessed pages. Furthermore they show that Moca is the only existing tool able to provide a trace that is *precise* enough to give an good overview of the memory use of an application. In short, not only our tool provides a *complete* trace at the granularity of the page but it is also significantly more *precise* than the other existing tools.

Figure 5.6 shows for each of the NPB, the slowdown factor when instru-

mented by Moca and the other existing tools on Intel (Figure 5.6a) and AMD (Figure 5.6b) Machines. Notice that the Y-axis is in log scale.



(a) Evaluation on `Edel` (Intel)



(b) Evaluation on `StRemi` (AMD)

Figure 5.6 – Slowdown factor of each tool. Y-axis in log scale.

From Figure 5.6a, we can see that Mitos, MitosTun overhead is almost negligible which is not the case for Moca and Tabarnac. This difference is explained by the results of the previous experiment, as these tools usually collects less than 10 % of the accesses collected by Moca and miss a significant part of the address space.

We can classify the benchmarks into three groups: for `BT`, `CG`, `DC`, `EP`, `LU`, `SP` and `UA`, Moca is significantly faster than Tabarnac. This set of benchmarks is interesting as it is made of varied application profiles as we can see in Table 5.3. Indeed, if `EP` is mostly doing parallel computation with only a few number of memory accesses, `CG` is described as memory intensive, `BT`, `LU` as well as `SP` are linear algebra solvers with regular memory access patterns, and both

UA and DC contain *unstructured computation, parallel I/O and data movement.* Furthermore, DC has a considerable memory footprint as described in Table 5.3. The second group only contains memory intensive benchmarks (FT and IS). For this group, Moca is as good as Tabarnac or a bit faster, probably because the balance between computations and memory accesses hides the overhead of the instrumentation.

For the last benchmark: MG, Moca is significantly slower than Tabarnac. By looking at our experiment logs, we found that MG generates a lot of conflicts in the hash map used by Moca to store false page faults. This issue is caused by applications that perform a very large number of sparse accesses to a large working set. This is not usual as parallel applications are often optimized to make memory accesses as local as possible in order to take advantage of all the levels of the memory hierarchy. Thus, we consider this benchmark as a pathological case. A solution could be to increase the size of this hash map, which is not always possible as memory space in the kernel is limited (and these experiments have been run with the largest hash map we could use). Another easier solution would consist in working on a smaller instance of MG and see if the trace is still useful. Although the results are not presented here, we have run Moca on MG with a smaller size (W) and we have been able to confirm that the performance becomes comparable to Tabarnac in this case.

Figure 5.6b shows the results of the evaluation on the AMD machine (StRemi). On this machine, Moca overhead is quite similar to the one obtained on Edel. MemProf exhibits a slowdown factor comparable to Mitos while providing traces a little more *precise*. Nevertheless, they are still *incomplete* and way less *precise* than Moca traces. Obviously MemProfTun has the same overhead as MemProf as it captures the same amount of data.

Finally, we can see, as expected, that adding one execution with a Pin instrumentation to retrieve data structures information (MocaPin) only adds a small overhead to the whole Moca execution. For several benchmarks this difference is so small that we cannot distinguish it from Moca usual overhead.

5.4.4 Results and discussion

We have tested Moca with various applications and using several parameters. Our experiments show that Moca has a good behavior for a wide range of parameters and helped us defining their default values. Our experiments also show that, with these parameters, Moca provides significantly more precise traces than state of the art tools. Of course, because of this increased precision, Moca is slower than two of these tools, MemProf and Mitos. Nevertheless, compared to the only other tool able to collect a superset of the memory space, Tabarnac, Moca exhibit a decent overhead. Collecting such a superset with MemProf and Mitos, and provide the same guarantee, would require to sample all the memory instructions, which is not possible. At the end of the day, Moca is the only tool able to provide a detailed trace with temporal, spacial and sharing information while providing guarantees about the information lost during the sampling.

5.5 Conclusions

We addressed the issue of memory accesses collection for multithreaded applications. This is a key challenge in high performance computing as memory is often a performance bottleneck. Memory traces can be used at runtime to improve data locality or offline by developers to understand and improve the memory behavior of their applications and, therefore, their performance. For online analysis the trace *precision* is limited by the volume of data that can be analyzed in real time, but for offline usage, highly accurate traces can provides a better understanding of the application memory behavior.

Our first attempt at collecting memory traces, Tabarnac, is designed specifically for NUMA related issue. It relies on a custom memory tracer based on the Pin dynamic binary instrumentation tool which records the number of memory reads and writes performed by all threads for each data structure. The advantage of instrumentation is that it is the most accurate and portable way to generate memory traces. Despite the overhead caused by the instrumentation, Tabarnac is efficient enough to analyze even huge applications in a reasonable time.

We analyzed two parallel applications with Tabarnac: *Ondes3D*, a real life application that simulates seismic waves, and `IS` from the NPB which is known for being memory intensive with a random memory access pattern. For both applications, Tabarnac helped us understand their performance issues. Using this knowledge, we proposed simple code modifications to optimize the memory behavior resulting, for each application, in significant speedups compared to the original version (up to 60% speedup). Improvements were also substantially higher than those provided by automated tools. Yet, Tabarnac traces are not *precise* as they only contains a global overview of the memory sharing patterns without temporal information. Therefore they only enable the observation of a limited number of memory related issues.

Our second memory trace collection tool, Moca addresses this challenge. It collects *precise*, *complete* and *detailed* memory traces. While existing tools rely on *incomplete* hardware sampling to provide such traces at a lower cost, Moca provides a *complete* trace, that contains all the accessed areas, at the granularity of the page. Moreover, Moca traces not only contain all the pages that are accessed during the execution, but also, for each trapped access, temporal, spatial and sharing information: accesses are timestamped and recorded along with their thread number, CPU number, and kind. While Moca works at the page granularity, it stores the exact address of each intercepted accesses. Therefore, it also provides an *incomplete* trace at the granularity of the Byte, similar to traces collected by instructions sampling. Furthermore, Moca is also able to relate accesses to data structures of the application by combining this efficient trace collection system with an examination of the application binary.

Most state of the art tools are relying on hardware technologies such as Intel PEBS or AMD IBS, and embed vendor (or processor) dependent code making them hard to maintain and not portable. On the contrary, Moca is based on page faults interception as well as false page faults injection mechanisms and does not use any architecture dependent code. It can work on any Linux kernel from 3.0 only by loading a module and without any kernel modification.

Several tools use page faults interception to retrieve information about the

memory use. As information provided by only intercepting regular page faults is not always *precise* enough, a few tools also inject false page faults on a regular basis to increase the trace *precision*. To our knowledge, all the existing tools relying on these mechanisms uses the collected data online and, thus, do not have to manage and store a large volume of data. Moca is the first tool able to generate and store *complete* and *precise* memory traces for offline analysis.

We evaluated Moca and Tabarnac by comparing them to two state of the art tools: Mitos (the collection tool from MemAxes) and MemProf. Both of these tools were evaluated with their default parameters and with some fine tunning of our own. For this comparison, we evaluated two criteria: the *precision* of the trace and the runtime overhead. We ran our evaluation on the NPB which are representative of multiple kinds of applications from simple kernels to realistic ones. Our evaluation has exposed the fact that the tools relying on hardware sampling miss a large part of the address space. It has also shown that Moca is able to provide both a *complete* trace at the page granularity and a sampling at the Byte granularity significantly more *precise* than the other sampling based tools. Generating comparable traces using MemProf or Mitos would require to sample more memory instructions than the hardware can. Finally, Moca overhead is more important than the overhead of sampling based tools but usually lower than the one induced by Tabarnac.

The visualization and exploitation of these memory traces is another challenge. Indeed, not only theses traces contains an important volume of data, but they are spread over several dimensions. The most obvious ones are the address space and the time. Yet, the threads responsible for the accesses also represent a dimension of the trace. Moreover, the Central Processing Unit (CPU) location of the access is not necessarily bound to the threads, hence it is one more dimension. Finally, the type of access (read or write, private or shared) is yet another dimension. Several of these five dimension can be observed from different point of view, for instance the address space can be physical or virtual, and the time can be seen either as a discrete sequence of timestamp or as the result of the code. In summary, our traces contain at least five dimensions, therefore designing meaningful and intuitive visualization is far from being trivial.

Chapter VI

Analyzing Fine Grained Memory Traces

Contents

6.1 Interactive visualization of aggregated trace 82
6.1.1 FrameSoc and Ocelotl **83**
6.1.2 Trace Description **85**
6.1.3 Sharing detection **85**
6.1.4 Example **86**
6.1.5 Discussion **89**
6.2 Programmatic exploration 90
6.2.1 Design **91**
6.2.2 Example of visualization **92**
6.3 Conclusions 96

Tabarnac traces contain two parts: informations about the data structures and the actual trace. For small applications with a limited number of data structures, it is relatively easy to present the first kind of information. The actual trace is spread over four dimensions: data structures, threads, pages and type of accesses, however, this last one can only take two values (read or write). Finding a meaningful and comprehensive representation for such data is slightly more complex but it is still doable. Moca traces are way more detailed. Indeed, they contain the same meta data and the actual trace also provides information about time and Central Processing Unit (CPU) location. Therefore, these traces are spread over five dimensions: time, addresses, type of access, thread and CPU location. Furthermore some of these dimensions can be seen from several point of view, for instance we can look either at the virtual address space or physical. As we are not used to visualize things in more than three dimensions, to analyze these traces, we must provide the user with a way to navigate through different representations. Additionally, we need to help the user identify and focus on the important parts.

The contribution presented in this chapter consists in two different methods to analyze Moca traces:

- The first method relies on FrameSoc [Pagano and Marangozova-Martin, 2014], an existing generic trace management tool, and more specifically one plugin called Ocelotl [Dosimont et al., 2014] that provide aggregated views of a trace.
  We have implemented an importer to analyze Moca traces in FrameSoc that is distributed on Github under GPL License:
  https://github.com/dbeniamine/framesoc_importer_moca.
  The proposed visualizations are presented in a Research Report [Beniamine et al., 2015a].

- The second method relies on R, it is an ongoing work, publicly available online at: https://github.com/dbeniamine/Moca_visualization

This chapter is organized as follows: first we present our analysis of Moca traces with FrameSoc and Ocelotl with an example and discuss the limits of this method in Section 6.1. Then we propose a second, more flexible approach based on R in Section 6.2. Finally we present some perspective of improvement of these analysis and our conclusions in Section 6.3.

6.1 Interactive visualization of aggregated trace

As Moca traces are spread over five dimensions and as the address space of an application can be quite large, we need a way to navigate easily in the trace and highlight interesting parts. Consequently, we are looking for a high level tool that is able to highlight potentially interesting parts of the trace. While several trace manager tools such as HPCToolkit [Adhianto et al., 2010] are able to import generic traces, we decided to use FrameSoc [Pagano and Marangozova-Martin, 2014]. This decision was mainly motivated by one of FrameSoc visualization tool, Ocelotl [Dosimont et al., 2014] that is designed specifically to aggregate similar parts of a trace and identify anomalies.

6.1.1 FrameSoc and Ocelotl

FrameSoc is a generic trace management infrastructure, it provides importers to read traces from many different formats. From its point of view a trace consists in five sets:

1. Some meta data about the trace, such as the name of the trace, the application traced, the number of CPUs used, the Operating System (OS) and so on. . .
2. A set of *Event Producers*: which are entities able to produce some *Events*. For classic performance traces the Event producers are the CPUs, threads or processes.
3. A set of *Events Types* used to classify the possible events. MPI function calls and system calls are two usual event types.
4. A set of *Events*, *Variables* and *States* that form the actual trace. For instance in a classic trace, a call to a call to `MPI_Send` could be an event, and a CPU could be in *idle* state after a call to `MPI_Receive`. Variables are used to represent values that evolves as the time passes.
5. A set of *Links* that can be used to represent causality between events, variables and states.

To analyze a trace from an unknown format in FrameSoc, we need to write an importer which is a relatively simple task. Indeed FrameSoc is implemented as an Eclipse plugin, consequently an importer is a small piece of java code that read a trace file, create the sets described above and store them in a database, using FrameSoc Application Programming Interface (API). The main challenge in the writing of an importer is to figure out how to represent the trace in FrameSoc internal model.

FrameSoc provides several functionalities to explore a trace such as filtering events by type, name, focusing on a time frame. Additionally it has a multi view representation which means that several views of the trace can be opened at the same time and synchronized. For instance a user can start inspecting a trace with a Gantt chart, focus on a small part and then look at a pie chart of the event distribution in this subset of the trace. FrameSoc is optimized to make such analysis as smooth as possible.

Ocelotl [Dosimont et al., 2014] is an analysis tool for FrameSoc. This tool is particularly interesting for us as it provides an aggregated overview of a trace. The idea behind Ocelotl is that a trace with too many entities (events or event producers) is not understandable, consequently, it should be analyzed with a *systemic approach*. This means considering the whole trace as a system and finding a macroscopic representation of that system that contains an amount of information understandable by a human. To do so, it uses an aggregation methodology proposed by Lamarache-Perrin [Lamarche-Perrin et al., 2014] and adapted for trace analysis. This methodology cuts the trace into small slices over the two dimensions: time and space (event producers). Then it considers each possible partition, benefiting from the structure enforced on time and space to reduce the number of possibilities. For instance, merging two slices that are not continuous over time is not allowed as it would not be meaningful.

In addition, it uses a parameter $p < [0,1]$ that controls the trade-off between information loss and data reduction and find the optimal partition for this parameters. Once the first visualization is generated, Ocelotl provide the ability to explore the trace (zoom, use FrameSoc filters . . .) and change the p parameter. The usual workflow with Ocelotl is starting with a high p, where the trace is mostly aggregated, zoom on anomalies or interesting parts and decreasing p to understand more precisely what phenomena we are observing.

Figure 6.1 is a screenshot of Ocelotl, showing on the main pane an overview of an example trace, aggregated over time and space (memory addresses). The X-axis represents time and the Y-axis represent the memory space, we can see on the left that the Y-axis is organized hierarchically. Indeed, the blocks on the left of the main page represents a tree where the root represent the whole memory and which is then divided in data structures and continuous allocations. Each of these entities are divided recursively until we reach the page level. This hierarchy is used to reduce the number of possible aggregations, indeed, Ocelotl can only merges complete nodes of the tree. Without any a priori knowledge about the trace application, we can distinguish three phases on this visualization: a short initialization (blue column at the beginning), the main execution with some pattern change around the middle of the execution (dark block in the middle) followed by a pattern change at the two third of the execution. The top right pane shows a summary of the trace, aggregated only over the time. On this summary, the initialization is a bit less visible. Finally, the bottom right pane shows a comprehensive visualization of the tradeoff between information and complexity depending on the value of p. The red (dark) curve shows the information gain and the green (light) one the complexity gain. The current value of p is showed by a blue vertical line, and displayed in a text block under the curve. We can set the p parameter either by clicking the curve or by entering the requested value inside the text block. At this point, the user could zoom in a subpart of the trace by selecting it, or change the parameter p to disaggregate the visualization.

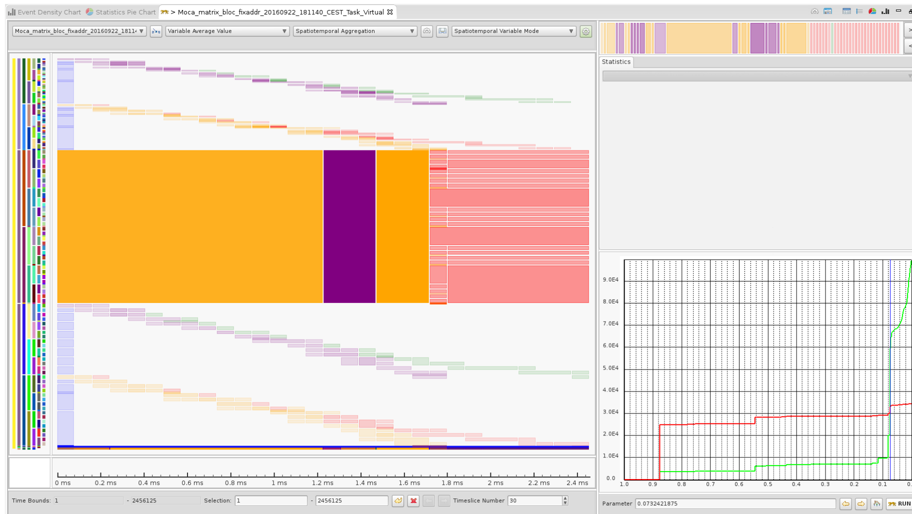


Figure 6.1 – Screenshot of Ocelotl.

6.1.2 Trace Description

As explained earlier, Moca traces are spread over five dimensions while FrameSoc is designed for two dimensional traces (time and event producers). Nevertheless, we can use the *event types* of FrameSoc to represent some of the missing dimensions. To provide different visualizations of the same trace, our importer produces four different FrameSoc traces with different event types. For all the traces, the event producers represent the memory addresses, but the two first are based on virtual addresses while the two others physical addresses.

The more event producer there are, the more partitions Ocelotl must consider to compute the aggregation. Yet, Ocelotl uses the structure of the event producers hierarchy to reduce the number of allowed partitions, indeed it can only merge event producers that have the same parent (and all of them or none). Consequently we can counterbalance the huge number of event producers by creating an artificial hierarchy in the memory. Still, we can build a meaningful hierarchy that also adds some semantic to the trace: to do so we create a virtual *Memory Root* event producer that is the parent of all the event producers. Then, the second level of event producers is composed of the stacks and data structures. All the addresses that are not in these set, are merged if they are contiguous, creating chunks of continuous addresses which are also second level event producers. Then, each subsequent level is obtained by splitting the previous one in two or three parts, until we reach the page level. The pages are the leaves of this artificial memory hierarchy. We could divide the pages in cache lines and keep going until the address granularity, but this would generate way more event producer that what Ocelotl is able to handle.

For both physical and virtual addressing, our importer creates two different traces. In the first type of trace, the events are spread on four event types: `private_read`, `private_write`, `shared_read`, `shared_write`. For the second trace, the event type represent the thread responsible of the access. As a result, for both physical and virtual addresses, we have two views, one representing the detailed usage of the memory by the threads and one global view presenting the sharing patterns and memory access types.

Each access is represented by a variable, whose value is the number of threads involved in the access. Finally, the CPU on which the access occurred is stored as an event parameter.

6.1.3 Sharing detection

Moca does not compute sharing directly but the traces contains enough information to detect shared accesses. Indeed, in Moca traces, each access is timestamped with the begin and end of the chunk to which it belongs, which means that an access can be seen as a time interval during which a thread is using a memory page. We do the sharing detection at the page level as Moca traces are complete at the page granularity. Consequently, we consider that a sharing occurs when two threads access the same page during and intersecting time interval. To compute this sharing, we retrieve the list of accesses for each page. Then we cut the global time interval each time a thread start or stop using a page and mark each access with the number of threads involved in the sharing. Figure 6.2 shows this transformation.

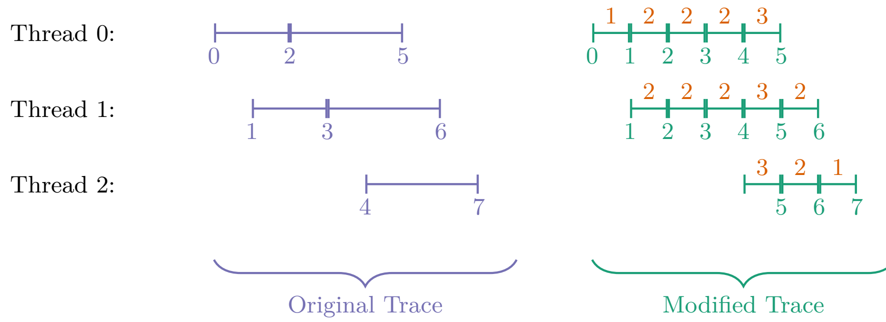


Figure 6.2 – Sharing detection in Moca traces.

6.1.4 Example

To illustrate these visualizations we implemented an extremely naive parallel matrix multiplication. In this example, a first thread does the whole initialization, then create four threads that will do the actual computations. Furthermore, we split the work by cutting the result matrix in four parts, resulting in the distribution of the data structures presented in Figure 6.3.

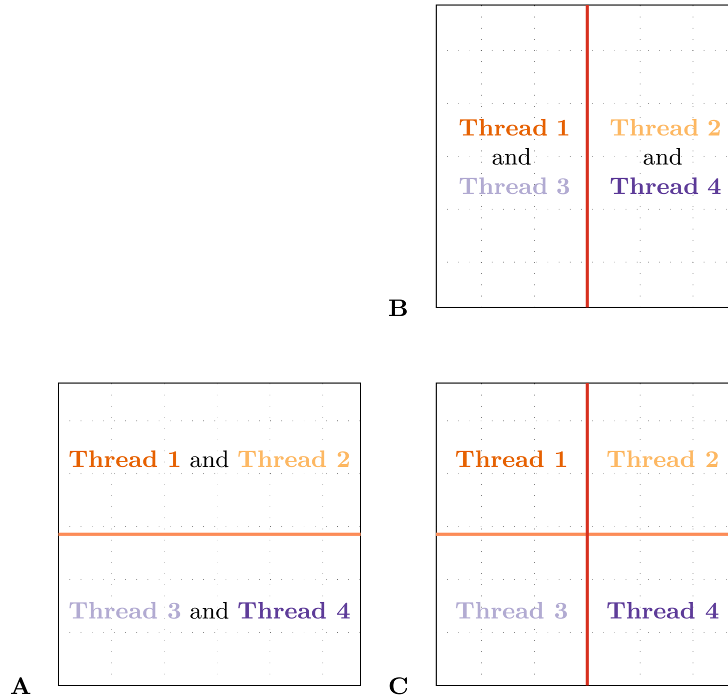


Figure 6.3 – Naive parallel matrix multiplication

Figure 6.4 is a screenshot of Ocelotl visualization by thread of a Moca trace. We can see the hierarchy on the Y-axis, starting with the memory root, then three data structures. The X-axis represent temporal evolution.

From this view, we see clearly that blue accesses occurs only during the ini-

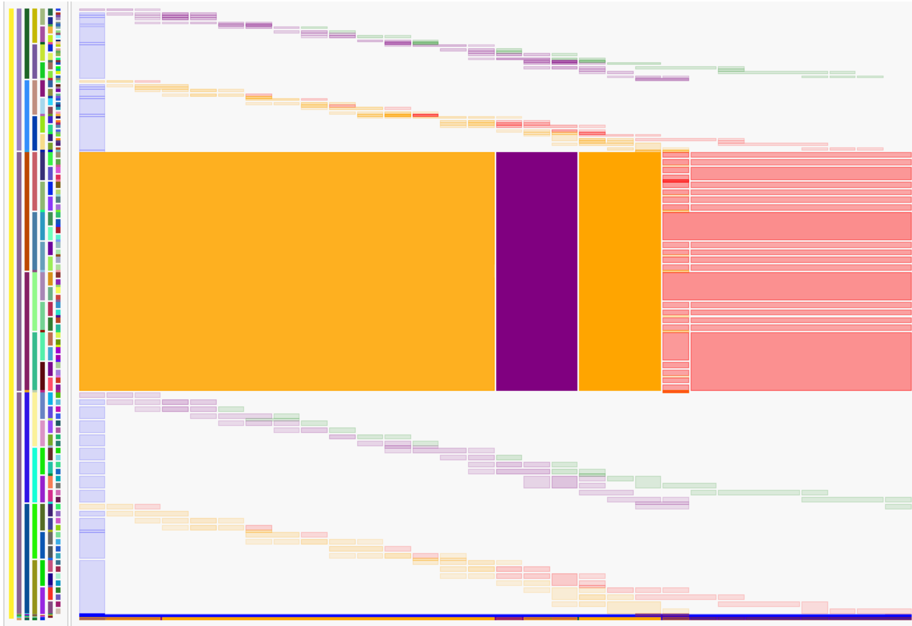


Figure 6.4 – By thread view of the memory usage of a naive parallel matrix multiplication.

tialization phase (vertical light line at the beginning) except for a small data structure on the bottom of the figure (thin dark line). Blue is the color of threads 0, therefore we can see that this is the master thread doing the initialization. Moreover, the master thread also access to some private data (the small structures in the bottom). Among these data, we can find a Process Identifier (PID) array used to wait the end of the slave threads. We can confirm that by filtering the accesses to show only the accesses done by thread 0. Or by zooming on the initialization phase.

During the rest of the execution, two data structures are accessed diagonally over the time, which means linearly. Furthermore, the colors confirms that two threads are sharing each series of accesses. The data structure in the middle is intensively accessed by all the threads. As these accesses are regular, all the accesses are aggregated.

At the two third of the execution, we can see a change of colors in the middle data structure. Additionally, it seems that at the same time the violet (dark) thread has completed his work and does not access the memory anymore. Moreover, just before this threads finished its accesses, the middle structures turns violet (dark) for a small time lapse which means that this thread is responsible for most of theses access at that time. This could be an effect of Linux scheduler that privileged temporarily this thread.

Now let's zoom on the initialization step. The result is shown in Figure 6.5. We can see that, during the initialization phase, only the master thread is working. We can identify a three diagonal patterns happening at the same time, it correspond to the matrix initialization. The private data structures in the bottom also appears during the initialization. Finally, we see that as soon as

the thread 0 has finished to initialize the data structure, the other four threads start working.

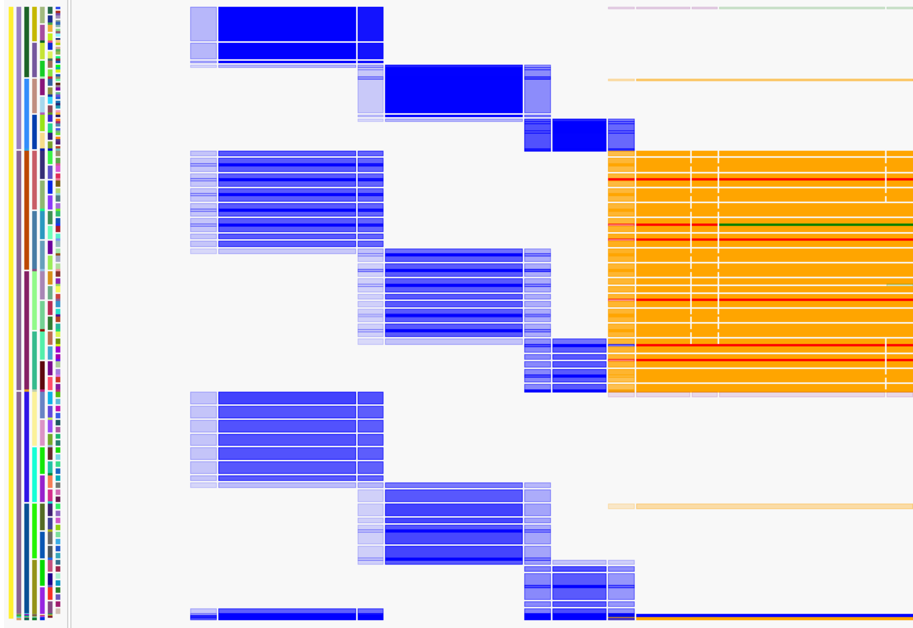


Figure 6.5 – By thread view of the memory usage of a naive parallel matrix multiplication, initialization.

In the previous view there was one access type per thread. The global view, designed to highlight sharing only, provides four event types independently from the number of threads. Therefore, it is easier to identify sharing patterns with this view. Figure 6.6 shows this visualization of our example traces. We can see that the trace looks like the previous one except that the order of the data structures is different, which is due to an artifact of the importer. Blue accesses are privates and reds are shared. Dark colors are for writes while light ones means reads. From this view, we can see that most accesses seems to be reads and except for the matrix B (on the bottom) they are all private. Afterwards, the visualization is aggregated. At this point it is interesting to zoom in the middle of the execution and disagregate the trace as much as possible.

By focusing on the middle of the execution and setting p to zero, we obtain the Figure 6.7. It is important to note that the trace is still aggregated due to the microscopic model of the trace. This aggregation explains the fact that we still see small, regular, blocks of accesses. We cannot identify a clear pattern on the bottom matrix, however, we can see a few private (blue) accesses appearing from time to time in this data structure. The access on this matrix seem dense and not designed to fit in a cache. This density of accesses is coherent with the behavior expected. Indeed the matrix B is accessed column first by all the threads. Due to the representation of 2D matrices in `C`, each access made by each thread is separated by sz doubles from the next one. Hence Ocelotl groups almost everything in huge chunks of accesses on all the matrix. We can also see in this figure that shared (orange) accesses appears regularly on the two other

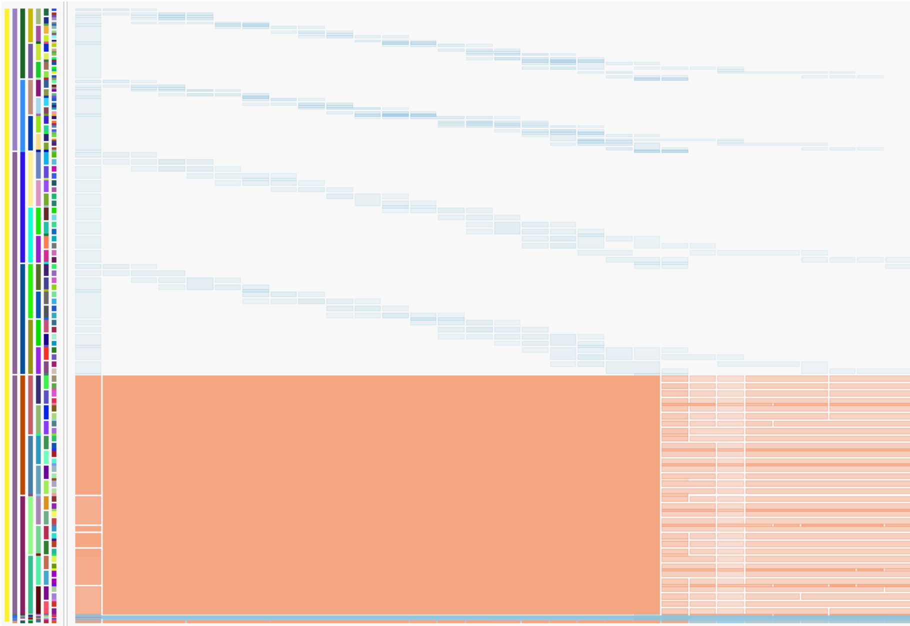


Figure 6.6 – Sharing view of the memory usage of a naive parallel matrix multiplication.

data structures (matrix A and C) which is also coherent with the expected behavior.

6.1.5 Discussion

Ocelotl enabled visualizing Moca traces and helped identifying inefficient patterns on a test application. We visualized Moca traces from two different point of view, the first one show the memory accesses by threads and helps understanding the division of the work. The second shows the access types independently from the threads it helps understanding how the memory is globally used.

However even for this extremely simple benchmark we started to see some scalability issues. The first limit comes from the fact that FrameSoc is designed for a small number of event producers. Indeed FrameSoc event producers usually represents threads, CPUs or nodes of a distributed system with at worst a few thousands of them. At the opposite, our small example we already had 20000 event producers for only 78 Mb of memory usage, a trace using 1 Gb would require more than 250000 event producers. We can mitigate the impact of the number of event producers on Ocelotl by organizing them in an artificial hierarchy but this only reduces the computation time of Ocelotl and makes the visualization even more aggregated. Filtering the trace based on the event producers is extremely slow with such traces in FrameSoc. This issue also impact other FrameSoc tools, and makes it almost impossible to analyze Moca traces on tools which rely on filtering. The second issue comes from the fact that we had to create several FrameSoc traces to represent all the information contained in Moca traces. As a result, changing analysis point of view means changing the trace, hence loosing all filtering already computed and all caches that speeds

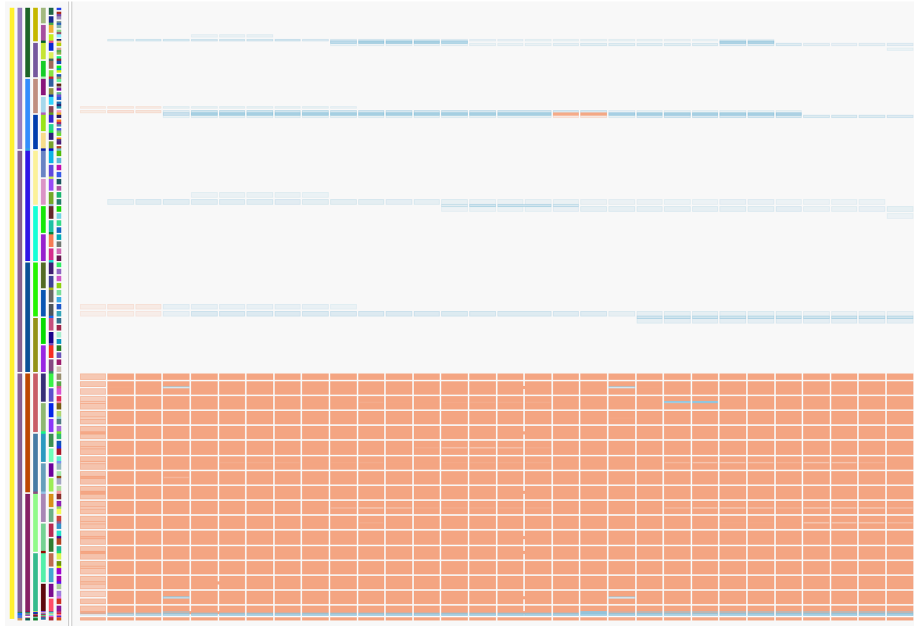


Figure 6.7 – Sharing view of the memory usage of a naive parallel matrix multiplication, computing phase.

Ocelotl up. Therefore it is a process extremely slow that breaks the analysis workflow. While it would have been technically possible to merge the threads view and the sharing view by creating complex event types holding information about access type, sharing and the thread responsible for it, this would result on a huge amount of event types. Such a trace would results in two issues: first the microscopic description of the trace would be more complex which might slow Ocelotl down, and second it would make filtering by type extremely complicated. Indeed there is no way to do filtering using regular expression or any programmatic way in FrameSoc, thus, on a huge trace the user would have to do tens or hundreds of clicks and might do some errors to switch from point of view. Finally, with these traces it is not possible to have information about the data structure (their size and number of acesses), to do so we would have to create yet another trace during the importation.

To conclude, visualizing our traces with Ocelotl proved that memory traces are helpful to identify performance issues. Nevertheless, we identify several scalability issues in our analysis workflow due to the fact that generic trace analysis tools are not designed for the specificities of memory traces. Consequently, a more flexible approach that ease switching from point of view and filtering would be more efficient to analyze our traces.

6.2 Programmatic exploration

The main drawback of using a generic visualization tool for analyzing Moca traces come from the difficulty to switch the perspective from which we visualize the data. More precisely, this limitation comes from the static representation

of traces as a two dimensional entities used in most tools. This is problematic for Moca traces because not only they are spread over five dimensions, but these dimensions are related to each other. Indeed, the placement of threads on the CPUs is not necessarily fixed but it matters and can impact the performance. Furthermore, this placement must be analyzed in relation to the placement of memory pages on the physical memory and on the underlying hierarchy. As a result, projecting such traces on two dimension is complex and different projections should be analyzed and correlated to understand the memory behavior of an application. Hence, a more programmatic approach may enable the analyst to work closer to the data and ease this point of view switch.

In R, data are usually stored in huge dataframes, each row of a dataframe represent one observation and each column represent one dimension of this observation. Consequently, in R, changing the analysis point of view for a set of data only means looking at another column of the dataframe which is made simple by its formalism. While this representation has a significant memory footprint, R is designed and optimized to do such analysis and to correlate different dimensions of a set of observations. For these reason, we analyzed several traces with R. For more reproducibility, we have stored the evolution of our work in an Org-mode labbook, as described in [Stanisic, 2015, Chapter 4, p 54], available online at: `https://github.com/dbeniamine/Moca_visualization`.

While this approach is extremely flexible, it is not user friendly. Indeed it requires to know how to write efficient R code, how to use Org-mode and Emacs and to read the labbook before doing an analysis. Anyway, after a few analysis we obtained a basic procedure to analyze traces (parsing, transforming data, showing some generic visualization) which we can adapt at each step to actual the trace. For instance, we might want to ignore some parts of the trace very soon to focus on some data structures, or after analyzing one plot we might think of another representation of data that might be meaningful in this case.

6.2.1 Design

Our analysis always starts with a typical pipeline that we can easily adapt to the specificities of the analyzed trace:

1. Parsing: reading Moca traces (to which we have applied the sharing detection written for FrameSoc and depicted in Figure 6.2) and storing them in R dataframes. At the end of this step, we have two dataframes, the main one contains all the accesses, and the other one the list of data structures.

2. Creating simplified data frames: At this point, the main dataframe contains a set of accesses where shared access appears one time for each thread involved in the sharing as described in Figure 6.2. We can aggregate all theses accesses, reducing the size of the main dataframe.

3. Retrieving the mapping between structures and pages: this step is the most costly one, but can be speeded up by several means:

    - At the end of the parsing step, we can reduce the interesting address space, usually we take the minimum and maximum addresses that are inside a data structure and filter out all the accesses that are not in this interval.

- By sorting both dataframes (accesses and data structures) by address, we can retrieve this mapping while going through each data frame only one time.

4. First set of predefined plots: we can present, using processed dataframes, some predefined plots. This first set is inspired from Tabarnac plots and show the size of data structures, the number of accesses, and amount of sharing.

5. Filtering: at this step we can easily identify data structures that are rarely used and might not have a significant impact on the application performance, we filter these data structures out to focus on the most important ones.

6. Second set of predefined plots: for all the data structures that are left we visualize the memory accesses over the time depending on their type. For these plots, the color represent either the number of accesses or the number of threads involved in the access.

At any step of the pipeline, it is possible to do more filtering using our knowledge of the analyzed application to speed up the process. Once the predefined plots are obtained, we can easily navigate in them using R selection operators to do complex filtering or zoom on a part of the trace. Moreover it is possible to design any other visualization that might be interesting.

6.2.2 Example of visualization

We illustrate our visualization with the *dgetrf* kernel from the fflas-ffpack [group, 2016] compiled against the OpenBlas [Chothia et al., 2016]. This trace was collected on a machine from the Edel cluster which hardware was presented in Table 5.1.

Figure 6.8 shows for each data structure[1] the number of accesses per page over the time. Furthermore we differentiate four types of access: `PrivateRead`, `PrivateWrites`, `SharedRead` and `SharedWrites`.

For each structures, some accesses seemed to appear private and shared at the same time which is not possible. Nevertheless, by zooming on a small part of the execution were able to confirm that those access are interleaved and never of both types at the same time. This means that several threads are working on data very close and often do actually share some pages.

From this visualization we can see that two of the stacks (1252 and 1250) are always used privately. Furthermore the data structure malloc#5 seems to be used mostly privately and only very rarely read in a shared way. The name malloc#N means that it is the Nth call to malloc intercepted by the library. These three structures seems also to be accessed mostly linearly, hence they should not be subject to memory optimization.

At this point, it is interesting to ignore these three data structures and focus on the others, we can do this with the simple line of code displayed in Listing 6.1.

These three structures present different and interesting memory access patterns that are presented in Figure 6.9. First the malloc#1 structure is very small

[1] A few, almost unused, data structures have been filtered out to make the image more readable.

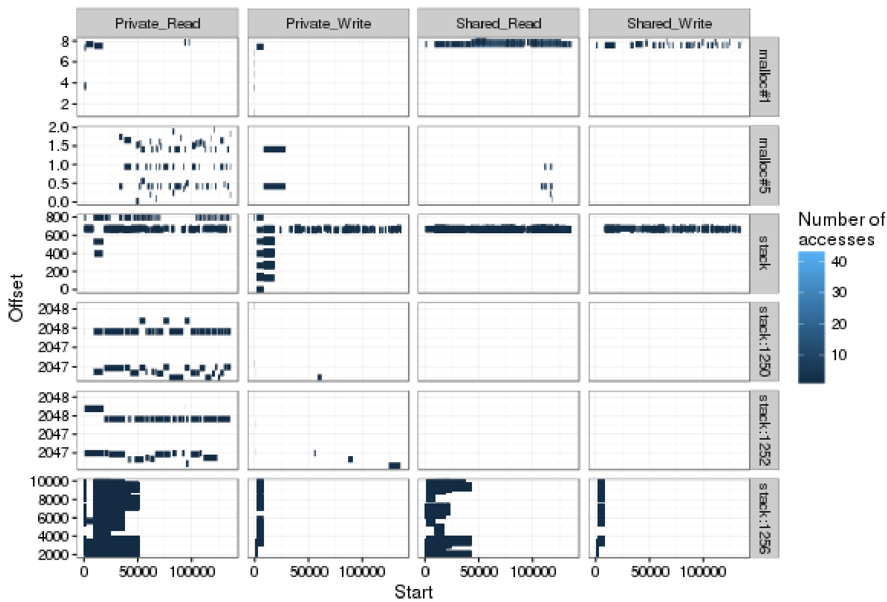


Figure 6.8 – Visualization of the memory access of the dgetrf kernel from ffplas-ffpack.

Listing 6.1 R code to focus on interesting data structures.

```
1  s <- r[r$Structure %in%
2          c("malloc#1", "stack", "stack:1256"),]
```

and accessed intensively in a shared way, during all the execution and always on the same page. We can presume that this data structure contains information about the threads status that must be updated quite often, maybe it is used for thread scheduling. Second the stack 1256 is only used during the first third of the execution and written only at the beginning and in parallel. For a generic data structure, this parallel initialization could probably have been designed to distribute first touch on the Non-Uniform Memory Access (NUMA) nodes of the machine, still on a stack it seems quite unusual. Last but not least, a small part of the main stack is accessed in read and write mode and in a shared way during all the execution. This pattern means that threads are probably organized in a master / slave way, where the master thread allocates data in its stack (not a dynamic allocation). This might be problematic on NUMA machines as the stack is usually used to store private data, thus not trivial to explicitly allocate a part of it on a chosen remote node. It seems interesting to zoom on the beginning of the execution to check if the initialization of this part of the stack is correctly spread among the thread or not. If not, Linux will allocate each page on the memory bank of the master thread independently of the repartition of the data between the threads.

Figure 6.10 shows the memory accesses occurring inside `stack:1256` during the first par of the execution. The color indicates the number of threads involved

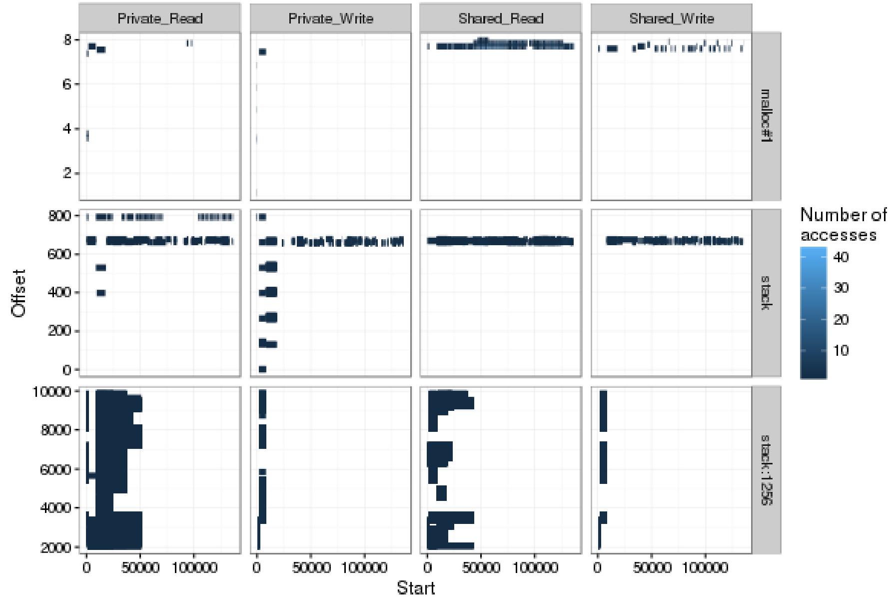


Figure 6.9 – Visualization of the memory access of the dgetrf kernel from ffplas-ffpack, zoom on the three interesting data structures.

in the memory access. The first thing we can notice in this figure is that the data structure seems to be read before being written. When a page that is not mapped to the memory is read before writing it, Linux does not map it but reads a page full of zeroes. Consequently, this behavior should not impact the first touch. Then, it seems that the first writes are both private and shared. As a memory access cannot be both shared an private, that means these accesses occurs in parallel and some parts are accessed by two or three threads at a time while some others stays private. We can confirm this behavior by zooming even more in the initialization.

By zooming even more, we obtain Figure 6.11 which shows that access are indeed not shared and private at the same time. Furthermore it confirms that the data in this stack are read before being written.

After discussing these results with some of the developers of the fflas-ffpack, it appears that the observed patterns can be due to the OpenBlas library and might be complex to improve. Yet an interesting thing to do would be to compare these traces, with the trace of the same kernel but compiled against the Intel Math Kernel Library (Intel MKL). Indeed, there are some performance differences between the two library that are hard to explain with traditional profiling tools as the Intel MKL code is proprietary. Therefore, memory traces might help understanding the underlying algorithms.

In the end, this approach ease the exploration of Moca traces, provided that the user know a minimum of R code and that read our labbook. This is more complicated than using FrameSoc and Ocelotl. Additionally, we had some troubles to analyze some traces from Lulesh and Sofa. Indeed these applications generates thousands of call to the malloc functions (hundreds of thousands for Lulesh), probably to create trees. Hence, retrieving the mapping page to data

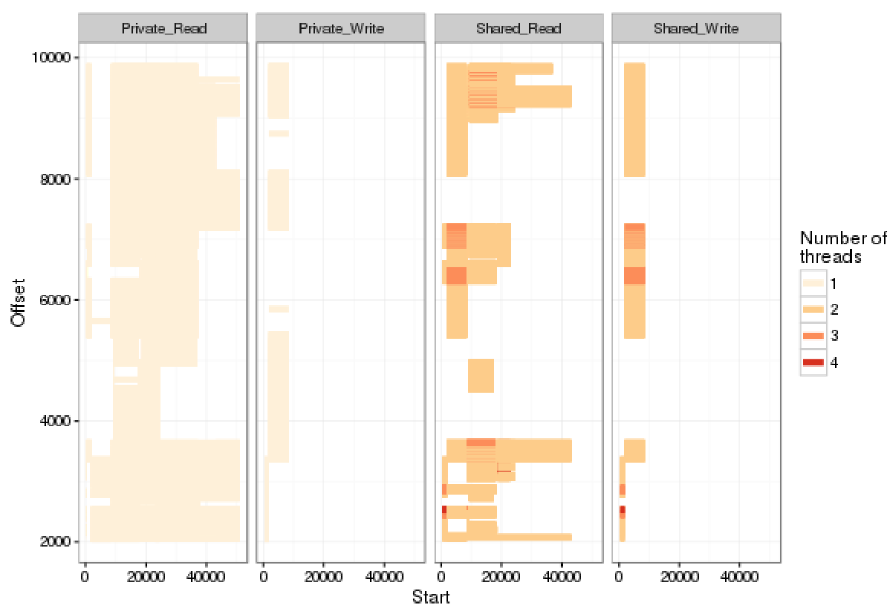


Figure 6.10 – Visualization of the sharing patterns of the dgetrf kernel from ffplas-ffpack, zoom on the initialization of `stack:1256`.

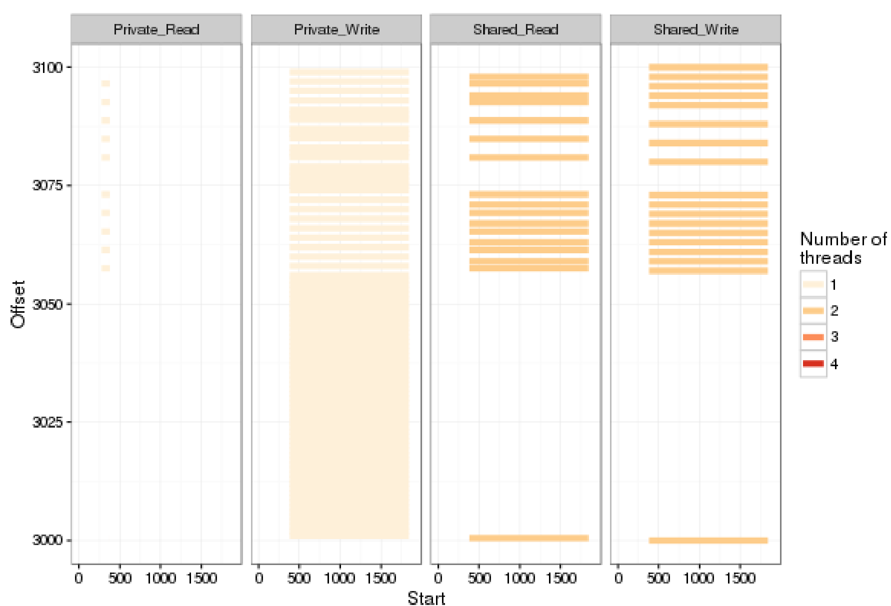


Figure 6.11 – Visualization of the sharing patterns of the dgetrf kernel from ffplas-ffpack, zoom on a part of the initialization of `stack:1256`.

structure was extremely slow or nearly impossible without merging some data structures. In the end it appears that we need to add a priori knowledge of the developers in the parsing step to merge data structures.

6.3 Conclusions

Visualizing Moca traces is a complex task. Indeed these traces are spread over five dimensions: time, addresses (virtual or physical), threads, CPU location and access type. Furthermore, the address space is quite large and we are mostly interested in some specific patterns. Therefore, we first used Ocelotl, a part of FrameSoc infrastructure, to analyze these traces. This tool is designed to aggregate traces in a meaningful way, trying to provide a good trade-off between information lost and data reduction. Importing Moca traces inside FrameSoc is interesting as it provides easy filtering and zoom on the traces. Still, FrameSoc considers that a trace has two dimensions: time and event producers, it is hard to represent the complexity of Moca traces in it. FrameSoc event types could be used to represent this complexity. In Ocelotl event types are represented as a color, if the event types represent several dimensions, it is impossible to interpret the colors of a visualization. Consequently, if a FrameSoc trace contains all the information, we must do some filtering by event types to be able to interpret it. In FrameSoc, the only way to filter a trace by event type is to select individually some types by clicking on them. As a result such a trace would not be usable and it is simpler to generate several distinct FrameSoc traces while importing one Moca trace. The main drawback of this workaround is that switching traces inside FrameSoc means loosing all zooms and filters along with the caches and the results of the aggregated views computed by Ocelotl. In the end this approach enable the visualization of small traces, yet the cost of changing the perspective is too high and makes the interaction too slow to be usable.

Our second approach to analyze these trace was more programmatic, we used R and saved all our attempts in a labbook. Using a labbook is particularly appropriate for such workflows as it is designed to reuse easily chunks of code and adapt them. Moreover it is also easy with a labbook to link a plot generate with a previous trace and, thus, compare two traces. While this is less user friendly, R is a powerful tool and it enables more complex visualization. Moreover, it is optimized to analyze large data sets spreed over several dimensions and provides powerful selection operators that can be used to do conditional zoom and filtering. Using R, we have provided several meaningful plots that can be used to start a memory trace analysis. These visualization could be used to compare the memory access patterns of applications in order to understand the underlying algorithm of proprietary programs such as the Intel MKL.

Another approach that have not been studied during this thesis would be to analyze traces automatically. Such analysis would have to detect memory patterns and possibly and highlight parts of code that should be improved. A memory pattern is an interaction between threads inside a memory area over a short laps of time. Yet, defining such a pattern in a more specific way is a hard task.

For any approach, it appears that the developers knowledge is useful to focus the analysis on the interesting parts. Therefore we need a way to use this

knowledge during the analysis and as soon as possible to reduce the amount of data to analyze. Nevertheless, we have seen in Chapter 2 that they might not know all the sources of performance issues. In the end, this knowledge can be used to focus the analysis but it is important to also have a global visualization of the trace, to spot issues that would have been missed by the developers. Hence the two approaches described in this chapter can be used in a complementary way.

Chapter VII

Conclusions and perspectives

Contents

7.1 Contributions **100**
7.2 Perspectives **101**

In this thesis, we have addressed the issue of memory performance analysis. It has been motivated by the fact that Central Processing Units (CPUs) are more and more parallel and their memory and caches are organized hierarchically. Therefore, writing efficient code requires to consider this architecture, and is complex even for High Performance Computing (HPC) specialists.

In Chapter 2, we presented a case study on the performance analysis of Sofa, a physical simulation tool. This case study highlighted the fact that generic performance analysis tools can help finding memory related issues, but are not sufficient to clearly understand the nature of theses issues in order to fix them. This is due to the fact most performance analysis tools focus on the CPU point of view. Indeed, they consider the memory as a monolithic entity, missing information on how the accesses are organized inside it.

This thesis present several experiments, for each of them, we used a well defined methodology described in Chapter 2, in order to ease reproducibility. All the files required to reproduce each step of each experiment presented in this thesis are available online.

Analyzing the memory behavior of an application raises two challenges: the first one is to collect a trace complete and precise enough to contain memory patterns. This is challenging as there is no hardware designed specifically for memory analysis comparable to the performance counters for CPUs traces. The second challenge is to provide a comprehensive visualization of the memory traces which are spread over five dimensions: time, address space, threads, CPU location and access types. A few tools were designed to analyze performance from the memory point of view, yet they rely on instruction sampling, which is a hardware assisted mechanism that enable interception of some instruction at a defined frequency. The limit of instruction sampling based tools is that they miss a significant part of the execution and therefore are not able to display memory patterns or to give a global overview of the memory sharing. As explained in Chapter 3, these patterns can have a significant impact on the performance. Therefore we consider that the existing memory analysis tools are not sufficient.

7.1 Contributions

We proposed two different tools to address the challenge of memory performance analysis. The first tool, called Tabarnac and presented in Chapter 4, is based on an existing binary instrumentation, which relies on Pin, an instrumentation library developed by Intel. We improved this instrumentation to add contextual information allowing to determine on which data structure the memory accesses occurred. Furthermore we designed several comprehensive visualizations to interpret Tabarnac traces. Finally we evaluated the overhead of Tabarnac and used the knowledge acquired thanks to this tools to improve the performance of two benchmarks, resulting on 20 % performance gain on a well studied benchmark. This work was published at Visual Performance Analysis (VPA) 2015 a Super Computing workshop [Beniamine et al., 2015b], and is the result of a collaboration with M. Diener and P.O.A Navaux from the GPPD of the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. Moreover Tabarnac is distributed as a free software under the GPL license: https://github.com/dbeniamine/Tabarnac. This provides sufficient traces to understand and improve the global sharing pattern of an application. Nev-

ertheless, more precise traces are required to understand temporal and complex patterns.

Our second tool, Moca, which is presented in Chapter 5, is an efficient fine grain memory trace collection system. This tool relies on a Linux kernel module that we implemented. It collects memory accesses by intercepting page faults at the operating system level. As page fault does not occurs frequently it also injects false page faults frequently to increase the number of intercepted accesses. We ran an extensive experimental evaluation of Moca, comparing it to two state of the art memory performance analysis tools and Tabarnac. We compared these tools in terms of overhead, trace precision, and completeness. This work is the subject of two Inria research reports [Beniamine et al., 2015a, Beniamine and Huard, 2016] and has been submitted at Cluster, Cloud and Grid Computing (CCGRID) 2017. As the previous tool, Moca is distributed under the GPL license: https://github.com/dbeniamine/Moca.

As Moca traces are more complex than the one from Tabarnac, we do not provide visualizations with the collection system. Nevertheless, we proposed two different techniques to analyze Moca traces, which are presented in Chapter 6. The first technique relies FrameSoc a generic trace management framework. More precisely it uses one of FrameSoc tools called Ocelotl. This tool is designed to aggregate traces based on a model of the trace, highlighting anomalies and pattern changes. The importer required to read Moca traces in FrameSoc is published as free software:
https://github.com/dbeniamine/framesoc_importer_moca.
With this tool we were able to visualize several inefficient pattern on a test application. However we encountered some scalability limits with this tool. Consequently we proposed a second approach to analyze Moca traces, based on a programmatic exploration of the trace using R. We analyzed several programs and were able to visualize memory patterns. Our analysis are stored on a labbook publicly available online for reproducibility purpose:
https://github.com/dbeniamine/Moca_visualization.

After these analysis, it appears that Moca traces are precise enough to identify temporal memory patterns. Moreover they are sufficiently small to be analyzed. Nevertheless, it seems that the sampling rate and the granularity of Moca might be too high to detect clearly very fine grain patterns such as false sharing on a few lines of caches.

7.2 Perspectives

Our contextual library used in both Moca and Tabarnac is extremely useful to understand which data structure inefficient memory patterns occurred in. However, this library could be improved by two means.

The first one would be to take into account the lifetime of data structures. Indeed, our library does not handle data structures suppression and reallocation. This could lead to erroneous interpretation in the analysis of complex applications that uses many temporary data structures. Adding temporal information in this library is not trivial: Indeed, we do not run Moca and the library at the same time, to avoid tracing memory accesses done inside the library. Therefore the execution time of both runs are different and we would have to synchronize them after tracing. This could be done by generating the

call tree of the application and using it as a temporal indicator. Furthermore, retrieving the mapping addresses to data structure will be more complex with this temporal information.

A second improvement that could be done to this library would be to identify complex data structures such as trees and lists. We might be able to do so by looking at the addresses accessed right after and before allocating a data structure. Nevertheless, this approach seems heuristic and requires to keep a huge amount of data online. Another approach could rely on the developer knowledge and provide some callbacks to annotate allocations and post process them after the trace collection.

While Moca traces enables visualization of memory patterns, it sometimes is hard to associate these patterns with some code a posteriori. At this point we could use the developers knowledge to annotate their code before tracing it to highlight the data structures and parts of code that seems inefficient to them and guide the analysis. However, as we saw in Chapter 2, the developers knowledge can help the analysis but they might miss some hotspots. Therefore, this knowledge should not be used to filtrate traces during the collection step, but to guide the exploration and interpretation during the analysis step.

Moca is not able to detect extremely fine grain patterns such as false sharing on a few lines of caches. However, the pages where such pattern occurs while appears as hotspots in Moca traces. Thus, it would be interesting to build an extremely fine grain collection traces that focuses on small parts of the execution, identified by a first analysis, to visualize these patterns.

Finally it would be interesting to use memory traces to understand proprietary code. More specifically, it is sometimes hard to understand the performance of some kernels of the Intel Math Kernel Library (Intel MKL) as its code is kept secret. Comparing the memory patterns of the kernels from this library to equivalent free kernels might help to understand the differences of performance and to improve those free kernels.

A longer term perspective would be to build a tool similar to Moca for recent memory oriented co-processors architectures, such as the Intel Xeon Phi, or for Graphical Processing Units (GPUs). This would require to identify a mechanism that can be used to collect memory traces which is far from being trivial as we have far less control on these architectures than on general purpose CPUs.

Contents

Acknowledgments iii

Abstract vii

Résumé ix

Résumé étendu xi

Outline xvii

I Introduction 3

1.1 Contributions 6
1.1.1 Global overview of the memory sharing patterns 6
1.1.2 Fine grain memory traces collection 6
1.1.3 Fine grain memory traces analysis 7
1.2 Thesis organization 7

II Case Study 11

2.1 Motivations 12
2.1.1 SOFA: a physical simulation framework 12
2.1.2 Previous efforts toward SOFA parallelization 14
2.2 Profiling tools 15
2.3 Experimental methodology 16
2.3.1 Reproducible research 16
2.3.2 Experimental workflow 18

2.3.3 Methodology 20
Construction of an experimental plan 20
Automation and documentation 21
Distribution 23
2.4 SOFA Analysis 25
2.4.1 Experimental plan 25
2.4.2 Results and discussion 26

III Memory Performance Analysis 31

3.1 Architectural considerations 32
3.1.1 Caches 32
Cache lines and alignment 32
Cache management policies 33
A naive example 34
Memory caches and parallelism 35
3.1.2 Memory hierarchy 37
3.2 Existing tools 39
3.2.1 Memory traces collection 40
3.2.2 Memory traces analysis 41
3.3 Conclusions 43

IV Collecting and Analyzing Global Memory Traces 45

4.1 Design 46
4.1.1 Trace collection 46
4.1.2 Ease of use and portability 47
4.1.3 Visualization 48
4.2 Experimental validation 50
4.2.1 Methodology 50
4.2.2 Ondes3D 51
4.2.3 The `IS` benchmark 52
4.2.4 Tracing overhead 56
4.3 Results and discussion 57

V Collecting Fine Grain Memory Traces 61

5.1 Moca components 62
5.2 Background knowledge 63
5.3 Design 63
5.3.1 Page faults interception and injection 64
5.3.2 Internal design 65
5.4 Experimental validation 70
5.4.1 Methodology 70
5.4.2 Moca validation 72
5.4.3 Comparison to other memory trace collection tools 72
5.4.4 Results and discussion 77
5.5 Conclusions 78

VI Analyzing Fine Grained Memory Traces 81

6.1 Interactive visualization of aggregated trace 82
6.1.1 FrameSoc and Ocelotl 83

6.1.2 Trace Description 85
6.1.3 Sharing detection 85
6.1.4 Example 86
6.1.5 Discussion 89
6.2 Programmatic exploration 90
6.2.1 Design 91
6.2.2 Example of visualization 92
6.3 Conclusions 96

VII Conclusions and perspectives 99

7.1 Contributions 100
7.2 Perspectives 101

Contents 105

List of Figures 109

List of Tables 113

Acronyms 115

Glossary 119

Bibliography 125

List of Figures

2.1 The simulation loop. 12
2.2 Example of SOFA scene graph. 13
2.3 Experimental workflow. 19
2.4 SOFA likwid results. 27

3.1 Example of bad alignment. 33
3.2 Example of non linear memory accesses. 35
3.3 Topology of a quad core parallel machine. 36
3.4 Example of false sharing. 37
3.5 Topology of a 32 cores NUMA machine. 38
3.6 Screenshot from MemAxes. 42

4.1 Global views of the memory usage. 48
4.2 Per structure view of the memory usage. 49
4.3 Access distribution and first-touch for structure `vz0` from `Ondes3D`. 52
4.4 Speedup for `Ondes3D`. 53
4.5 Original memory access distribution for `IS`. 54
4.6 Fair distribution of the pages of `key_buff2` among the threads. . 55
4.7 Modified memory access distribution for `IS`. 56
4.8 Speedup for `IS`. 57
4.9 Tabarnac's instrumentation overhead. 58

5.1 Accesses done by two threads on a set of pages, captured by instruction sampling tools and by Moca. 65
5.2 Interactions betweens Moca and Linux. 67
5.3 Flow chart of Moca's page faults 69
5.4 Influence of Moca wakeup intervals. 73
5.5 Precision of the traces generated by each tool. 75

5.6 Slowdown factor of each tool. 76

6.1 Screenshot of Ocelotl. 84
6.2 Sharing detection in Moca traces. 86
6.3 Naive parallel matrix multiplication 86
6.4 By thread view of the memory usage of a naive parallel matrix multiplication. 87
6.5 By thread view of the memory usage of a naive parallel matrix multiplication, initialization. 88
6.6 Sharing view of the memory usage of a naive parallel matrix multiplication. 89
6.7 Sharing view of the memory usage of a naive parallel matrix multiplication, computing phase. 90
6.8 Visualization of the memory access of the dgetrf kernel from ffplas-ffpack. 93
6.9 Visualization of the memory access of the dgetrf kernel from ffplas-ffpack, zoom on the three interesting data structures. . . . 94
6.10 Visualization of the sharing patterns of the dgetrf kernel from ffplas-ffpack, zoom on the initialization of `stack:1256`. 95
6.11 Visualization of the sharing patterns of the dgetrf kernel from ffplas-ffpack, zoom on a part of the initialization of `stack:1256`. 95

List of Tables

2.1 Hardware and software configuration of Naskapi. 26

3.1 Approximate access latency depending on the memory hierarchy level. 39

4.1 Hardware and software configuration of the evaluation systems for Tabarnac. 50

5.1 Hardware and software configuration of the evaluation systems for Moca. 70
5.2 Comparison of different memory traces tools. 71
5.3 Description of the NAS Parallel Benchmarks. 71

List of Algorithms and codes

2.1 Logging experimental informations. 22
2.2 Dependent runs. 23
2.3 Independent runs. 23
2.4 Execution of a run. 24

4.1 Handling of memory accesses by Tabarnac. 47
4.2 A simple allocation. 47
4.3 Original IS code. 53
4.4 Modified IS code. 54

5.1 Monitoring thread algorithm 68
5.2 Moca Logging daemon algorithm. 68

6.1 R code to focus on interesting data structures. 93

Acronyms

API
: Application Programming Interface. 14, 36, 81

CCGRID
: Cluster, Cloud and Grid Computing. xiv, 7, 60, 99

CPU
: Central Processing Unit. 4, 5, 12–14, 30, 31, 33, 35, 77, 80, 81, 83, 87, 89, 94, 98, 100

CSV
: Comma-Separated Values. 17, 45, 64, 69

DOI
: Digital Object Identifier. 22

GPL
: General Public License. xiv, 6, 7, 44, 60, 98, 99

GPPD
: Parallel and Distributed Processing Group. 6, 44, 98

GPU
: Graphical Processing Unit. 12, 100

HPC
: High Performance Computing. vii, 5, 10, 12–16, 18, 23, 35, 98

I/O
: Input / Output. 23, 24, 70

IBS
: Instruction Based Sampling. 18, 38, 39, 60, 68, 76

Intel MKL
: Intel Math Kernel Library. 92, 94, 100

Likwid
: "Like I Knew What I am Doing". xii, 5, 13, 14, 23, 24, 38

LRU
: Least Recently used. 32

MAQAO
: Modular Assembler Quality Analyzer and Optimizer. 13, 14

Moca
: Memory Organisation Cartography & Analysis. xiii–xv, 6, 7, 60–77, 80, 83, 84, 87–89, 92, 94, 99, 100

MPI
: Message Passing Interface. 13, 81

NPB
: NAS Parallel Benchmarks. 6, 48, 51, 54, 69, 70, 72, 73, 76, 77, 111

NUMA
: Non-Uniform Memory Access. vii, 4, 6, 35, 36, 39, 44, 46–48, 50, 53–55, 60, 68, 76, 91

OpenMP
: Open Multi-Processing. 12, 13, 23–25, 38, 52, 54, 68

OS
: Operating System. 6, 15, 16, 35, 36, 50, 54, 60, 61, 63, 81

PAPI
: Performance API. 13, 14, 38

PARAVER
: PARAllel Visualization end Events Representation. 13, 14

PCM
: Performance Counter Monitor. 38

PEBS

Precise Event Based Sampling. 18, 38, 39, 60, 68, 76

PGD

Page Global Directory. 67

PID

Process Identifier. 85

PMD

Page Middle Directory. 67

PTE

Page Table Entry. 61, 63, 64, 66, 67

Sofa

Simulation Open Framework Architecture. xii, xv, 5, 7, 10–12, 14, 23–26, 92, 98

Tabarnac

Tool for Analyzing the Behavior of Applications Running on NUMA ArChitecture. xiii–xv, 6, 7, 44–50, 53–56, 60, 68–70, 72, 74–77, 80, 90, 98, 99

TAU

Tunning and Analysis Utilities. 13

UFRGS

Universidade Federal do Rio Grande do Sul. xiv, 6, 44, 98

VCS

Version Control System. 21

VPA

Visual Performance Analysis. xiv, 6, 44, 98

Glossary

AMD

Advanced Micro Devices, Inc. 13, 55, 60, 68, 76

CodeAnalyst

CodeAnalyst is AMD performance analyser. 13, 38

CodeXL

CodeXL is CodeAnalyst successor's. 13, 14

FrameSoc

Framesoc is a generic trace management and analysis infrastructure. xiii–xv, 6, 7, 14, 80, 81, 83, 87–89, 92, 94, 99

Git

Git is a distributed Version Control System. 16, 17, 21, 22

Grid5000

Grid5000 is a large scale platform for experiment in computer science. 68, 69

HPCToolkit

HPCToolkit is an integrated suite of tools for measurement and analysis of program performance. 13, 14, 38, 40, 80

Intel

Intel corporation. xi, xiv, 4, 6, 13, 32, 35, 37, 55, 60, 68, 76, 98, 100

KAAPI

KAAPI is a parallel runtime with data flow dependencies. 12

Kameleon

Kameleon is a tool to generate customized software appliances. 15, 21

Linux

The Linux kernel. xiv, 6, 13, 35, 47, 60–64, 70, 76, 92, 99

MemAxes

MemAxes is a tool for visualizing memory access samples. 40, 68, 77

Memphis

Memphis is a toolset for indentifiying problematic memory accesses. 38, 39

MemProf

MemProf is a memory profiler for NUMA machines based on AMD IBS. 38, 39, 68, 71–73, 75, 77

Memspy

MemSpy is a performance debugging tool for memory bottlenecks. 39

Mitos

Mitos is a library and a tool for collecting sampled memory performance data to view with MemAxes. 38–40, 68, 70–73, 75, 77

Numalyze

Numalyze is a memory tracer designed to analyze the result of online NUMA mapping tools. 44, 45

Ocelotl

Ocelotl is a visualization tool providing aggregated overview for trace analysis.. xiv, xv, 7, 80–84, 86–88, 92, 94, 99

Org-mode

Org-mode is an organizing mode for the editor GNU Emacs. 16, 89

Perf

The Linux perf command, also called perf_events, allows to access performance counters. 13

Pin

Pin is a dynamic binary instrumentation framework designed by Intel for IA-32 and X86-64 architectures. xiv, 6, 13, 18, 44, 45, 55, 60, 71, 98

Pintool

Pintool is the name given (by Intel) to tools developed using Pin. 14

R

R is a programming language for statistical computing. xv, 6, 7, 14, 19, 45, 80, 89, 90, 92, 94, 99

R-markdown

R-markdown is a combination of the R and markdown languages that allow to mix statistical analysis with formated comments. 19, 45

SimGrid

SimGrid is a scientific instrument to study the behavior of large-scale distributed systems such as Grids, Clouds, HPC or P2P systems. 13

VTune

Intel VTune Amplifier. 13, 14, 38

Bibliography

[Adhianto et al., 2010] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701.

[Allard et al., 2007] Allard, J., Cotin, S., Faure, F. c., Bensoussan, P.-J., Poyer, F. c., Duriez, C., Delingette, H., and Grisoni, L. (2007). SOFA - an Open Source Framework for Medical Simulation. In *Medicine Meets Virtual Reality (MMVR 15)*, palm beach, États-Unis.

[AMD, 2016] AMD (2016). CodeXL Quick Start Guide. https://github.com/GPUOpen-Tools/CodeXL/releases/download/v2.0/CodeXL_Quick_Start_Guide.pdf. Version 2.0 Revision 1 [Online; last accessed 01-10-2016].

[Bae et al., 2012] Bae, C. S., Xia, L., Dinda, P., and Lange, J. (2012). Dynamic Adaptive Virtual Core Mapping to Improve Power, Energy, and Performance in Multi-socket Multicores. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 247–258, New York, NY, USA. ACM.

[Bao et al., 2008] Bao, Y., Chen, M., Ruan, Y., Liu, L., Fan, J., Yuan, Q., Song, B., and Xu, J. (2008). HMTT: A Platform Independent Full-system Memory Trace Monitoring System. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 229–240, New York, NY, USA. ACM.

[Beniamine, 2013] Beniamine, D. (2013). Cartographier la mémoire virtuelle d'une application de calcul scientifique. In *ComPAS'2013 / RenPar'21*, Grenoble, France.

[Beniamine et al., 2015a] Beniamine, D., Corre, Y., Dosimont, D., and Huard, G. (2015a). Memory Organisation Cartography and Analysis. Technical Report 8694, Inria.

[Beniamine et al., 2015b] Beniamine, D., Diener, M., Huard, G., and Navaux, P. O. A. (2015b). TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, VPA '15, pages 1–1, New York, NY, USA. ACM.

[Beniamine and Huard, 2016] Beniamine, D. and Huard, G. (2016). Moca: An efficient Memory trace collection system. Research Report RR-8931, Inria Grenoble Rhône-Alpes, Université de Grenoble.

[Boehm et al., 1991] Boehm, H.-J., Demers, A. J., and Shenker, S. (1991). Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, New York, NY, USA. ACM.

[Bosch et al., 2000] Bosch, R., Stolte, C., Tang, D., Gerth, J., Rosenblum, M., and Hanrahan, P. (2000). Rivet: A Flexible Environment for Computer Systems Visualization. *SIGGRAPH Comput. Graph.*, 34(1):68–73.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186.

[Browne et al., 2000] Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. (2000). A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204.

[Budanur et al., 2011] Budanur, S., Mueller, F., and Gamblin, T. (2011). Memory Trace Compression and Replay for SPMD Systems Using Extended PRSDs? *SIGMETRICS Perform. Eval. Rev.*, 38(4):30–36.

[Cappello et al., 2005] Cappello, F., Caron, E., Dayde, M., Desprez, F., Jegou, Y., Primet, P., Jeannot, E., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Quetier, B., and Richard, O. (2005). Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, page 8.

[Casanova et al., 2014] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. (2014). Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917.

[Cern and OpenAire, 2013] Cern and OpenAire (2013). Zenodo. https://zenodo.org. [Online; last accessed 01-10-2016].

[Chothia et al., 2016] Chothia, Z., Shaohu, C., and Luo, W. (2016). OpenBlas An optimized BLAS Libary. http://www.openblas.net/. [Online; last accessed 01-10-2016].

[Collberg et al., 2015] Collberg, C., Proebsting, T., and Warren, Alex, M. (2015). Repeatability and Benefaction in Computer Systems Research . A Study and a Modest Proposal. Technical report, University of Arizona TR 14-04.

[Corbet, 2012] Corbet, J. (2012). AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/486858/.

[DeRose et al., 2002] DeRose, L., Ekanadham, K., Hollingsworth, J. K., and Sbaraglia, S. (2002). SIGMA: a simulator infrastructure to guide memory analysis. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 1–13.

[DeRose, 2001] DeRose, L. A. (2001). The Hardware Performance Monitor Toolkit. In *Euro-Par 2001 Parallel Processing*, volume 2150, chapter Lecture Notes in Computer Science, pages 122–132. Springer Berlin Heidelberg.

[Diener et al., 2013] Diener, M., Cruz, E. H. M., and Navaux, P. O. A. (2013). Communication-Based Mapping Using Shared Pages. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 700–711.

[Diener et al., 2014] Diener, M., Cruz, E. H. M., Navaux, P. O. A., Busse, A., and Heiß, H.-U. (2014). kMAF: Automatic Kernel-level Management of Thread and Data Affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 277–288, New York, NY, USA. ACM.

[Diener et al., 2015] Diener, M., Cruz, E. H. M., Pilla, L. L., Dupros, F., and Navaux, P. O. A. (2015). Characterizing communication and page usage of parallel applications for thread and data mapping . *Performance Evaluation* , 88–89:18–36.

[Djoudi et al., 2005] Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., and Jalby, W. (2005). MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In *Proceedings of workshop on Explicitely Parallel Instruction Computing Techniques (EPIC'04)*, Santa Jose, California.

[Dosimont et al., 2014] Dosimont, D., Corre, Y., Schnorr, L. M., Huard, G., and Vincent, J.-M. (2014). Ocelotl: Large Trace Overviews Based on Multidimensional Data Aggregation. In *8th International Parallel Tools Workshop*, Stuttgart, Germany.

[Drepper, 2007] Drepper, U. (2007). What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf. [Online; accessed 01-10-2016].

[Drongowski, 2007] Drongowski, P. J. (2007). Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Technical report, AMD CodeAnalyst Project, Boston Design center.

[Drongowski, 2008] Drongowski, P. J. (2008). An introduction to analysis and optimization with AMD CodeAnalyst^{TM}. Technical report, AMD Boston Design center.

[Dupros et al., 2008] Dupros, F., Aochi, H., Ducellier, A., Komatitsch, D., and Roman, J. (2008). Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In *Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on*, pages 253–260.

[Faure et al., 2011] Faure, F., Gilles, B., Bousquet, G., and Pai, D. K. (2011). Sparse Meshless Models of Complex Deformable Solids. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 73–1, New York, NY, USA. ACM.

[Feitelson, 2015] Feitelson, D. G. (2015). From Repeatability to Reproducibility and Corroboration. *SIGOPS Oper. Syst. Rev.*, 49(1):3–11.

[Ganesan, 2016] Ganesan, B. (2016). Introduction to Multi-Core. http://www.ecs.umass.edu/ece/andras/courses/ECE668/Mylectures/Introduction_to_Multi_Core.pdf. [Online; last accessed 01-10-2016].

[Gautier et al., 2007] Gautier, T., Besseron, X., and Pigeon, L. (2007). KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA. ACM.

[Giménez et al., 2014] Giménez, A., Gamblin, T., Rountree, B., Bhatele, A., Jusufi, I., Bremer, P.-T., and Hamann, B. (2014). Dissecting On-Node Memory Access Performance: A Semantic Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 166–176, Piscataway, NJ, USA. IEEE Press.

[group, 2016] group, T. F.-F. (2016). *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*, v2.2.1 edition. http://github.com/linbox-team/fflas-ffpack.

[Heo et al., 2005] Heo, J., Yi, S., Cho, Y., Hong, J., and Shin, S. Y. (2005). Space-efficient Page-level Incremental Checkpointing. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 1558–1562, New York, NY, USA. ACM.

[Hermann et al., 2008] Hermann, E., Faure, F. c., and Raffin, B. (2008). Ray-traced collision detection for deformable bodies. In Braz, J., Nunes, N. J., and Pereira, J. M., editors, *GRAPP 2008 - 3rd International Conference on Computer Graphics Theory and Applications*, pages 293–299, Madeira, Portugal. INSTICC.

[Hermann et al., 2009] Hermann, E., Raffin, B., and Faure, F. c. (2009). Interactive Physical Simulation on Multicore Architectures. In *EGPGV - Eurographics Workhop on Parallel Graphics and Visualization*, pages 1–8, Munich, Germany.

[Husain et al., 2015] Husain, B., Giménez, A., Levine, J. A., Gamblin, T., and Bremer, P.-T. (2015). Relating Memory Performance Data to Application Domain Data Using an Integration API. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, VPA '15, pages 5–1, New York, NY, USA. ACM.

[Jiang et al., 2014] Jiang, T., Zhang, Q., Hou, R., Chai, L., Mckee, S. A., Jia, Z., and Sun, N. (2014). Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 22–30.

[Jin et al., 1999] Jin, H., Frumkin, M., and Yan, H. (1999). NPB-OpenMP 3.0. NAS Technical Report NAS-99-011, NASA Ames Research Center.

[Jones et al., 2006] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006). Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. *SIGARCH Comput. Archit. News*, 34(5):14–24.

[Kleen et al., 2005] Kleen, A., Brunner, R., Langsdorf, M., and Chabowski, M. (2005). A NUMA API for Linux. Technical report, SUSE Labs.

[Lachaize et al., 2012] Lachaize, R., Lepers, B., and Quema, V. (2012). MemProf: A Memory Profiler for NUMA Multicore Systems. In *USENIX 2012 Annual Technical Conference (USENIX ATC 12)*, pages 53–64, Boston, MA. USENIX.

[Lamarche-Perrin et al., 2014] Lamarche-Perrin, R., Schnorr, L. M., Vincent, J.-M., and Demazeau, Y. (2014). Agrégation de traces pour la visualisation de grands systèmes distribués. *Technique et Science Informatiques*. Forthcoming.

[Levinthal, 2009] Levinthal, D. (2009). Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. Technical report, Intel.

[Liu and Mellor-Crummey, 2013] Liu, X. and Mellor-Crummey, J. (2013). A Data-centric Profiler for Parallel Programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 28–1, New York, NY, USA. ACM.

[Liu and Mellor-Crummey, 2014] Liu, X. and Mellor-Crummey, J. (2014). A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA. ACM.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA. ACM.

[Majo and Gross, 2013] Majo, Z. and Gross, T. R. (2013). (Mis)understanding the NUMA memory system performance of multithreaded workloads. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 11–22.

[Malony et al., 2011] Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., and Lamb, C. (2011). Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176–185.

[Marchetti et al., 1995] Marchetti, M., Kontothanassis, L., Bianchini, R., and Scott, M. L. (1995). Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 480–485.

[Martonosi et al., 1992] Martonosi, M., Gupta, A., and Anderson, T. (1992). MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '92/PERFORMANCE '92, pages 1–12, New York, NY, USA. ACM.

[McCurdy and Vetter, 2010] McCurdy, C. and Vetter, J. (2010). Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96.

[Miller et al., 1995] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. (1995). The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46.

[Mytkowicz et al., 2009] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2009). Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA. ACM.

[Nesme et al., 2009] Nesme, M., Kry, P. G., JeRabkova, L., and Faure, F. (2009). Preserving Topology and Elasticity for Embedded Deformable Models. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 52–1, New York, NY, USA. ACM.

[Pagano et al., 2013] Pagano, G., Dosimont, D., Huard, G., Marangozova-Martin, V., and Vincent, J. M. (2013). Trace Management and Analysis for Embedded Systems. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 119–122.

[Pagano and Marangozova-Martin, 2014] Pagano, G. and Marangozova-Martin, V. (2014). The frameSoC Software Architecture for Multiple-view Trace Data Analysis. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 217–222, New York, NY, USA. ACM.

[Perarnau et al., 2011] Perarnau, S., Tchiboukdjian, M., and Huard, G. (2011). Controlling Cache Utilization of HPC Applications. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 295–304, New York, NY, USA. ACM.

[Pillet et al., 1995] Pillet, V., Labarta, J., Cortes, T., and Girona, S. (1995). PARAVER: A Tool to Visualize and Analyze Parallel Code. In Nixon, P., editor, *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31.

[Reinders, 2005] Reinders, J. (2005). *VTune performance analyzer essentials*. Intel Press.

[Renater, 2011] Renater (2011). FileSender. [Online; last accessed 01-10-2016].

[Ruiz et al., 2015] Ruiz, C., Harrache, S., Mercier, M., and Richard, O. (2015). Reconstructable Software Appliances with Kameleon. *SIGOPS Oper. Syst. Rev.*, 49(1):80–89.

[Servat Gelabert, 2015] Servat Gelabert, H. (2015). *Towards instantaneous performance analysis using coarse-grain sampled and instrumented data*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona.

[Shende and Malony, 2006] Shende, S. S. and Malony, A. D. (2006). The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311.

[SOFA, 2016] SOFA (2016). Sofa online documentation. https://www.sofa-framework.org/community/doc. [Online; last accessed 20-06-2016].

[Stanisic, 2015] Stanisic, L. (2015). *A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications*. PhD thesis, Université Grenoble Alpes.

[Tao et al., 2001] Tao, J., Karl, W., and Schulz, M. (2001). Visualizing the Memory Access Behavior of Shared Memory Applications on NUMA Architectures. In Alexandrov, V., Dongarra, J., Juliano, B., Renner, R., and Tan, C. J. K., editors, *Computational Science - ICCS 2001*, volume 2074, chapter Lecture Notes in Computer Science, pages 861–870. Springer Berlin Heidelberg.

[Torrellas et al., 1994] Torrellas, J., Lam, H. S., and Hennessy, J. L. (1994). False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663.

[Toss and Comba, 2013] Toss, J. and Comba, J. (2013). Parallel Voronoi Diagram computation on scaled distance planes using CUDA. In *WSPPD 2013 - XI Workshop de Processamento Paralelo e Distribuído*.

[Toss et al., 2014] Toss, J., Comba, J. L. D., and Raffin, B. (2014). Parallel Shortest Path Algorithm for Voronoi Diagrams with Generalized Distance Functions. In *XXVII SIBGRAPI, Conference on Graphics Patterns and Images*, Rio de Janerio, Brazil.

[Treibig et al., 2010] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA.

[Weaver et al., 2013] Weaver, V. M., Terpstra, D., McCraw, H., Johnson, M., Kasichayanula, K., Ralph, J., Nelson, J., Mucci, P., Mohan, T., and Moore, S. (2013). PAPI 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 124–125.

[Weyers et al., 2014] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Muller, M. S., and Hentschel, B. (2014). Visualization of Memory Access Behavior on Hierarchical NUMA Architectures. In *Visual Performance Analysis (VPA), 2014 First Workshop on*, pages 42–49.

[Wilhalm et al., 2012] Wilhalm, T., Dementiev, R., and Fay, P. (2012). Intel Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor. [Online; last accessed 01-10-2016].