UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Duco van Amstel

Thèse dirigée par **Fabrice Rastello**
et co-encadrée par **Benoît Dupont de Dinechin**

préparée au sein de l' **INRIA**
et de **l'École Doctorale MSTII**

Optimisation de la Localité des Données sur Architectures Manycœurs

Thése soutenue publiquement le **18** / **07** / **2016**,
devant le jury composé de :

Mr. François Bodin
Professeur, IRISA, Président
Mr. Albert Cohen
Directeur de Recherche, Inria, Rapporteur
Mr. Benoît Dupont de Dinechin
Directeur technique, Kalray S.A, Co-Encadrant de thèse
Mr. François Irigoin
Directeur de Recherche, MINES ParisTech, Rapporteur
Mr. Fabrice Rastello
Chargé de Recherche, Inria, Directeur de thèse
Mr. Ponnuswamy Sadayappan
Professor, Ohio State University, Examinateur

Data Locality on Manycore Architectures

a dissertation by

Duco Yalmar van Amstel

Acknowledgments

The work which is related in this manuscript is not that of a single person. It rather is the synthesis of a continuous flow of discussions, exchanges and collaborative effort. In this regard I extend my deepest gratitude towards both Kalray and Inria for giving me the opportunity to work on this most exciting project. The industrial context has proven to be a rich learning ground and I would like to thank Benoît de Dinechin for showing me the way. The same can be said about my research team where Fabrice Rastello has taught me the importance of persevering in my efforts even when you're trust in the chosen approach is waning. His guidance and inspiration will stay with me for a long time.

Many people have contributed in one way or another in this project and naming all of them would make for a very long list. Nonetheless I give my sincere thanks to Lukasz Domagala, Yves Durand and Dr. Sadayappan, my co-authors, who all have helped me at numerous times on a wide variety of subjects. I also would like to thank all my colleagues at Kalray: Céline, Marc, Frédéric, Nicolas, Patrice, Alexandre, Renaud & others, as well as those at Inria: François, Fabian, Diogo, Imma, Frédéric, ...

On a more personal note I am most gratefull to my parents Marit & Walter for teaching a young boy the importance of being curious, as curiosity is the brightest of torches and most certainly did not kill the cat (except for Schrödinger's). Last but not least I cannot leave out my beloved and most precious Bérengère for sharing both the best and the toughest moments of my work, with always an unconditionnal support.

Résumé

L'évolution continue des architectures des processeurs a été un moteur important de la recherche en compilation. Une tendance dans cette évolution qui existe depuis l'avènement des ordinateurs modernes est le rapport grandissant entre la puissance de calcul disponible (IPS, FLOPS, ...) et la bande-passante correspondante qui est disponible entre les différents niveaux de la hiérarchie mémoire (registres, cache, mémoire vive). En conséquence la réduction du nombre de communications mémoire requis par un code donnée a constitué un sujet de recherche important. Un principe de base en la matière est l'amélioration de la localité temporelle des données : regrouper dans le temps l'ensemble des accès à une donnée précise pour qu'elle ne soit requise que pendant peu de temps et pour qu'elle puisse ensuite être transféré vers de la mémoire lointaine (mémoire vive) sans communications supplémentaires.

Une toute autre évolution architecturale a été l'arrivée de l'ère des multicœurs et au cours des dernières années les premières générations de processeurs manycœurs. Ces architectures ont considérablement accru la quantité de parallélisme à la disposition des programmes et algorithmes mais ceci est à nouveau limité par la bande-passante disponible pour les communications entres cœurs. Ceci a amène dans le monde de la compilation et des techniques d'optimisation des problèmes qui étaient jusqu'à là uniquement connus en calcul distribué.

Dans ce texte nous présentons les premiers travaux sur une nouvelle technique d'optimisation, le *pavage généralisé* qui a l'avantage d'utiliser un modèle abstrait pour la réutilisation des données et d'être en même temps utilisable dans un grand nombre de contextes. Cette technique trouve son origine dans le pavage de boucles, une techniques déjà bien connue et qui a été utilisé avec succès pour l'amélioration de la localité des données dans les boucles imbriquées que ce soit pour les registres ou pour le cache. Cette nouvelle variante du pavage suit une vision beaucoup plus large et ne se limite pas au cas des boucles imbriquées. Elle se base sur une nouvelle représentation, le *graphe d'utilisation mémoire*, qui est étroitement lié à un nouveau modèle de besoins en termes de mémoire et de communications et qui s'applique à toute forme de code exécuté itérativement.

Le *pavage généralisé* exprime la localité des données comme un problème d'optimisation pour lequel plusieurs solutions sont proposées. L'abstraction faite par le *graphe d'utilisation mémoire* permet la résolution du problème d'optimisation dans différents contextes. Pour l'évaluation expérimentale nous montrons comment utiliser cette nouvelle technique dans le cadre des boucles, imbriquées ou non, ainsi que dans le cas des programmes exprimés dans

un langage à flot-de-données. En anticipant le fait d'utiliser le *pavage généralisé* pour la distribution des calculs entre les cœurs d'une architecture manycœurs nous donnons aussi des éléments de réponse pour modéliser les communications et leurs caractéristiques sur ce genre d'architectures.

En guise de point final, et pour montrer l'étendue de l'expressivité du *graphe d'utilisation mémoire* et le modèle de besoins en mémoire et communications sous-jacent, nous aborderons le sujet du débogage de performances et l'analyse des traces d'exécution. Notre but est de fournir un retour sur le potentiel d'amélioration en termes de localité des données du code évalué. Ce genre de traces peut contenir des informations au sujet des communications mémoire durant l'exécution et a de grandes similitudes avec le problème d'optimisation précédemment étudié. Ceci nous amène à une brève introduction dans le monde de l'algorithmique des graphes dirigés et la mise-au-point de quelques nouvelles heuristiques pour le problème connu de joignabilité mais aussi pour celui bien moins étudié du partitionnement convexe.

Abstract

The continuous evolution of computer architectures has been an important driver of research in code optimization and compiler technologies. A trend in this evolution that can be traced back over decades is the growing ratio between the available computational power (IPS, FLOPS, ...) and the corresponding bandwidth between the various levels of the memory hierarchy (registers, cache, DRAM). As a result the reduction of the amount of memory communications that a given code requires has been an important topic in compiler research. A basic principle for such optimizations is the improvement of temporal data locality: grouping all references to a single data-point as close together as possible so that it is only required for a short duration and can be quickly moved to distant memory (DRAM) without any further memory communications.

Yet another architectural evolution has been the advent of the multicore era and in the most recent years the first generation of manycore designs. These architectures have considerably raised the bar of the amount of parallelism that is available to programs and algorithms but this is again limited by the available bandwidth for communications between the cores. This brings some issues that previously were the sole preoccupation of distributed computing to the world of compiling and code optimization techniques.

In this document we present a first dive into a new optimization technique which has the promise of offering both a high-level model for data reuses and a large field of potential applications, a technique which we refer to as *generalized tiling*. It finds its source in the already well-known loop tiling technique which has been applied with success to improve data locality for both register and cache-memory in the case of nested loops. This new "flavor" of tiling has a much broader perspective and is not limited to the case of nested loops. It is build on a new representation, the *memory-use graph*, which is tightly linked to a new model for both memory usage and communication requirements and which can be used for all forms of iterate code.

Generalized tiling expresses data locality as an optimization problem for which multiple solutions are proposed. With the abstraction introduced by the *memory-use graph* it is possible to solve this optimization problem in different environments. For experimental evaluations we show how this new technique can be applied in the contexts of loops, nested or not, as well as for computer programs expressed within a dataflow language. With the anticipation of using *generalized tiling* also to distributed computations over the cores of a manycore

architecture we also provide some insight into the methods that can be used to model communications and their characteristics on such architectures.

As a final point, and in order to show the full expressiveness of the *memory-use graph* and even more the underlying memory usage and communication model, we turn towards the topic of performance debugging and the analysis of execution traces. Our goal is to provide feedback on the evaluated code and its potential for further improvement of data locality. Such traces may contain information about memory communications during an execution and show strong similarities with the previously studied optimization problem. This brings us to a short introduction to the algorithmics of directed graphs and the formulation of some new heuristics for the well-studied topic of reachability and the much less known problem of convex partitioning.

Contents

Introduction

The wind was not the beginning. There are neither beginnings nor endings to the turning of the Wheel of Time. But it was *a* beginning.

R. Jordan - *The Eye of the World*

The Memory-Wall

The science of code optimization, be it automatic through optimizing compilers or done manually by programmers, has seen numerous evolutions since the arrival of the first computers and the associated programming languages. These evolutions have often, if not always, held a close link with evolutions in the hardware on which the code is executed. As such, many changes in the micro-architecture of processor cores have been paired with corresponding modifications in the way code was optimized for these architectures. Compiler techniques such as instruction selection, scheduling and register allocation are some of the most iconic examples of the relationship between computer architecture and code optimization. Some techniques such as vectorization only exist because of the introduction of specific features in a processor's micro-architecture.

The topic of memory communications has been pushing its way into the spotlight of the compiler community for some years. In most if not all modern architectures, memory follows a hierarchical organization ranging from large but distant to small and close by storage. The further away a storage is from a computational core the more time and energy it takes to retrieve data from it. In the past a reduction of the number of communications between the different levels of a memory hierarchy has already been a target as it is usually linked to increased performances. However the current context of hardware evolutions and constraints is pushing both the academic and the industrial world to consider the subject with a renewed focus.

An interesting measure for a given computer architectures is the computation-to-bandwidth ratio. This ratio can be expressed as a number of arithmetic operations per byte loaded from one level of the memory hierarchy to the level below. If a program uses more operations per byte that needs to be loaded than the ratio this indicates that it is *computationally-bound* with respect to this memory level. In the opposite case the program is called *memory-bound*. This is an indication of which architectural element is the current bottle-neck that limits the performance of the program: the computational core or the memory bandwidth.

Data extracted from computer architectures that have appeared over the past decade shows a common trend in the evolution of computation-to-bandwidth ratios for both CPUs and GPUs (see Figure I.1). The computation-to-bandwidth ratio is growing which means that more and more existing applications move from being *computationally-bound* to being *memory-bound*. This phenomenon is not new and has been called the *Memory-Wall* when identified during the '90s [WM95; SPN96;

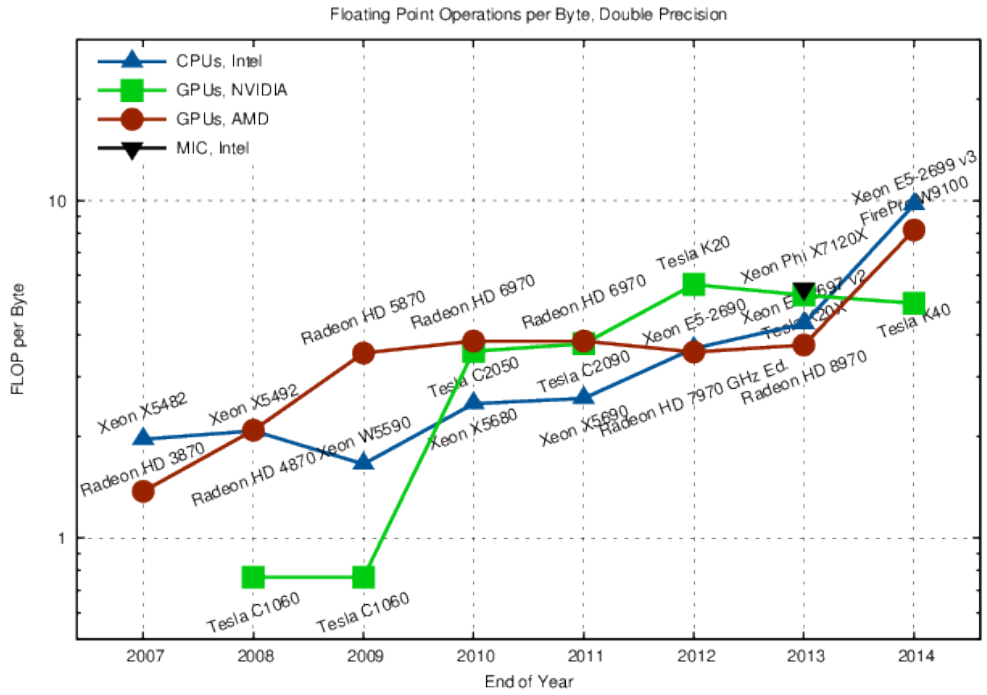


Figure I.1 – Computation-to-bandwidth ratio for various recent architectures[1]

McK04; SMCCL11].

An implication of this phenomenon is that the evolution of performances will be limited by the evolution of memory-bandwidth. It appears thus as a necessity to identify ways in which this memory-wall can be broken down or, at the least, ways to extend the time before its influence becomes prohibitive. The paths that are being researched in this respect include the modification of memory interfaces for a faster increase in bandwidths [JED15; Hyb15], the development of novel architectures based on the processor-in-memory concept [Dlu+14] as well as the improvement of bandwidth efficiency of existing software.

Some very recent steps in memory-interfaces have already proven to be able to increase the attainable bandwidths in the near-future by multiple orders of magnitude. This means the memory-wall can be pushed back significantly in its time-frame without being broken down [Rad+15].

In a similar fashion the current projects for novel architectures also promise a change of scenery in the way data is stored and transferred by memory. The grounds

[1]Source: www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

for this evolution lie in parallel computations that do not follow the traditional Von Neumann architectural model. Such a change of paradigm means a potential for considerable speed-ups for certain types of computations. The limits however lie in the parts of algorithms that are devoid of inherent parallelism.

The previous observations argue in favor of a continued effort by the compiler community in the search for improvements in bandwidth efficiency. Mainly this implicates that for the transfer of a single datum to a lower-level in the memory hierarchy a maximum amount of computations involving this particular datum should be scheduled before it is discarded again towards a higher level. The underlying concept here is that of *temporal locality*. Traditionally *temporal locality* is optimized through methods that perform some form of rescheduling on the order of execution of an algorithm. A typical example for such methods is loop tiling which can be applied either manually by a programmer or automatically through a compiler.

Tiling and the need for a communication model

Loop tiling as an optimization technique appeared in the 1980's [Pei86; IT88; Wol89]. The founding concept is to partition the iteration space of a loop nest in regular tiles and to reschedule the execution order of the loop iterations such that all iterations within a given tile are processed together. One of the main difficulties in this process is to compute the correct tile shape and size for a given loop nest. This is a problem that is quite similar in nature to the computation of block sizes for blocked matrix operations [MC69]. However determining the existing data dependencies within the iteration space has become a research subject on its own. Likewise the computation of the optimal tiles for a given dependency scheme remains a source for a great number of publications even if the nature of the loop transformation has remained the same.

Among the limits that exist for current loop tiling techniques we can identify some of those that appear as significant barriers to a more general application of this optimization. A first condition is that tiling only applies to perfect loop nests even if some extensions have been achieved. Furthermore most tiling models only account for situations where all dependencies have distances (*i.e* the amount of iterations between reuses) that can be expressed either as a constant or as an affine function of the induction variables.

A paramount element when evaluating tiling options is the memory for which the tiling is performed. Depending on the characteristics of the target memory (size, granularity, access, ...) a given tile will produce more or less communications with mem-

ory further away from the cores which in turn will influence the final memory bandwidth usage of the tiled code. As an example two cache-memories of the same size may exhibit two completely different behaviors if the sizes of their lines or their associativity differ. Similarly cache-memory does not behave in the same way as registers or off-chip *Dynamic Random Access Memory* (DRAM). In this light a model for memory communications appears to be an important tool without which tile computations fall back to educated guesswork.

When taking a step back to look at the initial motivations behind loop tiling it appears that the identification of parallelism is of an importance comparable to data locality itself. If all incoming dependencies of two separate tiles have been computed and the tiles have no dependencies between them their executions can be done in parallel. This observation makes it possible to draw a link between loop tiling and other examples of optimizations of frequently executed code such as dataflow programs and distributed run-time environments. In all these situations the questions of scheduling and data locality are intertwined and depend heavily on the memory hierarchy on which code is being executed. The whole should be viewed in the context of an amount of hardware parallelism in computer architectures that has been growing considerably over the last decade.

The advent of manycore architectures

In a general manner the usage of large distributed and parallel systems has been an exclusivity of high-performance computing. However even basic commercial processors have hit a frequency ceiling and have therefore resorted to an increase in the amount of hardware parallelism. Current top-of-the-line processors showcase up to 20 complex computational cores with the physical size of the chip being the only limit to this number. This type of processors has been called a *multicore processor*. As it has been possible to go from a single core to tens of cores, the next step in this evolution is to go from tens of cores to hundreds or even thousands of cores. Already industrial products and prototypes exist that have taken this direction and have thereby started a first generation of *manycore processors*. The issue of the physical size of the chip is solved by using much simpler core architectures to reduce the surface per core. Some examples of such architectures are the TILE-Gx range by Tilera (now Mellanox)[2], the

[2] http://www.mellanox.com/page/multi_core_overview

MPPA family by Kalray[3] and the Xeon Phi processors by Intel[4].

Among the main differences between multicore and manycore processors is the communication between the cores. For multicore architectures the general principle is direct communication through shared memory as these architectures usually have a single off-chip memory that is common to all the cores and which is accessed through multiple levels of coherent cache. In the case of manycore architectures memory communications are rarely direct and use an embedded *network-on-chip* that connects all the cores. Memory can be off-chip and accessed through the network-on-chip with caches that are local to some cores. In some cases non-cache memory can also be distributed among the cores thereby forming a *Non-Uniform Memory Access* (NUMA) typed architecture with distant accesses performed through the network-on-chip. Such a memory hierarchy makes it possible to relate a manycore processor to a computational cluster in many aspects. As a result *spatial locality*, a concept that is complimentary to *temporal locality*, encourages the computations involving a given datum to be performed on a single core.

In the context of the *Memory-Wall* a manycore architecture has the advantage of a higher degree of parallelism in the memory communications between on-chip cores. However the off-chip bandwidth must still be shared among all the cores. Furthermore the presence of a network-on-chip as well as the increased complexity of the memory-hierarchy contribute to higher average latencies for memory communications. Because of this both *spatial* and *temporal locality* are of great importance to ensure a correct performance level for computations that run on such architectures. The tiling principle can here be used to ensure the necessary locality but also to provide an adequate level of granularity for scheduling and resource allocation.

A high-level framework

Our observation of modern-day hardware characteristics, with the principle of loop tiling in mind suggests, that this existing technique could potentially be of use at a much broader scale than it is today. While it is currently used only on loop nests it would be interesting to see if it could be applied in a more general way to a larger set of loops. There is also much to say for an approach similar to tiling in the optimization of frequently executed code beyond the case of loops. Dataflow programs for example typically can be seen as a set of computations performed by iteration over

[3] http://www.kalray.eu

[4] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

the available input.

This manuscript's challenge is to expose a coherent vision of such a generalization of the tiling technique applicable in a variety of use-cases. In order to do so, the first point of attention is the study and development of both a communication and memory-cost model in Chapter 1. By defining a single representation for the optimization problems at hand, it is possible to evaluate the memory requirements of a given piece of code as well as the amount of communications that are required throughout the targeted memory hierarchy for an arbitrary scheduling.

For the extension of tiling towards distributed environments such as manycore processors it is also necessary to evaluate the costs and constraints related to the specifics of such platforms. Notably the presence of a network-on-chip requires that its behavior is modeled and parametrized in order to determine the latencies of communications on the network: an important factor when scheduling computations that use a common set of data on separate resources. For the purpose of this study the target platform here was the MPPA chip by Kalray[5] for which a specific modelisation is performed in Chapter 2.

With the theoretical foundations in place, the next step is to create a formal definition for our generalized tiling optimization problem and to propose methods for solving it. This is the subject of Chapter 3. Multiple approaches have been attempted both optimal and approximate with a preference for heuristics that allow to minimize the impact of the optimization on the compilation process. The time required to compute a solution is reduced at the cost of not being optimal.

The first application of the generalized tiling method specifically targets loops and tries to reduce the number of load and store instructions. This is also the point where the main differences between generalized tiling and classical tiling appear best. A full detail of the implementation is given in combination with experimental results in Chapter 4. A thorough analysis of the presented results enables us to identify both the strengths and weaknesses of this new approach.

A second application of generalized tiling is presented next in Chapter 5 and targets dataflow code at a higher level of granularity. This also offers the possibility to extend the tiling problem statement with resource allocation in a distributed environment. The experimental evaluation of this generalized tiling in this use-case uses a theoretical model of execution. The first elements of an implementation in a real dataflow

[5]The doctoral contract leading to this manuscript was obtained through a CIFRE agreement between Kalray and Inria. The author was a full-time employee of Kalray for the entire duration of his doctoral studies.

environment are also given within the ΣC[6] [Gou+11] toolchain.

As a final step of this study we try to go beyond the context of code optimization in Chapter 6 by reusing our memory-cost and communication model for the purpose of execution trace analysis. The analysis of traces of the memory operations of a program can be used to identify parts of the code that use significantly more memory-bandwidth than it would with a different scheduling. As execution traces do not exhibit the same kind of regularity as loop dependencies and dataflow graphs their analysis requires us to tackle some new algorithmic challenges in graph partitioning. Again an implementation of these newly developed algorithms is presented together with the associated experimental results and comments.

[6] ΣC is a dataflow extension of C that follows a cyclo-static execution model. It is developed by CEA LIST and was supported on the MPPA processor family.

Chapter 1

Memory Usage & Operational Intensity

'Thus is our treaty written; thus is agreement made.
Thought is the arrow of time; memory never fades.
What was asked is given. The price is paid.'

R. Jordan – *The Shadow Rising*

1.1 Context and goal

The first step in this study of a generalized form of tiling is the construction of an effective model for memory usage and Input-Output (IO) costs.

The memory usage of a given code snippet can be associated with the maximum amount of data throughout the execution that should be simultaneously available. From this point of view it bears a similarity with the notion of the *max-live* measure used in scheduling and register allocation.

In a similar fashion the IO cost of a piece of code represents the amount of memory communications that are performed during its execution. This can then be translated into the characteristic of operational intensity which is the ratio between the number of instructions that are executed and the number of memory communications that the execution of these instructions requires. The operational intensity is specific to a level of the memory-hierarchy. In the case of a standard x86 processor with registers, L1, L2 and LLC caches this means that a program has four different operational intensities:

- A register operational intensity for load / store instructions.
- An L1 operational intensity for L1 misses that will hit the L2 cache.
- An L2 operational intensity for L2 misses that will hit the LLC.
- An LLC operational intensity for LLC misses that will hit off-chip RAM.

Our primary goal is to increase the operational intensity of programs by reducing the amount of memory communications between close-by memory and distant memory. For this purpose we assume from now on that we have a target memory, which is limited in size, containing the data used by the program, and a distant memory of infinite size to which data is sent if the target memory is not large enough. This assumption is valid both for registers, where overflowing data is transferred from and to memory, and for cache memory, where cache misses and updates are transferred from and to a higher level of cache or main memory.

A second important challenge is to be able to perform our optimizations on different types of code, on a diversity of hardware targets as well as on the multiple levels of memory hierarchies. The implies that we need to describe the data dependencies existing within a program in a common representation for all these environments. We also need a representation with an abstraction level high enough to hide the granularity at which the code is viewed (instruction, basic-block, ...). As we want to represent code that executes iteratively it is also necessary for the representation to be identical

for each iteration. This reduces both the memory size of the representation and the time complexity of algorithms that are applied to it.

Our optimization being a form of tiling, the representation should make it easy to define a tiling solution for the code that is being optimized. And circling back to the memory and IO costs it should be straightforward to translate a tiling solution into the associated measures.

The solution that we propose is inspired by the representation of dataflow programs in the form of a graph of actors and links that transport data from one actor to another. Before diving into the definition of this new representation that we name the *Memory-use graph* of the code we first take a look at the existing state-of-the-art concerning the computation of memory and IO costs.

1.2 State-of-the-art

1.2.1 Cache analysis

The notion of reuse distance or *least-recent use* (LRU) distance was coined by Mattson et al. [Mat+70] in the context of the study of cache usage. It is an effective way to compute the number of cache-misses as a function of the cache size for a given access pattern in the case of an LRU cache and has been used in numerous studies. Similar metrics include the miss rate, the average footprint and the data lifetime. These notions were formalized by Denning in his seminal work on working-set locality for cache analysis [Den68]. They have been extended and formally linked within a unique framework for cache-performance evaluation by Xiang et al. in a *higher-order theory of locality* [Xia+13].

1.2.2 Register pressure

When the register file is the targeted memory, the most well-known and commonly used method is register pressure. It has been the main metric for graph-based register allocation and spilling since the work by Chaitin et al. [Cha+81; Cha82]. Briggs proposed some improvements in his thesis [Bri92] as well as in his work with Cooper and Torczon [BCT94]. The approach taken by Chow and Hennessey for their *priority-based register allocation* [CH90] balances the need to keep register pressure under a given limit while keeping the critical path of execution as short as possible.

1.2.3 IO complexity

Yet another approach to the issue of memory usage and IO cost exists under the form of *IO complexity*. It aims at providing the tightest possible lower-bound on the amount of memory communications that are required for a given algorithm to be executed on target memory and as such it is implementation independent. The first attempts at the computation of such lower-bounds can be attributed to Jia-Wei and Kung [JWK81] who used the mathematical red-blue pebble game to model the inputs and outputs of an algorithm and applied it to a limited set of algorithms. The most recent advances in the topic of IO complexity have been produced by Elango et al. [Ela+15b]. Among other advances they generalized the red-blue pebble game to arbitrary algorithms running on parallel architectures [Ela+14; Ela+15a].

1.3 The memory-use graph

As suggested in the goal statements in Section 1.1 we propose a representation of the memory usage and dependencies of a code snippet inspired by diagrams of dataflow programs. This most important advantage of this approach is that it can easily be applied to the various types of code that we target for optimization. It should be noted that when mentioning dataflow programs we restrict ourselves to static [LM87] and cyclo-static [Bil+95] dataflow languages, a subset of the more general dataflow paradigm. The representation, presented in detail below, is referred to as the *memory-use graph* of the analyzed code. Furthermore, in order to illustrate the choices made for this representation, we use the code presented in Figure 1.1 all along this section.

```
int N, *A;
long long int *B, *C;

/* ... Initializations of N, A, B and C ... */

for (int i = 2; i < N - 2; i++) {
  /* S1 */ A[i] = (A[i-2] + A[i-1] + A[i] + A[i+1] + A[i+2]) / 5;
  /* S2 */ B[i] = B[i] + A[i] * C[i];
  /* S3 */ B[i] = B[i] - (B[i-1] - B[i]) * C[i-1];
}
```

Figure 1.1 – Example code for the illustration of the *memory-use graph*

1.3.1 Flow of data in the semi-regular case

At the most basic level a computation, an algorithm, a program is nothing more than the specification of a set of statements to be executed in a specific order. The level of abstraction with which these statements are given can vary over a very wide range and is closely associated with the different types of programming languages that exist. The lowest level of abstraction typically corresponds to the lowest level language (assembly programming), *i.e* a statement representing a single machine instruction, whereas higher levels of abstraction may let a statement correspond to a set of instructions, a basic block or even an entire function. In the case of the example in Figure 1.1 the statements 1, 2 and 3 (S1, S2 and S3) correspond to individual lines of C code. From the point of view of the representation the invariant for all abstraction levels is that statements are the atomic elements.

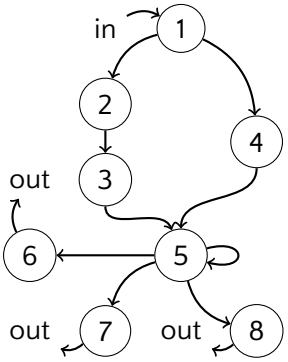


Figure 1.2 – Flow of data for a dataflow computation

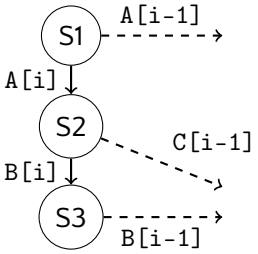


Figure 1.3 – Raw representation of the example code

Statements may take input, produce output or perform intermediate steps. As a result data may be used, modified or produced. This is illustrated in Figure 1.2 (no link with Figure 1.1) and forms the basis of the dataflow programming paradigm. We use the same principle for our *memory-use graph*: nodes represent statements and directed edges between them represent reuses by the target node of data produced by the source node. Following this the example of Figure 1.1 results in three separate nodes as shown in Figure 1.3. Dashed edges are used to represent inter-iteration reuses whereas full edges represent intra-iteration reuses. Because we have not yet described the graph's semantics, this example should not be interpreted as a correct and exact representation of the code in Figure 1.1. Starting from this point we now dive into the specifics of the *memory-use graph*.

An important observation to make is that in the case of static dataflow languages the entire set of available input is processed by the same computational graph. This means that the same code is applied by iterating over each separate input. For this

reason we will characterize the computation as being *semi-regular*: an irregular set of varying statements that are applied in a uniform way over a set of data. From this point of view, the execution of a loop in a more imperative-styled programming paradigm has exactly the same behavior with the different statements of the loop-body being applied over the iteration domain of the loop. Another consequence of the regularity over the iteration domain is that the amount of data that is transmitted between the same two nodes of the *memory-use graph* is identical at each iteration. We can thus annotate each edge with the amount of data that it represents. This is done in Figure 1.4 for each reuse of the example code: arrays `A` and `B` contain elements of size 8 whereas array `C` contains elements of size 4. Again the semantics of the edges being unknown this example should not be seen as an exact representation.

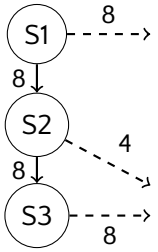


Figure 1.4 – Example representation annotated with byte sizes

By making a short detour through the world of loop tiling, we can improve our understanding of the above-described *semi-regular* characteristic by comparing it to the usual representation of loop nests. In the case of classical tiling, each element within the iteration domain of the loop is usually represented by a single point. An iteration of a loop nest of depth k is then associated with coordinates in $\mathbb{N}^k$ according to the values of the induction variables for this iteration. This correspondence between the loop nest and its representation is shown in Figure 1.5 where each point of the grid represents a single full iteration of the loop. A consequence is that this representation is fully *regular* as all elements of the grid are associated with the same code. Our dataflow-based approach

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    /* Loop statements */
    /* S1 */
    /* S2 */
    /* ... */
```

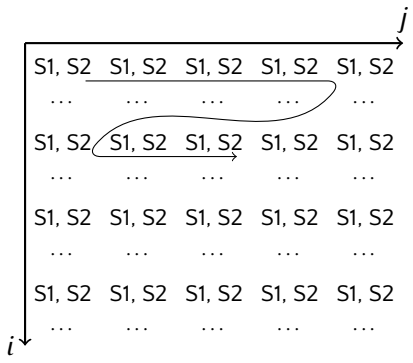


Figure 1.5 – Mapping from code to a regular grid for classical loop-tiling

would not showcase this regularity as can be observed in Figure 1.6.

The semi-regularity of the computation implies that it should be possible to represent the data reuses of all iterations with the same *memory-use graph*. However by its definition the graph only contains the statements of a single iteration whereas data reuses often cross multiple iterations. If we want to reflect this fact while keeping the form of the *memory-use graph* more information has to be added to the graph, again inspired by the dataflow programming paradigm.

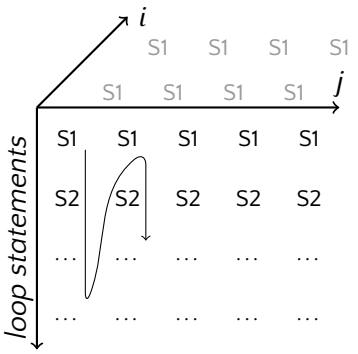
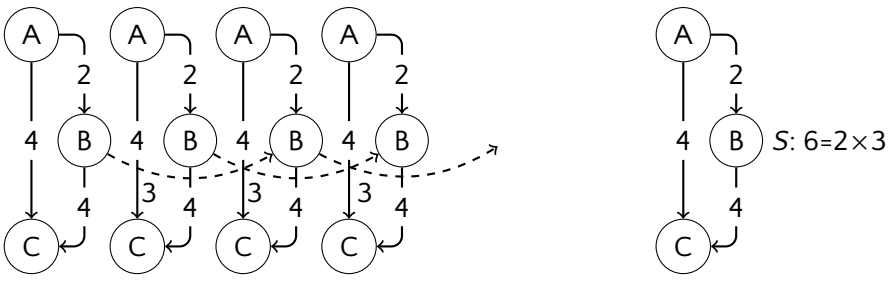


Figure 1.6 – Semi-regular grid

1.3.2 State data

When provided with a large set of input values, a dataflow program may need to keep specific data between the processing of separate input values. Examples of such values could be counters and accumulator variables that are associated with such stateful processing. The same concept can be applied to the reuse of data between different iterations of a loop.

For the structure of the *memory-use graph* we associate *state information* with each node by providing the size of the state data that should be transmitted from iteration to iteration. Because our goal is only to reduce the amount of IO operations of the analyzed code we choose not to distinguish between separate state data elements that are transmitted by the same node to subsequent iterations. This means that a node in our graph only has a single state data attribute that groups together all the



(a) Fully unrolled graph (b) Condensed graph

Figure 1.7 – Construction of state data annotations in a *memory-use graph*

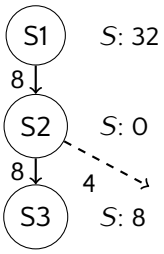


Figure 1.8 – Example with states

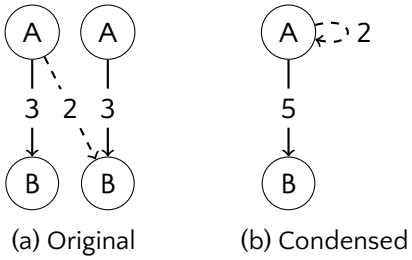


Figure 1.9 – Cross-node reuses

sizes of all state data elements that this node transmits.

The state data attribute is not necessarily equal to the combined sizes as the data reuse of a specific element may occur in the next iteration or in any other following iteration. In turn this means that a node may need to store several iterations worth of state data if the reuse is not immediate as illustrated in Figure 1.7. The exact amount of memory required is the size of the state date of a single iteration multiplied by the number of iterations between reuses of the data. This state data reuse can then be represented as a loop-back edge within the *memory-use graph* or as an attribute of the node (noted *S*). The effect of this phenomenon on the example representation can be observed in Figure 1.8 on the state data of S1.

In the case where a reuse is performed across multiple iterations, but not within the same node, it can still be described using both state data and intra-iteration reuses. The value is first transported as a state data of the node that produces it to the target iteration. From there a standard reuse within the iteration itself transfers the data to the target node. This transformation is shown in Figure 1.9 and applied to the example in Figure 1.10. Modifying the *memory-use graph* in such a way may introduce cycles which is problematic as we need the graph to be acyclic. Several solutions can be considered:

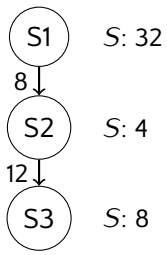


Figure 1.10 – Condensed example

- Simplifying the graph by considering the nodes within each strongly-connected component as a single super-node. This effectively masks the cycles in all further analyses performed on the *memory-use graph*. However by reducing the number of nodes in the graph this increases the granularity at which the code is considered and as such may decrease the quality of the tiling and indirectly increase the IO cost as well as the associated operational intensity.

- Not considering the reuse for the optimization of memory usage and removing the conflicting edge from the *memory-use graph*. This means that the reuse will in any case contribute to the code's IO cost. Even though this does not increases the granularity of the representation it still increases the IO cost. Furthermore, as the reuse is not included in the graph, this may raise issues when performing tiling because it remains equivalent to a Read-After-Write dependency without being accounted for in our representation. As a result this method for the removal of a cycle in the graph is not considered.

More specific information on the ways in which the state data information is precisely obtained and added to the graphs in the different use-cases is discussed below. For loops this is done in Chapter 4 and for dataflow programs this is detailed in Chapter 5.

Now that edges of the *memory-use graph* are annotated with data sizes and nodes are labeled with state data weights all reuses are taken into account and represented. One element however has still to be considered: the fact that nodes, while being atomic elements in the representation, are not atomic in their execution or their usage of memory.

1.3.3 Internal computational requirements

Because of the abstract notion of statements, a node may represent hundreds or thousands of lines of machine instructions. As such its execution may produce intermediate values that do not appear within our reuse representation but still require memory at some point. In a more general manner a node in the *memory-use graph* represents an instruction as described in Section 1.3.1 and an instruction can be anything from a single machine-instruction to a high-level statement that performs a complex computation on its arguments.

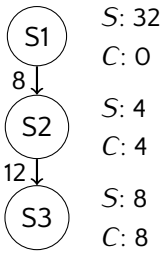


Figure 1.11 – Example with internal computation requirements

Because of this the computation of a single node may require more memory than the sum of the weight of its state data and the size of all incoming reuse edges. For such cases a second attribute is added to each node: the *internal computation requirement*. The value of this attribute is computed by subtracting the state data and the incoming reuses from the total amount of memory required by the nodes execution. For the example code we computed the values for the

internal computation requirement of each node (noted C) by writing a naive assembly code equivalent for each statement. The result is presented in Figure 1.11. In a general manner the computation of the exact memory use of a statement is not a focus of this study but an example can be given: when the exact content of the node is known under the form of machine instructions it suffices to count the number of registers that are used. A detailed application of this principle can be found in Chapter 4.

Adding the internal computation requirements to the *memory-use graph* completes its semantics. As a final step we will give a short and more formal definition.

1.3.4 Formal definition of a memory-use graph

For a code snippet containing a set S of statements that are considered to be executed in an atomic fashion, the corresponding *memory-use graph* is a directed-acyclic graph $G = (V, E)$ where:

- $V = \{v \in \mathbb{N}^2\}$
- $E = \{e \in V \times V \times \mathbb{N}\}$
- $\exists f : S \twoheadrightarrow V$
- $\forall (u, v, r) \in E$ at most r data is produced in $\{f^{-1}(u)\}$ and used in $\{f^{-1}(v)\}$
- $\forall v = (s, i) \in V$ at most s data is available during the computation of $\{f^{-1}(v)\}$ and should be available for the computation of $\{f^{-1}(v)\}$ at the next iteration
- $\forall v = (s, i) \in V$ the amount of memory required by the computation of $\{f^{-1}(v)\}$ is lower than or equal to $s + i + \max(\text{In}(v), \text{Out}(v))$, with $\text{In}(v) = \sum_{(i,j,r) \mid j=v} r$ and $\text{In}(v) = \sum_{(i,j,r) \mid i=v} r$

1.4 Memory usage of a code

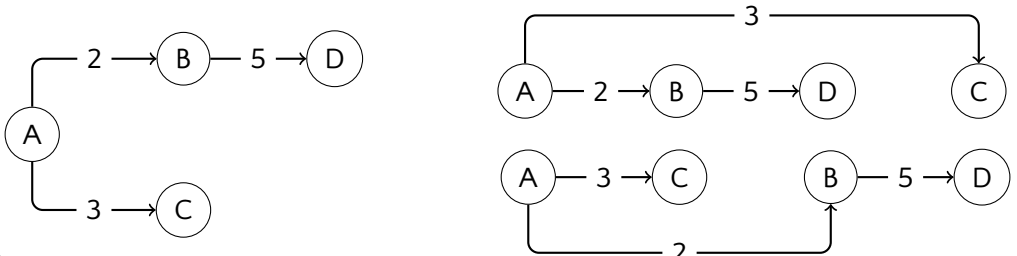
Once the *memory-use graph* of a code is generated, what is the amount of memory required to run the code. Even though we consider that the amount of memory required by the atomic execution of each node of the graph is a known parameter, the handling of several nodes is not straightforward. Multiple questions arise. Has the code already gone through register allocation? Do data objects have yet to be mapped to memory addresses or do we have to take an existing mapping into account? In which order are the nodes executed?

1.4.1 The impact of scheduling

When two statements in a code are linked through a reuse edge between their representative nodes in the *memory-use graph* their order of execution, *i.e* their schedule, is enforced by the direction of the edge. This is true whether they are part of the same iteration or correspond to the same node in two different iterations of the *memory-use graph*. If there is no such edge their schedule is unconstrained. In the case of two statements this is not a major issue. However if multiple statements are involved it becomes much more of an issue. This is of course not a new research question and the topic of scheduling has been a major topic since the advent of compiler research. Our goal here is not to introduce new approaches but instead to merely make the observation that the scheduling of the nodes of the graph, and their multiple instances, may have a significant influence on the associated memory usage.

See the example *memory-use graph* in Figure 1.12. Two different schedules are given for the corresponding statements of a single iteration. For the sake of simplicity the state and internal computation requirements are left out. For the first scheduling to be executed the maximum amount of memory necessary to store reused data is achieved at the end of node B and has a value of $3 + 5 = 8$. At the same time the maximum for the second scheduling is achieved at the end of both nodes A and D and has a value of $3 + 2 = 5$. Under the hypothesis of an available memory of size 6 the first schedule would generate communications with a distant memory whereas the second would not.

The main consequence of our observation here is that scheduling can influence memory usage. This has direct consequences on our tiling optimization presented in Chapter 3.



(a) Unscheduled *memory-use graph*

(b) Potential schedules.

Figure 1.12 – An unscheduled *memory-use graph* and two of its possible schedules

1.4.2 Register pressure & memory mapping

As mentioned in Section 1.3.3, we assume the memory size required for the execution of a specific node of the *memory-use graph* is known. But how is this quantity obtained?

On one hand, if the target is the optimization of register usage it is very likely that a node can be directly linked to one or more machine instructions. This means that we can evaluate the maximum register pressure during the execution of the code that the node represents.

On the other hand, if we want to optimize cache memory usage, a lot depends on whether the variables used within the code have already been assigned to specific memory addresses or not. In the case of such a memory-mapped code it is theoretically possible to count the number of cache lines that are referenced within each node. It is then possible to check if these referenced lines fit into the target cache considering both its size and its associativity. For non-memory-mapped code a similar approach would simply be to compute the amount of data that is accessed by the code's execution and compare it to the total cache size. This would however require the compiler and linker to perform a perfect memory-mapping later on and even then the result could potentially underestimate the real memory usage and IO cost.

Because the goal at this point is not to study the memory-mapping issues in depth, we assume for the moment that the memory use of every node is known when the *memory-use graph* is built. Some additional discussion is necessary when we attempt to extend our tiling approach to execution-trace analysis in Chapter 6 and we can now move on to the main focus of this section: the memory usage model.

1.4.3 A memory usage model

After the discussion about scheduling as well as the memory usage of individual nodes of the *memory-use graph* it should be clear that the model that we are looking for has one main goal: verifying that a given set of nodes subjected to a specified scheduling can be executed within a target memory without requiring any communications with a higher level in the memory hierarchy.

A key fact here is that such a set of nodes may potentially include the same node multiple times when the corresponding piece of code crosses iterations, *i.e* different instances of the same statement.

Evaluating the memory usage of a given set of node instances and schedule is similar to a *max-live* computation. A piece of data is considered to be *live* between

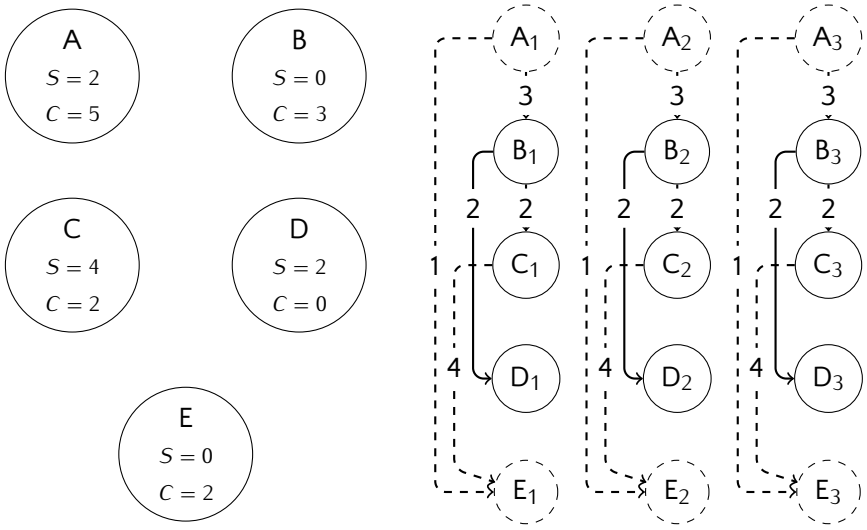


Figure 1.13 – Liveness of edges with respect to the memory usage of a partial schedule

the point of the schedule where it is produced and the point where it is last used. In other words, the edges of the *memory-use graph* actually represent *live ranges*. A particularity is that we only take into account data that flows between nodes belonging to the set of node instances for which we perform our memory usage evaluation. The set may not include all nodes of the *memory-use graph* and because of this edges that have an origin or a target outside the set are not considered to be *live*.

In Figure 1.13 the nodes of a *memory-use graph* are given with their state (S) and internal computation requirements (C). The graph is copied three times in order to represent three iterations. Node duplicates are indexed by the iteration to which each duplicate belongs. The set that is considered for evaluation corresponds to the nodes with a full circle. Following our definition of *live* reuses, only the edges drawn with a full line are considered as opposed to the edges drawn with dashed lines.

To illustrate the effect of scheduling we compute the memory usage of our set assuming two potential schedules: line-by-line and column-by-column.

Only reuse edges

At first, we do not consider state sizes and internal computation requirements. We introduce evaluation points between each node of the evaluated schedule. The total usage corresponds to the maximum amount of *live* reuses measured at each of the evaluation points. When applied, this produces two different values for the two proposed schedules:

1. **Schedule** $B_1 \to C_1 \to D_1 \to B_2 \to C_2 \to D_2 \to B_3 \to C_3 \to D_3$

   Memory usage $\max(4, 2, 0, 4, 2, 0, 4, 2) = 4$

2. **Schedule** $B_1 \to B_2 \to B_3 \to C_1 \to C_2 \to C_3 \to D_1 \to D_2 \to D_3$

   Memory usage $\max(4, 8, 12, 10, 8, 6, 4, 2) = 12$

State data

The next step is to include the state data. To do so we only need to extend the liveness principle to the related reuses. In our model this means that state data is equivalent to a *live range* between a node in one iteration and the same node in the next iteration. As state data may actually be the result of consolidated reuses over a distance of multiple, iterations this measure may be a conservative approximation of the real amount of data that is reused. We update the previous computations accordingly without changing the evaluation points. Again we find different usages for both schedules:

1. **Schedule** $B_1 \to C_1 \to D_1 \to B_2 \to C_2 \to D_2 \to B_3 \to C_3 \to D_3$

   Memory usage $\max(4, 6, 6, 10, 8, 6, 10, 4) = 10$

2. **Schedule** $B_1 \to B_2 \to B_3 \to C_1 \to C_2 \to C_3 \to D_1 \to D_2 \to D_3$

   Memory usage $\max(4, 8, 12, 14, 12, 6, 6, 4) = 14$

Internal computations

The last step is to include internal computation requirements. The memory usage that these represent is local to a node and does not influence edges. As such we need to add evaluation points at each node in the evaluated schedule. The memory use for a single node is considered to be equal to the maximum of the memory use before or after the the node's execution to which we add the internal computation requirement. This brings us to the final update to our memory-use computation:

1. **Schedule** $B_1 \to C_1 \to D_1 \to B_2 \to C_2 \to D_2 \to B_3 \to C_3 \to D_3$

   Memory usage $\max(7, 4, 8, 6, 6, 6, 13, 10, 12, 8, 8, 6, 13, 10, 12, 4, 4) = 13$

2. **Schedule** $B_1 \to B_2 \to B_3 \to C_1 \to C_2 \to C_3 \to D_1 \to D_2 \to D_3$

   Memory usage $\max(7, 4, 11, 8, 15, 12, 16, 14, 16, 12, 14, 6, 6, 6, 6, 4, 4) = 16$

With the previous example and the associated computations we introduced our memory usage model. We have also shown how this model can lead to a straightforward application on a *memory-use graph* and a (partial) scheduling of its nodes. The next step in the evaluation of a scheduling is its IO cost.

1.5 IO cost of a code

A tiling optimization of a piece of code attempts to reducte the number of IO operations between the different levels of a memory hierarchy through an increased data locality. For this to be possible we should in the first place be able to quantify the number of IO operations required by the code for a given scheduling and tiling. As we have not yet discussed and defined *generalized tiling* (see Chapter 3), we will give and use basic assumptions where necessary.

Because our goal is to target different levels of the memory hierarchy, we first look at some of the differences that may exist between these levels.

1.5.1 Registers are not cache

The major difference between processor registers and cache-memory resides in the manner in which they are accessed. Processor registers must be explicitly loaded with their content through load/store instructions but are the only ones that can be used within computations[1]. Cache memory, on the other hand, is accessed indirectly and IO operations with other levels of cache and distant memory are implicit, but the induced latencies for a cache-miss are much higher than for a cache-to-register load.

Another important difference is that a cache-line is much larger (typically 64 bytes) than a single register or load/store operation (typically 4 or 8 bytes). The result is that a cache-miss can in theory occur at every load or store operation with a new address if a new cache line is accessed each time. In such a case, we would be using only 1/16th or 1/8th of the capacity of each cache line in practice. The influence of this phenomenon can be anything from very extensive to almost non-existent and depends on the memory-mapping of variables combined with a schedule for the code as was already suggested in Section 1.4.2.

[1]We do not consider the possibility in x86 architectures to use memory-operands. Replacing such an operation by a load-compute or a compute-store sequence will not change the amount of IO operations and barely influences the general performance. Furthermore other target architectures do not support such a feature.

Because tiling is used on code that executes over a uniform iteration domain of significant size, it is reasonable to make the assumption that the underlying data also exhibits a fair amount of regularity in its organization and memory-mapping (arrays, buffers, ...). Considering this, we assume that cache lines are used in a near optimal fashion in code that is targeted by tiling.

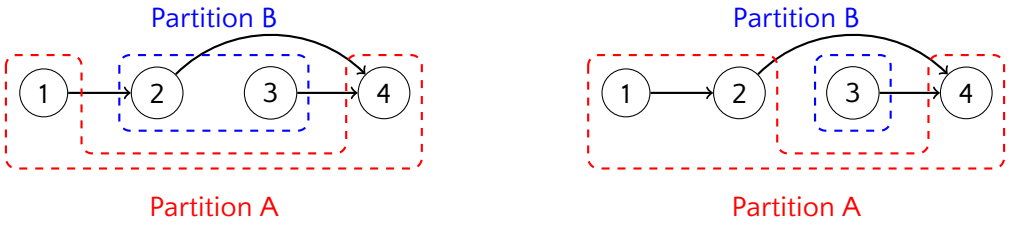
1.5.2 Node by node evaluation

Computing the IO cost of the entire code requires us to determine what reuses can be stored within local memory and what data must be transferred to a more-distant memory and thus contributes to the IO cost. Our approach here is to partition the execution of the code into separate pieces. Because the entire execution can be represented by k copies of the *memory-use graph*, k being the number of iterations that the code performs, the partitioning is done on the nodes of these k copies.

For our evaluation of IO the cost to be valid, two criteria are to be verified by the partitioning:

1. Each individual partition should admit a scheduling whose memory usage, according to our model, is lower than or equal to the target memory size.

2. Each partition should be executable in an atomic fashion. As such there should be no data dependency path that starts in a partition, passes outside of it and then flows back in. Such a path would require code exterior to the partition to be executed to complete the partition, thereby going against the atomic property. A graphical representation of this property is given in Figure 1.14 and it is further discussed in Chapter 6 under the form of convex partitioning of a directed acyclic graph.



(a) Path $1 \to 2 \to 4$ goes from A to B to A: it is a non-convex partitioning.

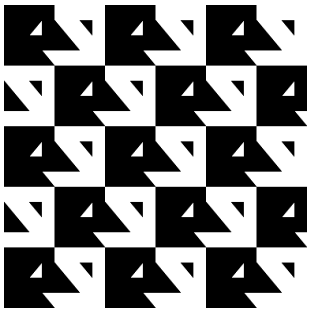
(b) There is no path from A to B to A or from B to A to B: it is a convex partitioning.

Figure 1.14 – Example and counter-example for convex partitioning

With such a partitioning of the code in hand we can define the IO cost of the code in our model. The first constraint on the partitions imposes that all reuses belonging to a single partition can flow through the target memory and as such do not generate any communications with distant memory. The remaining reuses between partitions are the only ones that generate IO operations that contribute to the cost of the code as a whole.

For each partition we count the amount of data that originates outside of it. This may include state data when the nodes representing the copies of a same node in two consecutive iterations do not belong to the same partition. The amount of data entering a given partition is potentially different from the sum of the weights of all incoming edges. Multiple edges may represent the same data element that is reused by different nodes of the same partition; such duplicates should not be counted in the IO cost computation.

We consider incoming data and thus we only account for load operations in the IO cost of the code. Store operations are not taken into account because a given data element is only produced once. If it is a result that should be kept the associated store will never be removed as it is part of the computation performed by the code. If it is an intermediate result and the scheduling allows it, it is never even written to the more distant memory. In case it can not be kept, it is written to a distant memory before being loaded at least once for a reuse. For these reasons the number of store operations related to reuses is less then the corresponding load operations count. As a consequence the minimization of load operations also directly gives a tighter upper-bound for the number of store operations.



Chapter 2

Communication Costs in a Manycore Environment

The bird launched itself north and east, straight as an arrow toward Tar Valon. After a moment's thought Ronde prepared another copy on another narrow strip of thin parchment and fastened it to a bird from another coop. That one headed west for she had promised to send duplicates of all of her messages.

R. Jordan – *The Fires of Heaven*

2.1 Context and goal

After considering the memory usage of a given code snippet as well as the number of IO operations that are required by its execution, we move on toward a different kind of communications. If we want to apply our tiling method to an environment with distributed resources, communications can go beyond a simple memory hierarchy. For example, it may be necessary to use explicit data communications between the different processors of a distributed system. Or, if we consider the case of manycore processors, these explicit transmissions flow through a Network-on-Chip that link the cores on a single chip. The communications supporting the distribution of computations over different resources require the highest degree of stability in characteristics such as latency and bandwidth. The current chapter aims at providing the necessary theoretical framework to provide this stability.

As the work described in this manuscript was performed during a collaboration with a manycore chip manufacturer, this study of the memory communications beyond the cache-hierarchy is focused on the case of Network-on-Chip-based architectures. The study of traffic behavior as well as its control on Network-on-Chips through Quality-of-Service protocols has been identified as a major issue since the inception of the technology [Mar+09]. For the case of distributed systems beyond manycores, a significant amount of literature exists on the topic of network performance, Quality-of-Service and latency evaluation.

Before looking at what elements and behaviors we want to model and describe, we first need to give some definitions and terminology related to the topic. In no way is this an in-depth survey of all existing Network-on-Chip variants. It should nonetheless allow the reader to clearly understand the context within which our work falls as well as the limits to the theoretical and practical results that will be presented.

After this short introduction, we focus on the presentation of existing network traffic characterizations. Such theoretical models are used both to describe traffic as it is observed on a network, and as a set of constraints that may be forced onto a network flow in order to achieve a desired behavior.

We then move on to the analysis of both the advantages and drawbacks of using a Quality-of-Service regulation on data flows within a Network-on-Chip. Such a regulation can take many different shapes: some rely on back-pressure generated by routers throughout the network, others require to model the behavior of the whole network.

The last part of this chapter reports on a case-study that applies a Quality-of-

Service regime to the MPPA processor family. Because of its architectural specifics, the regulation requires a theoretical network model that is able to compute Quality-of-Service parameters. This work was also published in two separate papers [Din+14b; Din+14a]. A second step is the redesign of the physical bandwidth-limiter that applies the computed parameters at hardware level. The new design from this second step, which is presented in Section 2.5.3, is implemented in the second generation of the MPPA chip family and is the subject of a filed patent.

2.2 Architecture of a general-purpose Network-on-Chip

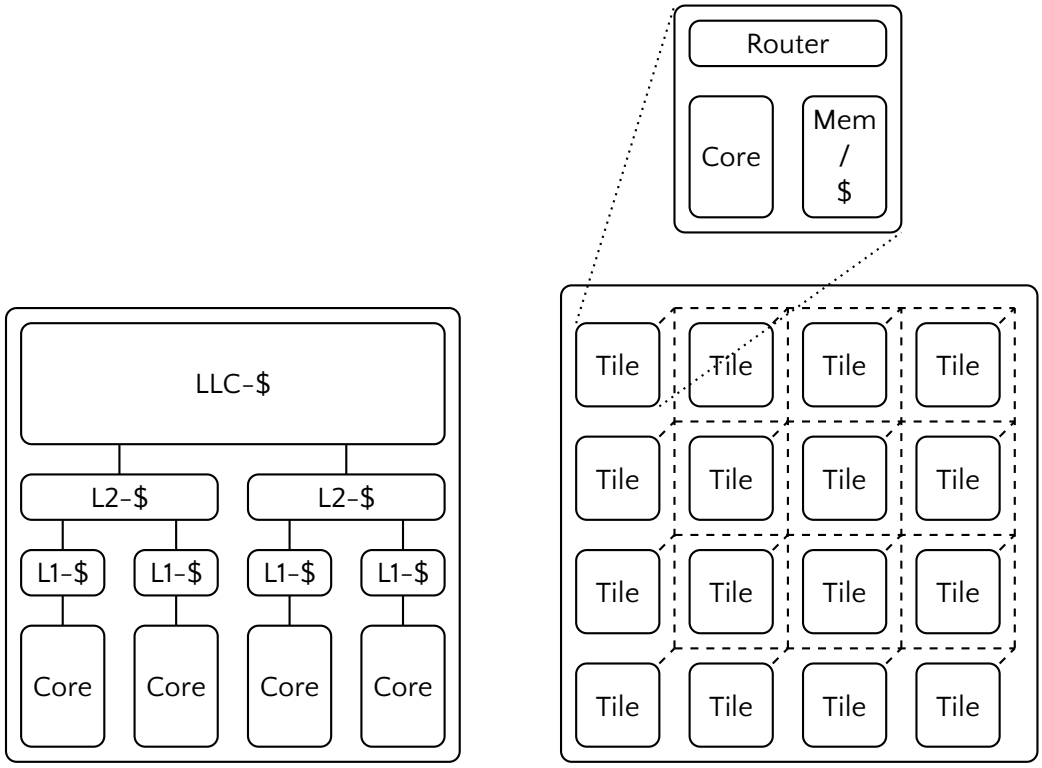
A Network-on-Chip is an infrastructure used by the computations of a manycore processor. In a general manner, it can be seen as a downsized version of the network within a distributed cluster or supercomputer. This is not surprising as the manycore concept is actually based on the idea of adapting the logic behind these larger scale systems to fit into a single chip. The most important elements for our study are:

1. The memory architecture for which the Network-on-Chip serves as interconnect
2. The topology and routing logic of the network
3. The types of traffic that are expected

2.2.1 Memory architecture

Existing multicore processors use a monolithic memory hierarchy. All cores share at least the same last-level cache. Lower cache levels may be divided into separate units that each serve a subset of the cores. This creates a tree-like shape that is referred to as the memory hierarchy. The different levels and units of this cache hierarchy implement a hardware-based chip-wide cache coherency in the overwhelming majority of such architectures. For manycore processors however, a similar approach would raise significant issues: required surface and power consumption for the coherency logic, exponentially increasing latencies for cache-misses, ...

The paths taken by existing manycore architectures, such as the Tile-Gx family [Jag+12] or the MPPA family [Din+14b], differ but a general tendency is to distribute the memory over the numerous cores to create small independent local memories that may better benefit from data locality within parallel computations. The association of a computational core and some local memory creates an element known as a



(a) Multi-core processor (b) Manycore processor

Figure 2.1 – Memory organizations for multicore and manycore processors

processing tile or *processing element*. These elementary units are then duplicated to achieve the desired core-count within a manycore processor and they are linked all together by a Network-on-Chip. This approach does not prohibit the implementation of hardware-based cache-coherency but the protocols involved in the process differ from their multicore counterparts because they are distributed by design, as for example directory-based coherency. Yet another alternative is to implement software-based cache-coherency or to forgo cache-coherency entirely. This does however puts a significant burden on the programmer or the run-time environment in order to maintain memory consistency.

The structural differences between the multicore and the manycore approaches are summarized and illustrated in Figure 2.1.

2.2.2 Data traffic

As can be deduced from the description of the memory architecture in Section 2.2.1, the Network-on-Chip's main purpose is to ensure the transfer of data between processing elements and potentially to off-chip interfaces such as DRAM ports or Ethernet interfaces. However this single purpose hides a multitude of different data communications that may or may not share common characteristics and constraints.

For example, depending on the architecture chosen for the memory layout, processing tiles may contain only cache memory or also some local memory accessible by the local core. In the latter case, it is entirely possible to allow cores to transfer data between the memory of a distant tile and the local memory. This generates explicit and thereby somewhat predictable data communications as the sequence of a program's execution is known. Cache communications, however, are implicit and unpredictable because their behaviour depends on a multitude of factors.

Cache communications should be treated in such a way as to minimize latency. Explicit communications however may, depending on the use-case, require bandwidth optimization instead. These potentially opposite goals may influence our choices when we want to apply a Quality-of-Service protocol in Section 2.4.

The communications themselves are performed in a packet-switched channel-based fashion. A channel is opened at the source of a data transmission with a given destination and some other parameters. The data that is pushed through a channel is cut into packets of fixed size which are then individually and sequentially sent through the Network-on-Chip. Each packet is prefixed with a header containing elements such as destination, target memory address and other administrative information. In turn a packet can be divided into so-called flits[1] which are the atomic transmission elements.

As the network is located on the chip itself, links between nodes are materialized by a parallel set of wires. Usual links are 32 or 64-bit wide. With respect to this fact a *flit* is a data unit equal to the width of a link. *Flits* are also the usual level of granularity at which other elements of the Network-on-Chip function, especially the routers. As can be seen in Figure 2.1 each processing tile of a manycore processor has a core and some local memory or cache but also contains a router. This router is both part of the tile and part of the Network-on-Chip on which the tiles are positioned.

[1] The term flit is a contracted form of the expression *Flow control digit*.

2.2.3 Routers & routes

The usage of a Network-on-Chip is a natural evolution of earlier architectures with multiple components that used 1-to-1 links or busses. The increase in the number of elements in the network made the area and energy costs of dedicated 1-to-1 links prohibitive whereas in the case of busses the global available bandwidth has become insufficient thereby forming a performance bottle-neck. As has been the case in the past with computer networks the next step was to introduce switching. Just like this technique has enabled the construction of vast networks with thousands of nodes it has brought the same kind of advantages to manycore architectures.

While switching can either be done in the form of *circuit switching* or *packet-switching* it is the latter one that is predominantly used in computer data networks and it is also the packet-switching method that is used by existing Network-on-Chip architectures. While transiting over the network, data packets need to be routed, an action that is performed by the previously described routers present in each processing tile. In turn this implies the computation of routes for each data packet so it can move from its source to its destination tile through the network topology[2].

Routers and their implementations in manycore architectures take up various designs as their exact function is heavily dependent on the type of routing on which the network is based. In the case of Network-on-Chips, the most frequently encountered routing methods are:

Virtual Channels The original goal of this technique was to implement virtual *circuit switching* over a packet-switched network. This means allocating resources at each router on the route of a communication before sending data which may guarantee better latency and/or bandwidth performances. The allocated resources include buffer space in the routers and access to the switching fabric leading to the next router. In the case of Network-on-Chips the main advantage compared to *wormhole switching* is to make a better use of resources at each router, especially buffer memory.

Wormhole Switching Wormhole switching can partially be seen as a simplified version of virtual channels, without resource allocation. The term wormhole is used because of the possibility for *flits* to traverse routers without waiting for the other *flits* of the same packet to arrive whenever the next link in the packet's route is not in use. This is in opposition to a *store-and-forward* mechanism. Routers do

[2]Network topologies are an entire research topic to themselves that predates the arrival of Network-on-Chip architectures and are not discussed in detail here.

not contain any logic except for route decoding as routes are determined and encoded in the packet's header before it is emitted.

A logical consequence of having different types of routing and router designs is that modeling the behavior of traffic on the network may be more or less challenging. As the goal of this chapter is to describe a model for Network-on-Chip communications we will focus on a specific architecture later on. Firstly, however, we should have a general view of the types of existing models.

2.3 State of the art

2.3.1 Describing network traffic

With the basic notions of Network-on-Chips and their architecture stated we now present an oversight of the existing literature on the modeling of communications over such a network. The foremost element in such a model is the traffic itself: data is injected at nodes throughout the network topology but it would be of great help if we could know how it is injected, the rate at which it is done, etc...If we can model such characteristics, it should then be possible to describe the behavior of the traffic when it flows through the different constituents of the Network-on-Chip.

Traffic regulation and its optimization have been explored in the macroscopic case since the first computer networks appeared in the sixties. Initially, the most common issues involved the over-subscription of link capacities, *i.e* too much data needed to be transferred through links with not enough bandwidth. Global source rate adaptation was identified as a convenient way to avoid network congestion and router buffer saturation. The work of Kelly [KMT98] jointly redefined fairness and optimality: a formalization for an optimal distributed method for rate calculation based on the Lagrangian method. Network calculus was initiated by Cruz [Cru91a; Cru91b] in order to compute tight bounds for end-to-end transfer delays in the presence of sporadic traffic bursts, which he named "distortion".

2.3.2 Network calculus

In the case of Network-on-Chip regulation and optimization most authors originally considered the problem from the bandwidth sharing viewpoint. Several adaptive rate control techniques have been proposed, such as "on/off" regulation using predictive control [OM06], proactive or reactive regulation based on buffer occupancy [TL11],

flow control with a max-min fairness guaranty [Jaf+08] and others. The arrival of the (min,+) network calculus [LBT01] opened up the possibility to model the behavior of traffic at a more detailed level, especially with respect to traffic backlog in router buffers. This framework was used by Jafari et al.[Jaf+10] to compute the optimal sizes for buffers that are required for a given Network-on-Chip traffic configuration.

2.3.3 The (σ,ρ) calculus

The (min,+) network calculus does however assume fluid flows and, because of that, ignores packetization effects. This results in pessimistic guarantees and bounds for traffic characteristics. The (σ,ρ) calculus of Cruz [Cru91a; Cru91b] and Zhang [Zha95] does not ignore these effects. It is this (σ,ρ) network calculus that we use in the remaining part of this chapter to describe both the behavior on the Kalray Network-on-Chip and control it through the use of Quality-of-Service implemented through source rate adaptation.

2.4 Quality-of-Service: why and how?

Being able to model the behavior of traffic on the Network-on-Chip brings has great advantages and enable a better understanding of what can or will happen in the case of specific data emission patterns. It does however not offer any control or possibility to avoid certain behaviors. Hence the use of a Quality-of-Service protocol.

2.4.1 Adverse network behavior

Without any form of control over the network and its behaviour there are many situations in which communications may generate situations that are detrimental to the global stability of the network. Some examples of such behaviors:

Unbounded latency Under heavy load the observed latencies in a network tend to grow in an unbounded way. This goes against the idea of being able to guarantee specific latencies for communications.

Deadlock The extreme case of unbounded latencies is actually infinite latency. Depending on factors such as network topology and routing policies it may be possible for a Network-on-Chip to experience deadlock. Even though most network topologies allow specific route computation schemes that guarantee the absence of deadlock by construction, this is not always the case.

In both cases, the underlying issue is a bad use of the available resources, *i.e* the links and buffers of the Network-on-Chip. A Quality-of-Service protocol may be used to balance the resource allocation which in turn guarantees the stability of the network.

Many such protocols use back-pressure, *i.e* network congestion information flowing back to the source of a flow of data, and balance the emission rate of flows accordingly. However when no information or back-pressure is sent back to the origin of the flows, beyond a total deadlock or saturation of the network, such a method can not be applied. Instead the entire network needs to be modeled and its behavior predicted. This allows for the computation of specific constraints for the emission of data by each individual channel. These constraints should be established to guarantee network stability in all cases.

A clear disadvantage of a Quality-of-Service protocol is that, through the enforcement of constraints the available bandwidth for each data flow may be significantly reduced in comparison to when no constraints are applied.

2.4.2 Predictable latencies

Beyond the prevention of adverse behavior imposed by a Quality-of-Service protocol, bounds on the latencies experienced by data communications are of great importance when mapping and scheduling a distributed computation on a manycore processor.

When a specific part of a computation depends on data from another part that is mapped onto a different processing tile, a transfer must take place, either explicit or implicit through cache misses. To prevent the stall of the execution due to unavailable data, some buffering is necessary. The amount of buffering that is necessary grows with the maximum latency that the data transfers may experience. This means more memory must be allocated thereby reducing the amount available for the computational code itself. Being able to predict maximum latencies as well as the possibility of reducing them has a great impact on the overall performance of the code: be it its execution time or its IO cost.

Again, a Quality-of-Service protocol can be used. The maximum latency target can be a parameter of the computation of the constraints that should be enforced on the data streams and the maximum latency can depend on the code's scheduling and mapping to the available processing tiles. As an example of a Quality-of-Service protocol we take the MPPA processor and its Network-on-Chip. This protocol is used later on in Chapter 5.

2.5 (σ,ρ) traffic on the MPPA manycore processor

The MPPA manycore processor is used as a target architecture for the code optimizations that are described within this study. Therefore it is also the architecture for which we wanted to create a Quality-of-Service protocol. Because of the specifics of its Network-on-Chip architecture, we could not use a method based on back-pressure. Instead we needed to fully model the behavior of flows on the network. The (σ,ρ) network calculus has been selected for this purpose. Before we go into the details of the adaptation of this model to the MPPA Network-on-Chip we give a description of its architecture.

2.5.1 Architecture of the Network-on-Chip

As a manycore processor, the first generation of the Kalray MPPA family [Din+14b] contains 288 cores. This number is divided into so-called clusters: 4 IO clusters and 16 computational clusters. Each IO cluster contains 4 cores and each compute cluster contains 17 cores among which one has a managing function over the cluster and is called the *resource manager* (RM). Additionally each cluster has a single network interface which manages data transfers as well as command and control messages via the Network-on-Chip that links all clusters and IO clusters together.

Clusters

Each computational cluster can be considered as a separate processing tile in the manycore model of Figure 2.1 (page 30). A single cluster has a shared local memory and can thus be considered as a classic multicore processor with 17 cores. Each core has an individual data-cache and instruction-cache. The caches of a cluster however do not benefit from hardware-based coherency between them and as such care should be taken when implementing the synchronizations of multi-threaded code. The access to the Network-on-Chip is done through its local network interface which is referred to as *Direct Memory Access* interface. This interface is directly linked to the network router associated with the tile. The organization of a single cluster is summed up in Figure 2.2 (page 37). IO clusters have a similar architecture from the perspective of access to the Network-on-Chip and their specificity is their direct access to their own set of IO interfaces for off-chip communication, including access to the DRAM memory.

Network interfaces

The *Direct Memory Access* (DMA) unit of each cluster, composed of receiving side (Rx) and a transmitting side (Tx), is able to hardware manage eight separate outgoing flows of data simultaneously. Such a flow is referred to as a *channel* and is defined by the combination of the source and destination clusters, a route between them, as well as the parameters for packet generation and emission. Before sending data a core has to request a channel from the DMA unit and if the request is granted it can push data from the shared memory of the cluster into the channel. The DMA then starts writing the data to a dedicated buffer memory that is physically seperated from the shared memory. A summary of the described design is found in Figure 2.3.

Each channel is also allocated with a packetshaper that is responsible for grouping data into packets, prefixing the packet payload with a header, acquiring access to the local router and sending the packet. Additionally each packetshaper also features a bandwidth limiter that can control the rate at which data packets are injected into the network. This is the only point where data streams on the network can be controlled in order to enforce a Quality-of-Service protocol.

Network topology

The Network-on-Chip itself is organized in order to maximize the available bandwidth both between computational clusters and between computational clusters and their IO counterparts. The topology that is used is an incomplete 2D-torus with some extra links. From a functional point-of-view the fact that the torus is not complete makes

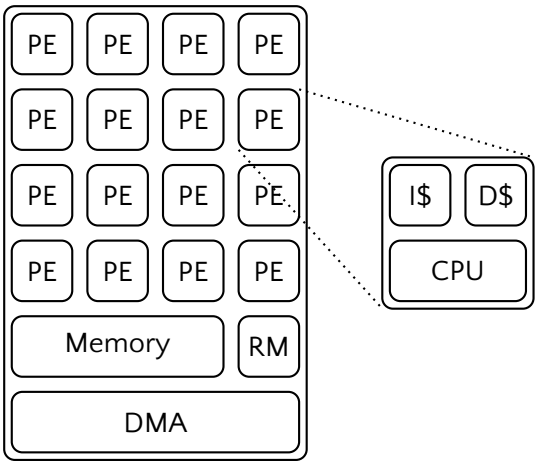


Figure 2.2 – Schematic description of an MPPA computational cluster

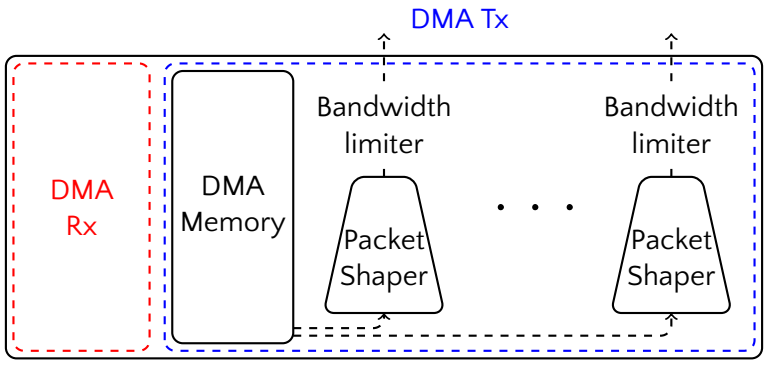


Figure 2.3 – Design of the DMA interface of a computation cluster

it impossible to avoid deadlocks through the use of specific routing strategies such as XY-routing [Sei+88] and turn-models [GN94]. In practice deadlocks have been observed. The full topology is presented in Figure 2.4. Some of the links are dashed just to increase the readability of the figure.

Routers

At the nodes of the network topology each computational cluster features a Network-on-Chip router directly linked to its DMA interface. Each IO cluster features one router per core (two cores in the second generation MPPA chips). All routers have an identical design: one local interface and four interfaces that connect the router to the neighboring nodes in the topology. The interfaces are distinguished by their direction: north, east, south, west and local. Routing in the MPPA chip is done in a wormhole fashion. Furthermore the routing is decided at the source cluster and encoded in each packets header under the form of a set of directions. The routers have no routing logic beside the ability to decode from a packet header the direction of the next router in its path. A representation of a router interface is shown in Figure 2.5.

When a packet arrives at the router the header is decoded and updated and the packet is forwarded to the target direction through a buffer. Whenever the link in the target direction is already in use by the another transmission the packet is stored in a buffer flit by flit. Each unique combination of source and target direction has its dedicated buffer with a capacity of 512 flits. Each outgoing direction is fed by four different buffers corresponding to four possible sources. The access to each link is arbitrated through a round-robin policy on its four buffers. Whenever a buffer is full, the upstream traffic is blocked at the hardware level to avoid any data loss.

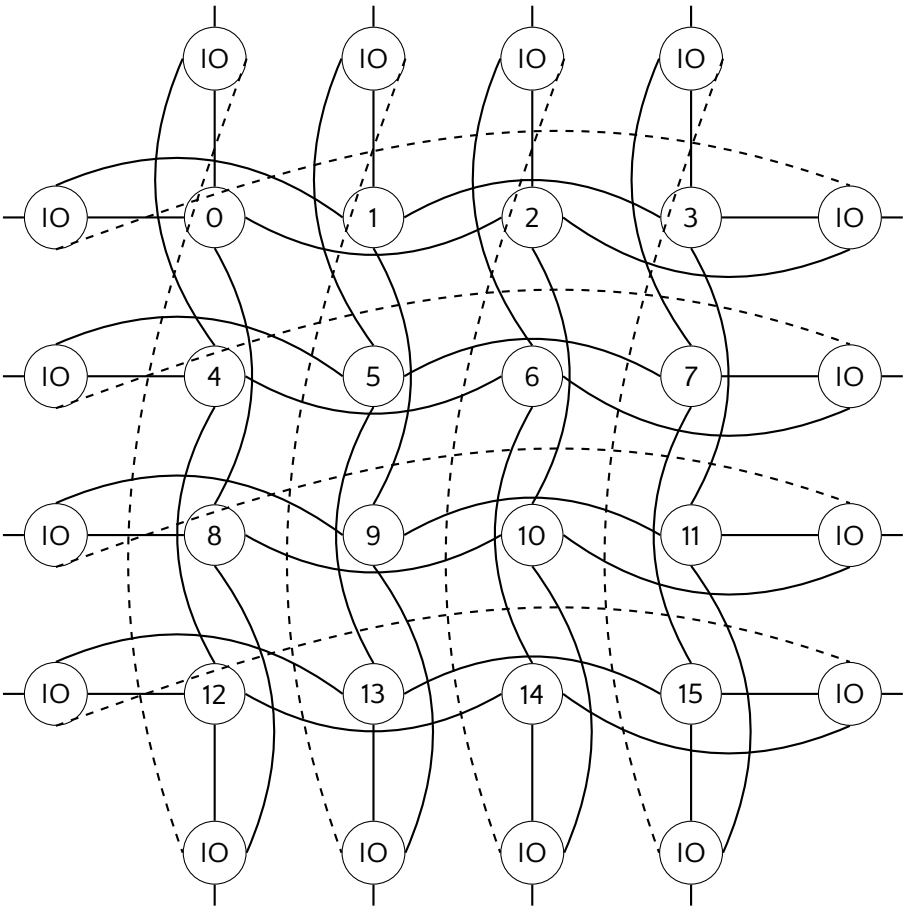


Figure 2.4 – Topology of the MPPA Network-on-Chip

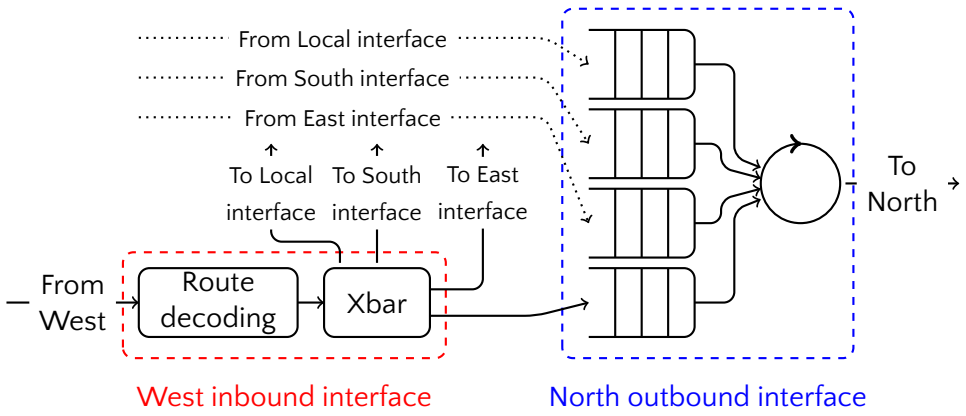


Figure 2.5 – Architecture of a router interface

σ_f, ρ_f	Burstiness and rate of flow f at injection
σ_f^q	Burstiness of flow f after buffer q
$\rho_f^{\min}$	Required minimum rate of flow f
m_q	Number buffers contending with buffer q
d_l	Local delay of link l (link traversal)
d_q	Local delay bound of buffer q
$F(l)$	Set of flows that use link l
$F(q)$	Set of flows that use buffer q
P_f	Links used by flow f
B_f	Buffers used by flow f
B_f^q	Buffers used by flow f up to buffer q
L	Fixed size of all packets
Q	Fixed size of a router buffer

Table 2.1 – Variables used in the (σ, ρ) model

Even though data loss is prevented, the fact that traffic blocks the upstream links and routers may generate deadlocks if a cycle is created among the existing channels that are active on the Network-on-Chip. For this reason we now look into the traffic model on the MPPA network.

2.5.2 Applying the model

As stated earlier the traffic model that use is the (σ,ρ) network calculus previously introduced by Cruz [Cru91a; Cru91b] and Zhang [Zha95]. The advantage of this model over other types of network calculus is its ability to account for the effect of packetization whereas other models mostly assume fluid flows. A second advantage, and consequence of the previous point, is that the (σ,ρ) network calculus tends to give less pessimistic values for the maximum latency. The last argument in its favor is that the design of the bandwidth limiters present in the DMA interfaces enables us to directly control both the ρ and the σ characteristics of their associated channels.

The goal of applying such a regulation at the source of each channel is to prevent any overflow in one of the buffers present in the Network-on-Chip routers. Such a guarantee would implicitly prevent any deadlock from happening as there would always be space for a flit to advance from its entry interface to its exit interface's buffer. This constraint is equivalent to formulating a backlog constraint on the flows that are using the Network-on-Chip. The variables that are used in the remaining part of this chapter are summarized in Table 2.1. Below we formulate a set of constraints that, when satisfied, should give the required guarantees.

The rate ρ_i of a flow f is expressed as a ratio of the bandwidth of one of the net-

works links. Because all links have the same bandwidth we can formulate a capacity constraint for each link l as:

$$\sum_{f \in F(l)} \rho_f \leq 1 \tag{2.1}$$

Furthermore a flow may be required to sustain a minimal rate in order to satisfy the application that produces it. Thus we have a minimal requirement constraint for each flow f:

$$\rho_f \geq \rho_f^{\min} \tag{2.2}$$

According to its definition the σ_f characteristic of a flow f is the current maximum backlog. Because we can regulate its value at the source we need to propagate the value of σ over the full path of the flow, adjusting the value where necessary at each router traversal. This is done through the relation between σ_f and the σ_f^q at each buffer q in B_f^q that is provided by Zhang [Zha95]:

$$\sigma_f^q = \sigma_f + \rho_f \cdot \left(\sum_{q \in B_f^q} d_q \right)$$

His formula makes use of the local delay bound d_q at each queue q that the flow traverses. In the case of the MPPA Network-on-Chip this delay bound is derived from the fact that link access arbitration among contending buffers is performed thru a round-robin policy. When m_q buffers are contending with buffer q (m_q includes buffer q):

$$d_q = (m_q - 1) \cdot L$$

If we apply this to our criterion that no buffer overflow may occur we obtain our backlog constraint for every buffer q:

$$\begin{aligned} Q &\geq \sum_{f \in F(q)} \sigma_f^q \\ Q &\geq \sum_{f \in F(q)} \sigma_f + \rho_f \cdot \sum_{b \in B_f^q} d_b \\ Q &\geq \sum_{f \in F(q)} \sigma_f + \rho_f \cdot L \cdot \sum_{b \in B_f^q} (m_b - 1) \end{aligned} \tag{2.3}$$

A strong emphasis should be put here on the fact that L should indeed be a constant for all flows. If flows make use of different packet sizes then the constraints in (2.3)

become invalid. In the case of the MPPA Network-on-Chip this is not a hindrance as the packetshapers in the DMA interfaces always form packets of the specified size.

By combining the three constraint sets deduced from (2.1), (2.2) and (2.3) we obtain a system that can be solved through *linear programming*. In this case the objective function should lead to an resource allocation that respects some kind of fairness criterion. We took the results from Kelly [KMT98] which lead to a logarithmic function $U((\rho_f)_f)$:

$$U((\rho_f)_f) = \sum_f \log(\rho_f)$$

As it is this function will minimize the σ_f of each flow which is not an ideal solution. Burstiness gives flexibility to data injection and is thus necessary. To counter this effect we introduce a second logarithmic term to the objective function:

$$U((\rho_f)_f, (\sigma_f)_f) = \omega \cdot \sum_f \log(\rho_f) + (1 - \omega) \cdot \sum_f \log(\sigma_f)$$

The ω factor allows to express a preference for either higher σ values or higher rates.

The ultimate goal of modeling traffic and using a Quality-of-Service protocol to limit the used bandwidths is to compute an upper-bound on the latency of data transfers. However the validity of a result working towards this goal [Din+14a] has been the subject of intense discussion. Multiple attempts have been made in order to provide another strong upper-bound on transfer latencies, but none of them has resulted in a viable and valid solution.

2.5.3 Packet injection - the design of a bandwidth limiter

Because the bandwidth limiters in each DMA interface are the elements responsible for injecting packets with the correct (σ, ρ) characteristics, their design should be flexible enough to allow a wide range of potential configurations.

First generation

The first generation of the MPPA chips features bandwidth limiters based on a sliding window design. A global time-window size T_w is defined for an entire DMA interface and each bandwidth limiter is allotted a certain transmission quota N_f for its associated flow f. The limiter then enforces that never more then N_f flits were emitted over the last T_w cycles. Going back to the (σ, ρ) characterization, ρ_f was directly defined

through:

$$\rho_f = \frac{N_f}{T_w}$$

Because T_w was set at cluster or even chip-level, this only leaves the N_f as parameter to individually configure each channel. As a result the σ_f value was thereby directly linked to ρ_f via:

$$\sigma_f = N_f \cdot \left(1 - \frac{N_f}{T_w}\right) = \rho_f \cdot (1 - \rho_f) \cdot T_w$$

An obvious consequence was the reduced flexibility in the configuration of each channel. And when this result is inserted into the backlog constraints (2.3), it creates quadratic terms in ρ_f thereby prohibiting the direct use of *linear programming* to solve the system. Therefore it is necessary to linearize the constraints. This can potentially reduce the quality of the solution and thereby increase the end-to-end delay.

A final, but nonetheless important drawback is that the design did not allow for a parametrization with fine granularity. It was thus impossible to configure the exact values obtained through *linear programming* which induced again a conservative approach and a decrease in performance.

Second generation evolution

For the second generation of the MPPA family we proposed a redesign of the bandwidth limiters that was accepted by Kalray. The new design had to satisfy the following criteria:

1. Allow the separate parametrization of both ρ and σ for each channel.
2. Allow for fine-grained parametrization.
3. Generate traffic with a characterization as close as possible to the specified ρ and σ while remaining conservative (*i.e* the observed values should be lower then or equal to the configured ones).

For the first criterion, it appeared that a credit counter would offer the easiest configuration. A packet can only be emitted if the number of credits available for the channel is greater or equal than the size of the packet. The accumulation of credits can be done through a so-called *leaky-bucket* mechanism where a credit is acquired every time a fixed number N of cycles elapses. In such a circuit the value of ρ could be easily configured as $\rho = 1/N$. The value for σ can be a configurable maximum amount of credits C_{max} that can be acquired.

To satisfy the second and third criteria the period (in cycles) defining the accumulation rate of credits should be configurable with great precision. On one hand using an integral value is not possible as this would result in configurable ρ values that go from $N = 1$ ($\rho = 1$) to $N = 2$ ($\rho = 0.5$) without any intermediary steps. On the other-hand floating-point arithmetic operators are significantly more expensive gate-wise, which was also not an acceptable solution.

A potential solution is to use fixed-point arithmetic. It has the gate-count of integer operations while offering better precision than simple integers. We however came up with yet another approach that could further minimize the gate-count of the bandwidth limiters for an equivalent precision. This design, which we present below, is the subject of a patent currently under review.

Non-linear integral counter

Our solution is based on the use of three distinct integral counters encoded on k bits:

Credit counter This counter contains the currently available number of credits within the bandwidth limiter. Its maximum value Λ is configurable.

Cyclic counter This is the main counter that directly implements the *leaky-bucket* mechanism as described above. The counter is incremented each cycle by 1 and when a configurable amount Φ of cycles has been reached a credit is added to the credit counter and the cyclic counter itself is reset to 0.

Shift counter This counter is incremented each cycle by a configurable amount Ω. As the representation of the counter is confined to k bits the incrementat is performed modulo 2^k. Each time an overflow occurs (*i.e* the increment result would be greater then $2^k - 1$) a signal is sent to the cyclic counter that prevents it from incrementing at the current cycle. As a result the credit counter will not be incremented after Φ cycles but after $\Phi + 1$.

The use of the *shift counter* and the subsequent shifting of *credit counter* increments simulates non-integral values for the increment period associated with the *leaky-bucket* approach. In order to compute the values of the parameters Λ, Φ and Ω we start from the (σ, ρ) characteristics that should be enforced.

Credits should be accumulated at rate ρ. Because the *shift counter* can only increase the increment period the base period should be the nearest inverse of an inte-

ger greater then or equal to ρ. Thus:

$$\Phi = \min\left(2^k - 1, \left\lfloor \frac{1}{\rho} \right\rfloor\right)$$

As a result we have to slow down the increment by ε cycles where:

$$\varepsilon = \frac{1}{\rho} - \Phi$$

This difference in rate should be covered every Φ cycles corresponding to a catch-up rate of $\Phi/\varepsilon + 1$. As the overflow of the *shift counter* occurs at 2^k this leads to a value for Ω of:

$$\Omega = \min\left(2^k - 1, \left\lfloor \frac{2^k}{\Phi/\varepsilon + 1} \right\rfloor\right)$$

Finally, because credits are accumulated at a rate ρ, the value of σ can directly be related to the maximum amount of credits that can be accumulated by a channel as this corresponds to maximum burst-size that can be injected:

$$\Lambda = \rho$$

A full diagram of a possible implementation for the corresponding circuit can be found in Figure 2.6. Within this diagram the ω (resp. φ and λ) register contains the current value of the *shift counter* (resp. *cyclic counter* and *credit counter*). Furthermore the α parameter allows for credits to be accumulated by increments different from 1, namely α.

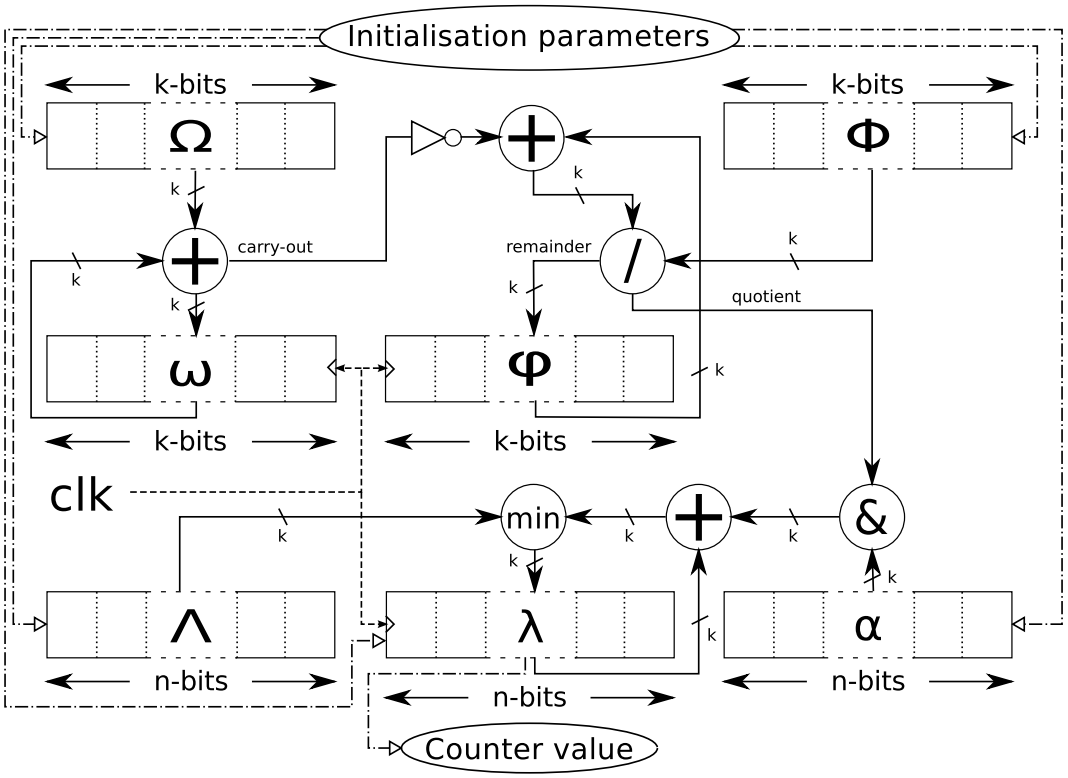
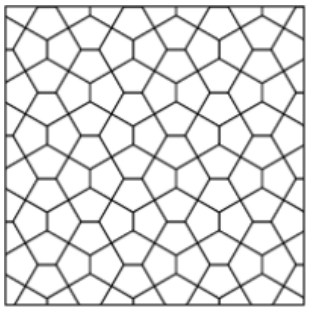


Figure 2.6 – Possible implementation of the non-integral leaky-bucket counter



Chapter 3

Generalized Tiling

She was impressed at how neatly the columns added up. Her mother had explained that often a quartermaster would make many messy notations, referencing other pages or other ledgers, separating different types of supplies into different books, all to make it more difficult to track what was going on.

R. Jordan & B. Sanderson – *Towers of Midnight*

3.1 Context and goal

The memory and IO cost models defined in Chapter 1 are mainly linked to the definition and evaluation of *memory-use graphs*. Even though the computation of a memory-cost is already related with a partial schedule of the considered graph and its iterations, the notion of tiling has not yet been introduced.

To clearly describe what we refer to as *generalized tiling* we need to draw some comparisons with the more classical tiling that has been used in the case of loop optimizations. From there on, we give a definition and representation of tiling in the new context of the *memory-use graph* as well as a formalization of the associated optimization problem. Next two approaches are presented for solving the previously stated problem. The first one is the use of the Constraint Programming paradigm and has the ability to find optimal solutions at the expense of high search times. The second approach is based on the use of heuristics that target non-optimal solutions and requires significantly less computations.

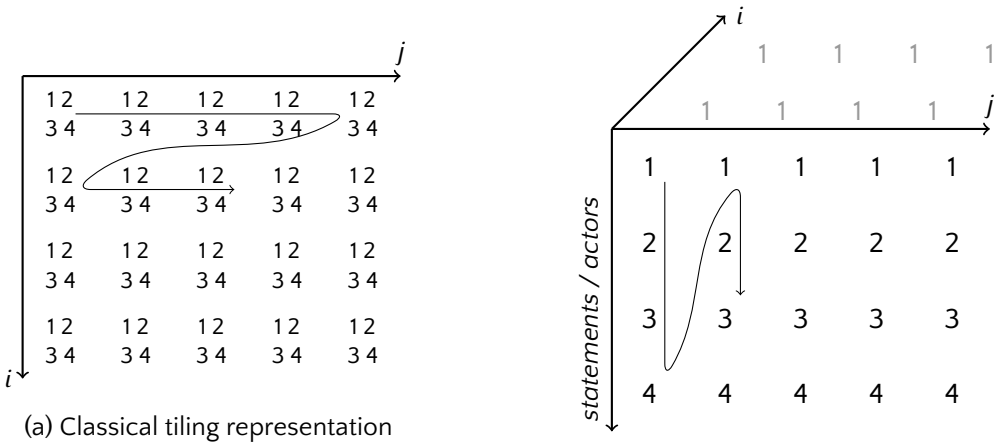
3.2 A change of perspective

Traditionally, tiling has been used in the case of loop optimization. It refers to the tessellation that this technique performs on the iteration space of a loop nest. As we described above in Section 1.3.1, the usage of the *memory-use graph* relies on a change in the representation of loop iterations: use the set of statements within a loop as an extra dimension that can be represented as illustrated in Figure 3.1.

In a similar way we can illustrate the modification that is generated by *generalized tiling* with respect to the existing methods. On one hand, the existing methods only act at the control-flow level by changing loop induction variables and adding extra loops within a loop-nest. On the other hand, *generalized tiling* also includes instruction rescheduling as well as loop splitting and unrolling. The result, with the modified order of execution of the loop body and its iterations is in Figure 3.2.

3.3 Tiling model

Applying tiling to the *memory-use graph* requires some modifications. First and foremost, the existing methods assume that the execution of an iteration of the considered loop body is atomic. Thus these methods model the iteration-space of a k-nested loop as a k-dimensional space of identical points with identical dependency vectors

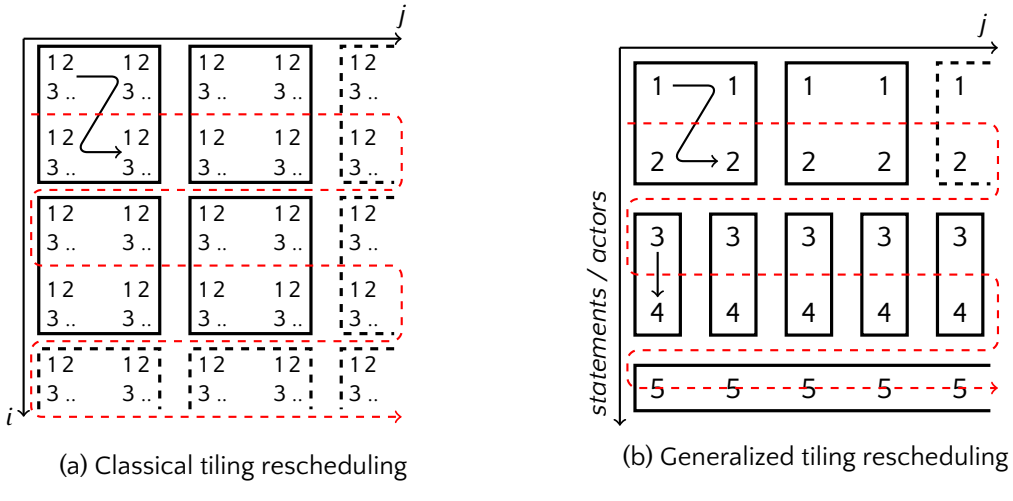


(a) Classical tiling representation

(b) Generalized tiling representation

The numbers 1, 2, 3 and 4 are the statements of the loop represented in its *memory-use graph*.

Figure 3.1 – Modification of the iteration space representation



(a) Classical tiling rescheduling

(b) Generalized tiling rescheduling

Intra-tile schedule

Inter-tile schedule

Figure 3.2 – A comparison of classical and generalized tiling

for each point. This leads to the fact that the resulting tiling is uniform over the entire iteration-space.

Taking another direction, the definition of the *memory-use graph* explicitly considers the statements of a loop body seperately. If we then map this representation back to the multi-dimensional space that is traditionally used for tiling, a k-nested loop will be represented with $k + 1$ dimensions. One of them is non-uniform with respect to the dependencies as it represents the computational steps of an iteration. Any tiling approach on such a space thus does not need to be necessarily uniform on this dimension.

The *generalized tiling* approach described in this Chapter starts with a representation of what tiles are and how that relates to the scheduling of the vertices within the *memory-use graph*. With such a representation at our disposal we can then formalize the optimization problem that needs to be solved in order to actually obtain an efficient tiling solution, which minimizes the number of memory communications. Both the representation as well as the formalization have been previously published in a research paper [Dom+16].

3.3.1 Tile shapes and representation

As stated above a *generalized tiling* solution includes a non-uniformity along the dimension representing the statements of the loop body. On one hand the tiles used to cover the iteration space do not all have the same size. On the other hand however the constraints and requirements of code generation, be it for loops or dataflow programs, require a certain regularity of the tiling solution along all the other dimensions. The code transformation associated with *generalized tiling* is a four-step process:

1. Linearize the *memory-use graph* on which the tiling is performed as the graph's edges may not enforce a unique order. This is equivalent to the issue of scheduling the instructions within a loop body or the actors of a dataflow program.

2. Separate the chosen graph linearization into multiple chunks. The non-uniformity of the tiling appears as each chunk contains different vertices and the number of included vertices per chunk may vary.

3. Copy each chunk multiple times in order to represent several iterations of the underlying vertices. This creates the effective tiles. The number of copies of each vertex may vary from tile to tile but is constant within the same tile. This value is also referred to as the width of the tile. A constant width within a tile ensures

the uniformity of the tiling over the other dimensions of the iteration-space while still propagating the non-uniformity over the intra-iteration dimension.

4. Within each tile perform some rescheduling of the vertices and their copies.

An illustration of these steps is given in Figure 3.3.

Using the same reasoning for a representation we can define a single tile on a graph $\mathcal{G} = (V, E)$ as follows

$$[v_1, \ldots, v_n]^k \quad | \quad (v_i)_{i \leq n} \in V, \;\; k \in [\![1; +\infty]\!]$$

where $(v_i)_{i \leq n}$ are the n vertices included in the tile in the order of the schedule used for the tiling and the parameter k represents the number of iteration copies that are made. A value of $+\infty$ for the number of copies indicates that the width of the tile is not limited. This may occur on tiles containing only a single vertex ($n = 1$) and these do not require any unrolling in the code generation.

A full *generalized tiling* solution for $\mathcal{G}$ can be defined by the consecutive tiles that it contains For the example featured in Figure 3.3a and if we suppose that the used schedule is the numerical order of the node IDs (*i.e* $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$) then the tiling obtained in Figure 3.3e can be written as

$$[1, 2]^3 \;\; [3, 4, 5]^2$$

A side effect of the constraints put on the shapes of tiles is that it enables a clear formalization of the optimization problem that needs to be resolved to find the best possible *generalized tiling* for a given *memory-use graph*.

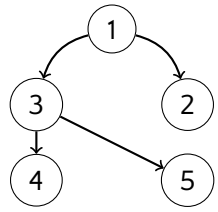
3.3.2 Formalization of the optimization problem

At the input of our optimization problem we have an instance of a *memory-use graph* representing a single iteration of repeatedly executed code. The form and content of the graph follows the definition and description given in Chapter 1. A second input is the size of the memory level for which tiling should be performed. The goal then is:

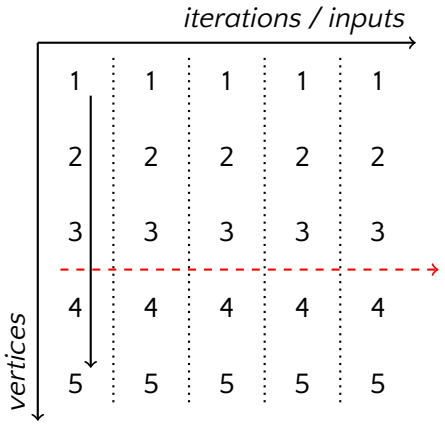
1. To find a topological ordering of the vertices of the graph.

2. To partition the ordering into tiles.

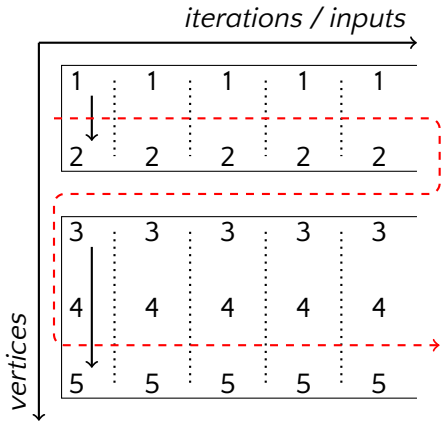
3. To find the a combined solution to steps 1 and 2 that minimizes the IO cost.



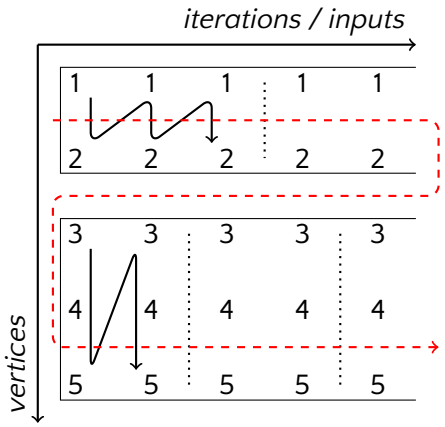
(a) Initial *memory-use graph*



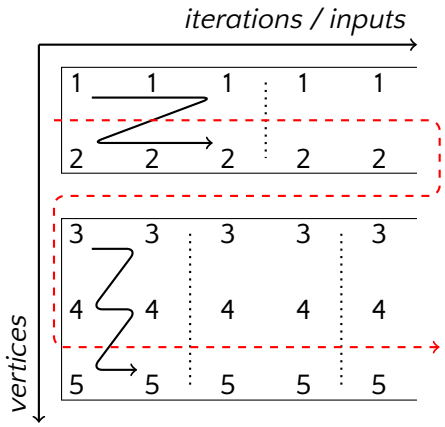
(b) Linearization of the graph



(c) Tile cutting



(d) Vertex copying



(e) Vertex rescheduling

Dotted lines indicate the separation between iterations in the generated code.

Figure 3.3 – Step-by-step code transformation by *generalized tiling*

For the computation of the IO cost, we consider the following indications:

- The memory-cost for each tile is a function of its width and computed under the following assumptions:

  Schedule - The vertices in a tile are executed row by row, top to bottom.

  Input - Data from edges entering the tile are loaded at the last possible moment, *i.e* just before the vertex that uses them.

  Output - Data produced by a vertex in the tile and used outside the tile is stored to a memory level higher than the targeted one as soon as possible, *i.e* just after the vertex that produced it.[1]

  IO free - Only input edges lead to loads, all internal edges are supposed to flow through the target memory.

- Tiles are assigned the largest possible width while not exceeding the target memory size with their memory-cost.

With these elements in mind we define the inputs, variables, cost functions and constraints of the optimization problem.

Inputs

- $\mathcal{G} = (V, E)$ the *memory-use graph* that is to be tiled. It's definition is given in Chapter 1 (page 18).
- C the size of the target memory.
- For each $v \in V$ we define $S(v)$ the state size and $C(v)$ internal requirement size of v.
- For each $e = (u, v, w) \in E$ w is the weight of the data carried by e from u to v and we define $W(e) = w$.

Variables

- $\mathcal{L} : V \leftrightarrow [\![1; |V|]\!]$ a bijective linearization function for $\mathcal{G}$.
- $T = \{t \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}\}$ the set of tiles represented by tuplets of (*start position*, *end position*, *width*). For each $t \in T$ we define $s(t)$, $e(t)$ and $w(t)$ the projectors on the members of the tuplet.

[1] This type of behavior can be achieved using non-temporal hints on store operations in x86 and with uncached writes on the Kalray architecture.

Cost function

- We define a point in the execution of a tile t as a position i in a linearization $\mathcal{L}$ of $\mathcal{G} = (V, E)$ and as the number $j \leq w(t)$ of the last executed instance of the vertex $\mathcal{L}^{-1}(i)$. The position $i, 0$ corresponds to the point before the first execution of $\mathcal{L}^{-1}(i)$.

- For a given tile $t \in T$ the edges internal to t under linearization $\mathcal{L}$ are defined as

$$L_{\mathcal{L}}(t) = \{(u, v) \in E \quad | \quad s(t) \leq \mathcal{L}(u) < \mathcal{L}(v) \leq e(t)\}$$

- For a given point (i, j) in $t \in T$ under linearization $\mathcal{L}$, the current memory footprint $\mathcal{F}_{\mathcal{L}}(i, j)$ is defined as

$$\begin{aligned}\mathcal{F}_{\mathcal{L}}(i, j) = \; & S(\mathcal{L}^{-1}(i)) \; + \; C(\mathcal{L}^{-1}(i)) \\ & + \sum_{\substack{e=(u,v)\in L_{\mathcal{L}}(t) \\ \mathcal{L}(u)<i<\mathcal{L}(v)}} w(t) \cdot W(e) \\ & + \sum_{\substack{e=(u,v)\in L_{\mathcal{L}}(t) \\ \mathcal{L}(u)=i}} j \cdot W(e) \\ & + \sum_{\substack{e=(u,v)\in L_{\mathcal{L}}(t) \\ \mathcal{L}(v)=i}} (w(t) - j) \cdot W(e)\end{aligned}$$

  The memory usage of the entire tile $\mathcal{M}_{\mathcal{L}}(t)$ is then expressed as

$$\mathcal{M}_{\mathcal{L}}(t) = \max_{s(t)\leq i\leq e(t)|0\leq j\leq w(t)} (\mathcal{F}_{\mathcal{L}}(i, j))$$

- For a given tile $t \in T$ under a linearization $\mathcal{L}$, the average IO cost per iteration $\mathcal{IO}_{\mathcal{L}}(t)$ can be expressed as

$$\mathcal{IO}_{\mathcal{L}}(t) = \frac{\sum_{s(t)\leq i\leq e(t)} S(\mathcal{L}^{-1}(i))}{w(t)} + \sum_{\substack{e=(u,v)\in E \\ \mathcal{L}(u)<s(t)\leq\mathcal{L}(v)\leq e(t)}} W(e)$$

- By extension, the IO cost for the full tiling T under linearization $\mathcal{L}$ is

$$\mathcal{IO}_{\mathcal{L}}(T) = \sum_{t\in T} \mathcal{IO}_{\mathcal{L}}(t)$$

Constraints

- Ensure that the linearization of $\mathcal{G}$ is valid

$$\forall (u, v) \in E, \;\; \mathcal{L}(u) < \mathcal{L}(v)$$

- Ensure that the tiles are well defined

$$\forall t \in T, \;\; s(t) \leq e(t), \;\; w(t) \in [\![1; +\infty]\!]$$

- Ensure that tiles are consecutive and non-overlapping

$$\exists F : T \leftrightarrow [\![1; |T|]\!] \quad | \quad t_1, t_2 \in T, \;\; F(t_2) = F(t_1) + 1 \implies s(t_2) = e(t_1) + 1$$

- Ensure that the memory-cost for each tile is inferior to the target memory size

$$\forall t \in T, \;\; \mathcal{M}_{\mathcal{L}}(t) \leq C$$

For all couples $(\mathcal{L}, T)$ that respect these constraints the goal is to minimize $\mathcal{IO}_{\mathcal{L}}(T)$.

Solvers

The shape of the optimization problem has a high complexity and finding an optimal solution may be very expensive from a computational point of view. With this in mind we now present two different approaches to find valid solutions to the tiling problem.

3.4 The use of Constraint Programming

3.4.1 The Constraint Programming paradigm

A first method that can be used to find a solution for the optimization problem is the *constraint programming* (CP) paradigm [RBW06]. This approach falls in the category of *optimal* solvers as their goal is to provide an optimal solution to the given problem with respect to the specified objective function. Another type of frequently used optimal solver is the *integer linear programming* (ILP). Both CP and ILP are based on the formulation of a set of constraint that characterize the problem to solve.

In the case of our specific optimization problem, the CP approach has the advantage of offering a large portfolio of *global constraints* [BD13] such as for example *all*

different [Rég94] and *global cardinality constraint* [Rég96] that encapsulate and "incrementally solve" parts of the problem by performing incremental domain filtering during the execution of the search algorithm. This removes the need for decomposition into linear constraints and auxiliary variables otherwise necessary in ILP.

Another, secondary, advantage of CP is that is can benefit from problem-specific search heuristics, based on expert knowledge. This may offer considerable speed-ups for the search by quickly filtering out sub-optimal solutions.

3.4.2 Solving the tiling problem

For the implementation of the generalized tiling algorithm the JaCoP [SK13] system was used. The problem formulation and the constraints are directly derived from those given in Section 3.3.2. The control variables, *i.e* those that are actually used to determine the values of all other variables and cost functions, are the linearization $\mathcal{L}$ and the set of tiles T with their positions and widths.

The search method that is used to explore the domain of all possible solutions is the depth-first algorithm with 2-way branching. The order in which variables get assigned values depends on the number of constraints in which they are involved. The ones involved in the most constraints get chosen first and the number of constraints for each remaining variable is updated. Assignment of values to the variables is done following different strategies. On one hand the linearization is chosen at random within its constraint-defined domain. Tile dimensions on the other hand start with maximized values within their domain.

By definition the search method is *complete* in that it covers the entire solution space if given enough time. The necessary time, however, may be beyond reasonable limits. This means that the search can be stopped at an arbitrary point and the best, possibly sub-optimal, solution found up to then is returned.

3.5 Tiling with heuristics

The *contraint programming* approach described in Section 3.4 finds optimal solutions to the tiling problem at the cost of computations with exponential time-complexity. The heuristic approach finds good or even near-optimal solutions while limiting itself to short and fast algorithms, typically with linear or quasi-linear complexity. Such heuristics can also be re-used in the CP solver to speed-up the search for optimal solutions.

3.5.1 Scheduling and tiling: isolated or intertwined?

As described in Section 3.3 our formulation of the optimization problem involves both the linearization of the tiled *memory-use graph* as well as the computation of tiles that cover the iteration space. From an algorithmic point of view this raises the question if these two distinct actions should be performed separately one after the other or if they should be done simultaneously in a collaborative fashion. This is quite similar to the question whether register allocation and scheduling should be done together or separately.

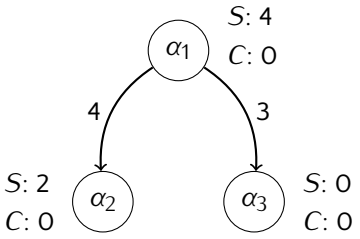


Figure 3.4 – Non-linearized *memory-use graph*

If we take the example of Fig. 3.4 we can see that there are two possibilities for the scheduling of vertices α_1, α_2 and α_3. Which schedule should be chosen depends, among other things, on the size of the target memory:

- With a size of 32, both schedules would allow the whole linearization to fit in the target memory for 4 iterations. Reloading the state of all three vertices at each tile border gives an average cost of 1.5 loads per iteration.
- With a size of 16, however, the best schedule would be α_1, α_2, α_3. This way α_1 and α_2 can form a tile of width 3 and α_3 forms an infinite-width tile. The average cost would be 5.33 loads per iteration. Inverting α_2 and α_3 would raise this cost to 7 loads per iteration with α_1 and α_3 in a tile of width 4 and α_2 in an infinite-width tile.

A possible interpretation of this behavior is that it is preferable to schedule the successors of a vertex by the weight of the reuses that lead to them in descending order. This reduces the loads in case not all successors fit into a single tile. When it comes to tiling the scheduled graph, we can also reduce the amount of spilling by sizing the tiles to include each of the most heavy-weighted data reuses in a single tile. Another option would be to size the tiles in a manner that locally minimizes the average IO cost per vertex.

3.5.2 Heuristics

We present three different heuristics that use the weight of edges as a criterion for optimization. The first one performs decoupled scheduling and tiling. The second

and third make decisions in a simultaneous fashion where the current state of tiling may influence the choices made for scheduling. An experimental evaluation of the proposed algorithms is carried out with the two implementations of *generalized tiling* presented in Chapters 4 and 5.

Heavy-edge / Greedy

The first proposed heuristic starts with the computation of a linearization of the *memory-use graph*. As preliminary work an arbitrary, valid, linearization is chosen.

Next all edges are ordered by their weight in a decreasing order. Edges are processed one after the other and each of them is *contracted* and *frozen*.

An *edge contraction* schedules the source and destination of the edge as close to each other as possible. This means that the only vertices that remain between the source and the destination are members of an alternative path between both vertices. An *edge freeze* means that the vertices between the source and destination cannot have their relative scheduling be modified.

A consequence of an *edge freeze* is that any further *edge contractions* may not schedule any vertices within the interval between the source and destination. It remains however possible to reschedule a vertex located after the interval to a position before the interval and vice versa, as long as the schedule remains legal. An illustration of the *edge contraction* and *freezing* is given in Fig. 3.5. Because the processing order of the edges is determined by their weight we refer to this heuristic as *Heavy-Edge* scheduling. The pseudo-code can be found in Algorithms 3.1 and 3.2.

Once all edges have been contracted, or all vertices have been frozen in their schedule, we can start the computation of a tiling solution. This is done in a greedy and iterative fashion and is therefor named *Greedy* tiling:

1. Take the first statement in our schedule that does not yet belong to a tile.
2. Compute the average spill-cost per super-actor and per iteration for all possible tiles starting at the selected statement:
   (a) Select a valid end-point for the tile and compute the maximum allowed width.
   (b) Iterate over all possible end-points.
3. Choose the tile with the lowest average spill-cost per statement and add it to our solution and select the next tile start-point at step 1.

Algorithm 3.1 Pseudo-code for Heavy-Edge scheduling

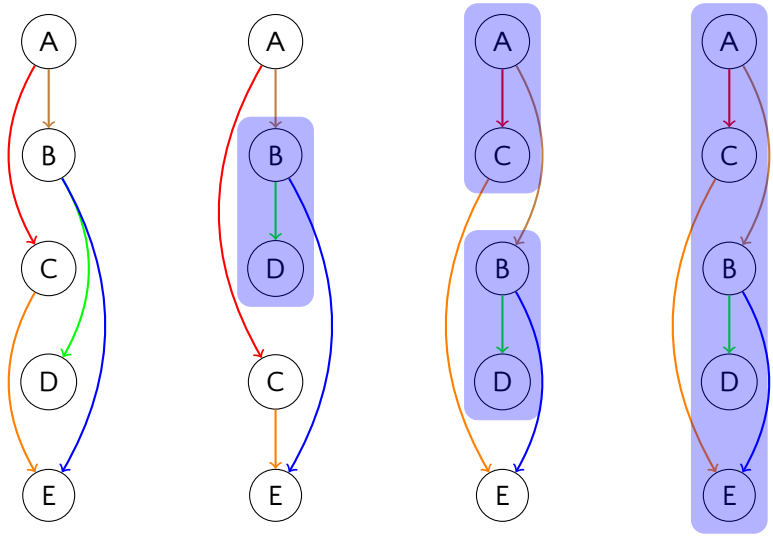
Input: `schedule` - Vector of the vertex schedule
Input: `frozen` - Interval map of the frozen parts of the schedule
Define: `workQueue` - Priority-queue of edges based on their weight

```
1:  function MoveForward(v, target)
2:      newSchedule ← target
3:      for all u ∈ v.successors do
4:          if u.schedule ≤ target then
5:              MoveForward(u, target)
6:          end if
7:          newSchedule ← min(newSchedule, u.schedule − 1)
8:      end for
9:      newSchedule ← frozen.lowerBound(newSchedule)
10:     Reschedule(v, newSchedule)
11: end function

12: function MoveBackward(v, target)
13:     newSchedule ← target
14:     for all u ∈ v.predecessors do
15:         if u.schedule ≥ target then
16:             MoveBackward(u, target)
17:         end if
18:         newSchedule ← max(newSchedule, u.schedule + 1)
19:     end for
20:     newSchedule ← frozen.upperBound(newSchedule)
21:     Reschedule(v, newSchedule)
22: end function

23: function ContractEdge(e)
24:     MoveForward(e.source, e.target.schedule − 1)
25:     MoveBackward(e.target, e.source.schedule + 1)
26:     frozen.insert(e.source.schedule, e.target.schedule)
27: end function

28: function FreezeSchedule(e)
29:     frozen.insert(e.source.schedule, e.target.schedule)
30: end function
```



Edge order by weight: green, red, orange, blue, brown

Blue zones have been frozen

Figure 3.5 – Example of scheduling by edge contraction and freezing

Algorithm 3.2 Pseudo-code for Heavy-Edge scheduling (ctd.)

```
31: function HeavyEdgeSchedule
32:     while workQueue.empty() do
33:         next ← workQueue.next()
34:         ContractEdge(next)
35:         FreezeSchedule(next)
36:         if frozen.contains(0, schedule.size() − 1) then
37:             break
38:         end if
39:     end while
40: end function
```

The pseudo-code corresponding to the combined *Heavy-Edge / Greedy* heuristic is given in Algorithm 3.3.

Tile-aware scheduling

The second and third proposed heuristics perform the linearization and tiling parts simultaneously and reuse the *edge contraction* from the first *Heavy-Edge / Greedy* heuristic. After performing a single edge contraction, one of two tiling heuristics, attempts to include the contracted edge in a single tile, possibly modifying previously created tiles. If this succeeds the next edge is contracted, if not, the edge contraction and the corresponding freeze are reversed before selecting the next edge for contraction. After all edges have been processed, a verification is performed to guarantee that all vertices belong to a tile. If some "holes" remain the *Greedy* tiling heuristic is used to fill them. The pseudo-code template for this *Tile-Aware* scheduling method with an arbitrary tiling heuristic is given in Algorithm 3.4.

Heavy-edge tiling

The idea behind the *Heavy-Edge* tiling heuristic is to maximize the reuses within an iteration and extending this to more iterations when possible. It should be combined with *Tile-Aware* scheduling to define the *Tile-Aware / Heavy-Edge* heuristic. Alternatively it can also be used with the previously defined *Heavy-Edge* scheduling heuristic. For each edge e processed by *Heavy-Edge* scheduling, three cases can occur:

1. No tile exists yet between the source and destination of e. A tile is created that covers the whole of e. If it is legal with respect to the size of the target memory it is added to the tiling solution and the tiling of e is successful. If not, nothing is done and the tiling of e is unsuccessful.

2. e is already internal to a tile. Nothing is done and the tiling of e is considered successful.

3. Multiple tiles exist between the source and destination of e. An attempt is made to create a unique tile to cover all of e as well as the entire range of the pre-existing tiles. If the resulting tile is legal it is added and replaces the existing tiles and the tiling is successful. If not, no modifications are made and the tiling is unsuccessful. It should be noted that the new, extended tile may be less wide than the pre-existing tiles.

The pseudo-code for this heuristic is Algorithm 3.5.

Algorithm 3.3 Pseudo-code for Greedy tiling

Input: `schedule` - Vector of a valid vertex schedule
Input: `capacity` - Size of the target memory
Output: `tiles` - List of the computed tiles
Output: `schedule` - Vector of the computed vertex schedule

```
1:  function CreateTile(begin, end)
2:      tile.begin ← begin
3:      tile.end ← end
4:      MaxWidth(tile)        ▷ Sets maximum width following memory usage model
5:      IOCost(tile)          ▷ Computes average IO cost per statement with model
6:  end function

7:  function GreedyTile()
8:      limit ← −1
9:      HeavyEdgeSchedule()
10:     while limit < schedule.size() − 1 do
11:         cost ← +∞
12:         horizon ← limit + 1
13:         repeat
14:             newTile ← CreateTile(limit, horizon)
15:             if newTile.width == 0 then     ▷ Means the tile is illegal for the model
16:                 break
17:             end if
18:             if newTile.cost < cost then
19:                 bestTile ← newTile
20:                 cost ← bestTile.cost
21:             end if
22:             horizon ++
23:         until horizon == schedule.size()
24:         tiles.push(bestTile)
25:         limit ← bestTile.end
26:     end while
27: end function
```

Algorithm 3.4 Pseudo-code for Tile-Aware scheduling

Input: `schedule` - Vector of a valid vertex schedule
Input: `capacity` - Size of the target memory
Input: `workQueue` - Priority-queue of edges based on their weight
Output: `tiles` - List of tiles
Output: `schedule` - Vector of the computed vertex schedule
Define: `memory` - Information necessary to reverse an edge contraction

```
1:  function TileAwareSchedule()
2:      limit ← −1
3:      while ¬workQueue.empty() do
4:          next ← workQueue.next()
5:          memory ← ContractEdge(next)
6:          memory ← FreezeSchedule(next)
7:          if ¬ComputeTile(next) then
8:              ReverseContractFreeze(memory)
9:          end if
10:     end while
11:     FillHoles()                ▷ Use Greedy tiling to fill up holes between tiles
12: end function
```

Tile-Aware / Conservative

The third and last proposed heuristic is *Conservative* tiling. It uses a variation on *Heavy-Edge* tiling. Together with *Tile-Aware* scheduling, this creates the *Tile-Aware / Conservative* heuristic. Instead of modifying existing tiles for a new edge, *i.e* case 3 in the previous paragraph, it considers the tiling of an edge unsuccessful if such modifications are necessary. The corresponding pseudo-code is presented in Algorithm 3.6.

3.6 From theory to practice

Both the Constrained Programming and the heuristics exposed in Sections 3.5 and 3.4 have been implemented seperatly as general solvers for the tiling of *memory-use graphs*. This allows for the computation of solutions to the abstract optimization problem. The next step from here is translating such abstract solutions to applications involving real code. The two specific targets selected for this purpose are the improvement of register reusage within imperative loops, presented in Chapter 4, and the reduction of cache-misses in the context of static and cyclo-static dataflow languages which is touched upon in Chapter 5.

Algorithm 3.5 Pseudo-code for Heavy-Edge tiling

Output: `tiles` - List of tiles in order of appearance in the schedule
Define: `tileStart`, `tileEnd` - Boundaries for a new tile

```
1:  function ComputeTileLimitsHeavyEdge(edge)
2:      tileStart ← edge.source.schedule
3:      tileStop ← edge.target.schedule
4:      for t ∈ tiles do
5:          if t.end < edge.source.schedule then
6:              continue
7:          else if t.begin > edge.target.schedule then
8:              break
9:          end if
10:         if t.begin < edge.source.schedule < t.end then
11:             tileStart ← t.begin
12:         end if
13:         if t.begin < edge.target.schedule < t.end then
14:             tileStop ← t.end
15:         end if
16:     end for
17: end function

18: function ComputeTilesHeavyEdge(edge)
19:     ComputeTileLimitsHeavyEdge(edge)
20:     tile ← CreateTile(tileStart, tileStop)
21:     if tile.width = 0 then
22:         return false
23:     else
24:         for all t ∈ tiles do
25:             if (tileStart ≤ t.begin) && (t.end ≤ tileStop) then
26:                 tiles.erase(t)
27:             end if
28:         end for
29:         tiles.insert(tile)
30:         return true
31:     end if
32: end function
```

Algorithm 3.6 Pseudo-code for Conservative tiling

Output: `tiles` - List of tiles in order of appearance in the schedule
Define: `tileStart`, `tileEnd` - Boundaries for a new tile

```
1:  function ComputeTileLimitsConservative(edge)
2:      tileStart ← edge.source.schedule
3:      tileStop ← edge.target.schedule
4:      for t ∈ tiles do
5:          if t.end < edge.source.schedule then
6:              continue
7:          else if t.begin > edge.target.schedule then
8:              break
9:          end if
10:         if t.begin < edge.source.schedule < t.end then
11:             tileStart ← −1
12:         end if
13:         if t.begin < edge.target.schedule < t.end then
14:             tileStart ← −1
15:         end if
16:     end for
17: end function

18: function ComputeTilesConservative(edge)
19:     ComputeTileLimitsConservative(edge)
20:     if tileStart = −1 then
21:         return false
22:     end if
23:     tile ← CreateTile(tileStart, tileStop)
24:     if tile.width = 0 then
25:         return false
26:     else
27:         for all t ∈ tiles do
28:             if (tileStart ≤ t.begin) && (t.end ≤ tileStop) then
29:                 tiles.erase(t)
30:             end if
31:         end for
32:         tiles.insert(tile)
33:         return true
34:     end if
35: end function
```

Chapter 4

Generalized Register Tiling

The Tower Library was divided into twelve depositories, at least insofar the world knew, and the Ninth was the smallest, given over to text on various forms of arithmetic, yet it was still a large chamber, a long oval with a flattened dome for a ceiling, filled with row on row of tall wooden shelves, each surrounded by a narrow walkway four paces above the seven-colored floor tiles.

R. Jordan - *Crossroads of Twilight*

4.1 Context & goal

Registers that are present in each core of a processor can be considered as the lowest level of the memory hierarchy. Therefore they form an important application domain of our *generalized tiling* approach. The goal is to represent the instructions contained within a loop with a *memory-use graph*. In the case of nested loops we will only tile the innermost loop.

The optimization of data reuse has already been extensively studied in the case of loop transformations and therefore the first step is to list the existing techniques and to show the differences with *generalized register tiling*. A second important point is to see how the general principle of the *memory-use graph* and the associated memory usage and IO cost model can be applied to register tiling.

After these theoretical considerations we present a practical implementation of *generalized register tiling*. The implementation has been attempted within two different contexts for which we expose the details, along with some observations on difficulties that were encountered from a software engineering point-of-view.

This leads us for a last part to the evaluation of experimental results obtained with the previously described implementation. These results, together with one of the implementations have been the subject of a paper [Dom+16] presented at the 2016 International Conference on Compiler Construction (CC'16).

4.2 State of the art

4.2.1 Register allocation and scheduling

The optimal register allocation problem is NP-complete [Cha+81]. Given an instruction schedule, good heuristics [GA96; CH90] or "optimal" formulations [AG01; CBD11] have been developed for register allocation to minimize spills. However, there is a strong interaction between the scheduling of instructions and optimizing register allocation. Goodman and Hsu presented a scheduling heuristic which tries to strike a balance between keeping the register pressure in check and delaying instructions on the critical path [GH88]. An integrated optimization of instruction scheduling and register allocation was shown to be NP-hard by Motwani et al. [Mot+95], who then developed a weighted heuristic that allowed control on the relative priority given to instruction level parallelism versus register pressure.

4.2.2 Scalar promotion

Scalar promotion, also referred to as register promotion, targets redundant loads and stored associated with array accesses. In non-optimized code such accesses transit all through memory as they are not seen as referring a single variable. A first version by Callahan et al. operated by transforming the source code of a program [CCK90]. Other approaches followed such as a loop-based form by Lu and Cooper [LC97] or one based on SSA by Sastry and Ju [SJ98]. Lo and al. [Lo+98] demonstrated that register promotion is closely related to partial redundancy elimination (PRE) [MR79] as this technique can be used to identify similar pointer computations for array accesses.

4.2.3 Re-materialization

Rematerialization [BCT92; BE11] corresponds to the recomputation of a value from the variables available in registers, instead of spilling it to memory and reloading it. It can be viewed as a form of a very limited rescheduling and it is the main source of performance when integrated in the spilling formulation [CBD11].

4.2.4 Classical register tiling

Register tiling [RRR05] considers perfectly nested multidimensional loops with uniform dependencies and represents the innermost loop body as an atomic unique instruction. These restrictions allow for the optimization of register reuse across multiple loop dimensions. In this paper we restrict ourselves to the inner-most loop level but expose register reuse inside the loop body itself. Although unroll-and-jam [Ma07] can be seen as a special case of register tiling with rectangular tiles, these two techniques are viewed quite differently in different communities. Unroll-and-jam and loop unrolling are typically used as a way of expanding the number of statements in the loop body in order to increase instruction level parallelism.

4.2.5 Comparison with generalized tiling

The transformations that are cited above all target register usage optimization but each one does so over a different space. A comparison between the optimization spaces of each transformation gives yet another point through which *generalized register tiling* differentiates itself from existing work. The different spaces are represented in Figure 4.1. The optimization spaces can best be represented as an overlay on an iteration space representation which is showcased in Figure 4.1a.

Classical register tiling (Figure 4.1b) Classical tiling only considers reordering the iterations of nested loops as can be seen in Figure 3.2a of previous Chapter 3. As a consequence it has no influence on reuses local to a single iteration but only on reuses between the iterations of a single tile.

Scalar promotion (Figure 4.1c) Scalar promotion functions in a greedy fashion over the code contained in the innermost loop of a nest. It modifies reuses of subscripted variables (*i.e* array accesses) to use scalar variables instead of involving redundant loads of the same address. When register pressure allows it this transformation also tries to optimize the reuse of subscripted variables from one iteration to the next.

Register allocation (Figure 4.1d) Register allocation can, depending on which algorithm and method are used, achieve optimal reuse within the code contained in the innermost loop of a nest. In some cases it can, in a similar fashion to scalar promotion, optimize reuses that are carried to the next iteration.

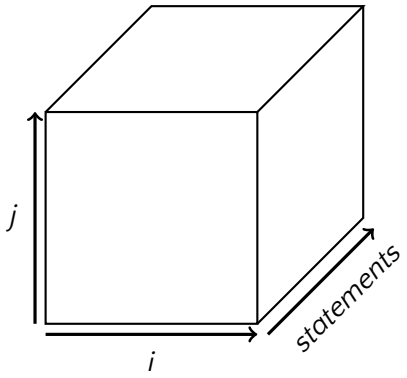
Generalized register tiling (Figure 4.1e) Generalized tiling considers the code within an innermost loop over its multiple iterations all at the same time. Therefore it can achieve optimal reuse over the entirety of the target space.

4.3 Implementing generalized tiling for registers

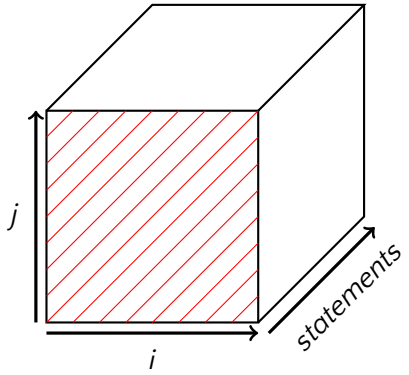
The implementation of generalized tiling within a compiler required us to confront ourselves both with the limits imposed by the targeted compiler framework and with the necessity to address all corner-cases that a compiler may generate depending on the source code. For example, the generation of the *memory-use graph* may vary depending on multiple implementation-related factors.

4.3.1 From data-dependency graph to memory-use graph

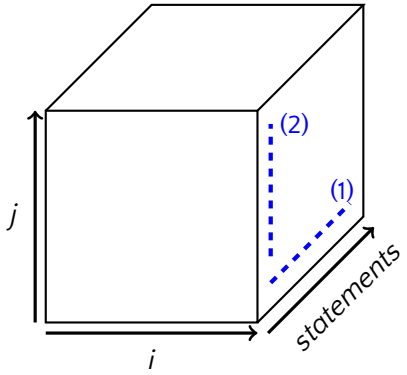
Ideally the *memory-use graph* should exactly reflect the data reuses between the computational instructions. However, with explicit memory communications, not all instructions in a given code may have a computational, *i.e* arithmetic, purpose. Depending on the point at which the *generalized tiling* is applied within an optimizing compiler the code may or may not include a variable amount of such explicit memory accesses. For this reason the most important element is the ability to analyze such communications in order to translate them into reuse edges when possible.



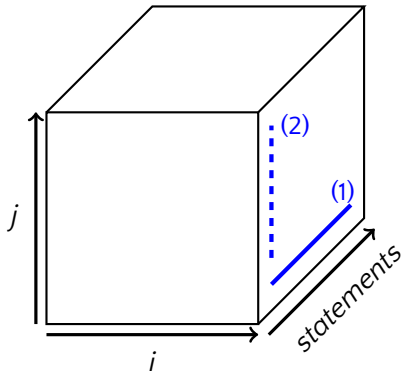
(a) Iteration space representation



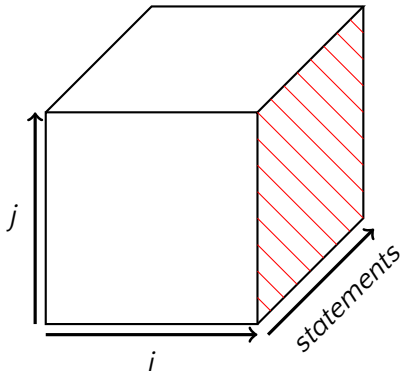
(b) Classical register tiling - reuse over the iteration dimensions



(c) Scalar promotion - greedy reuse within iterations and beyond if possible



(d) Register allocation - optimal reuse within iterations and greedy beyond if possible



(e) Generalized register tiling - optimal reuse both within an iteration and the innermost loop

Figure 4.1 – Optimization spaces for multiple code transformations

Furthermore, the reuse of a subscripted variable does not imply that the corresponding load and store instructions can be removed. On one hand store instructions should not be removed because the referenced values may still be used outside the scope of the loop that is being optimized. Load instructions, on the other hand can often safely be removed as the uses of the loaded data are already known with the data-dependency graph of the code.

We present two separate attempts at implementing *generalized tiling* in an industry-grade compiler.

4.3.2 Kalray K1 architecture

The MPPA processor family was the primary targets for the usage of *generalized tiling* before the scope of the work was extended to the more well-known x86 architecture. The computational cores of the MPPA manycore family use the K1 architecture. This is a *Very-Large Instruction Word* (VLIW) core with in-order execution. It can issue up to five instructions per cycle. The five execution units include two 32-bit *arithmetic & logic units* for integer and bitwise computations, one 64-bit *multiply & accumulator unit* for floating point computations, one *branch & conditional unit* and one *load & store unit*. The register file contains 64 uniform 32-bit registers and offers a fast register file bypass to immediately reuse results produced during the previous cycle. Memory accesses are based on a 32-bit address-space[1]and may use a 64kB L1 data cache which is private to each core.

From a compiler perspective, the K1 architecture offers a fully exposed pipeline, *i.e* the pipeline depth for all non-memory instructions is fixed and known. A precise scheduling of instructions on the critical path is thus possible.

4.3.3 The GCC - Tirex - LAO experience

As it was the only supported compiler, the direction that was initially chosen for implementing *generalized tiling* involved GCC (*GNU Compiler Collection*[2]). The complete framework also uses an intermediary representation (*Textual Intermediate Representation for compiler EXchange* - TIREX) [Pie12] as well as an external optimization tool (*Linear Assembly Optimizer* - LAO [Din+00]).

TIREX is a representation specification that is designed to enable cross-compiler optimizations as well as the use of external optimization tools. This is of great interest

[1]The second generation of the K1 architecture contains partial support for a 64-bit address-space.
[2]https://gcc.gnu.org/

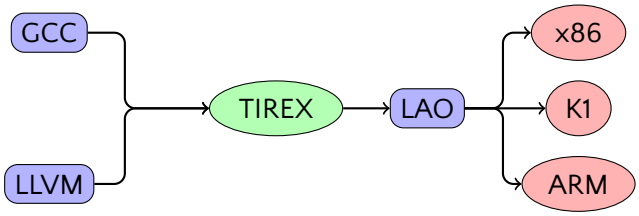


Figure 4.2 – Compilation paths with the use of TIREX and LAO

to reduce the numbers of compilers and optimizations within each of them that have to be maintained. In this particular case the goal was to promote the use of the LAO tool as an external back-end and architecture-specific optimizer for multiple existing compilers such as GCC and LLVM [LA04] (see Figure 4.2).

From an engineering point-of-view this meant that new compiler passes had to be developed within the back-end of each of these two compilers. Because GCC was the only compiler which supported the Kalray K1 architecture, at that time the unique target for *generalized register tiling*, and thus efforts were made to create a plugin for GCC that could emit TIREX.

A first set of technical problems soon appeared. They were related to the architectures of both GCC and LAO:

- The instruction selection available in LAO was not nearly as efficient as the one available in GCC. This meant that the TIREX dump would need to be performed after instruction selection so no performance would be lost at that point.
- The register allocator in LAO was not behaving as expected on the Kalray architecture and a serious amount of work would be needed to put it back into a working state. This meant that the TIREX dump would need to be delayed even further in the GCC back-end.
- An indirect consequence of the previous point was that, because *generalized register tiling* needs registers to be unallocated, LAO would need to perform *register de-allocation* as well as the associated spill-code removal before applying tiling.
- The information pertaining to the data-dependency analysis performed by GCC is not consistently maintained beyond the middle-end of the compiler. As a result data dependencies would need to be entirely recomputed within LAO.

Nonetheless the work on the implementation was started. A plugin for GCC was written and in the same time the TIREX specification [PBA13] was updated and extended

to include new high and low-level information that needed to be transferred between the optimization tools.

These first elements enabled to activate a first optimization pass within LAO: *post register allocation scheduling*. Even though the amount of rescheduling that can be done at such a late stage of the compilation process is very limited this did show some small performance increases. At this point work was started on the *register de-allocator* within LAO in order to progress towards the complete implementation of *generalized register tiling*.

Two separate events triggered the total discontinuation of the work on the GCC - TIREX - LAO optimization path:

1. An important amount of work remained to be done before there would be any hope of a *generalized register tiling* implementation. Given the limited time-frame available, an alternative approach was required.

2. Prototyping work on an LLVM back-end for the Kalray K1 architecture had been started. The LLVM framework had the reputation to be much easier to work with and it already possessed some specific code transformations that could potentially be reused for the tiling implementation.

It was thus decided to transfer the implementation to the LLVM compiler infrastructure. This had the advantage of reducing the number of different programs as well as intermediate representations that needed to be managed but it did not, however, remove all the technical difficulties encountered previously.

4.3.4 Moving to LLVM

Implementing *generalized register tiling* in LLVM needed to both enable the use of heuristics as well as the *constraint programming* approach for solving the tiling problem. As a consequence the optimization was designed to work in multiple passes. The tiling driver is actually a wrapper script that uses the real `clang` driver as well as the `opt` tool from the LLVM framework. Tiling is applied using the steps described in the following paragraphs, which corresponds to the passes described in Figure 4.3.

General optimizations

Source code files are first transformed into LLVM Intermediate Representations files. This is done via the original `clang` driver without performing any optimizations. The `opt` tool is used to apply all default optimizations with the `-O3` flag.

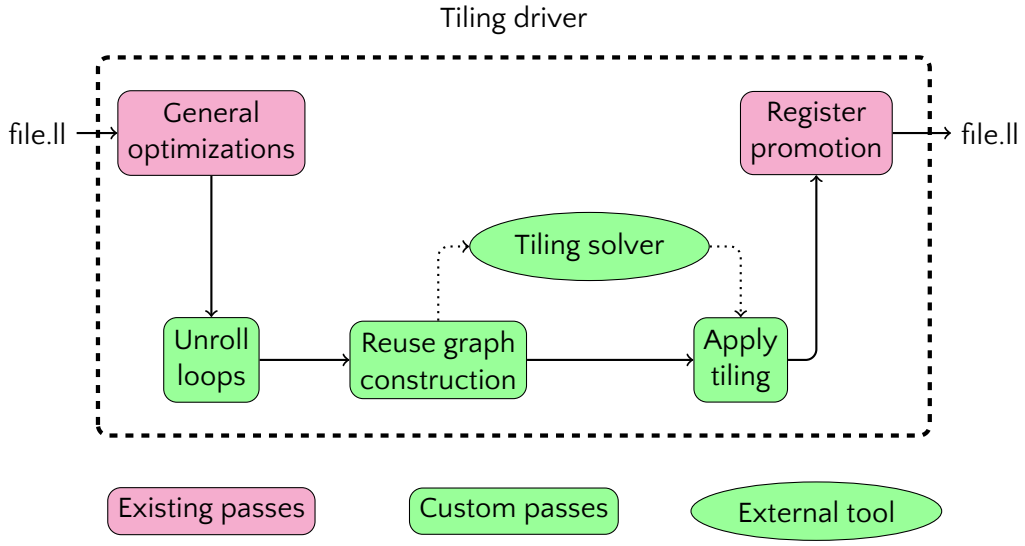


Figure 4.3 – Architecture of the LLVM implementation

Loop unrolling

Target loops are identified within the LLVM IR as not all loops can be tiled. For example loops with conditional statements cannot be tiled, unless the condition variables are constant over all iterations and the loop can be unswitched. Furthermore an induction variable should be identified for the correct analysis of inter-iteration dependencies and reuses. Each targeted loop is unrolled by a fixed factor that is specified by the user. The early unrolling, before the real tiling is applied, is necessary for the next step of analysis. Future work would need to remove the necessity for unrolling in order to reduce shortcomings and performance loss of the current implementation.

Memory-use graph construction & optimization

In the next step, the target loops are analyzed for their dependencies and a *memory-use graph* is constructed. The analysis of subscripted values that is currently implemented to detect reuses flowing through memory is a mix between custom code and some existing features of LLVM such as *Scalar Evolution* (SCEV) information. The graph of each loop is dumped to an independent external file. This external file can then be fed to a tiling solver (Constraint Programming or heuristics) in order to find a solution to the optimization problem. The solution is also dumped to an independent external file.

Tiling

The last custom step of the tiling uses as input the unrolled loops as well as the corresponding tiling solution files. The instructions in each target loop are then rescheduled accordingly. As a result of working on unrolled loops that have not been split, as opposed to what is suggested in Chapter 3, the order in which the instructions are executed is not entirely as the tiling model would assume. These differences are described in detail in Figure 4.4. An indirect result of this modified transformation is a larger code size than with the theoretical transformation. Consequences and other effects will be discussed in the experimental results and their analysis presented below.

4.4 Experimental results

4.4.1 Evaluation and choice of benchmarks

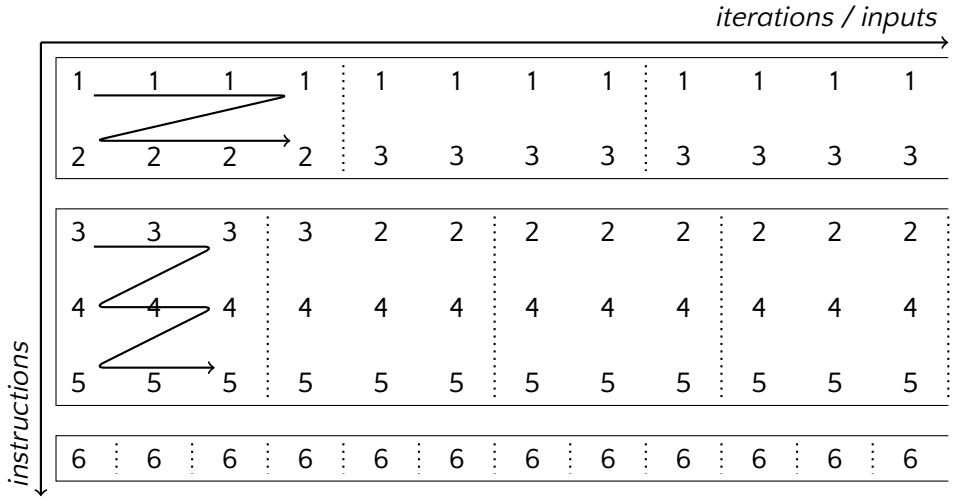
The 35 benchmarks that were used for the evaluation of the different tiling heuristics have been taken from a variety of DSP and other data processing algorithms with loop-intensive codes, such as FFT, FIR, Viterbi, dot-product, ... The target for this evaluation was the x86 architecture, the K1 back-end still being unavailabe. All benchmarks were compiled:

- With LLVM and only the `-O3` flag to create the reference code
- With the *generalized register tiling* toolchain based on LLVM targeting a memory size of 16 registers (x86 architecture). For each of the tiling heuristics from Section 3.5.2:
  - One version with an unroll factor of 3
  - One version with an unroll factor of 6
  - One version with an unroll factor of 12

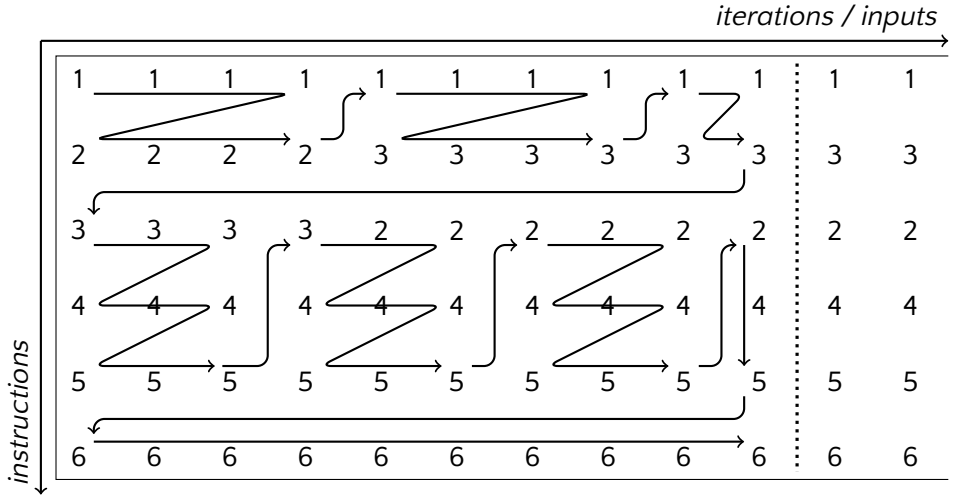
Each heuristics was used with three different unrolling factors as described in Section 4.3.4: 3, 6 and 12. For all the benchmarks the influence of the tiling on execution times is summarized in Table 4.1. The times were averaged over 100 consecutive runs of each individual benchmark starting with a cold cache.

4.4.2 Results and analysis

The execution time statistics do not allow to distinguish a clear winner among the three evaluated heuristics. It is not possible either to give a strong interpretation about the



(a) Theoretical code transformation suggested by a tiling solver



(b) Code transformation applied in the LLVM register tiling passes

Figure 4.4 – Discrepancies between the theoretical tiling and the LLVM tiling

Unroll	Worst	Best	Avg.	Std.Dev
3	+35%	-12%	0%	7%
6	+42%	-31%	-3%	14%
12	+69%	-7%	+6%	17%

(a) Heavy-edge / Greedy

Unroll	Worst	Best	Avg.	Std.Dev
3	+36%	-30%	-1%	10%
6	+60%	-26%	+4%	16%
12	+31%	-15%	0%	8%

(b) Tile-aware / Heavy-edge

Unroll	Worst	Best	Avg.	Std.Dev
3	+136%	-14%	+4%	24%
6	+50%	-17%	+4%	12%
12	+55%	-10%	+1%	11%

(c) Tile-aware / Conservative

Table 4.1 – Execution time comparison (tiled vs. non-tiled)

influence of unrolling. For the majority of the benchmarks, average times for the tiled versions remains stable at values equal to those of the untiled versions. However average stability hides strong variations specific to some benchmarks. Furthermore the best and worst values for each test cases tend to vary significantly when trying to replicate the results of Table 4.1, each time using the average of 100 runs for each benchmark. This indicates that tiling does have an impact on execution times, but this impact can be either beneficial or harmful in equal proportions.

Beyond the effect on execution times, the effect of tiling on the amount of IO operations has proven difficult to evaluate. Several attempts were made by using the various performance counters available on the x86 architecture but the returned values were not stable and could in general not be made sense of. An alternative option of instrumenting the benchmark executables did not produce stable results either, mainly due to the complexity of the implementation in the LLVM framework. Yet another alternative is to analyze the assembler code but this is complicated by the fact that a majority of arithmetic operations in the x86 instruction set allows for the direct use of memory operands and as such may generate implicit IO operations.

Chapter 5

Generalized Dataflow Tiling

It seemed to her, just for a heartbeat, as if time had suddenly slowed, as though the heartbeat took forever. She felt the flow inside her – *saidar* became a distant thought – felt the answering flow in the lightning. And she altered the direction of the flow. Time leaped forward.

R. Jordan - *The Great Hunt*

5.1 Context & goal

The dataflow paradigm is paramount in some cases of software development on embedded platforms. It organizes computations as a set of successive *actors* that apply distinct operations to one or more data elements. Intermediate values are transmitted between actors in order to create a chain of computations that are applied in order to transform input into output. Because each separate data element of the input stream follows exactly the same path through the dataflow actors, this model excels when applied to computations such as video processing, cryptographic encoding and decoding, etc. Hence its prevalence in the embedded community.

Embedded platforms can benefit from better cache memory usage through code optimization in a way that goes beyond that of more general-purpose processors as the amount of cache is in general much smaller. Among the more specific benefits of cache optimization, the reduced memory access latency is one of the most visible elements as it directly influences the execution speed of applications. Better cache usage implies fewer cache-misses and less time spent stalling for these misses to be fetched from distant memory. Another indirect effect is the reduced bandwidth usage on the memory bus as more computations can be performed before a fetch from memory is required. In other words, the *operational intensity* of the code increases. A last important benefit of better cache usage is the reduced power consumption or *energy-to-solution* of the code as memory communications typically use significantly more power than computations.

Just as it is the case for loops, data reuses in a dataflow program can take place between actors for the same input data element, *i.e* within the same loop iteration, as well as between computations for different inputs, *i.e* between different iterations of the loop-nest. While it is not possible to reschedule the order in which a dataflow program iterates over the input elements we can however modify the order in which actors are scheduled.

Firstly we give a summary of the existing work related to the optimization of programs expressed in dataflow languages. Some of these optimizations target specifically the data reuse at cache level whereas others also include the distribution and parallel execution over a multicore target. Next, we present some results obtained within the StreamIt framework which are also the subject of a paper presented at the 2016 Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'16) [DAR16]. We finish with some preliminary work on an implementation of *generalized tiling* on a manycore processor with the ΣC language [Gou+11].

5.2 State of the art

5.2.1 The dataflow paradigm and its variants

The numerous existing dataflow languages can be classified into multiple categories. Some like Lustre [Hal+91] are classified as synchronous because all actors share a common clock that indicates when they should advance to their next iteration. These languages are usually focused on real-time or time-critical systems and are not the main benefactors of cache optimization. Another category of languages is used to dynamically schedule large graphs of tasks on distributed systems [Den74; Gau+13] using techniques such as work stealing. Such languages can be found in high-performance applications and could benefit from cache optimizations, mainly because of the bandwidth and energy reductions. However the category that is of most interest for cache usage optimization consists of so-called *static* [LM87] and *cyclo-static* [Bil+95] languages. These find their most representative applications in the context of streaming computations where throughput, latency and energy consumption are primary constraints. Examples of such languages are StreamIt [TKA02], Spidle [Con+03] and ΣC [Gou+11]. As an extension of the *static* paradigm, dynamic dataflow languages such as OpenStream [Pop13] reintroduce the possibility of runtime optimizations and modifications. These dynamic languages can themselves be classified into a wide number of sub-categories such as *Parametric SDF*, *Variable-Rate Data-Flow* and others, depending on the manner in which communications and actor scheduling are constrained.

5.2.2 Cache-aware optimization of dataflow programs

Varied forms of optimizations already exist for dataflow languages that target among other metrics a better usage of cache memory [Hir+14]. Actor fusion [Ser+05] groups multiple actors together and performs inter-actor rescheduling. This may increase the amount of register reuse between the fused actors. Execution scaling [Car+03; Ser+05; WCB01], a technique that is also known as train scheduling or batching, increases the number of data elements that are processed in a single iteration by a multiplicative factor that is the same for each actor. Finally state sharing [ABW06; Bri+08; GTA06] optimizes the execution of stateful actors by co-scheduling multiple iterations that reuse the same state data.

5.2.3 Scheduling and multicore optimizations

Another objective of previously published optimizations for streaming applications is the effective use of multicore systems. The work of Kudlur and Mahlke [KM08] performs both application partitioning and resource allocation in a combined framework. They make use of optimal solvers, specifically *Integer linear programming*. Farhad et al. [Far+11] on the other hand use a heuristic approach to overcome the time-to-solution issues encountered with ILP in large problems.

5.3 Evaluation of a prototype: StreamIt

5.3.1 Experimental setup and choice of benchmarks

For the evaluation of our tiling approach, we compare ourselves to the results obtained with the approach described by Sermulins et al. [Ser+05]. The dataflow programs that we use are expressed in the StreamIt language [TKA02] and are part of the StreamIt benchmark suite. StreamIt is both a language and a compiler infrastructure designed to facilitate the programming of large streaming applications. The dedicated compiler is able to target a variety of multicore architectures as well as computational clusters. Details about each specific benchmark can be found in the work of Gordon et al. [GTA06].

As our approach aims at minimizing the number of cache-misses this is the measure used for evaluation. In order to estimate these misses we simulate the behavior of the cache based on a *least-recent use* (LRU) eviction policy. We target an embedded architecture in the form of a StrongARM 1100 with 16kB L1 instruction cache and 8kB L1 data cache. The main reason for chosing this architecture is that it is the only one for which performance measures with the state-of-the-art approach are available.

5.3.2 Results & analysis

The results of the cache-miss simulations are shown in Figure 5.1. Values for each dataflow program have been normalized with respect to the result obtained with the scaling techniques described in Sermulins et al [Ser+05].

For the *constraint programming* solver, we see that in a majority of cases the number of cache-misses is reduced. Only on the ChannelVocoder and the FilterBank benches does the number of misses increase slightly. The SAR benchmark contains an actor with a state that exceeds the cache-size and therefore cannot be significantly

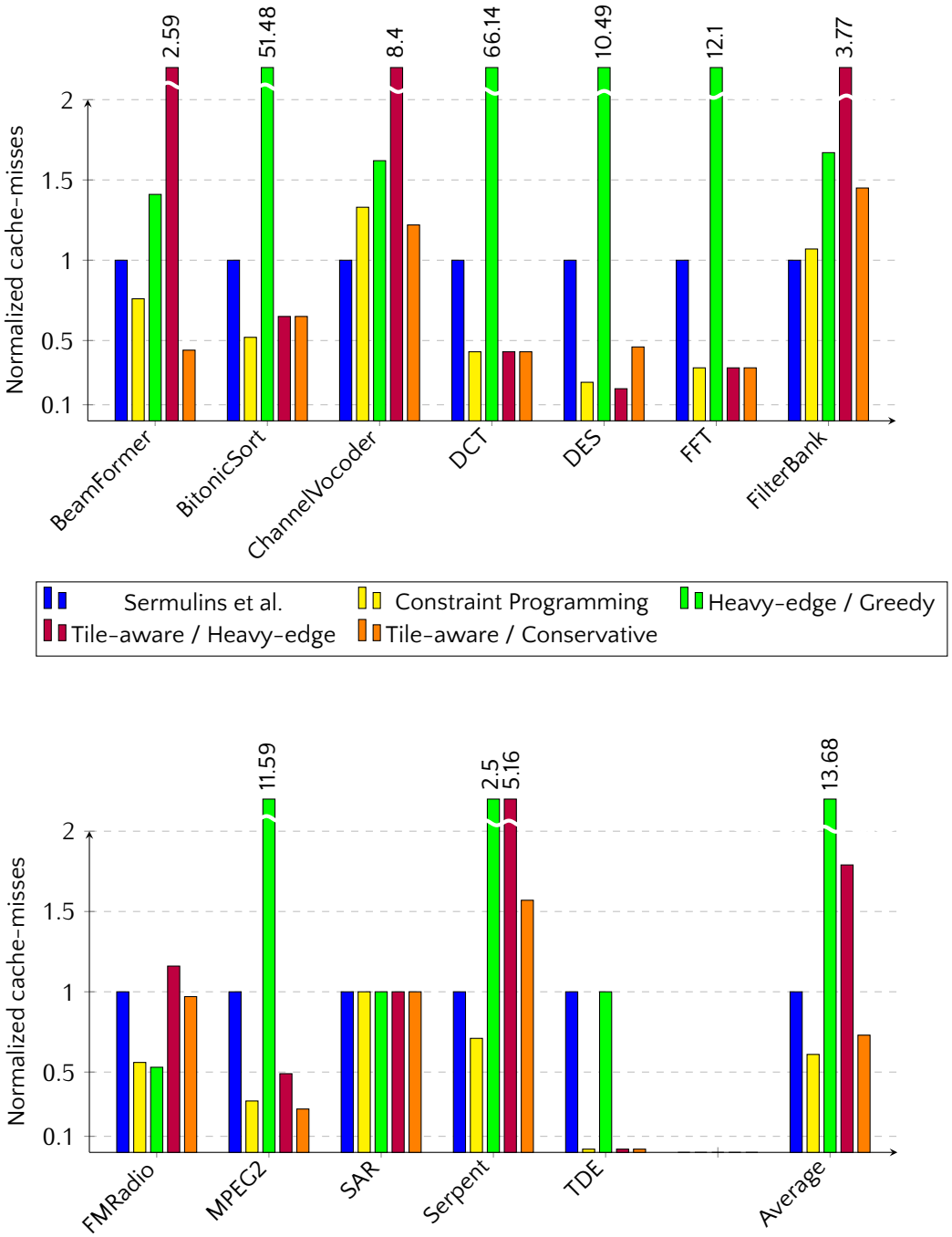


Figure 5.1 – Cache-miss results for dataflow tiling

improved. The configuration of the CP solver involved a 30 second timeout. In 7 out of the 12 benchmarks the solver timed-out and produced a sub-optimal solution as described in Section 3.4.2. Among these 7 benchmarks are ChannelVocoder and Filterbank, which suggests that with a higher time-out the CP solver might still be able to reduce the number of cache-misses compared to the state-of-the-art.

For our three heuristics, the results are much more varied. All three were applied using the same solver that was also used for *generalized register tiling* in Chapter 4. The *greedy* tiling with decoupled *heavy-edge scheduling* clearly makes too rough approximations and as such has a general performance that is significantly reduced compared to all other approaches, whereas the *tile-aware heavy-edge scheduling* heuristic performs somewhat better. For the latter, this indicates that the preference for including heavy-edges in tiles first does benefit the amount of reuse as we presumed in Section 3.5.2. However even the *tile-aware heavy-edge scheduling* heuristic does not improve on Sermulins et al. on at least half the benchmarks.

The *tile-aware heavy-edge scheduling* heuristic performs better than the greedy one but it still has a worse performance than Sermulins et al. on at least half the benches whereas on the others it compares to the *constraint programming* solver. This tends to indicate that the preference for including heavy-edges in tiles first does benefit the amount of reuse as we presumed in Section 3.5.2.

Last but not least, the *tile-aware conservative scheduling* improves again on the other two heuristics and is the only heuristic that has a global performance that is comparable to that of the CP solver. It also exhibits improvement over the previous Sermulins et al. scaling technique. From a detailed study of the tiling solutions it appears that this improvement is due to a better balance between the width and the number of actors in each tile.

In some cases, such as BeamFormer, ChannelVocoder, DES, FMRadio and MPEG2, one of the heuristics achieves a lower number of cache-misses then the CP solver. This is again due to the time-out enforced on the solver and confirms the sub-optimality of the resulting tiling solutions.

A lose correlation between the evolution in cache-misses with respect to Sermulins et al. and the ratio between the sizes of states and flow dependencies. An analysis of benchmark structures and of the results obtained with *generalized tiling* show that when flow dependencies exceed state data the potential for improvement is higher.

A reason for non-optimal results which affects both the CP approach and the heuristics is linked to our problem formulation in Section 3.3.2. We make the assump-

tion that each tile is spill-free. It may however be beneficial to allow for some spill locally in order to let tiles grow further and globally reduce spill. This is partly what is done by Sermulins et al. [Ser+05] when they perform scaling and allow a given percentage of the actors to have IO requirements that exceed the data cache size. Reducing the spill-free constraint is thus a sensible direction for future efforts and could increase even further the benefits of cache reuse through tiling.

Last but not least our model does not include any information pertaining to memory mapping or the associativity of the cache layout which reduces the precision of the cost evaluation.

5.4 The path towards an industrial implementation: ΣC

The previously exposed results where obtained through simulation and the rescheduling of existing dataflow programs. The next step for generalized dataflow tiling would be the implementation of the optimization within a dataflow compiler and runtime. The ΣC language and framework are already available for the MPPA processor. ΣC follows the *cyclo-static dataflow* model [Bil+95] and takes it sources from StreamIt [TKA02], Brook [Buc+04] and XC [Wat09]. Because this implementation was only undertaken very recently it has not been completed. This section aims at describing the challenges that were encountered as well as those that still have to be faced.

5.4.1 ΣC toolchain

As a framework ΣC [Gou+11] was developed as an extension of the C language with the introduction of the concepts related to dataflow languages: actors (called agents), communication channels, ...The additional constructs are used by the ΣC toolchain in order to gather all elements necessary for a runtime. These include the code for each actor as well as the dataflow graph with numerous annotations. The toolchain's action can be decomposed in multiple steps:

1. Extraction of elements from the source code: actor code and dataflow dependencies.

2. Actor fusion, multiplication and grouping. Groups of actors are constituted as to roughly match the available resources on the target architecture. Dependencies between groups take the shape of network communications, *e.g* Network-on-Chip transfers.

3. Physical mapping of actors to the resources available on the target. This involves the computation of buffer sizes for network and shared memory communications but also scheduling.

4. Generation of runtime code and modified actor code for the final resource allocation scheme.

With the kind of information necessary for the computation of a tiling solution it appears that the dependency analysis and the generation of the *memory-use graph* should be done between steps 1 and 2. Scheduling the actors is done at step 3 but this can be strongly influenced by the actor grouping that may occur at step 2. As a consequence the tiling solution is also necessary in order to influence this part of the runtime generation process: the actors of a single tile should be able to execute on the same resource, *e.g* a cluster on the MPPA architecture without overflowing the available local memory and should thus be grouped.

Nonetheless before applying tiling it is first necessary to extract the *memory-use graph* from the dataflow information.

5.4.2 Extraction of the graph: hierarchical actors

The representation that is used internally by ΣC as well as in its intermediate representation is a hierarchical graph. An actor may use other actors internally thereby creating the need for encapsulation. Because the *memory-use graph* does not accommodate such a structure we first perform a flattening of the intermediate representation. Actors are recursively replaced with their corresponding internal graph until only "atomic" actors remain. These are the actors that only contain C code and do not reference other actors. They are also the granularity at which our tiling will be done and correspond to the vertices of the *memory-use graph*.

The annotations of the dataflow graph include information on the communications between actors and make it possible to label the edges between vertices. The same is true for code size information which helps computing the *internal computation requirement* of each vertex. The remaining part of the requirement can be directly obtained with existing analysis tools as each atomic actor is written in C.

With the *memory-use graph* it is possible to compute a tiling solution with either the *constraint programming* or the heuristic methods presented in Chapter 3.

5.4.3 Using the Network-on-Chip

Aside from the computation of a tiling solution it is yet another challenge to implement it on a manycore platform. In the case of *cyclo-static* languages such as ΣC, the communications between the actors have to follow a specific static scheduling computed in advance, something that is potentially hindered by the presence of a Network-on-Chip in comparison with a single core or a shared-memory multicore processor. The complex structure and behavior of a Network-on-Chip leads to an uncertainty on the timing and latency of communications between actors that run on different cores of the processor.

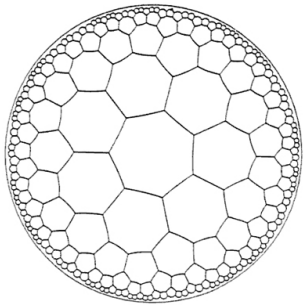
The *cyclo-static dataflow* model assumes that actors execute periodically following a predetermined scheme. Because there may be some time between the production of the input data and its consumption it is necessary to provision enough space in memory to absorb this latency. Variations of the latencies increase even further the required amount of buffer space. As a consequence it is of vital importance for a high performance implementation to limit latencies and their variations. The study performed in Chapter 2 could potentially be a useful tool in this regard.

5.5 Limits of the current implementation

As was the case for *generalized register tiling* in Chapter 4, the implementation presented for *generalized dataflow tiling* comes with a significant amount of limitations.

First and foremost is the fact that is has not moved beyond a prototyping stage in the StreamIt language. The engineering constraints and the complexity of a full implementation in a real-world environment are too broad for such work to be realistic within the context of this study. Nonetheless, the initial results are promising enough to continue further investigation into the *generalized tiling* approach as well as the underlying *memory-use graph* code modelisation.

The next step is to move beyond the optimization of code at compile-time and to apply the same abstract models to the analysis of existing code. The goal is to improve data locality through the use of *performance debugging*.



Chapter 6

Beyond Semi-Regularity

He wove something majestic, a pattern of interlaced *saidar* **and** *saidin* **in their pure forms. Not Fire, not Spirit, not Water, not Earth, not Air. Purity. Light itself.**

R. Jordan & B. Sanderson - *A Memory of Light*

6.1 Context and goal

Up to this point all our efforts have been turned towards static analysis and compile-time optimization of code at various granularity levels and in various use-cases. Our memory and IO cost models however are not limited in their applications to compile-time use. Any code for which data reuses are available under the form of a directed acyclic graph can potentially be analyzed.

Whereas compile-time optimization aims at enhancing performance as much as possible based on theoretical analyses and models, alternative approaches focus on the optimization of code that has already been compiled. To achieve this goal the target code can be observed through methods such as sandboxing and instrumentation. An example of instrumentation is the *Dynamic Dependency Graph* tool[1]. This tool instruments code at compile time so that the execution generates a graph of memory uses and their dependencies at variables granularities for every instruction executed during a run of the program. It can also record information such as the locations of load and store operations and their target addresses.

The graphs that are produced may be variable in size as this does not depend on the size of the code but on the length of the execution: each executed instruction or instruction group produces its own vertex in the graph. The graph may, for example, cover the execution of all the iterations of a loop while still generating individual vertices for each instruction at each iteration in opposition to the condensed representation that we used in the *memory-use graph*. As such the graphs may not exhibit all the expected regularities that formed the basis of the condensed forms and their typical sizes are within the range of 10^6 to 10^8 vertices or beyond.

As a consequence of the size and irregularity our heuristics for generalized loop tiling and generalized dataflow tiling are not applicable. The graphs, directed and acyclic by definition, can however still be used for analysis with our memory and IO cost models. The goal is not to find a fully-fledged optimization solution but to give indications on whether the code's current scheduling and organization can be improved with respect to its IO cost. This is called *performance debugging* and is used as a tool by developers to improve the performance of existing programs.

Following the same methodology as we previously used for the other forms of generalized tiling, we divide the graph of memory reuses in atomic units that can be executed within the target memory size after which we evaluate the IO cost of the communications between the units. As mentioned in Section 1.5.2 this refers to a con-

[1]Developed at Ohio State University and in the Corse research team at Inria

vex partitioning of the analyzed graph.

After a short presentation of the existing research context in both code instrumentation and graph partitioning we will present some contributions to the fields of convex partitioning as well as graph algorithmics in general. We can then move on to the concrete evaluation of the IO cost of such a partitioning of a graph. The final part of this chapter is dedicated to the presentation of the *GraphUtilities* library that was developed during this study for both research purposes and as a support for further applied work in the field of *performance debugging*.

6.2 State of the art

6.2.1 Graph reachability

Underlying the concept of convex graph partitioning is that of reachability. To characterize a given partitioning as convex immediately has numerous implications on the connectivity as well as the structure of a graph. Reversing this it appears that having an effective way of answering reachability queries may be of relevance for creating convex partitioning methods.

The topic of reachability queries has been well explored by the research community and remains today an active field of research. The naive approach of computing a full transitive closure has a space complexity in the square of the number of vertices, a cost which quickly becomes prohibitive except for the smallest of graphs. A wide variety of current industrial and research problems involve the processing and analysis of very large real-world graphs and this pushes for novel techniques and methods each year. It is not the goal here to give a thorough overview of all past research. For the interested reader, the more detailed papers in this area give extensive insight into the existing work [Jin+11; YCZ12].

For a short overview it is possible to reference the seminal work by Simon [Sim88] on chain decomposition. In more recent years the 2-HOP approach [Coh+02] and its 3-HOP generalization [Jin+09] have taken a different view of graph indexation by using a sub-graph that approximates the connectivity of the full graph. One of the resulting effects is the reduction of the index size. Jin et al. [Jin+11] also proposed a variant of the sub-graph approach in the form of the *PathTree* structure.

Another approach is the labeling of the vertices of the graph so that the label values provides implicit connectivity information. The GRAIL algorithm by Yildirim et al. [YCZ12] is the most well known representative of such approaches. Orthogonal

to these methods is the SCARAB [Jin+12] method which effectively uses any query method on sub-graphs in order to answer queries on the top-level graph.

The approach presented in this chapter also uses vertex labeling. It bears a strong resemblance to the FELINE algorithm [Vel+14] which was only published well after our work was started.

6.2.2 Graph partitioning

The topic of graph partitioning itself is again a very active field of research. However, the specific issue of the convex partitioning of directed graphs has not received much attention. To the best of our knowledge all existing algorithms have been created as solutions to solve specific computational problems relying on the concept of convex partitioning: resource allocation on multicore processors [CRA09] or trace analysis and scheduling [Fau+13].

In the case of the more general topic of non-convex graph partitioning, which is also used on undirected graphs, the work by Kernighan and Lin on partitioning [KL70] can be seen as a keystone to a large set of modern methods referred to as multilevel partitioning schemes [CW91; HK92; Man+93; KK95]. The general principle consists of successive steps of graph coarsening followed by a single rough partitioning and reciprocal steps of graph uncoarsening with partition refinement. The methods and criteria used for refinement may vary depending on the optimization target: min-cut, max-throughput, partition weight, ... Examples of libraries implementing such methods for standard graphs are Metis [KK95] and PaToH [ÇA95] for hypergraphs.

Whereas the multilevel methods produce good results in reasonable time, other approaches exists: geometric partitioning [MTV91; HR95] which uses geometric information on the graph the guide partition shapes, and spectral methods that rely on mathematical matrix computations as partitioning criteria [PSL90; HL95].

6.3 Graph algorithms

By getting back to the basic definition of a convex partition it appears that the question of reachability is of great importance when we want to create such a partition. For this reason our first approach was to look for an effective method to determine the reachability between two vertices in a DAG.

From there on, multiple attempts were made to design an algorithm that can efficiently provide a convex partitioning. In all aspects, it proved to be a difficult but

interesting challenge and we ended up with three potential heuristics effectively producing convex partitions. One was taken from the existing literature.

The implementation of both the reachability approach and the convex partitioning algorithms, which are presented in Section 6.5, can typically manage graphs of up to 10^7 vertices for reachability queries and of up to 10^6 vertices for convex partitioning, all within the limited resources of a laptop with 8GB of memory and an average processor. As a last step, in order to even further increase these limits, we then sought to reduce the size of the graph on which convex partitioning is performed through graph coarsening.

6.3.1 Reachability: a two-way post-order indexing

In the case of a non-directed graph $\mathcal{G} = (V, E)$ the question of reachability is equivalent to that of connectivity: if two vertices u and v are within the same connected component of a graph then u is reachable from v and vice-verse. This means that pre-computating the reachability set for each vertex can be done with a time complexity of $\mathcal{O}(|V| \cdot |E|)$ and stored with a space complexity of $\mathcal{O}(|V|)$.

However if $\mathcal{G}$ is a directed graph, the question becomes less trivial. Depending on the number of reachability queries one needs to perform, it can either be more effective to run individual searches for each query or to perform a pre-indexation of the graph that will help answering queries faster. Pre-computing the entire reachability set for each vertex is not an option here as this has a worst-case space complexity of $\mathcal{O}(|V|^2)$ which is prohibitive considering the sizes of our graphs. Since a convex partitioning potentially requires a large number of queries, the pre-indexation method was retained.

Core concept

As mentioned above, different categories of pre-indexing can be identified. For our purpose we decided to develop an algorithm based on the labeling of vertices like the GRAIL algorithm [YCZ12]. The core concept of our approach is that a post-order traversal provides an order that can give a negative answer to a reachability query in some cases (Theorem 6.1).

Theorem 6.1: *Let u and v be two vertices in a directed acyclic graph $\mathcal{G}$. If there exists a post-order traversal of $\mathcal{G}$ in which u precedes v then $\mathcal{G}$ does not contain a path from u to v.*

Proof. Let $\mathcal{G} = (V, E)$ be a directed acyclic graph with V its vertex set and $U = \{(x, y) \in VxV\}$ its edge set. Let $l_{\mathcal{G}}$ be a post-order traversal labeling function which associates to each $x \in V$ the position of x in an arbitrary post-order traversal. We observe by the definition of a post-order traversal that:

$$\forall (x, y) \in E \qquad l_{\mathcal{G}}(x) > l_{\mathcal{G}}(y)$$

Suppose $u, v \in V$ two vertices such that $l_{\mathcal{G}}(u) < l_{\mathcal{G}}(v)$. We assume by contradiction that there exists a path from u to v with $(x_i)_{i \leq n}$ the vertices of the path in order. Then $l_{\mathcal{G}}(u) > l_{\mathcal{G}}(x_0)$,$l_{\mathcal{G}}(x_n) > l_{\mathcal{G}}(v)$ and:

$$\forall i \in [\![1; n]\!] \qquad l_{\mathcal{G}}(x_{i-1}) > l_{\mathcal{G}}(x_i)$$

Which gives:

$$l_{\mathcal{G}}(u) > l_{\mathcal{G}}(x_0) > \cdots > l_{\mathcal{G}}(x_n) > l_{\mathcal{G}}(v)$$

contradicting our initial hypothesis. □

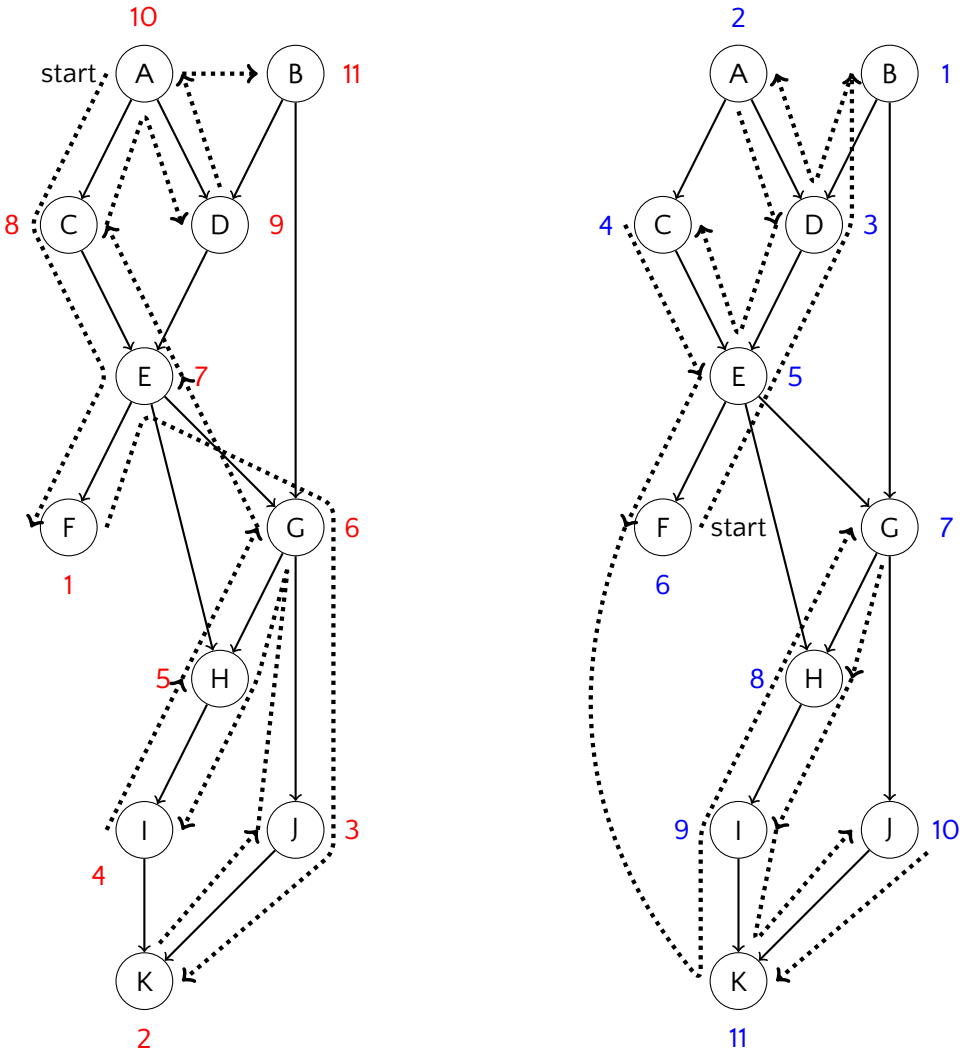
Index construction

With this property in mind we can describe our approach for the construction of a reachability index as a step-by-step process:

1. Perform a first labeling $l_{\mathcal{G}}^{(1)}$ of the vertices through a *depth-first search* traversal of the graph from the source vertices. During this visit register the order in which each vertex is visited by its predecessors.

2. Perform a second labeling $l_{\mathcal{G}}^{(2)}$ of the vertices through a *depth-first search* traversal on the reversed graph $\mathcal{G}' = (V, E')$ where $(u, v) \in E \iff (v, u) \in E'$ from the sink vertices of $\mathcal{G}$ (*i.e* source vertices of $\mathcal{G}'$). For each vertex visit its predecessors in $\mathcal{G}$ in the order obtained by reversing the one registered during the first labeling.

An illustration of the order in which vertices are visited during the two visits is given in Figure 6.1. As can be seen the first visit effectively starts from a source vertex of the graph, whereas the second visit starts from a sink vertex.



(a) First visit order

(b) Second visit order

Figure 6.1 – Example of a vertex visit orders during graph pre-indexing

Query processing

Once each vertex has received its two labels a reachability query from u to v is performed as follows:

1. If $l_{\mathcal{G}}^{(1)}(u) < l_{\mathcal{G}}^{(1)}(v)$ then return false.
2. If $l_{\mathcal{G}}^{(2)}(u) > l_{\mathcal{G}}^{(2)}(v)$ then return false.
3. Perform a *depth-first search* graph traversal of $\mathcal{G}$ between u and v. During the traversal only consider vertices $x \in V$ that verify the following criteria:
   - $l_{\mathcal{G}}^{(1)}(u) > l_{\mathcal{G}}^{(1)}(x)$
   - $l_{\mathcal{G}}^{(2)}(u) < l_{\mathcal{G}}^{(2)}(x)$
   - $l_{\mathcal{G}}^{(1)}(v) < l_{\mathcal{G}}^{(1)}(x)$
   - $l_{\mathcal{G}}^{(2)}(v) > l_{\mathcal{G}}^{(2)}(x)$

   If a path is found return true, else return false.

Variants

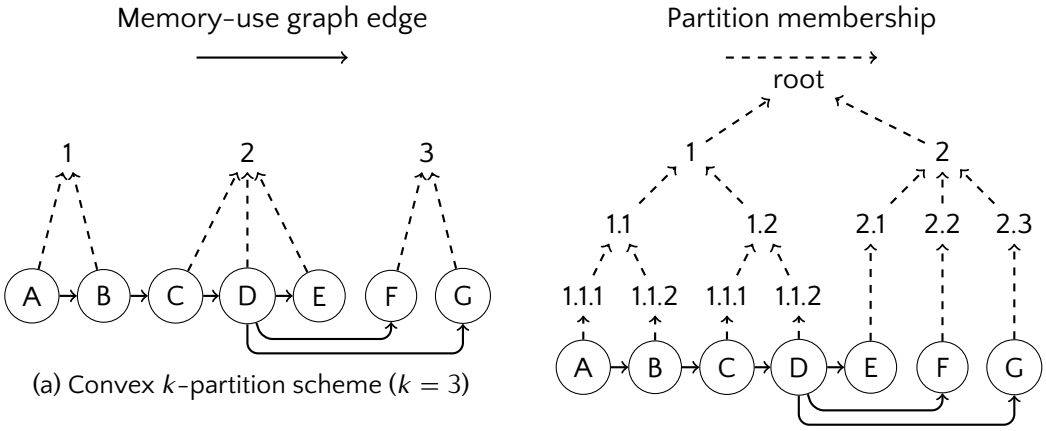
Additionally we experimented with some variations to the above described labeling and query protocols. We refer to this basic version as **standard DFS** because the queries are performed with a *depth-first search*:

Retro Additionally to the two first labeling traversals, two other labeling traversals are performed. Again one on $\mathcal{G}$ and one on $\mathcal{G}'$. In both cases, the order in which successors (or predecessors) are visited for each vertex during the DFS traversals is opposite to that of the two original traversals.

LabelOrder After the labeling traversals the lists of successors and predecessors of each vertex are ordered according to their label values.

BBFS When performing reachability queries the DFS traversal for searching a path is replaced by a *bi-directional breadth first search* (BBFS) traversal while still using labels to prune non-relevant vertices from the search.

These three variants can be used independently from each other, separately or together. For the experimental results see Section 6.5.



(a) Convex k-partition scheme ($k = 3$)

(b) Convex hierarchical partition scheme

Figure 6.2 – Types of partition schemes

6.3.2 Convex partitioning

As mentioned in Section 6.2, it appears that previous work for the subject of convex partitioning is close to non-existant. One of the few papers that brushes the topic by Fauzia et al. [Fau+13] introduces a first method which we present and use for comparison when evaluating other algorithms. The three algorithms that we present each produce one of two types of partition schemes: a k-partition with k distinct vertex sets identified within the graph or a hierarchical partition that divides the graph into multiple parts that are in turn divided into parts recursively down to the moment where each part is composed of only a single vertex. These two different types are illustrated in Figure 6.2. Additionally we enforce one specific constraint on hierarchical partition schemes:

Constraint 1: *Let $\mathcal{G} = (V, E)$ be a directed acyclic graph. Let $N(c_1, \ldots, c_k)$ be a non-leaf vertex of a hierarchical partition tree T of $\mathcal{G}$ where $(c_i)_i \in T$ are the children of N. For a vertex $v \in V$ the notation $v \in c_i$ states that v is part of the partition of $\mathcal{G}$ represented by c_i. Then:*

$$\exists (u, v) \in E \quad | \quad u \in c_i, \;\; v \in c_j, \;\; i \neq j \implies k = 2$$

In other words, if the sub-partitions of a given partition have edges between them then it is a bisection and there are only two sub-partitions. If there are more then two sub-partitions then these are not linked by any edges.

Greedy scheduling

The partitioning method by Fauzia et al. [Fau+13] is based on a *greedy scheduling* approach and will be referred to as such. It produces a k-partition scheme by sequentially filling up partitions. The order in which the vertices are added corresponds to an actual scheduling of the instructions represented by the vertices. This is enforced by the use of a priority-queue containing the vertices for which all predecessors have already been scheduled, *i.e* have been assigned to a partition. The priority-queue is ordered in such a fashion that it tends to insert neighboring vertices of the partition that is currently being filled. The exact order can be modified thereby potentially influencing the quality of the partition.

Because this method was designed with the goal of computing upper-bounds on the number of IO operations, it takes a second input parameter that corresponds to the size of the target memory. This parameter is used to determine how many vertices can be added to each partition without exceeding the memory constraint. The resulting partition scheme is convex by construction and the proof is provided in Theorem 6.2. For the details of the algorithm used for the implementation of this *greedy scheduling* approach one should refer to the original paper.

Theorem 6.2: *Let $\mathcal{G} = (V, E)$ be a directed acyclic graph. Let $f : V \to \mathbb{N}$ be a partition scheme that assigns each vertex to the identifier of its target partition following the greedy scheduling method. Then f is convex.*

Proof. Let $u, v \in V$ be two vertices assigned to the same partition, i.e $f(u) = f(v)$. Suppose that v is reachable from u, then

$$\exists (x_i)_{i \leq n} \quad | \quad (u, x_1) \in E, \ \ (x_n, v) \in E, \ \ \forall i \in]1, n] \ \ (x_{i-1}, x_i) \in E$$

For a vertex x_i to be inserted into the ready-set and thus into a partition all its predecessors should have been inserted themselves into partitions. As a consequence the vertices $(x_i)_{i \leq n}$ will be inserted in order into partitions. Because the partitions are generated in sequence this means that

$$f(x_1) = f(x_n) \implies \forall 1 < i < n \ \ f(x_i) = f(x_1) = f(x_n)$$

thereby proving the convexity of the partition scheme f as the same conclusion holds for the vertices of every path from u to v □

Convexify

A second approach to convex partitioning is based on non-convex graph-bisection and produces a convex hierarchical partition-scheme. It was established during a short internship by A. Olivry, a Bachelor student. A prerequisite for this approach is the ability to transform a non-convex bisection into a convex one. The main idea is to categorize vertices in a given non-convex bisection either as *safe* or *unsafe*.

Because of the convex criterion, all edges between the two partitions of a convex bisection of a directed graph should flow from one to the other. Let us call the source partition A and the sink partition B. A vertex is considered *safe* with respect to A if does not belong to A or B but all its predecessors belong to A. It is considered *safe* with respect to B if it does not belong to A or B but all its successors belong to B. If a vertex is not considered *safe* and it is *unsafe*. Source vertices are by definition *safe* with respect to A, as are sink vertices with respect to B. With this in mind, the steps to perform one single bisection with this *convexify* heuristic are as follows:

1. Perform a bisection with a non-convex partitioning algorithm.
2. Compute the initial *safe* vertices for A, *i.e* the sources, and B, *i.e* the sinks.
3. Targeting alternatively A and B, select a vertex that is *safe* relatively to the current target, add it to the corresponding partition and update its *safe* vertices. Continue until no *safe* vertices remain for neither A nor B.
4. If vertices remain that have not been assigned to A or B consider the sub-graph that they form and perform a recursive bisection starting at step 1. On completion the A (resp. B) partition of the bisection of this sub-graph is added to the A (resp. B) partition of the whole graph.

The corresponding pseudo-code is given as Algorithm 6.1.

As it is stated, the algorithm does not necessarily terminate. For the termination to be guaranteed we need to create at the least one new *safe* vertex at each recursive bisection. The non-convex bisection at Line 11 however does not give such a guarantee.

The solution is a forcing mechanism: whenever no progress is made at iteration, all the sources of the sub-graph of *unsafe* vertices are added to A (0 in the `convexId` variable) and all the sinks are added to B (1 in the `convexId` variable).

A proof of correctness of this *convexy* method with respect to the convex criterion is provided with Theorem 6.3.

Algorithm 6.1 Convexification of a non-convex bisection

Input: $\mathcal{G} = (V, E)$ Graph to be bi-sected
Define: `safeSet` Set of safe (assigned) vertices
Define: `convexId` Vector of convex bisection assignements of vertices
Define: `bisectId` Vector of bisection assignements of vertices
Define: `workQueue` Vertex FIFO queue
Define: `predCount`, `succCount` Vectors of integers

```
1:  for all v ∈ V do
2:      predCount[v.id()] ← |v.predecessors()|
3:      succCount[v.id()] ← |v.successors()|
4:  end for
5:  while |safeSet| < |V| do
6:      for all v ∈ (V \ safeSet do
7:          if (predCount[v.id()] = 0) OR (succCount[v.id()] = 0) then
8:              checkQueue.push(v)
9:          end if
10:     end for
11:     bisectId ← NonConvexBisect(V \ safeSet)
12:     while workQueue ≠ ∅ do
13:         v ← workQueue.pop()
14:         if (bisectId[v.id()] = 0) AND (predCount[v.id()] = 0) then
15:             convexId[v.id()] ← 0
16:             safeSet.insert(v)
17:             for all u ∈ v.successors() do
18:                 predCount[v.id()] −−
19:                 if predCount[v.id()] = 0 then
20:                     workQueue.push(u)
21:                 end if
22:             end for
23:         else if (bisectId[v.id()] = 1) AND (succCount[v.id()] = 0 then
24:             convexId[v.id()] ← 1
25:             safeSet.insert(v)
26:             for all u ∈ v.predecessors() do
27:                 succCount[v.id()] −−
28:                 if succCount[v.id()] = 0 then
29:                     workQueue.push(u)
30:                 end if
31:             end for
32:         end if
33:     end while
34: end while
35: return convexId
```

Theorem 6.3: *Let $\mathcal{G} = (V, E)$ be a directed acyclic graph. Let $f : V \to \{A, B\}$ be a bisection scheme that assigns each vertex to the identifier of its target part (A or B) following the convexify method. Then f is convex.*

Proof. The convexity of the bisection is defined by the algorithm as

$$\nexists (u, v) \in E \quad | \quad f(u) = B, \;\; f(v) = A$$

By contradiction let $(u, v) \in E$ be an edge such that $f(u) = B$ and $f(v) = A$. For a vertex such as v to be added to A it must be considered safe. As such all its predecessors should be safe too and belong to A. This is direct contradiction with our starting hypothesis and thus proves the theorem. □

Max-distance criterion

The last method for convex partitioning that we present also produces a convex hierarchical partition scheme. Because of this the partitioning must be performed in a recursive manner similar to the *convexify* method. For the bisection we use the combination of two integral values, *MaxSource* and *MaxSink*, defined for each vertex that contain a partial information about the global connectivity of the graph.

MaxSource This value corresponds for each vertex v to the length of the longest path leading from a source of the graph to v.

MaxSink This value corresponds for each vertex v corresponds to the length of the longest path leading from v to a sink of the graph.

Multiple methods can be used to effectively compute the values of these two characteristics for each vertex of a graph: modified Bellman-Ford, extended DFS, ...

When all vertices of a graph $\mathcal{G} = (V, E)$ have received a *MaxSource* and *MaxSink* value the bisection can be performed. With $f : V \to \{A, B\}$ the partition scheme:

$$\forall v \in V \;\; f(v) = \begin{cases} A & \text{if } \mathit{MaxSource}(v) < \mathit{MaxSink}(v) \\ B & \text{if } \mathit{MaxSource}(v) \geq \mathit{MaxSink}(v) \end{cases}$$

The proof for the convexity of the result of this bisection method is given with Theorem 6.4.

Theorem 6.4: *Let $\mathcal{G} = (V, E)$ be a directed acyclic graph. Let $f : V \to \{A, B\}$ be a bisection scheme that assigns each vertex to the identifier of its target part (A or B) following the max-distance method. Then f is convex.*

Proof. Suppose by contradiction that there exists a path from B to A, i.e

$$\exists (u, v) \in E \quad | \quad f(u) = B, \;\; f(v) = A$$

By the definition of f

$$MaxSource(v) < MaxSink(v)$$

Additionally because $(u, v) \in E$

$$MaxSource(u) < MaxSource(v) \quad \text{and} \quad MaxSink(v) < MaxSink(u)$$

Together these relations give

$$MaxSource(u) < MaxSink(u)$$

Which contradicts the fact that $f(u) = B$. □

While the initial graph $\mathcal{G}$ may be connected there will be sub-graphs during the recursive bisections that are not. Instead of bisecting using the max-distance criterion, we can trivially partition such sub-graphs into their connected components while conserving the convex property of the hierarchical partition scheme. These leads to a partition with potentially more than the two child partitions that the bisection based on *MaxSource* and *MaxSink* would produce.

6.3.3 Convex coarsening

Coarsening is a much used technique for non-convex graph partitioning. The main principle is to repeatedly reduce the size of the graph to be partitioned in a hierarchic way. Partitioning can then be performed on the most coarse graph in order to speed-up the process. Because the size of the graph is considerably reduced, the partitioning is much faster. The graph is then uncoarsened one level and the partitions resulting from the previous level can then be refined to improve the objective function of the partitioning (cutsize, partition weights, etc.). The uncoarsen-refine process is iterated until the original graph is partitioned. This type of partitioning is referred to as multi-level partitioning [HL95].

The same kind of approach could also benefit the sub-topic of convex partitioning. Most notably, when processing smaller graphs it would be possible to use a broader range of heuristics with time complexities that go beyond the quasi-linear limit that is required for large graphs. This would enable such heuristics to take more criteria into account besides convexity, *e.g* cut minimization. However, in order to guarantee the convexity of the final partition scheme; the coarsening should also be performed in a convex manner. Because of this convex graph coarsening can be directly associated with a convex k-partitioning for which the partitions are collapsed into a single vertex. The only method which performs convex k-partitioning is the *greedy scheduling* algorithm exposed previously and can thus also be used as a convex graph coarsening method.

An important observation at this point is that the reachability indexation approach that was presented in Section 6.3.1 has not been put to use in the context of convex partitioning. Over the course of the development of the various approaches to convex partitioning, it appeared that the interactions between partially constructed partitions and the reachability index were difficult to control. For example, no straightforward method partially recomputes the index after merging two or more vertices as is the case in graph coarsening.

Nonetheless, with graph indexing, convex partitioning and convex coarsening addressed we can now turn to the next step in the analysis of execution traces: the evaluation of the memory-cost and the computational intensity of the obtained partitioning schemes.

6.4 Cost analysis of a partitioning scheme

As stated in the introduction of this chapter the goal of performing convex partitioning of execution trace graphs is to improve performance debugging. We want to give an estimate for a computational intensity that should be achievable for the code that is being analyzed. The convex property of a partitioning enables us to apply the memory usage model defined in Chapter 1. The trace graph follows semantics that are similar to that of a *memory-use graph*: vertices represent instructions and edges represent reuses. The most notable differences is that trace graphs represent all iterations of the underlying code for a given input and may not even exhibit a regular structure with respect to these iterations.

As the linearization, scheduling in other words, of the *memory-use graph* influences the memory-cost and therefor the computation intensity, we first need to de-

cide how the graph is linearized based on a partitioning scheme. Once a schedule has been chosen, we can then study how the IO cost can be computed. Two methods are proposed: one uses the obtained schedule and the principle of reuse distance and the second method only uses the partitioning scheme.

6.4.1 From partitioning to schedule

For a linearization of a *memory-use graph* to be legal, all edges should be compatible with the ordering of vertices. If we suppose $\mathcal{G}$ to be a graph and f a convex partitioning scheme of $\mathcal{G}$ then we can use the convex property of f to define a partial order between the partitions. Two types of partitioning schemes lead to two different cases.

k-partitioning

The ordering of vertices within each individual partition cannot be inferred through the partitioning scheme. Similarly there is no single way to derive an ordering of the k partitions. As such, any pre-order traversal of the partition graph can be valid, and any pre-order traversal of the vertices of each individual partition can be valid too.

Minimizing computational complexity, or in other words reducing the max-live of a set of instructions below a given threshold, is an NP-complete problem [CB76] even though many effective heuristics exist that produce near-optimal results. Nonetheless the cost of even polynomial heuristics is still prohibitive given the size of a trace graph. Furthermore, as our goal is not to give a schedule but merely to indicate if the current computational intensity observed in the trace graph could possibly be improved, a non-optimal scheduling is sufficient. A default approach would be for each partition to use the schedule implied by the execution trace.

Hierarchical partitioning

The leaves of the partition tree all correspond to partitions that contain a single vertex. All internal nodes of the tree represent either a convex bisection or a multisection of unconnected sub-graphs. As a result, a hierarchical convex partition scheme can be directly translated into a valid schedule. This enforces an explicit ordering of the partitions in the case of a bisection. In the case of a multi-section the partition order can be chosen arbitrarily, the influence of such a choice on the computational intensity is discussed below in Section 6.4.3.

Note, it is possible, given a linearization of the trace graph, to construct a partition

tree which has the property of being convex. This can be achieved through the recursive bisection of the linearization where each new bisection creates a new node in the partition tree. A potential use-case would be the evaluation of a known schedule with the same methods that are used on hierarchical convex partition schemes.

6.4.2 Reuse distance

For a given schedule the most straightforward way for evaluating the computational intensity is the *reuse distance* mesure [Mat+70]. The IO cost is equal to the number of reuse distances greater than the size of the target memory. Using the approach of Bennett and Kruskal [BK75] for the computation of reuse distances, the evaluation of the computational complexity of a schedule for a trace graph $\mathcal{G} = (V, E)$ can be done in $\mathcal{O}(|V| \cdot B \cdot \log(|V|))$ where B is the blocksize of the reuse distance computation. The IO cost for a given memory size C can be computed in $\mathcal{O}(|V|)$ by iterating over the reuse distances. This kind of evaluation does not use any form of tiling as input and it is therefore necessary to transpose a partition scheme of the trace graph into a valid schedule as described above in Section 6.4.1.

Furthermore, because the evaluation of reuse distances is based on a given memory allocation the modification of the global schedule may go against implicit *Write-after-Read* dependencies that execution traces do not represent. This can be solved by analyzing the execution trace for such anti-dependencies and representing them on the trace graph with edges with no weight. A second effect of the fixed memory allocation is that it may oppose some of the beneficial effects of rescheduling: extra IO operations may be incurred with a new scheduling because seperate data elements share the same cache-line which could have been avoided with a different memory allocation. In order to study trace graphs from the point of view of pure data reuses and leaving the issues of memory allocation on the side we present a novel cost analysis method.

6.4.3 Greedy evaluation in a hierarchical partition

In case the computational intensity is computed with a hierarchical partition scheme, it is possible to use the partition scheme for the evaluation directly.

As stated in the definition of the memory usage in Section 1.4.3, the memory usage of a given partition corresponds to the *max-live* measure over the execution of its instructions. Starting with the leaf nodes of the partition tree we know the memory

usage of these partitions: it is directly equal to the size of the data element manipulated by the instruction represented by the vertex.

We also define a new measure, the *FrontierLive*, for nodes in the partition tree. It corresponds to the total amount of reuses that occurs between child nodes. An important observation here is that the characteristics of trace graphs imply that each vertex of the graph corresponds to the execution of a single machine instruction. This means that all edges originating at a given vertex are associated with the reuse of the same data element, *i.e* the result of that machine instruction. As a result, if a vertex in one child partition is the source of multiple edges that all have their destination vertex in the same other child partition, the corresponding data element is reused only once between these two child partitions. As such, it should only contribute once to the *FrontierLive* measure of the parent partition.

From here on, we can define the memory usage of a non-leaf partition N recursively as the maximum of the individual memory usages of child partitions $(c_i)_{i\in\mathbb{N}}$ or its *FrontierLive*:

$$\text{MemoryUse}(N) = \max\left(\max_i\left(\text{MemoryUse}(c_i)\right), \text{FrontierLive}(N)\right)$$

An illustration of the *MemoryUse* and *FrontierLive* of partition tree nodes is given in Figure 6.3. It contains an example with multiple edges having the same source vertex. These edges only contribute once to the *FrontierLive* of the parent partition: vertex D has E, F and G as successors but the *FrontierLive* of the root partition is only 1.

The computation of the labels of all nodes in a partition tree T requires the analysis of all edges in the target graph $\mathcal{G} = (V, E)$ for the *FrontierLive* measure and a traversal of all the nodes in T for the *MemoryUse* labels. Its complexity is $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of vertices in the trace graph. Because of the nature of trace graphs and the data reuses they represent we can reasonably assume that $|E| \sim k\dot{|}V|$. In the average case the complexity is thus actually of $\mathcal{O}(|V|)$.

For a given target memory size C, the computation of the IO cost of the hierarchical partition scheme is done by a recursive traversal of the partition tree. If a node's *MemoryUse* is greater than C, its *FrontierLive* is added to the total IO cost of the partition tree and its children are scheduled for traversal. If a node's *MemoryUse* is lower than or equal to C, the node can be executed without generating any IO operations with a distant memory, similarly to the situation of a tile in the *generalized tiling* optimizations. In Figure 6.3, a target size of 1 would have given an IO cost of 3 whereas a target size of 2 would correspond to a cost of 0.

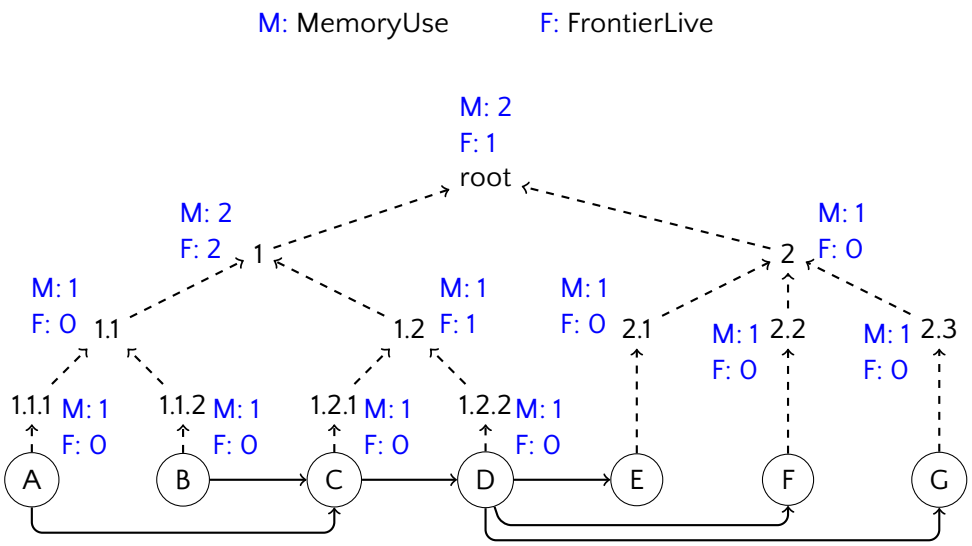


Figure 6.3 – Memory usage labeling of the nodes in a partition tree

In an alternative approach, a similar traversal of the partition tree computes a diagram of the IO cost associated with a trace graph for *any* target memory size. This can also be done once *reuse distances* have been computed.

A consequence of the traversal is that the complexity is $\mathcal{O}(|V|)$. This method has thus a lower global complexity then the reuse distance method but it is less precise in its evaluation because:

- The recursive definition of the *MemoryUse* measure may underestimate the real amount of memory that is required.
- The IO cost model may overestimate the number of real IO operations that are required as the *FrontierLive* measure assumes that all reuses between partitions require a communication with distant memory.

6.5 The GraphUtilities library

The graph algorithms presented in Section 6.3 have all been implemented in C++ in a single framework: the *GraphUtilities* library[2]. It can be used as a general toolbox for the exploration and prototyping of novel methods for the evaluation of the computational intensity and memory-cost of execution traces without being limited to this context. Where possible, depending on the required algorithm, the library supports

[2] https://www.github.com/Helcaraxan/GraphUtilities

multi-threading in order to speed-up graph queries or to process multiple queries in parallel. Nearly all features have been implemented from scratch. Only the implementation of the *convexify* partitioning method makes use of an external tool: the PaToH hypergraph partitioning tool [ÇA96].

The choice of the non-convex partitioning algorithm for the *convexify* method is open as long as the algorithm is able to create two seperate partitions. However, the same reasoning which was used for the definition of the *FrontierLive* measure, leads to prefer a hypergraph partitioning algorithm: multiple edges from the same vertex of one partition to a set of vertices in another partition imply only one reuse. As such hypergraph partitioning is a much closer match for our IO cost metric.

For all the acquired data points used in the experimental evaluations, the average of 5 separate runs was used. Unless specified otherwise, the benchmarks were performed on a small platform composed of an Intel Core i5-3230M dual-core CPU with 8GB of RAM. The hyper-threading functionality was activated in order to enable the use of 4 simultaneous threads. The memory hierarchy includes two separate 32kB L1 caches for data and instructions per core, a unified 256kB L2 cache per core, as well as a unique and unified 3MB L3 cache that is used for chip-wide cache-coherency. The size of the platform was deliberately chosen in order to showcase the possibilities and performances offered, even with reduced resources, by the reachability and partitioning algorithms presented in this chapter.

6.5.1 Evaluation of reachability indexing

Experimental setup

As noted in Section 6.3.3, we have not been able to use our reachability indexing method in the context of convex partitioning. As a consequence, the experimental evaluation of the indexation and the processing of subsequent reachability queries was done in realistic use cases, taken from other applicative domains. The corresponding graphs have also been used in previous work on the subject. The total of 20 graphs include small sparse graphs, small dense graphs and large real graphs. For detailed descriptions as well as more characteristics for each individual graph, the paper by Yildrim et al. [YCZ12], authors of the GRAIL tool, is a good reference.

We compare the performance of the *GraphUtilities* implementation to that of the GRAIL method. It can be considered state-of-the-art for the graph reachability labeling-based methods that fall in the same category as our post-order labeling. The GRAIL implementation is provided as an open-source package by its authors and was used

with only some minimal modifications for benchmarking purposes. Timing information was obtained with the PAPI library [Bro+00].

Results & analysis

The indexation being necessary to process reachability queries, it is the first functionnality that was evaluated. Different variants of the indexation method were compared: *Standard*, *LabelOrder*, *Retro* and the combined *Retro LabelOrder*. The resulting speed-up and slow-down values with respect to GRAIL are shown in Figure 6.4.

As can be observed, all test cases show a considerable speed-up. For each variant of the indexing method, the values observed for the small graph tests vary only slightly from their average value. However, the values obtained for the large graph tests are considerably less consistent. This is mainly due to the varying structures of these graphs where differences in behavior are exacerbated by their size. The general speed-up can be attributed to the fact that the structure of each labeling traversal is much simpler in the case of the post-order numbering and predecessor / successor reordering than it is for GRAIL.

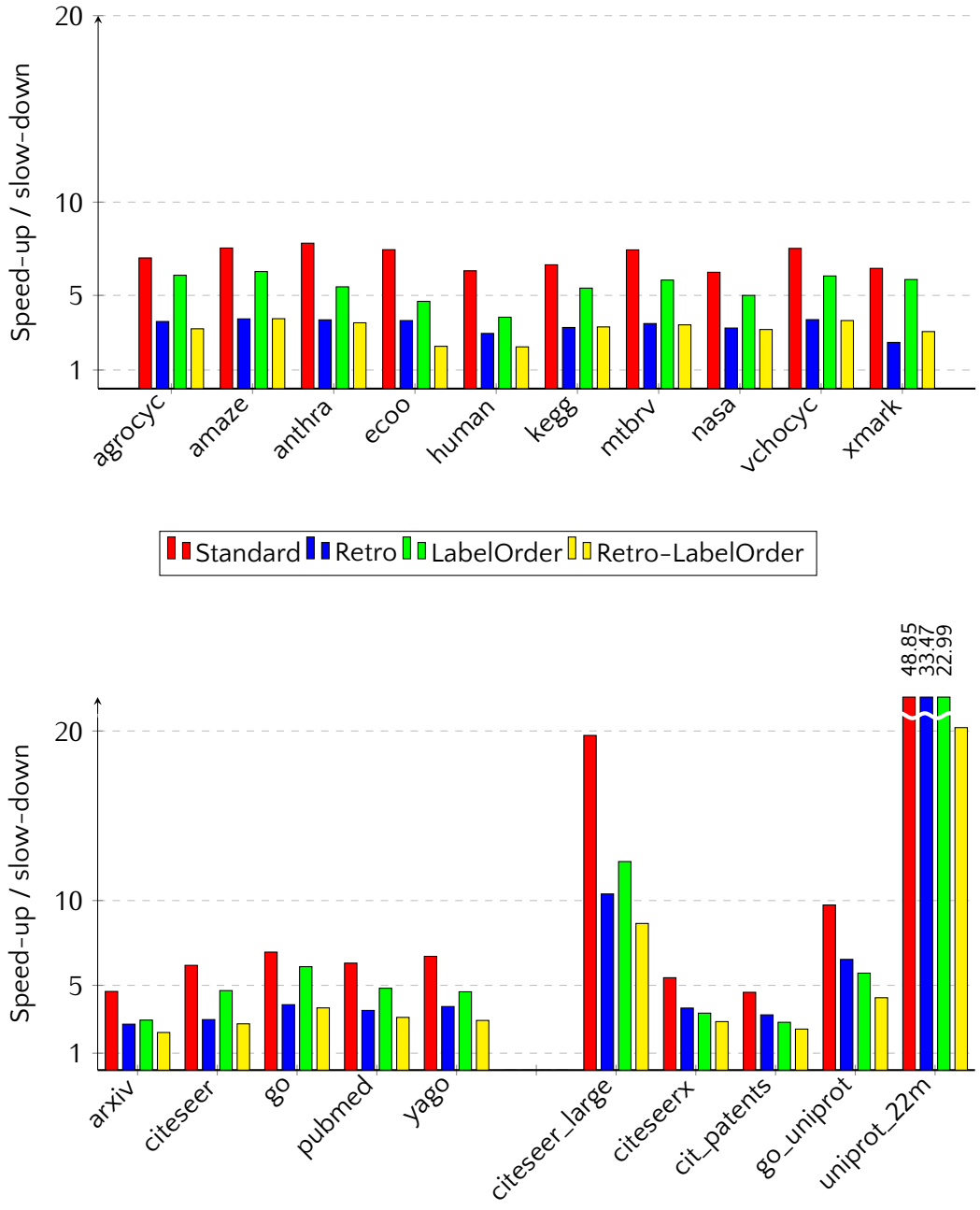
The varying speed-up values of the different variants of the indexing method can also be directly linked to the complexity of each labeling traversal. With the simplest structure, the *Standard* variant achieves the greatest speed-up in every test case. Adding the reordering of labels to the two initial traversals, the *LabelOrder* variant comes in second for all small graphs and is on near equal footing with the *Retro* variant on the large graphs. Quite naturally, the *Retro* variant with its total of four traversals has a speed-up that is reduced with respect to the *Standard* version. The combined *Retro-LabelOrder* variant comes in last.

6.5.2 Evaluation of reachability queries

Experimental setup

The evaluation of reachability query performances was performed on the same graphs that were used for the indexing evaluation. Each run of a query test typically involves 10^5 queries. Different sets of queries were created:

Random For each graph a set of random queries was created. A consequence of this randomness is the fact that the overwhelming majority ($> 90\%$) of the queries should return negative answers.



Small dense graphs agrocyc, amaze, anthra, ecoo, human, kegg, mtbrv, nasa, vchocyc, xmark

Small sparse graphs arxiv, citeseer, go, pubmed, yago

Large real graphs citeseer_large, citeseerx, cit_patents, go_uniprot, uniprot_22m

Figure 6.4 – Indexation time speed-up / slow-down compared to GRAIL

Positive For a some specific graphs a set of only positive queries was generated. The source vertex of each query was chosen at random followed by a random DFS through the graph. The vertex visited after $n \in [\![3; 8]\!]$ hops was then chosen as the target vertex of the query.

Deep positive For the same specific graphs as in the case of the *positive* queries a second set of only positive queries was created. The method remains the same except for the fact that $n \in [\![10; 15]\!]$.

Results & analysis

Results of the benchmarks involving the *random* query sets are separated in two distinct sets. The first set, using the DFS search policy when confronted with a not-immediate negative query answer, is given in Figure 6.5. The second set, displayed in Figure 6.6, uses the BBFS search policy instead.

From these first two result set for which the majority of queries resulted in a negative answer we can make multiple observations:

- Modifying the ordering of the predecessors and successors of each vertex according to their labels does not provide a significant improvement of performances. Only in one case (*go* - DFS and BBFS) is there any noticeable difference between the *Standard* and *LabelOrder* variants.
- A potential benefit of four labels per vertex over two labels does not appear on the *random* query sets. Only in the case of the *arxiv* graph does the difference between the *Standard* and *Retro* variants appear clearly. This difference however does not designate a winner because of the inversion of the situation between the DFS and the BBFS tests.
- In the majority of the tests the difference between the DFS and the BBFS performances is minor. Again the *arxiv* graph is an exception, together with the *citeseerx* graph.
- On a global level the results tend to indicate a speed-up of around 1.2 compared to the GRAIL method for the *random* query sets.

Figure 6.7 contains the results for the *positive* and *deep positive* query sets with a DFS policy. The results for the corresponding BBFS policy tests are shown in Figure 6.8. Because the queries of the involved query sets all should return a positive answer they

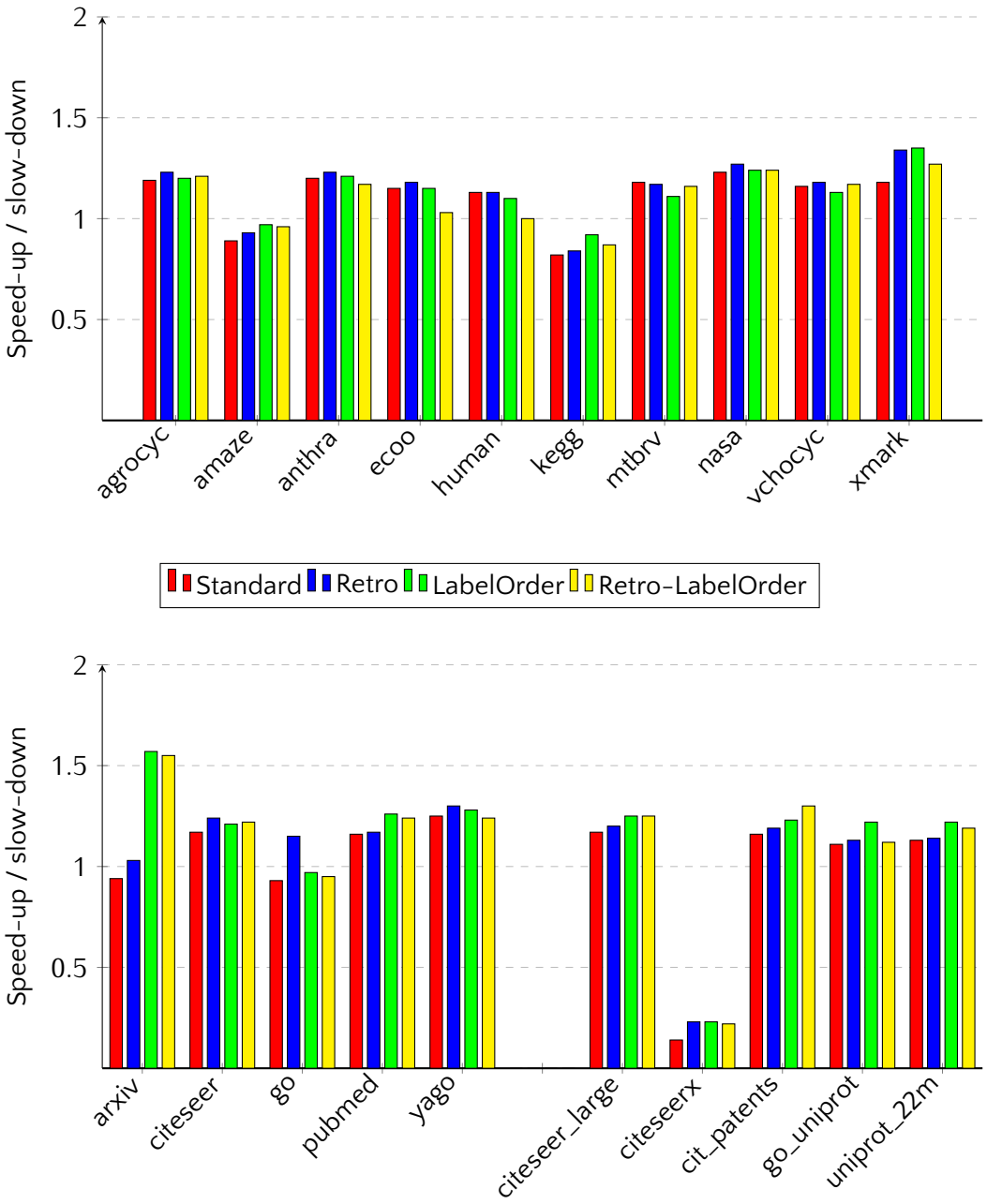


Figure 6.5 – Random query with DFS search policy speed-up / slow-down

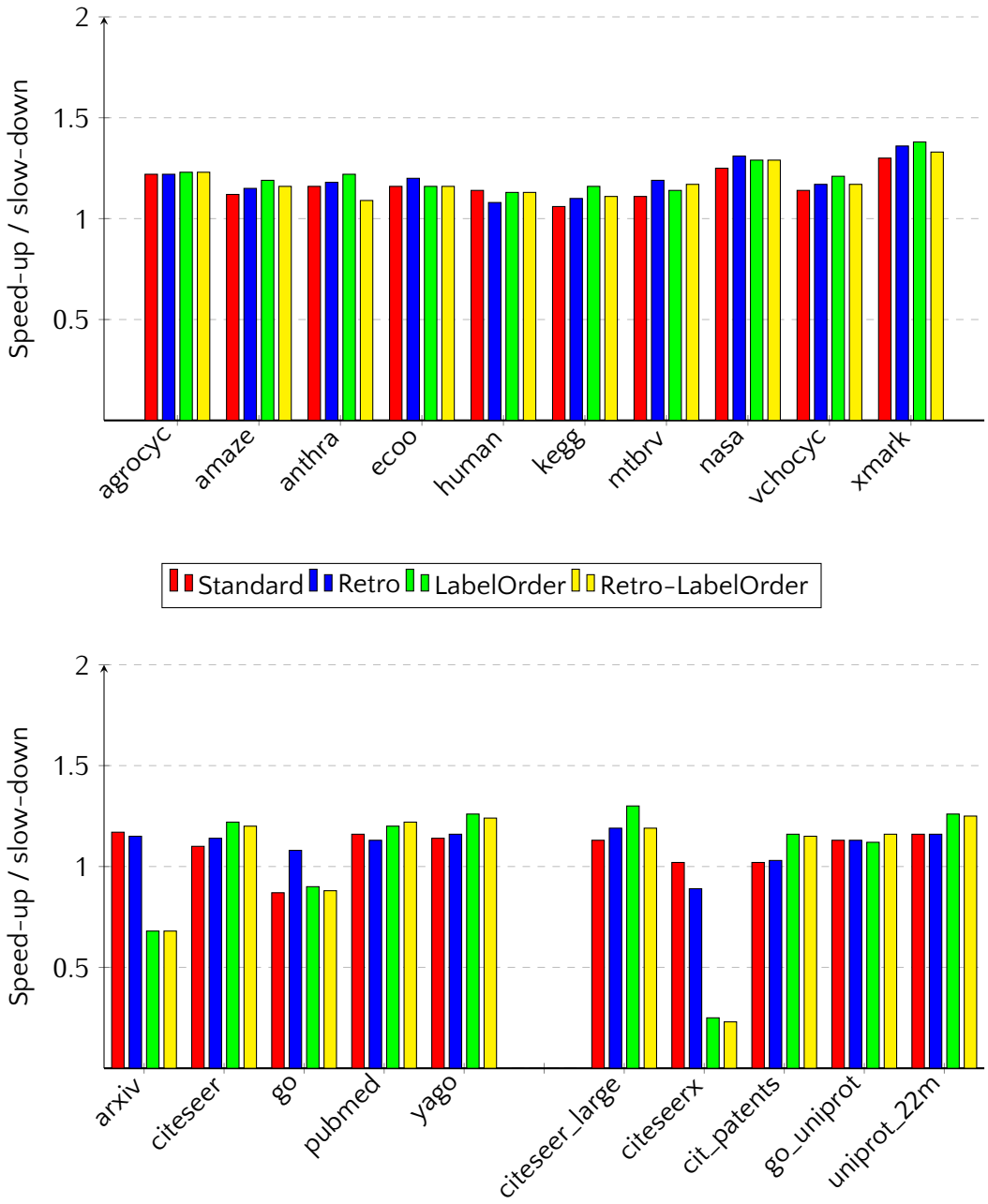


Figure 6.6 – Random query with BBFS search policy speed-up / slow-down

all imply a search of the graph for a path from the source to the target vertex. The obtained results give a good indication about whether the label-based vertex filtering is effective or not in comparison to the GRAIL method.

- The reordering of predecessors and successors with the *LabelOrder* variants are again not much more successful than the variants without reordering. In the case a difference appears (*arxiv*, *citeseerx* and *go_uniprot*), it may either be a speed-up or a slow-down and the difference between the DFS and BBFS search policies is even greater in these cases.

- The four label variants make a difference on some specific graphs. The *go_uniprot*). test-case has the particularity of having some vertices with a vary high arity. As the two consecutive forward and backward labeling traversals are performed by iterating over the successors and predecessors in opposite directions. The large arity can have a very big influence in the case of a BBFS traversal by filtering out far more vertices at each breadth-wide sweep. The effect is reversed on the *citeseerx* graph as the vertices have a much lower arity. The extra labels do not provide any significant filtering while requiring more label comparisons for each visited vertices, as can also be observed on the *arxiv* and *cit_patents* graphs.

- Contrary to the *random* and mostly negative query sets, the graph searches of the *positive* and *deep positive* query sets show some strong differences between the DFS and BBFS search policies, especially for the large real graphs. This can again be attributed to the varying structures of the graphs. DFS search performs better on a graph with high arity vertices whereas BBFS performs better on graphs with low arity vertices.

- The global speed-up observed on the *random* query sets is only partially confirmed on the new query sets. However if for each query set and variant the best result of both the DFS and BBFS search policy is taken, the global speed-up remains. This led to the introduction of an automatic selection of either a DFS or a BBFS search policy in the *GraphUtilities* library. The first queries both a DFS and BBFS search are performed and timed. The fastest policy is considered as winner. When a search policy achieves k wins (k configurable) it is automatically used for all future queries.

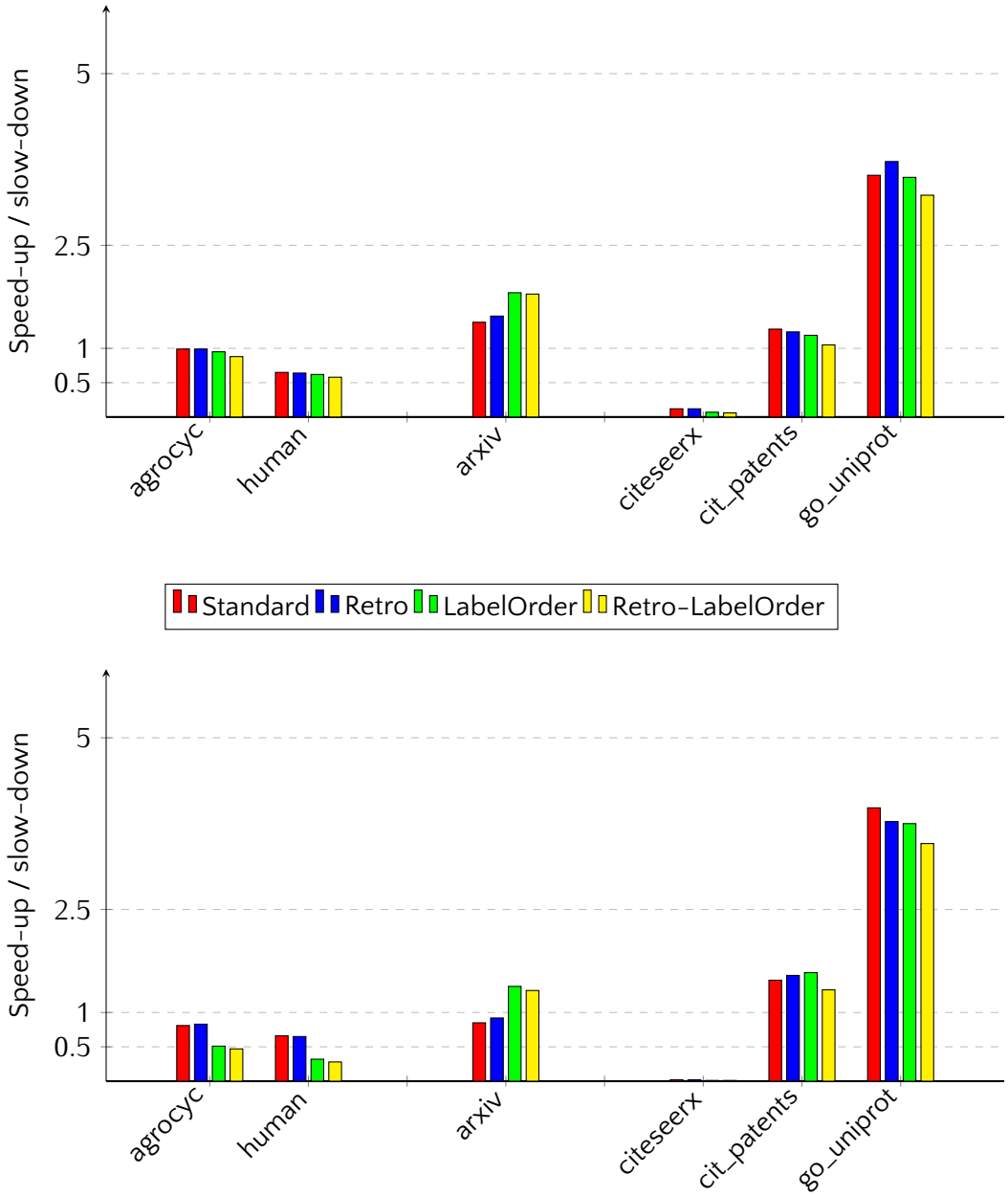


Figure 6.7 – Positive and deep-positive queries with DFS speed-up / slow-down

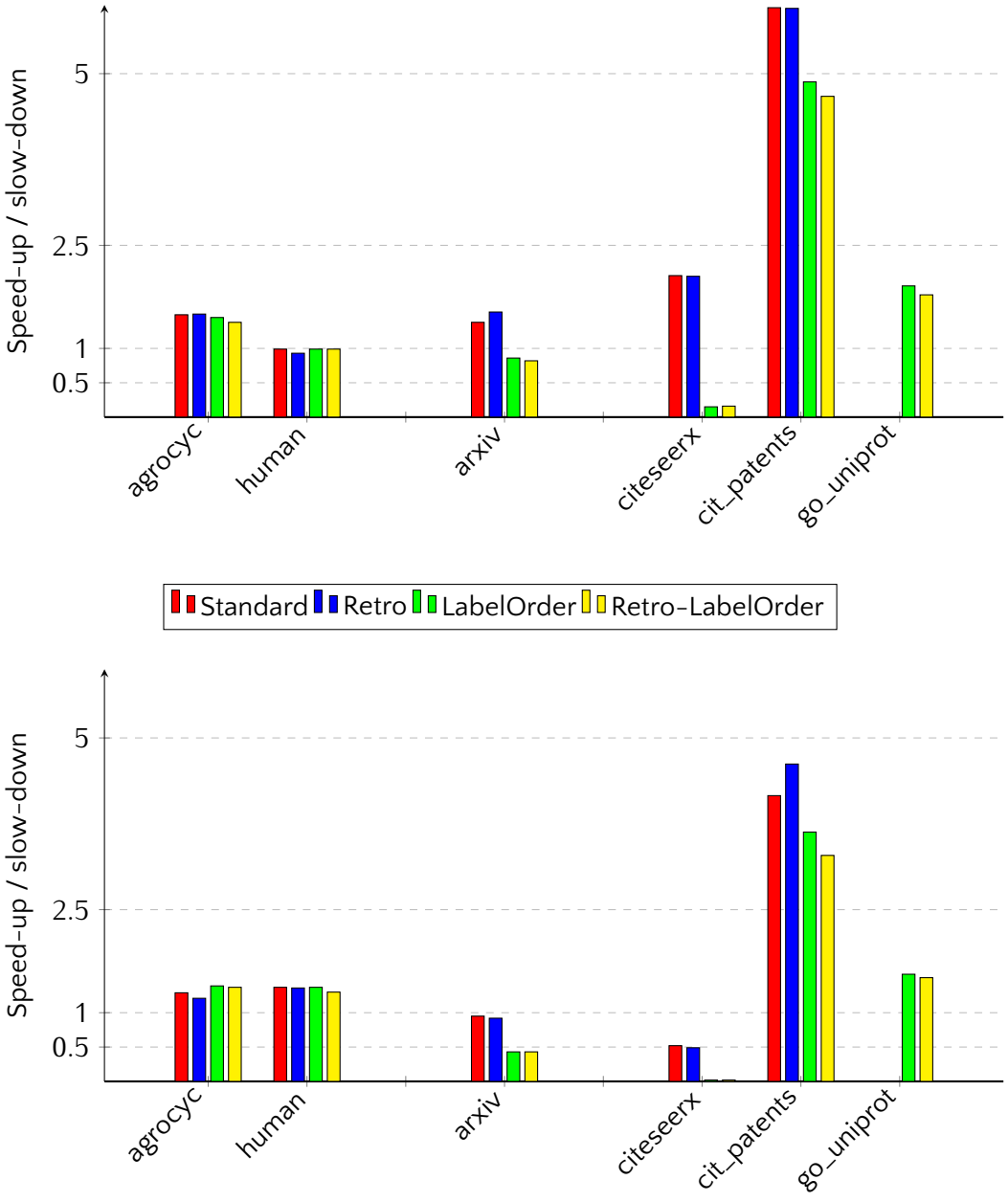


Figure 6.8 – Positive and deep-positive queries with BBFS speed-up / slow-down

6.5.3 Evaluation of convex partitioning

Experimental setup

For the evaluation of our convex partitioning heuristics a set of real-world execution traces was produced with the *Dynamic Dependency Graph*[3] tool on the benchmarks of the Polybench[4] suite. The traces do not represent the full execution of a loop as they were obtained by iteration sampling.

The value of interest for analysis is that of the communication cost upperbounds that the evaluated methods achieve, the lowest value obviously being the best. We compare three methods: the *greedy scheduling* heuristic, the *convexify* method and the *max-distance* criterion. For each method and benchmark couple the IO costs were computed for different cache sizes, ranging from 256 to 4096 bytes with 128 byte increments. Questions of cache associativity were ignored as we have been assuming a perfect memory mapping since Section 1.5.1. The method used for the IO cost estimation is the greedy evaluation presented in Section 6.4.3 in the case of the *convexify* and *max-distance criterion* heuristics. In the case of the *greedy scheduling* we compute the number of cuts between the partitions with a hypergraph metric, *i.e* multiple edges from the same vertex targeting the same partition counts only once.

A subset of the results measurements is presented in Figures 6.9 and 6.10.

Results & analysis

When observing a single method, we note that both the *convexify* approach and the *max-distance criterion* produce strictly decreasing communication costs, a logical result given the increase of the target cache size. The *greedy scheduling* heuristics differs in that its IO costs are only globally decreasing while experiencing some small local increases. This can be explained by the fact that the memory size is used as input for the partitioning and that as a result the partitioning changes with each test.

The comparison of the results for different methods leads to a first general observation. For every benchmark and target cache size the *convexify* method outperforms the *max-distance* one. Both methods are based upon the construction of a hierarchical partition tree, which is then used for the computation of the communication cost as described in Section 6.4.3. Because of this resemblance, a possible conclusion would be that the tree produced by the *convexify* method is of better quality than the one

[3] Developed at the Ohio State University and the Corse research team at Inria
[4] https://sourceforge.net/projects/polybench/

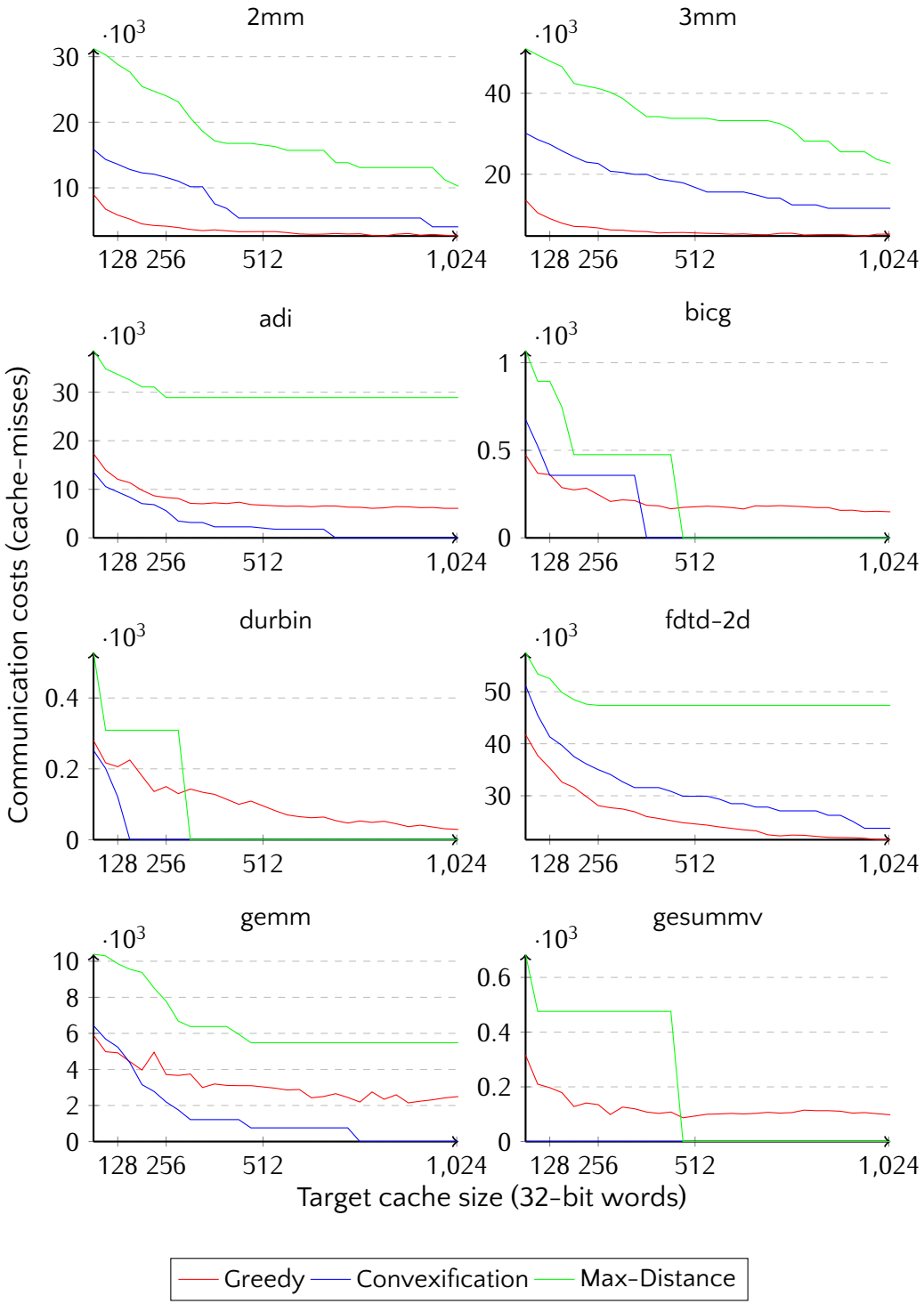


Figure 6.9 – Communication cost upper-bounds for multiple execution traces (1)

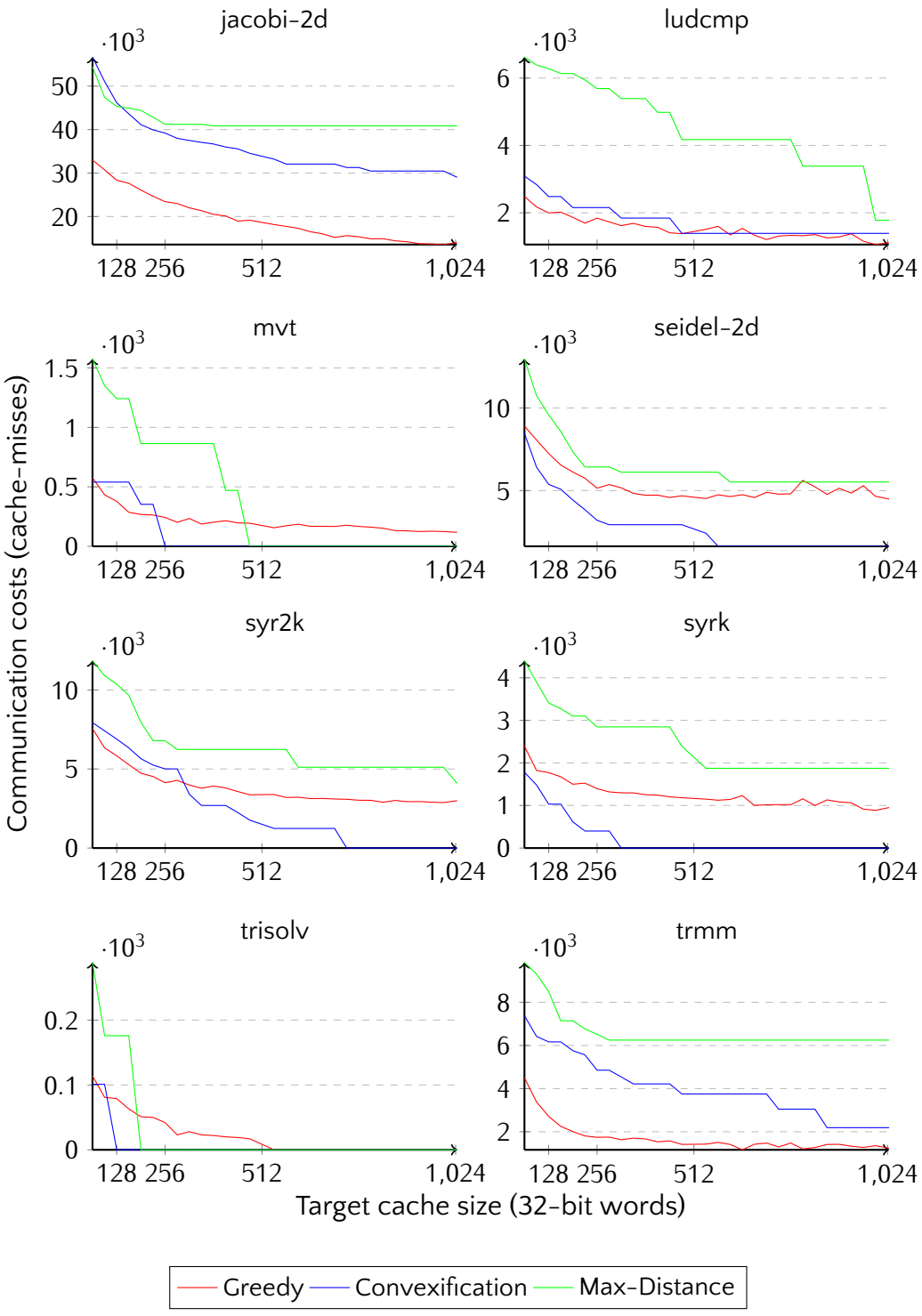


Figure 6.10 – Communication cost upper-bounds for multiple execution traces (2)

Benchmark	Min	Max	Std.Dev
adi	15	19	0.19
bicg	9	13	0.17
durbin	10	12	0.24
fdtd-2d	15	20	0.20
gemm	12	17	0.27
gesummv	11	13	0.13
jacobi-2d	15	20	0.26
ludcmp	11	16	0.15
mvt	12	14	0.15
seidel-2d	12	17	0.27
syr2k	13	17	0.24
syrk	12	15	0.12
trisolv	9	11	0.25
trmm	14	17	0.22

(a) Convexify tree-depth characteristics

Benchmark	Min	Max	Std.Dev
adi	4	30	13.71
bicg	3	21	10.60
durbin	2	29	19.97
fdtd-2d	2	40	22.89
gemm	3	27	15.17
gesummv	4	23	11.06
jacobi-2d	2	35	18.74
ludcmp	2	42	23.71
mvt	4	27	13.56
seidel-2d	4	46	39.66
syr2k	3	23	14.05
syrk	2	23	12.58
trisolv	2	26	19.90
trmm	3	29	17.91

(b) Max-Distance tree-depth characteristics

Table 6.1 – Hierarchical partition tree characteristics

generated with the max-distance criterion. A detailed observation of the characteristics of these trees, beyond the communication cost sheds more light on the issue. The *convexify* method produces well balanced trees whereas the trees from the max-distance criterion are far from balanced, with very significant variations in the depths of the leaf partitions. Table 6.1 contains a summary of the tree characteristics for some of the benchmarks. Because of the recursive evaluation method used for the communication cost computations, such unbalanced trees may have exhibit structures that are far from optimal. A potential solution for this phenomen would be to design some tree balancing heuristics that remove these suboptimal configurations. In practice, if such a heuristic were applied, a great potential for improvement is shown by the existence of corresponding convexified trees.

A second comparison involves the *convexify* and *greedy scheduling* methods. The results are mixed as the *greedy scheduling* heuristic and the *convexify* method each outperform the other on multiple benchmarks. It appears however that in general the *convexify* method dominates in a majority of cases: in 11 out of the 16 benchmarks, its communication costs are the lowest.

Last but not least, on multiple benchmarks (*bicg*, *durbin*, *gesummv*, ...), both the *convexify* method as well as the *max-distance* criterion generate schedulings for the trace graphs that do not involve any spill-related cache misses for relatively small cache sizes whereas the *greedy scheduling* heuristic generates misses even for larger cache sizes.

Conclusion

This wind, it was not the ending. There are no endings, and never will be endings, to the turning of the Wheel of Time. But it was *an* ending.

R. Jordan & B. Sanderson – *A Memory of Light*

A few steps back

Over the course of this work, we have attempted to present a coherent approach to the topic of data locality on manycore architectures and beyond, in a wide range of application fields. Below, we summarize the high-level framework that has been designed and implemented during this study as well as the main results that were obtained.

We started with the observation that the data reuse information in the case of loops is similar in structure to that of a dataflow program. This lead us to define a common representation to represent the data reuse within a code that is executed multiple times, in an iterative fashion: the *memory-use graph*. This representation can be used at different granularity levels with each atomic element corresponding to either machine instructions, basic-blocks or even functions as well as any intermediary levels. The reuses that are represented in the graph only include communications induced by the code structure and not those corresponding to the spilling of values from register to memory. With the *memory-use graph* designed, we then created a precise memory-usage model. It computes the amount of memory required to execute an arbitrary sequence of statements taken from *memory-use graph* without spilling values to higher levels of the memory hierarchy. At the same time, this model also gives a over-approximated estimate of the amount of IO operations that are required for the code's execution.

Another goal was to optimize data locality for manycore architectures. In such an environments memory communications may involve the usage of a Network-on-Chip. This is a source of undetermined latency and bandwidth bottlenecks. To overcome this limitation it was necessary to create a traffic model for the MPPA manycore chip, the main target for our optimization work. Indirectly this also lead to a redesign of the bandwidth limiter circuit to enable fine-tuned control over communications on the network in combination with the traffic model.

The next step was to apply the memory and communication cost model established in Chapter 1 towards a generalized form of tiling, an optimization that has traditionally been used on loops. The key difference between the existing tiling optimizations and our generalized version lies in the representation of the code that is being optimized. The polyhedral model for loop tiling relies on the representation of iterations as an n-dimensional grid and it is in fact the grid that is being tiled, not the code. In the same way our *generalized tiling* relies on the representation of the code and its iterations as a *memory-use graph* and it is the graph that is being tiled and not the

code. This abstraction has two consequences:

- The *memory-use graph* can represent multiple granularities of code and as such *generalized tiling* is not limited to the optimization of loops but can be extended to dataflow languages.
- The *memory-use graph* does not exhibit the same uniformity as an iteration space in the polyhedral model and because of this tiles in the case of *generalized tiling* may be of varying irregular shapes.

To find an optimal tiling we propose two different approaches. A first method, using *constrained programming*, finds optimal solutions when no timeout occurs. A second method uses of heuristics which attempt to find near-optimal solutions. We have proposed three different heuristics.

As a first application field for *generalized tiling*, we chose the improvement of register usage in loops, as loops are also the field where the classical tiling optimization originates. The effective implementation within a compiler framework has clearly shown the considerable distance between a theoretical model and a practically functional compiler pass. Not only is it very difficult to extract the required input information for the construction of a *memory-use graph*, but it has also been impossible up to now to generate efficient code that corresponds to the chosen tiling solution. These issues have severely reduced experimental results and the observed performance degradation cannot convincingly be attributed to either the tiling or the far-from-optimal code generation. Alternatively when only the theoretical values of the amount of spill code is compared between tiling and non-tiled code, it appears that tiling does result in a significant improvement. This leaves the use of *generalized tiling* in the case of loops with a mixed evaluation.

The second field in which *generalized tiling* was applied is cache optimization for dataflow languages. In this context, our tiling can be compared to the existing optimization of *execution scaling* but instead of having the same scaling factor for all dataflow actors we scale dynamically with different factors for different actors. The evaluation was done within in the StreamIt framework, which had also been the target for the state-of-the-art to which we compared our technique. Results in this case have been mainly positive, as significant decreases in the amount of cache-misses have been estimated. Furthermore, these experiments provided insight on the differences between the three proposed tiling heuristics and a comparison with the *constrained programming* solver: one heuristic obtained performance improvements compared to the state-of-the-art and achieved similar performances compared to the optimal

solutions in a number of benchmarks. On the downside, these evaluations were performed with a cache simulator and the implementation itself was not done through a fully functional framework. This last point has been partially addressed through preliminary work on an industrial grade implementation of *generalized dataflow tiling* for the ΣC language.

As a final part of our work, it appears hat the memory and communication cost models from which we started could be used beyond the case of *generalized tiling*. We show this with performance debugging through the study of execution traces. In the context of our study, these traces contained information about data reuses between load and store instructions. By reusing some of the assumptions made for the *generalized tiling* optimization problem, we focused on partitioning the trace graphs. A partition corresponds to a tile without having the regularity constraints associated with code generation constraints. This meant that the partitioning had to be convex, a criteria for which very few algorithms could be found. At the same time, the concept of convex partitioning is strongly associated with that of reachability within a directed graph. We propose novel heuristics for both problems. In the case of the reachability queries, we compared our implementation based on vertex labeling to another state-of-the-art algorithm and observed important performance improvements. Performance evaluation for the proposed convex partitioning heuristics suffered from the lack of comparable existing algorithms, but it did showcase better results with our new heuristics compared to the sole existing greedy approach.

This concludes the summary our contributions but it is not a conclusion for the two concepts of the *memory-use graph* and *generalized tiling*.

Future directions

From the start of this project it became clear that this work would merely be a first step towards the concept of *generalized tiling*. The *memory-use graph* is based on a deep modification of the way the code optimization problem is formulated. It meant it would be difficult to reuse existing results. The *memory-use graph* itself had to be defined correctly as well as the associated memory and communication cost model.

As the first set of experiments have lead to mixed to positive results, it seems worth the effort to dig further into the fine-tuning of the model. A first point of interest is the refinement of the way in which the tiling model accounts for spill because not all data marked for spill by our model will really be spilled; this is a direct consequence of the conservative character of the memory usage model. A second point is the inclusion of

an alternative scheduling within a tile: in the current model tiles are scheduled "line-by-line" but it may be more advantageous to schedule the content of some tiles on a "column-by-column" basis which could lead to an infinite tile width and even further reduced spill code. As a third point, whether the model is extended or not, it is of great importance to further improve and create new tiling heuristics to remove some of the caveats that remain in the proposed ones.

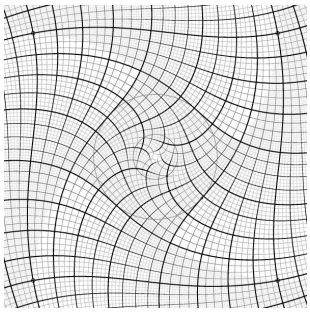
Beyond the memory and communication cost model, the *generalized tiling* itself could still be vastly improved. One of the major topics that has not yet been discussed is that of parallelism. The new organization of the tiles is not only profitable for the sake of data locality. It also opens whole new prospects for the automatic parallelization of the underlying computations. The existing body of work on automatic parallel execution includes elements such as DOACROSS loop constructs and *decoupled software pipelining* (DSWP) [Ran+04] but also a great part of the literature on dataflow programming in general.

Considering the DOACROSS and the DSWP techniques, it should be noted that both are compatible with *generalized tiling* but that they produce two radically different situations: the DSWP technique would allocate rows of tiles to resources, with different rows potentially running on different resources, whereas the DOACROSS technique would instead allocate columns of tiles to different resources. In other words, DSWP cuts the content of iterations in pieces and allocates the different pieces to separate resources whereas DOACROSS groups successive iterations together and allocates them to the same resource. Results presented by Boulet et al. [Bou+99] suggest that the DOACROSS method would allow for a considerably better load-balancing between resources.

On the side of *generalized register tiling* the topic of parallelism is partially represented by the integration of *generalized tiling* with vectorization. A first potential solution would be to apply vectorization first before tiling the code. However the representation of vectorized instructions in the *memory-use graph* is far from trivial as they exceed the single-iteration scope that has been the basis up to now. It is also not easy to apply vectorization after tiling because the tiling rescheduling makes it harder to detect the vectorizable code and it may even limit the potential for vectorization itself. In this regard the recent work by Zhou and Xue [ZX16] on the use of SIMD instructions in loops is very interesting as it coincides on some points with our code representation. Beyond the tiling model, the implementation would benefit greatly from additional work as the current state is far from optimal and has proved itself to be the source of many ongoing problems.

For *generalized dataflow tiling* the challenge is even greater as no fully functional framework exists today. Even though we have been able to create a model for Network-on-Chip communications, it has not yet been integrated into the tiling model to account both for transfer latencies and induced buffer sizes. The interaction between the hardware flows of the Network-on-Chip and the software that performs the send and receive commands is yet another source of uncertainty which needs to be taken into account. An integration of all these elements would be far from trivial because of the numerous direct and indirect parameters that may influence the validity and quality of a given solution. One direction, however, that could be of interest in this regard, would be to model the software / hardware interactions as yet another network buffer behind the real *direct memory access* interface of each computational cluster. This could even potentially be extended to also include the data buffers provisioned by the dataflow actors running on the cluster itself. The result would be a single and unique model for all communications between dataflow actors on separate clusters of the manycore architecture.

The graph-related topics detailed in Chapter 6 present yet another set of challenges that call for further research. Whereas the proposed approach to reachability queries does not leave much questions open, the opposite can be said of convex partitioning. The hierarchical partition schemes that are produced by our two heuristics contain a great amount of information about the structure of the partitioned graph. However the partition trees are rarely balanced and frequently contain both deep and shallow leaf partitions. One possible approach, for further improving the quality of the hierarchical partition schemes, is to use a second set of heuristics to balance the corresponding tree. Such heuristics may be less complex than can be initially expected as the structure of the tree itself contains an important part of the convexity information obtained through the initial hierarchical partitioning.



Annexes

The ancient literature is quite clear, though little studied I fear. It gathers dust rather than readers.

R. Jordan – *Crossroads of Twilight*

Les figures, tableaux et algorithmes auxquels est fait allusion dans les résumés sont ceux de la version anglaise. Les listes contenant ces éléments et les pages ou ils sont représéntés se trouvent à la fin du manuscrit aux pages 159, 161 et 163.

A Résumé - Introduction

Les évolutions des architectures matérielles des ordinateurs a depuis longtemps été un moteur pour la recherche en optimisation de code, les compilateurs s'adaptant aux nouvelles possibilités offertes par chaque génération de processeurs. Le sujet des transferts mémoire a pris une place de premier rang avec le constat que toutes les architectures modernes reposent sur une organisation mémoire en hiérarchie avec de petites mémoires proches des cœurs et des mémoires de taille croissante de plus en plus lointaines et partagés entre cœurs. La diminution des transferts entre ces mémoires a déjà été une cible privilégiée par le passé.

Un deuxième constat est que le rapport entre la puissance de calcul disponible au sein d'un cœur et la bande-passante qui relie ce cœur aux différents niveaux de la hiérarchie mémoire ne cesse de croître. Ce phénomène avait déjà été remarqué dans les années '90 et avait alors été désigné par le terme de « *Memory-Wall* » [WM95 ; SPN96 ; McK04 ; SMCCL11]. En conséquence de plus en plus de programmes ne sont plus limités dans leurs performances par la puissance de calcul mais par la bande-passante entre les différents niveaux de la mémoire.

Plusieurs solutions peuvent être envisagées à cette limitation : l'agrandissement des interfaces mémoire [JED15 ; Hyb15], de nouvelles architectures [Dlu+14] ou l'amélioration du rendement à bande-passante constante par l'optimisation de code. Les deux premières pistes, quoique prometteuses, ne permettront toutefois pas d'abolir le « *Memory-Wall* », seulement de le repousser [Rad+15]. Pour cette raison il semble primordial de continuer à investir dans la recherche liée à l'optimisation de l'utilisation de la bande-passante existante.

Dans ce sens la technique du pavage de boucles, une méthode d'optimisation apparue dans les années '80 [Pei86 ; IT88 ; Wol89] apparaît comme une piste prometteuse. Elle repose sur la division des itérations d'une boucle en « tuiles » régulières dont le contenu sera exécuté en bloc. Dans les faits cela revient à changer l'ordre dans lequel des boucles imbriquées parcourent l'ensemble de leurs itérations. Le calcul du format et de la taile optimaux des tuiles fait l'objet d'une litérature abondante depuis l'apparition du pavage sans que la technique elle-même n'ait fondamentalement évolué.

En revenant aux évolutions architecturales, la multiplication des cœurs a donné naissance au cours de la dernière décennie à la première génération de processeurs dits « manycœurs ». Celles-ci se distinguent des « multicœurs », aujourd'hui omniprésents, par le fait que les communications entre les différents éléments du processeur passent désormais par un *réseau-sur-puce*, un « Network-on-Chip », qui porte un certain nombre de ressemblances avec les réseaux des super-ordinateurs à architecture distribuée. Ce réseau est alors le support des transferts mémoire évoqués précédemment et a de ce fait une influence non-négligeable sur ceux-ci. Des exemples de ces processeurs « manycœurs » sont la famille des TILE-Gx de Tilera (aujourd'hui Mellanox)[5], la famille des MPPA de Kalray[6] et les Xeon Phi d'Intel[7].

Si l'on observe le nouveau degré de parallélisme offert par ces processeurs « manycœurs » avec en tête la technique du pavage de boucles il apparaît que cette dernière aurait des applications potentielles dans des situations bien plus diversifiées : les tuiles ne permettent non seulement d'optimiser les transferts mémoire à l'échelle d'un cœur mais aussi les transferts entre les cœurs.

Dans cette optique un premier pas est de sortir le modèle à la base du pavage de boucles de son cadre restreint. Ceci constitue le sujet du Chapitre 1 (*i.e* B). Pour l'intégration d'un tel modèle plus généraliste dans le cadre des processeurs « manycœurs » nous modélisons de la même façon les communications sur leur *réseau-sur-puce* dans le Chapitre 2 (*i.e* C). Ces éléments nous permettrons de formaliser dans le Chapitre 3 (*i.e* D) le problème d'optimisation que nous devons résoudre pour notre *pavage généralisé*. Le Chapitre 4 (*i.e* E) sera l'occasion de montrer l'application du *pavage généralisé* au cas classique des boucles alors que dans le Chapitre 5 (*i.e* F) nous présentons une application alternative aux langages « dataflow ». Enfin le Chapitre 6 (*i.e* G) dépasse le cadre du *pavage généralisé* pour montrer que les modèles sous-jacents trouvent des utilisations dans le domaine de l'analyse de traces d'exécution, toujours dans l'optique de réduire les transferts mémoire.

B Résumé - Chapitre 1

Coût Mémoire & Intensité Opérationelle

Un morceau de code donné requiert une certaine quantité de mémoire pour stocker l'ensemble des données dont il a besoin pour s'exécuter. L'accès à ces données

[5] http://www.mellanox.com/page/multi_core_overview
[6] http://www.kalray.eu
[7] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

peut à son tour engendrer des transferts entre les niveaux de la hiérarchie mémoire du processeur. Le nombre d'instructions que compte le morceau de code est à mettre en rapport avec le nombre de transferts que celui-ci engendre. Il s'agit là d'une caractéristique du code qu'on nomme *intensité opérationelle*. Celle-ci est définie par rapport à une taille de mémoire donné puisqu'une mémoire plus grande, pouvant contenir plus de données en même temps, engendrera moins de transferts avec les niveaux de mémoire plus éloigné du cœur sur lequel s'exécute le code.

Pour représenter l'utilisation que fait un code des données dans l'optique d'une optimisation généraliste nous devons respecter trois formes d'indépendance :

Architecture La représentation ne doit pas dépendre de l'architecture qui est ciblé par le code à optimiser.

Granularité Nous devons pouvoir représenter des codes à des échelles différentes : des instructions individuelles d'une boucle, aux tâches complexes d'un programme « dataflow ».

Données La taille ne doit pas varier avec la taille des données sur lequel s'exécute le code itératif, comme c'est le cas par exemple pour des traces d'exécution. Seul la taille du code représenté et la granularité à laquelle cela est fait doit influencer la taille de cette représentation.

Afin de respecter ces trois principes nous proposons une représentation qui s'inspire de celle utilisé dans le cadre des langages « dataflow » statiques [LM87] et cyclostatiques [Bil+95] et que nous nommerons le *graphe d'utilisation mémoire* d'un code. Les éléments qui y sont représentés (instructions, fonctions, tâches, ...) forment les nœuds d'un graphe et la réutilisation de données entre les éléments sont les arêtes reliant ces nœuds. Les arêtes du graphe sont annotées avec la taille des données produites par le code du nœud source et qui sont réutilisées par le code du nœud destination.

Si une donnée est réutilisée par le même nœud dans une itération ultérieur elle fait partie d'un état interne du nœud en question comme le montre la Figure 1.7. La taille des données véhiculés au sein d'un tel état interne constitue une première annotation des nœuds du *graphe d'utilisation mémoire*. Elle est associée avec une contrainte forte imposée sur le graphe : les arêtes ne doivent représenter que des réutilisations de données au sein d'une même itération du code.

Dans le cas d'une réutilisation de données entre itérations et entre deux nœuds différents il faut opérer une transformation du graphe, comme par exemple pour la

réutilisation d'un résultat à l'itération i d'une boucle dans l'itération $i + 1$. Cette transformation découpe la réutilisation dans un état interne du nœud source passant la donnée vers l'itération cible et une arête reliant le nœud source au nœud destination au sein de cette dernière itération. Cette transformation est illustré avec la Figure 1.9.

Une deuxième et dernière annotation associée à chaque nœud du *graphe d'utilisation mémoire* est ce que l'on appelle le *besoin mémoire interne*. Dans la mesure où un nœud peut représenter un ensemble plus ou moins grand de code, allant de l'instruction machine unique à une tâche de calcul complexe, il se peut qu'elle produise en interne des variables intermédiaires pour s'exécuter. Ainsi il se peut qu'elle ait un besoin mémoire qui excède la somme de son état interne et de ses données entrantes ou sortantes. Cette différence correspond au *besoin mémoire interne*, une valeur qui peut, bien évidemment, être nulle.

Une définition mathématique formelle du *graphe d'utilisation mémoire* peut être trouvée au sein de la Section 1.3.4.

En se recentrant sur le but premier de chapitre, à savoir le fait de calculer efficacement l'utilisation mémoire d'un morceau de code donnée, il s'avère que notre représentation permet justement cela. Comme expliqué précédemment l'utilisation mémoire équivaut à la valeur du *max-live* et avec le graphe à notre disposition il convient seulement de considérer l'ordonnancement des nœuds afin d'en déterminer l'utilisation mémoire. En effet, tout comme dans le cas de l'ordonnancement d'instructions, l'ordonnancement des nœuds peut influer sur l'utilisation mémoire comme le montre la Figure 1.12.

Afin d'illustrer correctement le calcul de l'utilisation mémoire à partir d'un *graphe d'utilisation mémoire* et d'un ordonnancement donnés nous prenons l'exemple de la Figure 1.13. Cette figure montre un graphe à cinq nœuds déroulé sur trois itérations. Les valeurs des annotations (poids des réutilisations, de l'état interne et du *besoin mémoire interne*) précédemment décrits sont indiqués. La partie du code que nous voulons évaluer est constitué des nœuds affichés en traits plein, les autres éléments sont laissés de côté. Il est possible alors de considérer deux ordonnancements, à savoir le *ligne-par-ligne* et le *colonne-par-colonne*, qui donneront des utilisations mémoire distincts. La Section 1.4.3 montre les valeurs qui sont obtenues ; d'abord en ne considérant que les transferts entre nœuds, ensuite en incluant le poids des états internes et enfin en rajoutant les *besoins mémoire internes*.

Le principe du calcul est que l'utilisation mémoire d'un bout de code donné est la valeur du *max-live* prise en différents points de l'ordonnancement considéré :

Sur un nœud de l'ordonnancement Le *max-live* équivaut à la somme de quatre éléments : le poids de l'état interne, le poids du besoin mémoire interne, le maximum des poids des données entrantes ou des données sortantes et le poids total des arêtes qui relient au sein du bout de code évalué un nœud situé avant le point courant dans l'ordonnancement à un nœud après le point courant.

Entre deux nœuds de l'ordonnancement Le *max-live* est constitué de la somme de deux éléments, à savoir : le poids total des arêtes reliant un nœud situé avant le point courant à un nœud après le point courant et le poids des états internes des nœuds avant le point courant dont il existe une autre itération après le point courant.

Pour l'évaluation du nombre de transferts mémoire d'un code donné nous réutilisons le calcul de l'utilisation mémoire. En effet le nombre de transferts est relative à la taille de la mémoire ciblée (registres, cache, ...). Il s'agit alors de découper le code en blocs pour lesquels il existe un ordonnancement tel que chaque bloc pris individuellement à une utilisation mémoire inférieure à la taille de la mémoire cible. Ainsi chaque bloc, ne générant que des réutilisations internes peut s'exécuter atomiquement sans transferts vers des mémoires plus lointaines. Le coût en transferts mémoire de l'ensemble est alors égale à la somme des arêtes entrant dans chaque bloc.

C Résumé - Chapitre 2

Communications sur une Plateforme Manycœurs

Le deuxième élément pour lequel nous avons besoin d'un modèle est la communication au sein d'une architecture « manycœurs ». La conception et l'architecture des *réseaux-sur-puce* de ces processeurs, *i.e* le « Network-on-Chip », constituent un domaine à elles seules dans la recherche académique. De multiples variations existent selon la topologie du réseau, la façon dont les données sont acheminées jusqu'à leur destination et bien d'autres paramètres. Nous ne ferons pas ici de bilan de ses travaux dans la mesure où elles n'apporteraient que peu ou pas d'informations pertinentes à notre étude et le lecteur intéressé pourra toujours se référer à la littérature du domaine. En suivant ce même constat il apparaît inenvisageable de former un modèle unique qui pourrait s'adapter à l'ensemble des architectures et c'est pour cette raison que nous limiterons notre modélisation au cadre de la famille des processeurs MPPA de Kalray[8]. Les travaux décrits ici ont donné lieu à la publication de deux articles [Din+14b ;

Din+14a].

Dans l'ensemble les problématiques auquel peut être confronté un *réseau-sur-puce* sont relativement connues :

Latences inconnues Dans le cas d'une charge importante du réseau les latences observées sur les transferts de données augmentent en général de façon incontrôlables sans pouvoir donner des garanties sur une latence maximale.

Interblocage Dans certains la cas le trafic sur le réseau peut engendrer des situations d'interblocage ou des données s'empêchent mutuellement d'arriver à leurs destinations respectives.

De nombreuses solutions ont été proposés dans la littérature pour le problème des interblocages comme le « XY-routing » [Sei+88] ou les « turn-models » [GN94] alors que pour les latences la façon la plus usité pour les limiter est d'appliquer un protocole de *qualité de service* (QoS). Ce dernier a pour but de limiter les débits des différents flux circulant sur le réseau de façon à contrôler leurs interactions et la croissance des latences qui y est associée. C'est ici qu'intervient le fait de modéliser les transferts sur le réseau interne puisque pour calculer les débits à accorder à chaque flux il faut d'abord savoir quel situation résulte d'une configuration donnée.

Le *réseau-sur-puce* des processeurs MPPA est composé des éléments suivants : 16 groupes de cœurs de calcul (Figure 2.2) et 4 groupes de cœurs responsables des entrées & sorties, un routeur (Figure 2.5) combiné à une interface réseau (Figure 2.3) par groupe de cœurs et un réseau à la topologie d'un tore 2-dimensionnel partiel complété de liens traversants (Figure 2.4). L'ensemble de l'architecture a le désavantage de ne pas se prêter aux diverses techniques qui garantissent l'absence d'interblocages. Néanmoins il a été établi formellement qu'une condition suffisante pour éviter ceux-ci est le fait de ne pas saturer les mémoires tampon des routeurs du réseau. L'élément architectural indispensable à une régulation empêchant de tels débordements se trouve aux interfaces d'émission des groupes de cœurs : un limiteur de débit permettant de réguler les caractéristiques de chaque flux dès son émission. D'un point de vue plus généraliste cette méthodologie de régulation s'oppose à une régulation faite par les routeurs à partir d'informations de congestion remontant chaque flux en sens inverse.

Pour modéliser le trafic réseau sur les processeurs MPPA nous avons retenu le calcul réseau de type (σ,ρ) [Cru91a ; Cru91b ; Zha95]. Son avantage par rapport à des modèles plus fluides comme le calcul réseau $(\min,\text{plus})$ est qu'il permet de mieux rendre

[8] La thèse à l'origine de ce manuscrit a été financé par le dispositif CIFRE. L'auteur a été, durant l'ensemble de ses travaux, un employé à temps plein de Kalray.

compte des effets de la packetisation, à savoir le fait que les données ne sont pas transmises comme un flux continu mais comme des émissions discontinues de sous-ensembles de données. Dans la pratique le calcul réseau (σ,ρ) permet de formuler des limites sur le remplissage des mémoires tampon des routeurs à partir des caractéristiques à l'émission de chaque flux. Il devient alors possible en retournant le sens des équations d'obtenir des contraintes sur l'émission de chaque flux en donnant des contraintes sur le remplissage des mémoires tampons. Les détails mathématiques du modèle ne sont pas reproduits ici mais elles sont présentées dans la Section 2.5.2 du manuscrit.

L'application du modèle aux processeurs MPPA en pratique a mis en évidence le fait que les limiteurs de débit des interfaces réseau avait un paramétrage difficile à utiliser et qui n'était de plus pas assez fin dans sa granularité de configuration. Dans cette optique un nouveau limiteur de débit a été conçu dont un potentiel schéma d'implémentation est suggéré en Figure 2.6. Ce nouveau limiteur de débit a été intégré au sein de la deuxième génération du MPPA. Son fonctionnement a répondu à l'ensemble des attentes et a été le sujet d'un dépôt de brevet en 2015.

D Résumé - Chapitre 3

Pavage Généralisé

Afin d'expliquer au mieux le concept d'un *pavage généralisé* il faut d'abord mettre en avant sa comparaison avec le pavage « classique ». C'est dans la représentation du code que réside la différence principale. Dans le premier cas le code est traditionnellement représente de par son domaine d'itération : une grille sur $\mathbb{Z}^n$ ou les dimensions représentent les différents niveaux d'un nid de boucles, chaque coordonnée distincte de la grille représentant une itération complète de la boucle interne. Les tuiles sont alors toutes de taille identique et permettent de paver la grille en question. L'élément atomique de cette représentation étant l'itération elle-même il n'est pas possible de distinguer les instructions constituant cette itération. Pour le *pavage généralisé* nous utilisons au contraire le *graphe d'utilisation mémoire* du code de l'itération interne ce qui nous conduit au fait de qualifier le pavage classique de régulier alors que le *pavage généralisé* est semi-régulier. La différence de représentation est illustrée dans les Figures 3.1.

Une première conséquence de la semi-régularité est le fait qu'avec une représentation élargie aux instructions d'une itération, l'ordonnancement de ceux-ci intervient comme une première étape dans le problème d'optimisation. Un deuxième

point concerne la forme des tuiles : celles-ci ne sont pas nécessairement toutes de la même taille. Là où le pavage classique, pour des raisons de génération de code, se doit d'avoir des tuiles de même taille sur chaque dimension de l'espace d'itération ceci ne concerne plus qu'une des deux dimensions dans le cas du *pavage généralisé*. En effet la dimensions des instructions d'une itération ne nécessite pas des tuiles de même taille dans la mesure où chaque élément sur cette dimension représente un bout de code différent. Un exemple de tuiles avec des formes semi-régulières est montré dans la Figure 3.2.

En pratique, afin d'appliquer le pavage donné à un morceau de code itératif représenté par un *graphe d'utilisation mémoire* il faut suivre une série de transformations qui est représenté dans la Figure 3.3.

Pour formaliser les tuiles et leurs formes nous imposons quelques contraintes sur leur organisation. Ainsi de par leur construction les tuiles sont constitués de groupes de nœuds consécutifs dans l'ordonnancement choisi du *graphe d'utilisation mémoire*. Le nombre de nœuds consécutifs est référencé comme étant la *hauteur* de la tuile. Ces nœuds sont alors dupliqués sur un nombre d'itérations ce qui constitue la *largeur* de la tuile. L'ordre de parcours des différentes instances des nœuds est restreinte à un ordonnancement en ligne-par-ligne.

Cette formalisation des tuiles permet dans un second temps de formaliser le problème d'optimisation lui-même. Sans reprendre l'ensemble des contraintes, qui interviennent pour assurer qu'une solution de pavage soit bien formée il est possible de nommer le fait que l'ensemble des instances des nœuds d'une tuile doivent pouvoir s'exécuter sans que les réutilisations de données internes nécessitent des transferts vers ou depuis une mémoire lointaine selon le modèle décrit en Chapitre 1 (voir B). Les contraintes détaillés sont disponible en Section 3.3.2. L'optimisation a pour but de minimiser le nombre de transferts mémoires moyen par itération du code et pour la résolution deux stratégies sont possibles.

La première tente de produire une solution optimale en utilisant des méthodes comme la *Programmation par Contraintes*. Celle-ci a le désavantage de nécessiter un temps de recherche important même pour un *graphe d'utilisation mémoire* de taille réduite. Il est alors possible d'imposer un temps limite au bout duquel la recherche retourne la meilleure solution trouvé jusqu'à là et qui est potentiellement sous-optimale.

Une deuxième approche fait appel à des heuristiques. Celle-ci se prêtent beaucoup mieux à une intégration dans un environnement comme un compilateur puisque leur temps de recherche est réduit. Les solutions, sans être optimales, peuvent être de très bonne qualité si l'heuristique utilisée est correctement choisie. Nous proposons

ainsi trois algorithmes différents pour le *pavagé généralisé*.

Dans la recherche d'une solution de pavage il faut garder en tête qu'il s'agit à la fois de choisir un ordonnancement valide des nœuds du graphe et de choisir les tailles des tuiles. Ces deux facettes du problème peuvent être résolues séparément ou conjointement. Pour l'ordonnancement l'idée principale est de raccourcir le plus possible d'abord les arêtes les plus lourdes en rapprochant leurs sources et leurs destinations. Pour le pavage plusieurs solutions s'offrent à nous :

- Il est possible, de la même façon que pour l'ordonnancement, d'inclure le plus d'arêtes possible au sein d'une même tuile afin de diminuer la quantité de transferts mémoire en traitant avec priorité les arêtes les plus lourdes. Puisque ceci aura tendance à créer des tuiles hautes au détriment de leur largeur il est possible d'envisager une variante qui impose de garder une certaine largeur minimale ou de ne pas trop « étirer » les tuiles.
- Une deuxième possibilité est de faire usage d'un algorithme glouton. En partant d'un ordonnancement préalablement choisi il est possible de choisir des tuiles au coût optimal localement en progressant le long de l'ordonnancement.

E Résumé - Chapitre 4

Pavage Généralisé pour Registres

Un premier domaine d'application dans lequel nous montrerons une implémentation du *pavage généralisé* est celui du pavage des boucles et qui vise à optimiser l'utilisation qui est faite des registres dans le cœur d'un processeur.

Il est tout d'abord important de comprendre les différences entre le pavage pour registres existant, d'autres optimisations de boucles et notre *pavage généralisé pour registres*. Pour cela la Figure 4.1 représente les domaines d'itérations et les instructions d'une boucle (Figure 4.1a) et y superpose les types de réutilisations qui sont ciblées pour optimiser l'utilisation des registres. On peut noter par exemple que le pavage « classique » (Figure 4.1b) cibles une réutilisations des données entre les différentes itérations sans se préoccuper des réutilisations au sein d'une même itération. Réciproquement la promotion scalaire (Figure 4.1c) et l'allocation de registres (Figure 4.1d) se préoccupent principalement des réutilisations locales à une itération et ne vont que partiellement au-delà.

Le *pavage généralisé* (Figure 4.1e) se démarque ici en considérant en une seule fois les réutilisations inter et intra-itération et ce de façon optimale. Bien que le mo-

dèle et l'optimisation de *pavage généralisé* tels qu'ils sont présentés dans ce rapport se limitent aux réutilisations portées par la boucle interne des nids de boucles il n'y a à priori pas d'obstacles théoriques majeurs au fait d'envisager leur extension à l'ensemble des itérations. Les problématiques les plus importantes porteraient sur la croissance exponentielle du domaine des pavages à envisager par le solveur et les heuristiques présentés en Chapitre 3 (voir D), ainsi que sur la génération de code qui en suit.

Pour l'implémentation du *pavage généralisé* dans un compilateur il faut d'abord se préoccuper de la génération du *graphe d'utilisation mémoire*. Les moyens disponibles pour le générer et le graphe qui en résulte dépendent notamment du moment auquel est appliqué l'optimisation dans la chaîne de compilation. Dans la mesure ou le graphe est sensé refléter les réutilisations de données ceux-ci doivent pouvoir être analysé facilement. Toutefois la mémoire cible étant les registres du cœur, les communications mémoire sont effectuées de façon explicite dans le code du programme et peuvent donc intervenir elles-mêmes au sein du *graphe d'utilisation mémoire*. Une distinction importante ici est encore une fois le moment auquel le code est analysé.

Si le code est représenté sous forme *Static Single Assignment* (SSA) [Cyt+91] alors les seuls accès mémoire qui appartiennent au code sont en principe ceux associés aux entrées-sorties du programme. Ces opérations de communications sont donc obligatoires pour le bon fonctionnement ce ce dernier et aucune optimisation ou transformation ne saurait les éliminer. Disposer du code sous une telle forme permet donc d'analyser aisément les réutilisations.

Deux tentatives d'implémentation ont été faites. La première s'est placé dans le cadre du compilateur GCC[9], de la représentation intermédiaire TIREX et de l'outil d'optimisation externe LAO. La représentation TIREX (Textual Intermediate Representation for compiler EXchange) [Pie12] a été spécifiquement conçue pour permettre des échanges inter-compilateurs ainsi que l'utilisation d'outils optimisations externes. Le Linear Assembly Optimizer (LAO) [Din+00] est une outil qui réalise des optimisations spécifiques à certaines architectures, notamment celles de type *Very Large Instruction Word* (VLIW) comme le K1 des processeurs MPPA. Schématiquement l'utilisation de TIREX et LAO permet de combiner de multiples compilateurs avec de multiples architectures cibles tel que le représente la Figure 4.2.

L'utilisation de GCC était obligatoire au départ puisqu'il s'agissait à l'époque du seul compilateur qui supportait efficacement l'architecture K1. Malheureusement un nombre d'obstacles technologiques est vite apparu :

[9] https ://gcc.gnu.org/

- Seul GCC était capable de faire une sélection d'instruction éfficace. Il en découlait que la représentation Tirex devait être générée après cette étape là.

- L'allocateur de registres de LAO montrait des comportements anormaux sur l'architecture K1 et il aurait fallu une quantité de travail conséquente pour le remettre dans un état fonctionnel. Ceci nécessitait de retarder encore plus la génération du TIREX et avait l'inconvénient de produire une représentation où les registres étaient déjà alloués.

- Les informations des dépendances de données générées par GCC ne sont plus maintenus au moment de l'allocation de registres. Ces informations devaient par conséquent être régénérées par LAO.

Malgré un travail considérable il n'a jamais été possible d'activer effectivement le pavage par l'intermédiaire de LAO. Après l'apparition de nouvelles difficultés il a été décide de changer de la chaîne GCC-TIREX-LAO vers une solution centrée autour du compilateur LLVM [LA04]. Celui-ci avait depuis peu un prototype de support de l'architecture K1.

L'implémentation dans LLVM s'est fait selon le schéma de la Figure 4.3 en fournissant de nouvelles passes d'optimisation à l'outil `opt` qui sont à leur tour pilotées par un outil remplaçant l'exécutable `clang` standard de LLVM. La construction du *graphe d'utilisation mémoire* s'appuie sur les analyses déjà existantes dans LLVM comme le *scalar evolution*.

Pour chaque boucle susceptible d'être pavé un graphe est généré et sauvegardé dans un fichier externe. Une solution de pavage est ensuite calculée par un outil externe qui utilise soit la *programmation par contrainte*, soit les heuristiques décrites en Section 3.5 (voir D).

Enfin le pavage choisi est appliqué par une dernière passe d'optimisation qui déroule la boucle ciblée et réordonne les instructions en suivant la forme des tuiles. La transformation ainsi appliquée produit malheureusement un code qui ne respecte pas entièrement le pavage de départ comme le montre la Figure 4.4.

Une évaluation de l'efficacité de cette implémentation a été fait sur un ensembles de codes issus du domaine du traitement du signal et d'autres sources de données. L'implémentation ne générant pas exactement le pavage prévu la taille du code augmente avec le facteur de déroulement. Ceci peut potentiellement avoir un impact négatif sur la vitesse d'exécution d'ensemble du code comme le suggèrent les données du Tableau 4.1 qui retracent l'évolution des temps d'exécution.

F Résumé - Chapitre 5

Pavage Généralisé pour Langages « Dataflow »

Une deuxième d'application du *pavage généralisé* se place dans le cadre des langages « dataflow ». Il s'agit encore une fois ici de se limiter au cas des langages statiques et cyclo-statiques comme par exemple StreamIt [TKA02], Spidle [Con+03] et Σ-C [Gou+11]. Les programmes exprimés dans ces langages ont déjà été le sujet d'un certain nombre de techniques d'optimisation concernant leur utilisation de la mémoire cache. Les plus représentatifs sont la fusion d'acteurs [Ser+05] et l'exécution multiple [Car+03 ; Ser+05 ; WCB01].

L'exécution multiple consiste à changer la granularité avec laquelle un programme « dataflow » lit les données en entrée. Ainsi au lieu de s'exécuter sur chaque entrée individuelle une fois, les différents acteurs sont exécutés n fois pour traiter n entrées sous forme de tâche unique. Tel qu'elle a été utilisé jusqu'à aujourd'hui l'ensemble des acteurs voyaient leur nombre d'instances multipliées par la même quantité, n ici. En pratique le *pavage généralisé* revient à appliquer le même principe mais avec potentiellement un facteur multiplicatif différent pour chaque acteur ou groupe d'acteurs.

Afin d'évaluer l'effet d'une telle modification dynamique du facteur multiplicatif nous l'avons en premier lieu utilisé un simulateur de mémoire cache combiné à des programmes exprimés dans le langage StreamIt. Le critère d'évaluation a encore une fois été le nombre de transferts mémoire, ici le nombre de défauts de cache, produit par une exécution. Les résultats sont regroupés au sein de la Figure 5.1. L'analyse de ces résultats met en évidence que dans une grande majorité des situations l'utilisation du *pavage généralisé* permet d'améliorer sensiblement le nombre de défauts de cache par rapport aux solutions existantes. Notamment il apparaît que l'heuristique *Tile-aware / Conservative*, soit la dernière décrite dans le Chapitre 3 (voir D), est de loin la plus efficace alors que celle-ci avait des performances légèrement moindre dans le cas du pavage pour registres précédemment exposé. Le fait le plus notable est que dans un certains cas cette heuristique arrive à quasi-égalité avec l'approche basée sur la *programmation par contraintes* alors que celle-ci donne soit une solution optimale, soit une solution sous-optimale si elle dépasse le temps de recherche maximal qui est alloué au solveur. L'ensemble des résultats permet de mettre en évidence que l'utilisation du *pavage généralisé* possède un potentiel considérable pour améliorer l'utilisation du cache des langages « dataflow » statiques et cyclo-statiques.

Le constat de ce potentiel de réduction du nombre d'opérations IO nous a poussé à étudier l'intégration du *pavage généralisé* dans le cadre du langage ΣC. Même si

cette intégration n'est aujourd'hui pas encore dans un état fonctionnel nous avons pu identifier la méthodologie à appliquer, les endroits de la chaîne de compilation ΣC où il faut intervenir ainsi que certains obstacles qui restent encore à surmonter. Parmi ceux-ci le plus notable reste l'utilisation combinée de notre modèle de pavage et ses solveurs avec la régulation à l'émission des flux sur le *réseau-sur-puce* d'un processeur « manycœurs ».

G Résumé - Chapitre 6

Par-delà la Semi-Régularité

Jusqu'ici l'ensemble des efforts se sont placés dans le cadre de l'analyse statique du code et de son optimisation au moment de la compilation. Nos modèles d'utilisation mémoire et de coût en transferts mémoire ne se limitent pourtant pas par définition à cette situation. Dans le dernier chapitre nous voulons donner un aperçu d'une autre application de ces modèles : celui de l'analyse de traces d'exécution et du débogage de performances.

Les traces d'exécution s'obtiennent par l'instrumentation de code, à savoir l'ajout d'instructions enregistrant certains comportements du code visé. Si cette instrumentation est faite en visant les accès mémoire il est possible d'obtenir un graphe dirigé acyclique retraçant ces accès et les réutilisations des données sous-jacentes. Dans ce sens il apparaît possible, à partir d'un tel graphe de réutilisation, d'utiliser nos modèles pour calculer une limite supérieure à la quantité de transferts mémoire que le code instrumenté nécessite par rapport à une taille de mémoire cible. Pour ce faire il convient de modifier l'ordonnancement des instructions dans la trace d'exécution afin de voir si cela permet de réduire le nombre de transferts requis. Si cette limite supérieure se situe en deçà du nombre de transferts enregistrés par la trace d'exécution cela constitue une indication indirecte du potentiel d'amélioration du code instrumenté : c'est le débogage de performances.

La représentation du graphe des réutilisations extraite d'une trace d'exécution possède de nombreuses similitudes avec le *graphe d'utilisation mémoire* mais elle diffère sur un plan structurel : elle ne possède pas obligatoirement la régularité du *graphe d'utilisation mémoire* et ne peut de ce fait pas être repliée à l'échelle d'une unique itération du code instrumenté pour représenter l'ensemble de l'exécution. Une conséquence indirecte de cet absence de simplification est que la taille des graphes issus des traces d'exécution est fonction du nombre d'itérations qui s'exécutent et varie typiquement de 10^6 à 10^8 nœuds ou plus encore. Ceci impose de trouver de nouvelles

méthodes d'analyse de ces graphes puisque les approches utilisés pour le *pavage généralisé* possèdent des complexités temporelles trop importantes pour ces tailles de données.

Lors de la présentation du modèle d'utilisation mémoire au Chapitre 1 (voir B) nous avions évoqués que chaque bloc de code, *i.e* une tuile dans le cas du *pavage généralisé*, doit pouvoir s'exécuter atomiquement sans transferts mémoires vers ou depuis une mémoire lointaine en considérant la taille de la mémoire ciblée. Ceci implique du point de vue du graphe que l'ensemble des blocs ainsi sélectionnés pour l'évaluation du coût en transferts mémoire constitue ce que l'on appelle un partitionnement convexe. Un tel partitionnement se définit par le fait que le graphe réduit des partitions est lui-même acyclique, ou autrement dit qu'il n'existe pas de chemin repassant dans une partition après en être sorti.

Alors que le partitionnement de graphes est un sujet qui a été étudié de façon approfondie par une quantité importante d'études existantes ce constat ne s'étend pas au cas particulier du partitionnement convexe de graphes dirigés.

Au premiers abords il semblerait qu'une notion fondamentale qui est en jeu dans le partitionnement convexe est celle de la connectivité du graphe : existe-t-il un chemin reliant un nœud u à un autre nœud v. Cette question est elle-même un domaine de recherche dans l'algorithmique des graphes et nous y avons apporté une contribution en élaborant une nouvelle méthode d'indexation des nœuds d'un graphe qui se base sur des parcours en profondeur. Notre méthode permet d'accélérer considérablement la recherche de chemins par rapport à des algorithmes naïfs et se compare aussi favorablement avec d'autres méthodes extraites de l'état de l'art comme l'outil GRAIL [YCZ12], que ce soit pour le temps d'indexation (Figure 6.4) ou pour les temps de recherche (Figures 6.5, 6.6, 6.7 et 6.8).

Nos efforts se sont ensuite concentrés sur le partitionnement convexe lui-même. Un des rares algorithmes existants a été introduit par Fauzia et al. [Fau+13] et a aussi été appliqué dans le cadre de l'analyse de traces d'exécution. L'approche en question est un algorithme glouton que nous avons pris comme point de référence pour l'évaluation de deux nouvelles méthodes que nous présentons ci-dessous.

Alors que l'algorithme glouton construit un partitionnement convexe en k partitions qui est spécifique à une taille de mémoire cible, nos deux nouvelles méthodes construisent des partitionnements convexes hiérarchiques. Ceci implique que chaque partition subit elle-même un partitionnement récursif jusqu'au point où une partition correspond à un unique nœud du graphe. Une telle structure peut être représenté par un arbre de partitions où les fils d'une partition donnée forment un partitionnement

de celle-ci. Des exemples d'une k-partition et d'un partitionnement hiérarchique sont données en Figure 6.2.

Notre première méthode, dite de *convexification*, utilise une méthode de partitionnement non-convexe quelconque et modifie les partitions obtenues pour les rendre convexes. Le pseudo-code correspondant à cette méthode est donnée dans l'Algorithme 6.1. La deuxième méthode repose elle sur la structure du graphe : pour chaque nœud nous calculons les longueurs du plus long chemin venant d'une source et du plus long chemin menant à un puits. Les nœuds sont ensuite repartis entre les partitions selon que l'une ou l'autre des ces longueurs est la plus grande. La correction de cette méthode vis-à-vis du critère de convexité est le sujet du Théorème 6.4.

Le dernier élément qui reste à préciser est l'évaluation du coût en transferts mémoire d'un partitionnement donné, ou comment appliquer le modèle déjà utilisé pour le *pavage généralisé*. Cette évaluation se fait de façon récursive sur un partitionnement convexe hiérarchique : l'utilisation mémoire de chaque partition est calculée en fonction de celles de ses fils et des réutilisations ayant lieu entre ceux-ci. Ceci permet de décider si ces mêmes réutilisations conduisent, oui ou non, à des transferts mémoire en fonction de la taille de la mémoire cible et contribuent ainsi au coût global du code.

Une étude faite à partir d'un groupe d'une trentaine de traces d'exécution, dont seize sont effectivement représentées dans les Figures 6.9 et 6.10, calcule les coûts en transferts mémoire avec nos deux méthodes ainsi que l'algorithme glouton de Fauzia et al., et ce pour des tailles mémoires de 256 à 4096 octets. Les résultats montrent que la méthode de *convexification* permet dans au moins la moitié des cas d'obtenir un coût entrée-sortie inférieur à l'algorithme glouton. Il s'avère aussi que la méthode basé sur les longueurs de chemins produit des résultats sensiblement moins bons que la méthode de *convexification*. Ceci peut s'expliquer par le fait que les arbres de partitions obtenus avec cette dernière sont bien mieux équilibrés comme le montre le Tableau 6.1.

H Résumé - Conclusion

Tout au long du travail présenté ici nous nous sommes efforcés de construire une approche cohérente de la localité des données sur des architectures « manycœurs ».

Nos efforts ont commencé par la mise-au-point d'un modèle permettant d'évaluer à la fois l'utilisation mémoire et le coût en transferts mémoire d'un code donnée. Ce modèle s'est lui-même construit sur la base d'une représentation du code inspiré

des langages « dataflow » : le *graphe d'utilisation mémoire*. Cette représentation fait abstraction du niveau de granularité à laquelle est analysé le code mais aussi de l'architecture ciblée. Dans un deuxième temps nous avons appliqué les principes du calcul réseau (σ,ρ) au cas spécifique des processeurs MPPA afin d'obtenir un deuxième outil théorique, à savoir la modélisation des communications transitant sur le *réseau-sur-puce* de ces processeurs.

Le modèle d'utilisation mémoire et de coût en transferts mémoire a ensuite trouvé une première application dans le cadre du *pavage généralisé*, une nouvelle optimisation de code qui s'inspire du pavage de boucles. Cette optimisation utilisé la même représentation de code déjà identifiée dans le premier chapitre. Une formalisation mathématique du problème combinatoire sous-jacent au *pavage généralisé* a ensuite permis d'identifier différents techniques pour calculer des paramètres pour l'optimisation en question. Suite à la présentation généraliste du *pavage généralisé* deux applications en ont été décrites.

La première se plaçait dans le cadre des optimisation de boucles. En dehors de résultats peu concluants ce cas d'utilisation a surtout permis d'identifier quelques problématiques liées à l'implémentation concrète du *pavage généralisé* au sein de différents compilateurs.

La deuxième application s'est placée elle dans le cadre des langages « dataflow ». Celle-ci a fait d'abord l'objet d'un prototypage avec le langage StreamIt. Les résultats expérimentaux ont cette fois-ci permis de montrer des améliorations considérables en comparaison avec une autre méthode identifié comme étant l'état de l'art en la matière. De même ces résultats ont permis de constater les différences entre les multiples techniques préalablement identifiés pour résoudre le problème d'optimisation du *pavage généralisé*. Une véritable implémentation pour un langage « dataflow » sur une plateforme « manycœurs » n'a pu être terminée mais celle-ci devrait *in fine* se baser à la fois sur le *pavage généralisé* tel que celui-ci a été défini au Chapitre 3 (voir D) et sur le modèle de communications du Chapitre 2 (voir C).

Un dernier chapitre visait à montrer la diversité des utilisations du modèle d'utilisation mémoire et de coût en transferts mémoire. Nous nous sommes ainsi intéressé au débogage de performances à base de traces d'exécution. Les traces permettent d'établir des graphes de réutilisations similaires au *graphe d'utilisation mémoire*. Pour le traitement de ces nouveaux graphes nous avons présenté plusieurs contributions, comme par exemple une nouvelle méthode d'indexation des nœuds pour accélérer les requêtes de connectivité ainsi que des heuristiques de partitionnement convexe. Ces partitions permettent une nouvelle application de nos modèles afin de calculer

rétrospectivement des limites supérieures aux nombre de transferts mémoire qu'un code nécessite en explorant les possibilité de ré-ordonnancement des instructions.

L'ensemble des travaux décrits ne constituent bien évidemment pas un environnement clos et terminé. Nos modèles s'inspirent d'éléments pré-existants et visent à fournir une base pour des travaux futurs sur le *pavage généralisé*.

Dans les points qui nécessitent d'être explorés plus en profondeur il y a notamment le fait que le modèle d'utilisation mémoire actuel rend imparfaitement compte du véritable besoin en transferts mémoire. Toutes les données réutilisées entre deux tuiles du *pavage généralisé* passent automatiquement par une mémoire lointaine alors que ceci ne sera pas forcément le cas en pratique. D'autre part le modèle présuppose un ordonnancement des instructions au sein d'une tuile en ligne-par-ligne alors qu'il serait probablement plus avantageux dans certains cas d'utiliser des ordonnancements alternatifs.

Un autre question qui n'a pas été abordé au cours de notre étude est celle du parallélisme. En dehors de l'objectif de favoriser la localité des données, le découpage en tuile du *pavage généralisé* peut constituer la base d'une parallélisation automatique de l'exécution du code visé. Les travaux concernant les boucles de type DOACROSS ainsi que ceux sur le *decoupled software pipelining* (DSWP) [Ran+04] sont compatibles avec le *pavage généralisé*. Ces techniques reposent chacune sur des hypothèses et une philosophie propres et leurs effets en seront potentiellement d'autant plus différents.

En regardant spécifiquement le *pavage généralisé pour registres* le parallélisme évoque notamment la question de la vectorization. L'intégration de cette technique avec notre pavage est loin d'être simple, à commencer par la question du moment de la vectorization : avant ou après le pavage. Dans le cas du *pavage généralise pour programmes « dataflow »* le travail est beaucoup plus pragmatique dans la mesure où il n'existe à ce jour aucune implémentation fonctionnelle. Si une telle implémentation vise en plus une architecture « manycœurs » elle nécessiterait encore de surmonter un certain nombre de barrières théoriques pour intégrer les communications du réseau-sur-puce au sein du modèle de pavage.

En dernier il reste les travaux liées au graphes de réutilisations issus de traces d'exécution. Alors que notre approche de l'indexation de graphes et de la connectivité ne laissent que peu de questions ouvertes il en est tout autrement pour le partitionnement convexe. Les heuristiques que nous avons présentés créent des arbres de partitions hiérarchiques mais dans certains cas ceux-ci sont fortement déséquilibrés. Puisque notre approche a quand-même montré des améliorations vis-à-vis de l'ap-

proche gloutonne existante il serait judicieux d'étudier les possibilités de rééquilibrage des arbres de partitions afin d'augmenter encore plus l'efficacité de nos techniques de partitionnement convexe.

Bibliography

Author references

[DAR16] **L. Domagala / D. van Amstel / F. Rastello** – "Generalized cache tiling for dataflow programs". *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. ACM. 2016 (Accepted). (cit. on p. 80).

[Din+14a] **B. Dupont de Dinechin / Y. Durand / D. van Amstel / A. Ghiti** – "Guaranteed Services of the NoC of a Manycore Processor". *Proceedings of the 2014 International Workshop on Network on Chip Architectures*. ACM. 2014. p.11–16. (cit. on p. 29, 42, 133).

[Din+14b] **B. Dupont de Dinechin / D. van Amstel / M. Poulhiès / G. Lager** – "Time-critical Computing on a Single-chip Massively Parallel Processor". *Proceedings of the 2014 Conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2014. p.97:1–97:6. (cit. on p. 29, 36, 132).

[Dom+16] **L. Domagala / D. van Amstel / F. Rastello / P. Sadayappan** – "Register allocation and promotion through combined instruction scheduling and loop unrolling". *Proceedings of the 25th International Conference on Compiler Construction*. ACM. 2016. p.143–151. (cit. on p. 50, 68).

[PBA13] **A. Pietrek / F. Bouchez / D. van Amstel** – "A textual target-level intermediate representation for compiler exchange". Tech. rep. Kalray S.A. 2013. (cit. on p. 73).

Other references

[ABW06] **A. Arasu / S. Babu / J. Widom** - "The CQL continuous query language: semantic foundations and query execution". *The International Journal on Very Large Data Bases*. 152 - 2006. Springer-Verlag New York, Inc. p.121–142. (cit. on p. 81).

[AG01] **A. W. Appel / L. George** - "Optimal Spilling for CISC Machines with Few Registers". *Proceedings of the 22nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press. 2001. p.243–253. (cit. on p. 68).

[BCT92] **P. Briggs / K. D. Cooper / L. Torczon** - "Rematerialization". *SIGPLAN Notifications*. 277 - 1992. ACM. p.311–321. (cit. on p. 69).

[BCT94] **P. Briggs / K. D. Cooper / L. Torczon** - "Improvements to Graph Coloring Register Allocation". *ACM Transactions on Programming Languages and Systems*. 163 - 1994. ACM. p.428–455. (cit. on p. 11).

[BD13] **N. Beldiceanu / S. Demassey** - "Global Constraint Catalog" (2013). `http://www.emn.fr/z-info/sdemasse/gccat/`. (cit. on p. 55).

[BE11] **M. Bahi / C. Eisenbeis** - "Rematerialization-based register allocation through reverse computing." *Conference on Computing Frontiers*. ACM. 2011. p.24. (cit. on p. 69).

[BK75] **B. T. Bennett / V. J. Kruskal** - "LRU stack processing". *IBM Journal of Research and Development*. 194 - 1975. IBM Corp. p.353–357. (cit. on p. 105).

[Bil+95] **G. Bilsen / M. Engels / R. Lauwereins / J. Peperstraete** - "Cycle-static Dataflow". *International Conference on Acoustics, Speech, and Signal Processing*. 1995. p.3255–3258 vol.5. (cit. on p. 12).

[Bou+99] **P. Boulet / J. J. Dongarra / F. Rastello / Y. Robert et al.** - "Algorithmic Issues on Heterogeneous Computing Platforms". *Parallel Processing Letters*. 92 - 1999. World Scientific Publishing. p.197–213. (cit. on p. 125).

[Bri+08] **A. Brito / C. Fetzer / H. Sturzrehm / P. Felber** - "Speculative out-of-order event processing with software transaction memory". *Proceedings of the 2nd International Conference on Distributed Event-based Systems*. ACM. 2008. p.265–275. (cit. on p. 81).

[Bri92] **P. Briggs** – "Register Allocation via Graph Coloring". PhD thesis. Rice University. 1992. (cit. on p. 11).

[Bro+00] **S. Browne / J. Dongarra / N. Garner / K. London et al.** – "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters". *Proceedings of SuperComputing 2000*. 2000. (cit. on p. 109).

[Buc+04] **I. Buck / T. Foley / D. Horn / J. Sugerman et al.** – "Brook for GPUs: stream computing on graphics hardware". *ACM Transactions on Graphics*. ACM. 2004. p.777–786. (cit. on p. 85).

[CB76] **E. G. Coffman / J. L. Bruno** – "Computer and job-shop scheduling theory". John Wiley & Sons. 1976. (cit. on p. 104).

[CBD11] **Q. Colombet / F. Brandner / A. Darte** – "Studying Optimal Spilling in the Light of SSA". *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM. 2011. p.25–34. (cit. on p. 68, 69).

[CCK90] **D. Callahan / S. Carr / K. Kennedy** – "Improving Register Allocation for Subscripted Variables". *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. ACM. 1990. p.53–65. (cit. on p. 69).

[CH90] **F. C. Chow / J. L. Hennessy** – "The Priority-based Coloring Approach to Register Allocation". *ACM Transactions on Programming Languages and Systems*. 124 - 1990. ACM. p.501–536. (cit. on p. 11, 68).

[CRA09] **P. M. Carpenter / A. Ramirez / E. Ayguade** – "Mapping Stream Programs Onto Heterogeneous Multiprocessor Systems". *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM. 2009. p.57–66. (cit. on p. 92).

[CW91] **C. K. Cheng / Y. C. A. Wei** – "An improved two-way partitioning algorithm with stable performance [VLSI]". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1012 - 1991. p.1502–1511. (cit. on p. 92).

[Car+03] **D. Carney / U. Çetintemel / A. Rasin / S. Zdonik et al.** – "Operator scheduling in a data stream manager". *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB Endowment. 2003. p.838–849. (cit. on p. 81, 139).

[Cha+81] **G. J. Chaitin / M. A. Auslander / A. K. Chandra / J. Cocke et al.** – "Register Allocation via Coloring". *Computer Languages*. 61 – 1981. Pergamon Press, Inc. p.47–57. (cit. on p. 11, 68).

[Cha82] **G. J. Chaitin** – "Register Allocation & Spilling via Graph Coloring". *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. ACM. 1982. p.98–105. (cit. on p. 11).

[Coh+02] **E. Cohen / E. Halperin / H. Kaplan / U. Zwick** – "Reachability and Distance Queries via 2-hop Labels". *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2002. p.937–946. (cit. on p. 91).

[Con+03] **C. Consel / H. Hamdi / L. Réveillère / L. Singaravelu et al.** – "Spidle: a DSL approach to specifying streaming applications". *Generative Programming and Component Engineering*. Springer. 2003. p.1–17. (cit. on p. 81, 139).

[Cru91a] **R. Cruz** – "A calculus for network delay. I. Network elements in isolation". *IEEE Transactions on Information Theory*. 371 – 1991. p.114 –131. (cit. on p. 33).

[Cru91b] **R. Cruz** – "A calculus for network delay. II. Network analysis". *IEEE Transactions on Information Theory*. 371 – 1991. p.132–141. (cit. on p. 33).

[Cyt+91] **R. Cytron / J. Ferrante / B. K. Rosen / M. N. Wegman et al.** – "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Transactions on Programming Languages and Systems*. 134 – 1991. ACM. p.451–490. (cit. on p. 137).

[Den68] **P. J. Denning** – "The working set model for program behavior". *Communications of the ACM*. 115 – 1968. ACM. p.323–333. (cit. on p. 11).

[Den74] **J. B. Dennis** – "First version of a data flow procedure language". *Programming Symposium*. Springer. 1974. p.362–376. (cit. on p. 81).

[Din+00] **B. Dupont de Dinechin / F. de Ferrière / C. Guillon / A. Stoutchinin** – "Code Generator Optimizations for the ST120 DSP-MCU Core". *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM. 2000. p.93–102. (cit. on p. 72, 137).

[Dlu+14] **P. Dlugosch / D. Brown / P. Glendenning / M. Leventhal et al.** - "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing". *IEEE Transactions on Parallel and Distributed Systems*. 2512 - 2014. p.3088–3098. (cit. on p. 3, 128).

[Ela+14] **V. Elango / F. Rastello / L.-N. Pouchet / J. Ramanujam et al.** - "On Characterizing the Data Movement Complexity of Computational DAGs for Parallel Execution". *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM. 2014. p.296–306. (cit. on p. 12).

[Ela+15a] **V. Elango / F. Rastello / L.-N. Pouchet / J. Ramanujam et al.** - "On Characterizing the Data Access Complexity of Programs". *Proceedings of the 42th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 2015. p.567–580. (cit. on p. 12).

[Ela+15b] **V. Elango / N. Sedaghati / F. Rastello / L.-N. Pouchet et al.** - "On Using the Roofline Model with Lower Bounds on Data Movement". *ACM Transactions on Architure and Code Optimization*. 114 - 2015. ACM. p.67:1–67:23. (cit. on p. 12).

[Far+11] **S. M. Farhad / Y. Ko / B. Burgstaller / B. Scholz** - "Orchestration by Approximation: Mapping Stream Programs Onto Multicore Architectures". *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2011. p.357–368. (cit. on p. 82).

[Fau+13] **N. Fauzia / V. Elango / M. Ravishankar / J. Ramanujam et al.** - "Beyond Reuse Distance Analysis: Dynamic Analysis for Characterization of Data Locality Potential". *ACM Transactions on Architure and Code Optimization*. 104 - 2013. ACM. p.53:1–53:29. (cit. on p. 92).

[GA96] **L. George / A. W. Appel** - "Iterated Register Coalescing". *ACM Transactions on Programming Languages and Systems*. 183 - 1996. p.300–324. (cit. on p. 68).

[GH88] **J. R. Goodman / W.-C. Hsu** - "Code scheduling and register allocation in large basic blocks". *Proceedings of the 2nd International Conference on Supercomputing*. ACM. 1988. p.442–452. (cit. on p. 68).

[GN94] **C. J. Glass / L. M. Ni** - "The Turn Model for Adaptive Routing". *Journal of the ACM*. 415 - 1994. ACM. p.874–902. (cit. on p. 38, 133).

[GTA06] **M. I. Gordon / W. Thies / S. Amarasinghe** - "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs". *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2006. p.151–162. (cit. on p. 81, 82).

[Gau+13] **T. Gautier / J. Lima / N. Maillard / B. Raffin** - "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures". *27th IEEE International Symposium on Parallel Distributed Processing*. 2013. p.1299–1308. (cit. on p. 81).

[Gou+11] **T. Goubier / R. Sirdey / S. Louise / V. David** - "ΣC: A programming model and language for embedded manycores". *Algorithms and Architectures for Parallel Processing*. Springer. 2011. p.385–394. (cit. on p. 8).

[HK92] **L. Hagen / A. B. Kahng** - "A new approach to effective circuit clustering". *IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers.* 1992. p.422–427. (cit. on p. 92).

[HL95] **B. Hendrickson / R. Leland** - "A Multilevel Algorithm for Partitioning Graphs". *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. ACM. 1995. (cit. on p. 92, 102).

[HR95] **M. T. Heath / P. Raghavan** - "A Cartesian parallel nested dissection algorithm". *SIAM Journal on Matrix Analysis and Applications*. 161 - 1995. SIAM. p.235–253. (cit. on p. 92).

[Hal+91] **N. Halbwachs / P. Caspi / P. Raymond / D. Pilaud** - "The synchronous data flow programming language LUSTRE". *Proceedings of the IEEE*. 799 - 1991. p.1305–1320. (cit. on p. 81).

[Hir+14] **M. Hirzel / R. Soulé / S. Schneider / B. Gedik et al.** - "A Catalog of Stream Processing Optimizations". *ACM Computing Surveys*. 464 - 2014. ACM. p.46:1–46:34. (cit. on p. 81).

[IT88] **F. Irigoin / R. Triolet** - "Supernode Partitioning". *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 1988. p.319–329. (cit. on p. 4, 128).

[JED15] **JEDEC** - "High Bandwidth Memory (HBM) DRAM" (2015). (cit. on p. 3, 128).

[JWK81] **H. Jia-Wei / H. T. Kung** - "I/O Complexity: The Red-blue Pebble Game". *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*. ACM. 1981. p.326–333. (cit. on p. 12).

[Jaf+08] **F. Jafari / M. H. Yaghmaee / M. S. Talebi / A. Khonsari** – "Max-Min-Fair Best Effort Flow Control in Network-on-Chip Architectures". *Proceedings of the 8th International Conference on Computational Science, Part I*. Springer-Verlag. 2008. p.436–445. (cit. on p. 34).

[Jaf+10] **F. Jafari / Z. Lu / A. Jantsch / M. H. Yaghmaee** – "Optimal Regulation of Traffic Flows in Networks-on-chip". *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2010. p.1621–1624. (cit. on p. 34).

[Jag+12] **D. Jagtap / K. Bahulkar / D. Ponomarev / N. Abu-Ghazaleh** – "Characterizing and Understanding PDES Behavior on Tilera Architecture". *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society. 2012. p.53–62. (cit. on p. 29).

[Jin+09] **R. Jin / Y. Xiang / N. Ruan / D. Fuhry** – "3HOP: A High-compression Indexing Scheme for Reachability Query". *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM. 2009. p.813–826. (cit. on p. 91).

[Jin+11] **R. Jin / N. Ruan / Y. Xiang / H. Wang** – "Path-tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs". *ACM Transactions on Database Systems*. 361 - 2011. ACM. p.7:1–7:44. (cit. on p. 91).

[Jin+12] **R. Jin / N. Ruan / S. Dey / J. Y. Xu** – "SCARAB: Scaling Reachability Computation on Large Graphs". *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012. p.169–180. (cit. on p. 92).

[KK95] **G. Karypis / V. Kumar** – "Multi-level graph partitioning schemes". *Proceedings of the 1995 International Conference for Parallel Processing*. 1995. p.113–122. (cit. on p. 92).

[KL70] **B. W. Kernighan / S. Lin** – "An efficient heuristic procedure for partitioning graphs". *Bell Systems Technical Journal*. 492 - 1970. Wiley Online Library. p.291–307. (cit. on p. 92).

[KM08] **M. Kudlur / S. Mahlke** – "Orchestrating the Execution of Stream Programs on Multicore Platforms". *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2008. p.114–124. (cit. on p. 82).

[KMT98] **F. Kelly / A. Maulloo / D. Tan** - "Rate control for communication networks: shadow prices, proportional fairness and stability". *Journal of the Operational Research Society*. 493 - 1998. Palgrave Macmillan. p.237–252. (cit. on p. 33, 42).

[LA04] **C. Lattner / V. Adve** - "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society. 2004. p.75–. (cit. on p. 73, 138).

[LBT01] **J.-Y. Le Boudec / P. Thiran** - "Network calculus: a theory of deterministic queuing systems for the internet". Springer Science & Business Media. 2001. (cit. on p. 34).

[LC97] **J. Lu / K. D. Cooper** - "Register Promotion in C Programs". *Proceedings of the 18th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 1997. p.308–319. (cit. on p. 69).

[LM87] **E. Lee / D. Messerschmitt** - "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". *IEEE Transactions on Computers*. C-361 - 1987. p.24–35. (cit. on p. 12, 81, 130).

[Lo+98] **R. Lo / F. Chow / R. Kennedy / S.-M. Liu et al.** - "Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores". *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 1998. p.26–37. (cit. on p. 69).

[MC69] **A. C. McKellar / E. G. Coffman** - "Organizing Matrices and Matrix Operations for Paged Memory Systems". *Communications of the ACM*. 123 - 1969. ACM. p.153–165. (cit. on p. 4).

[MR79] **E. Morel / C. Renvoise** - "Global Optimization by Suppression of Partial Redundancies". *Communications of the ACM*. 222 - 1979. ACM. p.96–103. (cit. on p. 69).

[MTV91] **G. L. Miller / S. H. Teng / S. A. Vavasis** - "A unified geometric approach to graph separators". *Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science*. 1991. p.538–547. (cit. on p. 92).

[Ma07] **Y. Ma** - "Register Pressure Guided Loop Optimization". PhD thesis. Michigan Technological University. 2007. (cit. on p. 69).

[Man+93] **N. Mansour / R. Ponnusamy / A. Choudhary / G. C. Fox** – "Graph Contraction for Physical Optimization Methods: A Quality-cost Tradeoff for Mapping Data on Parallel Computers". *Proceedings of the 7th International Conference on Supercomputing*. ACM. 1993. p.1–10. (cit. on p. 92).

[Mar+09] **R. Marculescu / U. Y. Ogras / L.-S. Peh / N. E. Jerger et al.** – "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives". *Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 281 - 2009. IEEE Press. p.3–21. (cit. on p. 28).

[Mat+70] **R. L. Mattson / J. Gecsei / D. R. Slutz / I. L. Traiger** – "Evaluation techniques for storage hierarchies". *IBM Systems journal*. 92 - 1970. IBM Corp. p.78–117. (cit. on p. 11, 105).

[McK04] **S. A. McKee** – "Reflections on the Memory Wall". *Proceedings of the 1st Conference on Computing Frontiers*. ACM. 2004. p.162–. (cit. on p. 3, 128).

[Mot+95] **R. Motwani / K. V. Palem / V. Sarkar / S. Reyen** – "Combining register allocation and instruction scheduling". *Courant Institute, New York University*. - 1995. (cit. on p. 68).

[OM06] **U. Y. Ogras / R. Marculescu** – "Prediction-based Flow Control for Network-on-Chip Traffic". *Proceedings of the 43rd Annual Design Automation Conference*. ACM. 2006. p.839–844. (cit. on p. 33).

[PSL90] **A. Pothen / H. D. Simon / K.-P. Liou** – "Partitioning sparse matrices with eigenvectors of graphs". *SIAM Journal on Matrix Analysis and Applications*. 113 - 1990. SIAM. p.430–452. (cit. on p. 92).

[Pei86] **J.-K. Peir** – "Program partitioning and synchronization on multiprocessor systems". PhD thesis. Illinois Univ., Urbana, USA. 1986. (cit. on p. 4, 128).

[Pie12] **A. Pietrek** – "TIREX : A textual target-level intermediate representation for virtual execution environment, compiler information exchange and program analysis". PhD thesis. Université de Grenoble. 2012. (cit. on p. 72, 137).

[Pop13] **A. Pop** – "OpenStream: A Data-flow Approach to Solving the Von Neumann Bottlenecks". *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*. ACM. 2013. p.2–2. (cit. on p. 81).

[RBW06] **F. Rossi / P. v. Beek / T. Walsh** – "Handbook of Constraint Programming (Foundations of Artificial Intelligence)". Elsevier Science Inc. 2006. (cit. on p. 55).

[RRR05] **L. Renganarayanan / U. Ramakrishna / S. V. Rajopadhye** – "Combined ILP and Register Tiling: Analytical Model and Optimization Framework". *18th International Workshop on Languages and Compilers for Parallel Computing*. 2005. p.244–258. (cit. on p. 69).

[Rad+15] **M. Radulovic / D. Zivanovic / D. Ruiz / B. R. de Supinski et al.** – "Another Trip to the Wall: How Much Will Stacked DRAM Benefit HPC?" *Proceedings of the 2015 International Symposium on Memory Systems*. ACM. 2015. p.31–36. (cit. on p. 3, 128).

[Ran+04] **R. Rangan / N. Vachharajani / M. Vachharajani / D. I. August** – "Decoupled Software Pipelining with the Synchronization Array". *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 2004. p.177–188. (cit. on p. 125, 144).

[Rég94] **J.-C. Régin** – "A Filtering Algorithm for Constraints of Difference in CSPs". *Proceedings of the 1994 National Conference on Artificial Intelligence*. American Association for Artificial Intelligence. 1994. p.362–367. (cit. on p. 56).

[Rég96] **J.-C. Régin** – "Generalized arc consistency for global cardinality constraint". *Proceedings of the 1995 National Conference on Artificial Intelligence*. AAAI Press. 1996. p.209–215. (cit. on p. 56).

[SJ98] **A. V. S. Sastry / R. D. C. Ju** – "A New Algorithm for Scalar Register Promotion Based on SSA Form". *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 1998. p.15–25. (cit. on p. 69).

[SK13] **R. Szymanek / K. Kuchcinski** – "JaCoP – Java Constraint Programming solver" (2013). http://jacop.osolpro.com/. (cit. on p. 56).

[SMCCL11] **P. Stanley-Marbell / V. Caparros Cabezas / R. Luijten** – "Pinned to the Walls: Impact of Packaging and Application Properties on the Memory and Power Walls". *Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design*. IEEE Press. 2011. p.51–56. (cit. on p. 3, 128).

[SPN96] **A. Saulsbury / F. Pong / A. Nowatzyk** – "Missing the Memory Wall: The Case for Processor/Memory Integration". *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. ACM. 1996. p.90–101. (cit. on p. 2, 128).

[Sei+88] **C. L. Seitz / W. C. Athas / C. M. Flaig / A. J. Martin et al.** – "The Architecture and Programming of the Ametek Series 2010 Multicomputer". *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1*. ACM. 1988. p.33–37. (cit. on p. 38, 133).

[Ser+05] **J. Sermulins / W. Thies / R. Rabbah / S. Amarasinghe** – "Cache Aware Optimization of Stream Programs". *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM. 2005. p.115–126. (cit. on p. 81).

[Sim88] **K. Simon** – "An improved algorithm for transitive closure on acyclic digraphs". *Theoretical Computer Science*. 581 - 1988. p.325–346. (cit. on p. 91).

[TKA02] **W. Thies / M. Karczmarek / S. P. Amarasinghe** – "StreamIt: A Language for Streaming Applications". *Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag. 2002. p.179–196. (cit. on p. 81).

[TL11] **M. Tang / X. Lin** – "Injection Level Flow Control for Networks-on-Chip (NoC)." *Journal of Information Science and Engineering*. 272 - 2011. p.527–544. (cit. on p. 33).

[Vel+14] **R. R. Veloso / L. Cerf / W. Meira Jr / M. J. Zaki** – "Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach." *Proceedings of the 17th Conference on Extending Database Technology*. OpenProceedings.org. 2014. p.511–522. (cit. on p. 92).

[WCB01] **M. Welsh / D. Culler / E. Brewer** – "SEDA: an architecture for well-conditioned, scalable internet services". *ACM SIGOPS Operating Systems Review*. 355 - 2001. ACM. p.230–243. (cit. on p. 81, 139).

[WM95] **W. A. Wulf / S. A. McKee** – "Hitting the Memory Wall: Implications of the Obvious". *SIGARCH Computer Architecture News*. 231 - 1995. ACM. p.20–24. (cit. on p. 2, 128).

[Wat09] **D. Watt** - "Programming XC on XMOS devices". XMOS Limited. 2009. (cit. on p. 85).

[Wol89] **M. Wolfe** - "Iteration Space Tiling for Memory Hierarchies". *Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics. 1989. p.357–361. (cit. on p. 4, 128).

[Xia+13] **X. Xiang / C. Ding / H. Luo / B. Bao** - "HOTL: A Higher Order Theory of Locality". *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2013. p.343–356. (cit. on p. 11).

[YCZ12] **H. Yildirim / V. Chaoji / M. J. Zaki** - "GRAIL: A Scalable Index for Reachability Queries in Very Large Graphs". *The International Journal on Very Large Databases*. 214 - 2012. Springer-Verlag New York, Inc. p.509–534. (cit. on p. 91).

[ZX16] **H. Zhou / J. Xue** - "Exploiting Mixed SIMD Parallelism by Reducing Data Reorganization Overhead". *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM. 2016. p.59–69. (cit. on p. 125).

[Zha95] **H. Zhang** - "Service disciplines for guaranteed performance service in packet switched networks". *Proceedings of the IEEE*. 8310 - 1995. p.1374–1396. (cit. on p. 34).

[Hyb15] **Hybrid Memory-Cube Consortium** - "HMC Specification 2.1" (2015). (cit. on p. 3, 128).

[ÇA95] **Ü. V. Çatalyürek / C. Aykanat** - "A hypergraph model for mapping repeated sparse matrix-vector product computations on multicomputers". *Proceedings of the International Conference on High Performance Computing*. 1995. (cit. on p. 92).

[ÇA96] **Ü. V. Çatalyürek / C. Aykanat** - "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication". *Parallel Algorithms for Irregularly Structured Problems*. Springer. 1996. p.75–86. (cit. on p. 108).

List of Figures

List of Tables

List of Algorithms

Résumé

L'évolution continue des architectures des processeurs a été un moteur important de la recherche en compilation. Une tendance dans cette évolution qui existe depuis l'avènement des ordinateurs modernes est le rapport grandissant entre la puissance de calcul disponible (IPS, FLOPS, …) et la bande-passante correspondante qui est disponible entre les différents niveaux de la hiérarchie mémoire (registres, cache, mémoire vive). En conséquence la réduction du nombre de communications mémoire requis par un code donnée a constitué un sujet de recherche important. Un principe de base en la matière est l'amélioration de la localité temporelle des données : regrouper dans le temps l'ensemble des accès à une donnée précise pour qu'elle ne soit requise que pendant peu de temps et pour qu'elle puisse ensuite être transféré vers de la mémoire lointaine (mémoire vive) sans communications supplémentaires.

Une toute autre évolution architecturale a été l'arrivée de l'ère des multicœurs et au cours des dernières années les premières générations de processeurs manycœurs. Ces architectures ont considérablement accru la quantité de parallélisme à la disposition des programmes et algorithmes mais ceci est à nouveau limité par la bande-passante disponible pour les communications entres cœurs. Ceci a amène dans le monde de la compilation et des techniques d'optimisation des problèmes qui étaient jusqu'à là uniquement connus en calcul distribué.

Dans ce texte nous présentons les premiers travaux sur une nouvelle technique d'optimisation, le *pavage généralisé* qui a l'avantage d'utiliser un modèle abstrait pour la réutilisation des données et d'être en même temps utilisable dans un grand nombre de contextes. Cette technique trouve son origine dans le pavage de boucles, une techniques déjà bien connue et qui a été utilisé avec succès pour l'amélioration de la localité des données dans les boucles imbriquées que ce soit pour les registres ou pour le cache. Cette nouvelle variante du pavage suit une vision beaucoup plus large et ne se limite pas au cas des boucles imbriquées. Elle se base sur une nouvelle représentation, le *graphe d'utilisation mémoire*, qui est étroitement lié à un nouveau modèle de besoins en termes de mémoire et de communications et qui s'applique à toute forme de code exécuté itérativement.

Le *pavage généralisé* exprime la localité des données comme un problème d'optimisation pour lequel plusieurs solutions sont proposées. L'abstraction faite par le *graphe d'utilisation mémoire* permet la résolution du problème d'optimisation dans différents contextes. Pour l'évaluation expérimentale nous montrons comment utiliser cette nouvelle technique dans le cadre des boucles, imbriquées ou non, ainsi que dans le cas des programmes exprimés dans un langage à flot-de-données. En anticipant le fait d'utiliser le *pavage généralisé* pour la distribution des calculs entre les cœurs d'une architecture manycœurs nous donnons aussi des éléments de réponse pour modéliser les communications et leurs caractéristiques sur ce genre d'architectures.

En guise de point final, et pour montrer l'étendue de l'expressivité du *graphe d'utilisation mémoire* et le modèle de besoins en mémoire et communications sous-jacent, nous aborderons le sujet du débogage de performances et l'analyse des traces d'exécution. Notre but est de fournir un retour sur le potentiel d'amélioration en termes de localité des données du code évalué. Ce genre de traces peut contenir des informations au sujet des communications mémoire durant l'exécution et a de grandes similitudes avec le problème d'optimisation précédemment étudié. Ceci nous amène à une brève introduction dans le monde de l'algorithmique des graphes dirigés et la mise-au-point de quelques nouvelles heuristiques pour le problème connu de joignabilité mais aussi pour celui bien moins étudié du partitionnement convexe.

Abstract

The continuous evolution of computer architectures has been an important driver of research in code optimization and compiler technologies. A trend in this evolution that can be traced back over decades is the growing ratio between the available computational power (IPS, FLOPS, …) and the corresponding bandwidth between the various levels of the memory hierarchy (registers, cache, DRAM). As a result the reduction of the amount of memory communications that a given code requires has been an important topic in compiler research. A basic principle for such optimizations is the improvement of temporal data locality: grouping all references to a single data-point as close together as possible so that it is only required for a short duration and can be quickly moved to distant memory (DRAM) without any further memory communications.

Yet another architectural evolution has been the advent of the multicore era and in the most recent years the first generation of manycore designs. These architectures have considerably raised the bar of the amount of parallelism that is available to programs and algorithms but this is again limited by the available bandwidth for communications between the cores. This brings some issues that previously were the sole preoccupation of distributed computing to the world of compiling and code optimization techniques.

In this document we present a first dive into a new optimization technique which has the promise of offering both a high-level model for data reuses and a large field of potential applications, a technique which we refer to as *generalized tiling*. It finds its source in the already well-known loop tiling technique which has been applied with success to improve data locality for both register and cache-memory in the case of nested loops. This new "flavor" of tiling has a much broader perspective and is not limited to the case of nested loops. It is build on a new representation, the *memory-use graph*, which is tightly linked to a new model for both memory usage and communication requirements and which can be used for all forms of iterate code.

Generalized tiling expresses data locality as an optimization problem for which multiple solutions are proposed. With the abstraction introduced by the *memory-use graph* it is possible to solve this optimization problem in different environments. For experimental evaluations we show how this new technique can be applied in the contexts of loops, nested or not, as well as for computer programs expressed within a dataflow language. With the anticipation of using *generalized tiling* also to distributed computations over the cores of a manycore architecture we also provide some insight into the methods that can be used to model communications and their characteristics on such architectures.

As a final point, and in order to show the full expressiveness of the *memory-use graph* and even more the underlying memory usage and communication model, we turn towards the topic of performance debugging and the analysis of execution traces. Our goal is to provide feedback on the evaluated code and its potential for further improvement of data locality. Such traces may contain information about memory communications during an execution and show strong similarities with the previously studied optimization problem. This brings us to a short introduction to the algorithmics of directed graphs and the formulation of some new heuristics for the well-studied topic of reachability and the much less known problem of convex partitioning.